Langage de programmation

C++

Année Universitaire: 2009/2010

Plan

Programmation Orienté Objet
Structure de base du langage C++
Programmation en langage en C++

Programmation Orienté

Objet

Programmation Orienté Objet

La programmation orienté objet est une nouvelle méthode de programmation qui tend à se rapprocher de notre manière naturelle d'appréhender le monde. La P.O.O. s'est surtout posé la question "Sur quoi porte le programme ?".

En effet, un programme informatique comporte toujours des traitements, mais aussi et surtout des données. Si la programmation structurée s'intéresse aux traitements puis aux données, la conception objet s'intéresse d'abord aux données, auxquelles elle associe ensuite les traitements, il est donc intéressant d'architecturer le programme autour de ces données.

Caractéristiques de Programmation Orienté Objet

Les caractéristiques de la Programmation Orienté Objet sont :

- Exactitude : aptitude d'un programme à fournir les résultats voulus, dans des conditions normales d'utilisation.
- Extensibilité : facilité avec laquelle un programme pourra être adapté pour satisfaire à une évolution des spécifications.
- Réutilisabilité : possibilité d'utiliser certaines parties (modules) d'un programme pour résoudre un problème.
- Portabilité : facilité avec laquelle on peut exploiter un même programme dans des différentes implémentations.
- Efficience : temps d'exécution, taille mémoire…

Concepts de Programmation Orienté Objet

1- Objet:

L'Objet est une représentation informatique des éléments du monde réel. La P.O.O est fondé sur le concept d'objet, à savoir une association des données et des procédures (qu'on appelle des méthodes) agissant sur les données.
L'objet est une entité atomique, caractérisé par des données et des méthodes.

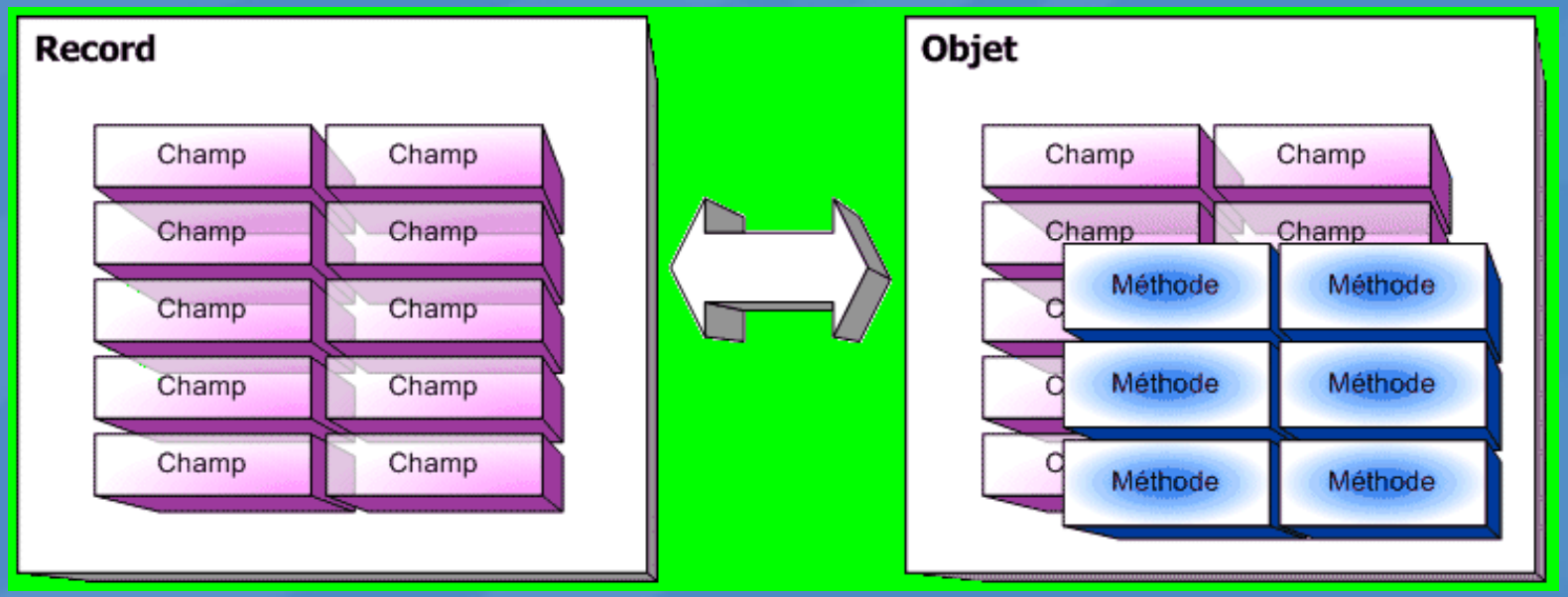
Méthode+Données= Objet

Les **données** :sont à l'objet ce que les variables sont à un programme, ce sont des variables stockant des informations d'état de l'objet.

Les **méthodes** : sont les éléments d'un objet qui servent d'interface entre les données et le programme.

Programmation Orienté Objet

1- Objet:



Programmation Orienté Objet

2- Classe:

On appelle classe la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Les objets de même nature ont en général la même structure et le même comportement. La classe factorise les caractéristiques communes de ces objets et permet de les classifier. Un objet est donc, une instanciation d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'objet ou d'instance (éventuellement d'occurrence). Toutes les instances d'une classe constituent l'extension de la classe.

Programmation Orienté Objet

3- Encapsulation :

Le concept d'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intègrité des données contenues dans l'objet.

L'encapsulation permet de définir des niveaux de visibilité des éléments de la classe. ces niveaux de visibilité définissent les droits d'accès aux données selon que l'on y accède par une méthode.

Programmation Orienté Objet

3- Encapsulation :

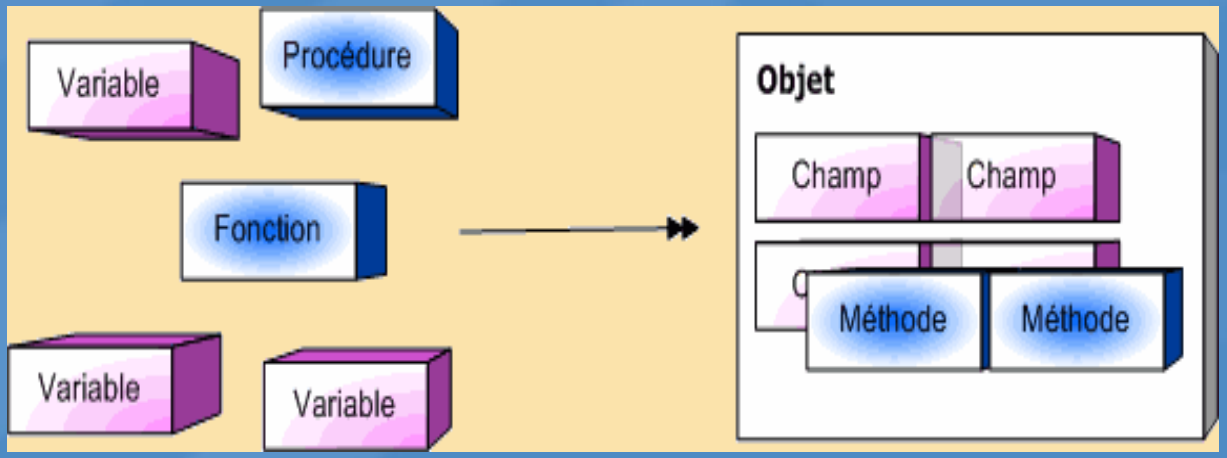
Il existe trois niveaux de visibilité :

- **Publique** : Les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité public. Il s'agit du plus bas niveau de protection des données.
- **Protégée** : l'accès aux données est réservé aux fonctions des classes héritières, c'est-à-dire par les fonctions membres de la classe ainsi que des classes dérivées.
- **Privée** : l'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé.

Programmation Orienté Objet

3- Encapsulation :

L'encapsulation permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.



Programmation Orienté Objet

3- Encapsulation :

Il existe trois niveaux de visibilité :

- **Publique** : Les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité public. Il s'agit du plus bas niveau de protection des données.
- **Protégée** : l'accès aux données est réservé aux fonctions des classes héritières, c'est-à-dire par les fonctions membres de la classe ainsi que des classes dérivées.
- **Privée** : l'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé.

Programmation Orienté Objet

4- Héritage:

L'héritage est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Le nom d'héritage (ou parfois dérivation de classe) provient du fait que la classe dérivée (la classe nouvellement créée ou bien classe fille) contient les attributs et les méthodes de sa superclasse (la classe dont elle dérive ou bien classe mère).

L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées.

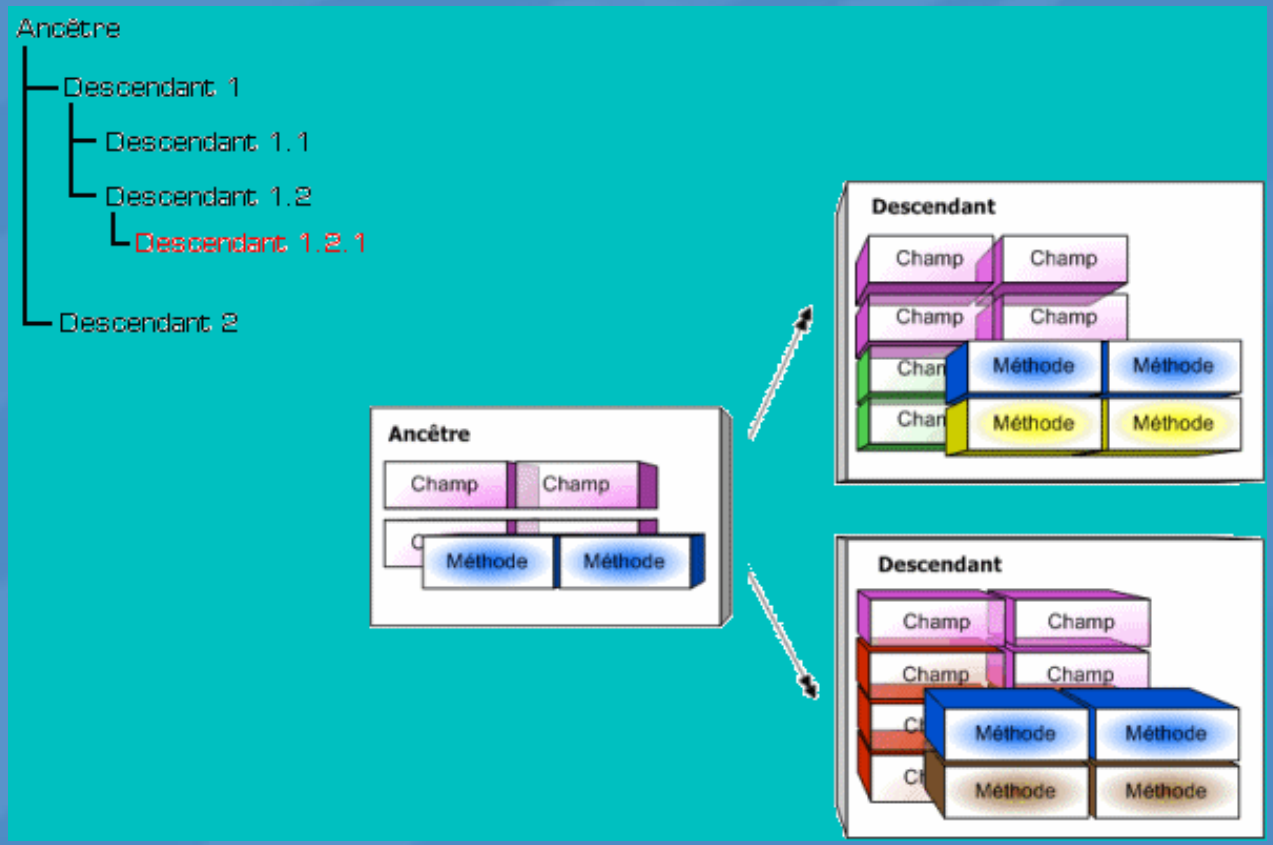
Programmation Orienté Objet

4- Héritage:

Certains langages orientés objet, tels que le C++, permettent de faire de l'héritage multiple, ce qui signifie qu'ils offrent la possibilité de faire hériter une classe de deux superclasse. Ainsi, cette technique permet de regrouper au sein d'une seule et même classe les attributs et les méthodes de plusieurs classes.

Programmation Orienté Objet

4- Héritage:



Programmation Orienté Objet

5- Polymorphisme :

Le nom de polymorphisme vient du grec et signifie qui peut prendre plusieurs formes. Cette caractéristique essentielle de la programmation orientée objet caractérise la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents (en nombre et/ou en type), si bien que la bonne fonction sera choisie en fonction de ses paramètres lors de l'appel.

Structure de base du langage C++

Langage C++

Le langage C++ est une amélioration du langage C (le langage C a été mis au point par M.Ritchie et B.W.Kernighan au début des années 70). Bjarne Stroustrup, un ingénieur considéré comme l'inventeur du C++, a en effet décidé d'ajouter au langage C les propriétés de l'approche orientée objet. Ainsi, vers la fin des années 80 un nouveau langage, baptisé C with classes (C avec des classes), apparaît. Celui-ci a ensuite été renommé en C++, clin d'œil au symbole d'incrémentation ++ du langage C, afin de signaler qu'il s'agit d'un langage C amélioré (langage C+1).

Langage C++

Le C++ reprend la quasi-intégralité des concepts présents dans le langage C, si bien que les programmes écrits en langage C fonctionnent avec un compilateur C++. En réalité le langage C++ est un surensemble du C, il y ajoute, entre autres, des fonctionnalités objet :

- L'encapsulation
- L'héritage (simple et multiple)
- Le polymorphisme

Ainsi qu'un ensemble de nouvelles fonctionnalités, parmi lesquelles :

- Le contrôle de type
- Les arguments par défaut
- La surcharge de fonctions
- Les fonctions inline…

Types des variables

1- Types:

Les données manipulées en langage C++ sont typées, c'est-à-dire que pour chaque donnée que l'on utilise (dans les variables par exemple) il faut préciser le type de donnée, ce qui permet de connaître l'occupation mémoire (le nombre d'octets) de la donnée ainsi que sa représentation.

Le type des données peut être :

- Simple;
- Composé.

Types des variables

Les types élémentaire en langage C++ sont :

- **int** : entiers (au min 16 bits, pour des valeurs 32767), d'autres formats existent: long int (min 32 bits), short int, unsigned int ;
- **float**, double et long double : nombres à virgule flottante ;
- **char** : caractères ('a','b',. . . 'A',. . . ,':',. . .);
- **bool** : booléens ('true' ou 'false).

(Et: 0 équivalent à false, tout entier non nul équivaut à false)

Définition des variables

Le syntaxe pour la définition d'une variable est :

Type var1[=val1], var2[=val2], …;

Exemple: int p=2 ;
double x ;

- Une variable peut être déclarée n'importe où dans un bloc pourvu que sa déclaration soit placée avant son utilisation. Contrairement, en langage C, il est obligatoire de la déclarer au début de bloc.
- Les valeurs d'initialisations (val1, val2, …) peuvent être des expressions quelconques, elles peuvent faire intervenir des variables déjà déclarées, mais initialisée n'importe où dans un bloc, alors qu'en C, elles ne peuvent faire intervenir que des variables initialisées lors de leurs déclaration au début du bloc.

Définition des variables

Code correcte en C et en C++. Dans cet exemple, l'expression d'initialisation de q lors de sa définition fait intervenir la variable n déjà initialisée lors de sa définition au début du bloc.

```
int   n = 2;
int   q = 2 * n -1;
```

Code correcte en C++, mais pas en C. Dans cet exemple, la déclaration tardive de q permet de l'initialiser avec une expression qui fait intervenir la variable n dont la valeur n'était pas connue au moment de sa définition au début du bloc.

```
int   n;
 n = 2;
 int   q = 2 * n -1;
```

Constantes

La Syntaxe d'un constante en langage C++ est : **const** type nom = val ;

Par exemple: const float Pi = 3,14 ;

Il ne sera pas possible de modifier Pi dans le reste du programme (erreur à la compilation).

Chaînes de caractères

Pour le langage C++, il existe une classe string, ce n'est un pas un type élémentaire.
Pour l'utiliser, il faut placer en tête du programme :

include <string>

- string s: définit s comme une variable;
- string s = "bonjour" ;
- string t= « tout le monde"
- s.size() représente la longueur de s;
- s[i] est le i-ème caractère de s (i = 0,1,. . . s.size()-1)
- s+t est une nouvelle chaîne correspondant à la concaténation de s et t.

Tableaux

Le syntaxe pour la déclaration des tableaux en langage C++:

type Nom_du_tableau [Nombre d'éléments]

- L'indice du premier élément du tableau est 0 ;
- Un indice est toujours positif ;
- L'indice du dernier élément du tableau est égal au nombre d'éléments – 1;
- int tab [5]: tableau de 5 entiers ;
- tab.size() : correspond à la taille de tab;
- tab[7] = 6 : affecte la valeur 6 au huitième élément du tableau;
- int Toto[10] = {0}: initialise tous les éléments du tableau à zéro;
- int Tableau [3][4] : définit Un tableau d'entiers positifs à deux dimensions (3 lignes, 4 colonnes).

Expressions

1- Affectation:

En C/C++, l'affectation est une expression. Soient v une variable et expr une expression.

v = expr

Ce instruction permet d'affecter la valeur de expr à la variable v et retourne la valeur affectée à v comme résultat.
Par exemple, i = (j = 0) affecte 0 à j puis à i et retourne 0.

Expressions

2- Opérateurs classiques:

- Opérateurs arithmétiques : *, +, -, / (division entière et réelle), % (modulo);
- Opérateurs de comparaison :<, <= , ==, >, >=, !=;
- Opérateurs booléens: && représente l'opérateur "ET", || représente le "OU", et ! représente le "NON".

Expressions

3- Pré et Post incrément :

- ++var incrémente la variable var et retourne la nouvelle valeur (++i équivaut à i=i+1).
- var++ incrémente la variable var et retourne l'ancienne valeur (i++ équivaut à (i=i+1)-1).
- --var décrémente la variable var et retourne la nouvelle valeur.
- var-- écrémente la variable var et retourne l'ancienne valeur.

Entrées-Sorties

1- Affichage :

Pour utiliser les entrées/sorties en langage en C++, il faut ajouter cette ligne:

include <iostream>

Pour afficher des données sur l'écran, on utilise le flot de sortie cout.
cout désigne un flot de sortie prédéfini associé à la sortie standard (par défaut, c'est l'écran).

cout << expr1 << expr2 << … << exprn;

- expr1, expr2, …, exprn sont des expressions de type quelconque (caractère, entier, flottant, chaîne de caractères, pointeur, …).
- << : est un opérateur binaire dont l'opérande de gauche est un flot de sortie (ici cout) et l'opérande de droite une expression de type quelconque.
- Le résultat fourni par l'opérateur <<, quand-il reçoit un flot en premier opérande est ce même flot.

Entrées-Sorties

1- Affichage :

Exemple:

```
int   n = 25;
cout  <<  "Valeur  :  ";
cout  <<  n;
```

Ces deux instructions peuvent s'écrire :

```
cout << "Valeur  :  " << n;
```

➢ L'instruction : cout << "Valeur : "; signifie que le flot cout reçoit la valeur du type chaîne de caractères "Valeur : ".

➢ L'instruction : cout << "Valeur : " << n; est équivalente à :

(cout << "Valeur : ") << n;

= cout

Entrées-Sorties

1- Affichage :

- cout est un objet de la classe prédéfinie ostream ;
- Le mot clé endl sert à insérer un saut de ligne et à vider le tampon de sortie ;
- Pour sauter une ligne, on peut aussi utiliser le caractère '\n'.

Entrées-Sorties

2- Lecture :

Pour lire des données à partir du clavier sur l'écran, on utilise le flot d'entré cin.
cin désigne un flot d'entrée prédéfini associé à l'entrée standard (par défaut, c'est le clavier).

cin >> var1 >> var2 >> … >> varn;

- var1, var2, …, varn sont des variables de type de base quelconque (caractère, entier, flottant, chaînes de caractères).
- >> : est un opérateur binaire dont l'opérande de gauche est un flot d'entrée (ici cin) et l'opérande de droite est une variable dans laquelle on souhaite écrire une information.
- Le résultat fourni par l'opérateur >>, quand-il reçoit un flot en premier opérande est ce même flot.

Commentaires

Les commentaires en langage C++ sont représentés par deux barres obliques accolées (//). Le commentaire peut démarrer à n'importe quel endroit et s'étend jusqu'à la fin de ligne.

Instructions

1- Boucle IF :

Le syntaxe de la boucle « if else » dans le langage C++ est :

if (expr) instruction

if (expr) instruction1 else instruction 2

Exemples:

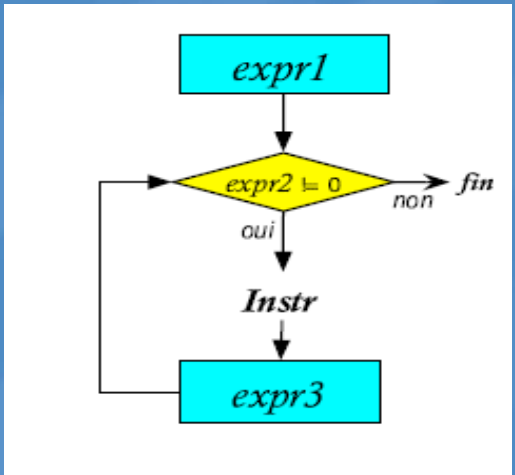
```
if (v == 3) i =i+4 ;
if ((v==3) && (i<5))
        { i=i+4 ;
        v=v*2 ;}
else v=i ;
        r = r*3 ;
```

Instructions

1- Boucle FOR :

Le syntaxe de la boucle « for » dans le langage C++ est :

for (expr1 ;expr2 ;expr3) instruction (s)



Exemples :

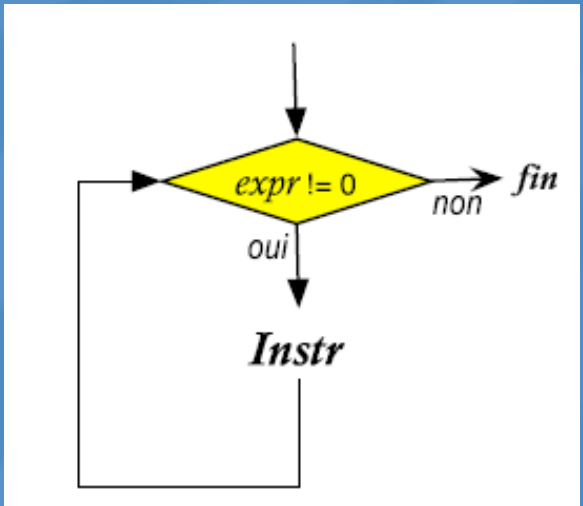
```
for(i=0 ;i<235 ;i=i+1) cout << T[i] ;
for(i=0,j=1 ;i<235 ;i=i+1,j=j+3) cout << T[i][j] ;
```

Instructions

1- Boucle WHILE :

Le syntaxe de la boucle « while » dans le langage C++ est :

while (expr) instruction (s)

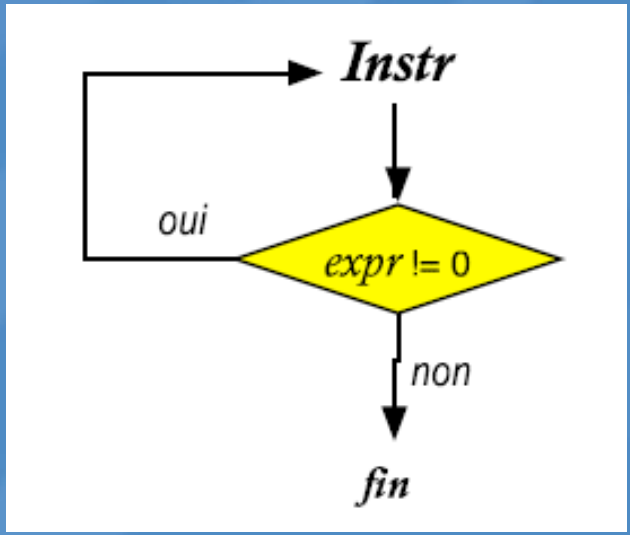


Instructions

1- Boucle DO :

Le syntaxe de la boucle « do while » dans le langage C++ est :

do instruction (s) while (expression(s)) ;



Instructions

Le langage C++ permet l'utilisation d'autre instructions:

- break provoque l'arrêt de la première instruction do, for, switch ou while englobant.
- continue provoque (dans une instruction do, for, ou while) l'arrêt de l'itération courante et le passage au début de l'itération suivante.
- switch (expression) {

```
case cas1 : liste d'instructions1
...
case casn : liste d'instructionsn
default: liste d'instructions
}
```

Instructions

(exp) ? exp1 : exp2 vaut exp1 si exp est évaluée à vrai et exp2 sinon.

```
m = (i < j) ? i : j
```

var op= expr désigne l'expression var = var op expr
(op prend les opérandes {+,-,*,/,%})

```
int i=2,j=34 ;
i += 2 ;
j *= 10 ;
j *= (i++) ;
i = i+1 ;
```

```
int i=2,j=34 ;
i = i+2 ;
j = j*10 ;
j = j*i ;
```

Structure générale d'un programme

Un programme C++ est reparti dans un ou plusieurs fichiers. Chacun peut contenir des définitions/déclarations de fonctions, des définitions de types et des définitions de variables globales.
Il existe une seule fonction main: c'est la fonction qui sera exécutée après la compilation.
Le profil de main est : int main() ou void main().

Portée, visibilité, durée de vie des variables

1- Portée:

La portée d'un identificateur correspond aux parties du programme où cet identificateur peut être utilisé sans provoquer d'erreur à la compilation.
La portée d'une variable globale ou d'une fonction globale est égale au programme.
La portée d'une variable locale va de sa définition jusqu'`a la fin de la première instruction composée ({...}) qui contient sa définition.
Deux variables peuvent avoir le même nom mais dans ce cas, elle doivent avoir deux port ´ es différentes.

Portée, visibilité, durée de vie des variables

1- Portée:

Exemple :

```
{
        int i=3 ;
{
        int i=5 ;
        cout << i ; // affiche 5
}
        cout << i ; // affiche 3
}
```

Les deux variables ont des portées différentes

Portée, visibilité, durée de vie des variables

2- Visibilité:

- La visibilité d'une variable dit quand elle est accessible;
- La durée de vie d'une variable correspond à la période depuis création é sa destruction.

L'espace mémoire d'une variable locale déclarée static est alloué une seule fois et n'est détruit qu'à la fin du programme !

```
void Test(int v)
        {
        static int compteur=0 ;
        compteur++ ;
        cout << compteur << "-eme appel de Test"<< endl ;
        cout << " v = " << v << endl ;
        }
```

Sa durée de vie est celle du programme.

Fonctions

1- Définition d'une fonction :

Le C++ ne permet de faire que des fonctions, pas de procédures. Une procédure peut être faite en utilisant une fonction ne renvoyant pas de valeur ou en ignorant la valeur retournée.
La définition des fonctions se fait comme suit :

```
type identificateur(paramètres)
    {
/* Instructions de la fonction */
    }
```

Type : le type de la valeur renvoyée;
Identificateur : le nom de la fonction;
Paramètres : la liste de paramètres.

Fonctions

1- Définition d'une fonction :

Remarque : L'initialisation des paramètres de fonctions n'est possible qu'en C++, le C n'accepte pas cette syntaxe.

La valeur de la fonction à renvoyer est spécifiée en utilisant la commande return, dont la syntaxe est : return valeur.

Fonctions

1- Définition d'une fonction :

Exemple d'une fonction en C++:

```
int somme(int i, int j)
{
        return i+j;
}
```

Si une fonction ne renvoie pas de valeur, on lui donnera le type void. Si elle n'attend pas de paramètres, sa liste de paramètres sera void ou n'existera pas. Il n'est pas nécessaire de mettre une instruction return à la fin d'une fonction qui ne renvoie pas de valeur.

Fonctions

1- Définition d'une procédure :

Exemple d'une procédure en C++:

```
void rien() /* Fonction n'attendant pas de paramètres */
{           /* et ne renvoyant pas de valeur. */
return;     /* Cette ligne est facultative.
}
```

Fonctions

2- Appel d'une fonction:

L'appel d'une fonction se fait en donnant son nom, puis les valeurs de ses paramètres entre parenthèses.
S'il n'y a pas de paramètres, il faut quand même mettre les parenthèses, sinon la fonction n'est pas appelée.

Si la déclaration comprend des valeurs par défaut pour des paramètres (C++ seulement), ces valeurs sont utilisées lorsque ces paramètres ne sont pas fournis lors de l'appel. Si un paramètre est manquant, alors tous les paramètres qui le suivent doivent eux aussi être omis. Il en résulte que seuls les derniers paramètres d'une fonction peuvent avoir des valeurs par défaut.

Fonctions

2- Appel d'une fonction:

Exemple d'une fonction:

```
int test(int i = 0, int j = 2)
{
return i/j;
}
```

L'appel de la fonction test(8) est valide. Comme on ne précise pas le dernier paramètre, j est initialisé à 2. Le résultat obtenu est donc 4. De même, l'appel test() est valide : dans ce cas i vaut 0 et j vaut 2. En revanche, il est impossible d'appeler la fonction test en ne précisant que la valeur de j. Enfin, l'expression « int test(int i=0, int j) {...} » serait invalide, car si on ne passait pas deux paramètres, j ne serait pas initialisé.

Fonctions

3- Déclaration d’une fonction:

Toute fonction doit être déclarée avant d'être appelée pour la première fois. Il peut se trouver des situations où une fonction doit être appelée dans une autre fonction définie avant elle. Comme cette fonction n'est pas définie au moment de l'appel, elle doit être déclarée. Le rôle des déclarations est donc de signaler l'existence des fonctions aux compilateurs afin de les utiliser, tout en reportant leur définition plus loin ou dans un autre fichier.

Fonctions

3- Déclaration d'une fonction:

Exemple de déclaration d'une fonction:

```
int Min(int, int);              /* Déclaration de la fonction minimum */
int main(void) {                /* Fonction principale. */
        int i = Min(2,3); /* Appel à la fonction Min, déjà déclarée. */
        return 0; }

int Min(int i, int j) {         /* Définition de la fonction min. */
        if (i<j) return i;
        else return j;}
```

Fonctions

4- Surcharge d'une fonction:

Il est donc possible de faire des fonctions de même nom (surcharge) si et seulement si toutes les fonctions portant ce nom peuvent être distinguées par leurs signatures. La surcharge qui sera appelée sera celle dont la signature est la plus proche des valeurs passées en paramètre lors de l'appel.

Exemple d'une surcharge d'une fonction:

```
float test(int i, int j)
 {
        return (float) i+j;
}
float test(float i, float j)
{
        return i*j;
}
```

Fonctions

4- Surcharge d'une fonction:

Ces deux fonctions portent le même nom, et le compilateur les acceptera toutes les deux.
Lors de l'appel de test(2,3) la 1ère fonction est appelée (2 et 3 sont des entiers).
Lors de l'appel de test(2.5,3.2) la 2ème fonction est appelée (2.5 et 3.2 sont réels).
Lors de l'appel de test(2.5,3) la 1ère fonction est appelée (le flottant 2.5 sera converti en entier).

Fonctions

5- Fonctions inline:

Une fonction en ligne se définit et s'utilise comme une fonction ordinaire, à la seule différence qu'on fait précéder son entête de la spécification inline.

Le mot inline indique au compilateur de remplacer au sein du programme les appels de la fonction par les instructions compilées (c'est-à-dire en langage machine) du corps de la fonction. Cela permet de réduire le temps d'appel de la fonction. En revanche, les instructions correspondantes au corps de la fonction sont générées à chaque appel de la fonction, ce qui consomme une quantité de mémoire croissante avec le nombre d'appels.

Fonctions

5- Fonctions inline:

- Le code de la fonction inline est inséré à la place de chaque appel ;
- La fonction est intéressante pour des petites fonctions ;
- Le compilateur peut l'implémenter comme une fonction classique.

Exemple:

```
inline int Max(int i, int j)
            { if (i>j) return i;
            else return j; }
 void main(){
            int x=5, res,  y=2 ;
            res = max( x, y) ;
            }
```

Passage de paramètres

Passage par valeur :

Lors du passage par valeur, les arguments sont copiés dans des objets temporaires qui sont crées lors du passage d'arguments et qui sont détruit à la fin de l'exécution de la fonction : toutes les modifications effectuées sur ces objets temporaires seront donc perdues à la fin de la fonction et n'auront aucune incidence sur l'objet passé en argument lors de l'appel de la fonction.

Exemple de passage par valeur:

```
void incremente(int i)
            { i++; }
int main()
            {int j = 12;
            incremente(j);
            cout << j << endl; // Affiche 12
            }
```

l'échange n'a pas lieu

Passage de paramètres

Passage par adresse :

Dans le passage par adresse, on manipule des pointeurs (c'est le passage d'arguments classique du C lorsqu'on veut modifier les arguments) : les arguments sont des pointeurs qui contiennent l'adresse en mémoire des objets que l'on veut manipuler. Il n'y a qu'une copie de pointeur lors du passage par adresse.

Exemple de passage par adresse:

```
void incremente(int * i)
            { (*i)++; }
 int main()
            {
             int j = 12; incremente(&j);
            cout << j << endl; // Affiche 13
            }
```

l'échange n'a pas lieu

Passage de paramètres

Notion de référence :

En C, la notation &n signifie l'adresse de la variable n.
En C++, cette notation possède deux significations:

- Il peut toujours s'agir de l'adresse de la variable n;
- Il peut aussi s'agir de la référence à n.

Seulement le contexte du programme permet de déterminer, s'il s'agit de l'une ou l'autre des deux significations.

Exemple:

```
int n ;
int &p=n; //p est une référence à n
          //p occupe le même emplacement mémoire que n
n=3;
cout<<p;// l'affichage donnera 3
```

Passage de paramètres

Passage par référence :

En C, comme en C++ , un sous programme ne peut modifier la valeur d'une variable locale passée en argument de fonction. Pour se faire, en C, il faut passer l'adresse de la variable. En C++, on préférera le passage par référence.

Le C++ permet le passage par référence : la fonction appelée manipule directement l'objet qui lui est passé par référence (cela correspond au var du langage Pascal). Il n'y a pas de copie ni de construction d'objet temporaire lors le passage par référence.

Passage de paramètres

Exemple de Passage par référence :

```
void incremente(int & i)
        { i++; }
int main()
        { int j = 12; incremente(j);
        cout << j << endl; // Affiche 13
        }
```

l'échange a lieu

Lorsqu'on manipule de gros objets (des matrice,...), il est particulièrement intéressant d'effectuer des passages par référence.

Pointeurs

- Une variable permet de désigner par un nom, un emplacement dans la mémoire.
- Il est aussi possible de désigner directement les cases mémoire par leur adresse.
- Un pointeur permet de stocker ces adresses mémoire.
- Un pointeur est type : on distingue les pointeurs sur un entier , un double,…
- Définir un pointeur: type * nom, le nom est un pointeur sur type, pouvant recevoir une adresse désignant un objet de type type.
- &V représente l'adresse mémoire d'une variable V.
- *P désigne l'objet pointée par le pointeur P.

Pointeurs

Exemple:

```
int i, pi, k ;
i =3;
pi = &i ;            // pointeur sur i
k = (pi )+5;       // identique à k=i+k
(pi) = k+2;       // identique à i = k+2
(pi )++;          // i c i , i =11, k=8
```

New & Delete

L'objectif de ces deux instructions New et Delete est d'allouer et désallouer de la mémoire sans dépendre la durée de vie des variables.

- "new type " : alloue un espace mémoire pour contenir un objet de type type et retourne l'adresse ce cette zone mémoire.
- "P = new type[N] :" alloue un espace mémoire capable de contenir N objets de type type .
- "delete P :" désalloue l'espace mémoire d'adresse P. (ou "delete [ ] P ;")
- Une zone allouée par new n'est desallouée que par l'appel de delete. . .

Exercices

Exercice 1 : Ecrire un programme qui demande de saisir un entier et qui indique si cet entier est premier ou non

Exercice 2 : Déclarer un tableau de 5 réels. Calculer et afficher la moyenne.

Exercice 3 : Un programme contient la déclaration suivante:

Int tab[20]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11, 12, 13, 14, 15, 16, 17, 18, 19};

Complétez ce programme de sorte d'afficher les éléments du tableau avec la présentation suivante:

```
0   1   2   3   4
5   6   7   8   9
10  11  12  13  14
15  16  17  18  19
```

Exercice 4: Saisir un tableau d'une chaine de caractères. Afficher le nombre de la lettre e et d'espace de cette chaine.

Exercice 5: Saisir une matrice d'entiers 2*2, calculer et afficher le détermianat.

Exercices

Exercice 6 : Ecrire un programme qui permet de faire des opérations sur un entier (valeur initiale à 0). Le programme affiche la valeur de l'entier puis affiche le menu suivant :

1. Ajouter 1
2. Multiplier par 2
3. Soustraire 4
4. Quitter

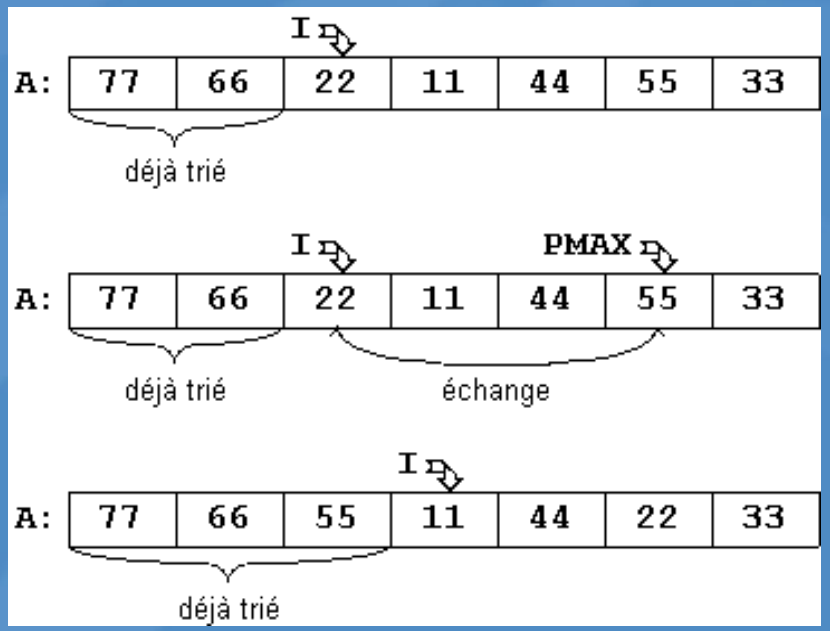
Le programme demande alors de taper un entier entre 1 et 4. Si l'utilisateur tape une valeur entre 1 et 3, on effectue l'opération, on affiche la nouvelle valeur de l'entier puis on réaffiche le menu et ainsi de suite jusqu'à ce qu'on tape 4. Lorsqu'on tape 4, le programme se termine.

Exercices

Exercice 7 (Tri par sélection du maximum): Ecrire une programme permet de classer les éléments d'un tableau par ordre décroissant (le programme permet de remplier et afficher le tableau ainsi que d'afficher le résultat de tri).

Méthode: Parcourir le tableau de gauche à droite à l'aide de l'indice I. Pour chaque élément A[I] du tableau, déterminer la position PMAX du (premier) maximum à droite de A[I] et échanger A[I] et A[PMAX].

Exemple:



Exercices

Exercice 8: Ecrire un programme qui demande à l'utilisateur de saisir un entier N et qui affiche la figure suivante:

N=1

*

N=2

**

*

N=3

***

**

*

Exercice 9: Ecrire une fonction qui a en paramètre une chaîne de caractères (paramètre en entrée et en sortie) et qui supprime toutes les voyelles.

Exercice 10: Ecrire un programme qui demande à l'utilisateur de taper une chaîne de caractères et qui indique si cette chaîne est un palindrome ou non.

Exercices

Exercice 11: Ecrire une fonction f ayant en paramètres un tableau t de taille quelconque et un entier n indiquant la taille du tableau. f doit renvoyer par un return un booléen b indiquant s'il existe une valeur comprise entre 0 et 10 dans les n premières cases du tableau t.

Exercice 12: Ecrire un programme qui saisit 2 tableaux de 10 entiers a et b. c est un tableau de 20 entiers. Le programme doit mettre dans c la fusion des tableaux a et b. On copiera dans les 10 premières cases de c le tableau a, dans les dix dernières le tableau b. Le programme affiche ensuite le tableau c.

Programmation en langage C++

Classe en C++

Lorsqu'on définit un type de données, on fournit des fonctions de manipulation et impose leur usage: les détails d'implémentation n'ont pas à être connus par l'utilisateur. On veut donc offrir une interface et protéger certaines données.
Les classes permettent de:

- Regrouper des données;
- Associer des fonctions aux objets de la classe;
- Restreindre l'accès aux certaines données.

Classe en C++

On veut regrouper des points pour construire une courbe.
Pour chaque point, on dispose les fonctions suivantes:

- Initialiser le point;
- Déplacer le point;
- Afficher le point .

Chaque point doit contenir les données suivantes:

- x;
- y.

Classe en C++

On intègre donc les données et les méthodes suivantes dans une classe point :

```
#include<conio.h>
#include<iostream.h>
class point
        {
        int x, y;
        public: void initialiser(int, int);
                void deplacer(int, int);
                void afficher();
        } ;
```

Classe en C++

« point » est une classe. Cette classe est constituée des **données** x, y et des **fonctions membres (méthodes)** « initialiser », « deplacer » et « afficher ». On déclare d'abord la classe en début de programme (données et prototypes des fonctions membres). Puis, on définit le contenu des fonctions membres. Les données x et y sont dites **privées**, ceci signifie que l'on ne peut les manipuler qu'à travers des fonctions membres. On dit que le langage C++ réalise **l'encapsulation des données**.

Classe en C++

```
void point :: initialiser(int abs, int ord)
        {x=abs; y=ord;}
void point :: deplacer(int dx, int dy)
        {x=x+dx; y=y+dy;}
void point :: afficher()
        {cout<<«" je suis en " <<x<< "  "  <<y<<endl;}
void main(){
        Point a, b;
        a.initialiser(3,6);
        a.afficher();
        a.deplacer(6,9);
        a.afficher();
        a=b;
        b.afficher(); getch();
        }
```

Classe en C++

a et b sont des objets de classe « point », c'est-à-dire des variables de type « point ».
Suivant le principe de l'encapsulation des données, la notation a.x est interdite.

Constructeurs

Le constructeur se définit comme une méthode normale. Cependant, pour que le compilateur puisse la reconnaître en tant que constructeur, les deux conditions suivantes doivent être vérifiées :

- Le constructeur doit porter le même nom que la classe ;
- Le constructeur ne doit avoir aucun type, pas même le type void.

Le constructeur en langage C++:

- Est appelé automatiquement lors de l'instanciation de l'objet;
- Est appelé après l'allocation de la mémoire de l'objet;
- Pourra avoir des paramètres, il peut donc être surchargé.

Constructeurs

Constructeur sans argument:

```
#include<conio.h>
#include<iostream.h>
class point
        {
        int x, y;
        public: point();//noter le type de constructeur (pas de void)
                void afficher();
                void deplacer(int, int);
        } ;
point :: point()
        {x=20; y=10;}//initialisation par défaut grâce au constructeur
void point :: deplacer(int dx, int dy)
        {x=x+dx; y=y+dy;}
```

Constructeurs

```
void point :: afficher()
        {cout<<«" je suis en " <<x<< "  "  <<y<<endl;}
void main()
        {
        Point a, b;// les deux points sont initialisé en 20 et 10
        a.afficher();
        a.deplacer(6,9);
        a.afficher();
        b.afficher();
        getch();
        }
```

Constructeurs

Constructeur avec des arguments:

```
#include<conio.h>
#include<iostream.h>
class point
        {
        int x, y;
        public: point(int , int );
                void afficher();
                 void deplacer(int, int);
        } ;
point :: point(int abs , int ord)// initialisation par défaut grâce au constructeur,
        {x=abs; y=ord;}// ici  paramètre à passer
void point :: deplacer(int dx, int dy)
        {x=x+dx; y=y+dy;}
```

Constructeurs

```
void point :: afficher()
        {cout<<" je suis en " <<x<< "  "  <<y<<endl;}
void main(){
        point a(20,10), b(30,10); // les deux points sont initialisé : a en 20,10
                                  //et b 30,10
        a.afficher();
        a.deplacer(6,9);
        a.afficher();
        b.afficher();
        getch();
        }
```

Constructeurs

Plusieurs constructeurs :

```
#include<conio.h>
#include<iostream.h>
class point
        {
        int x, y;
        public: point();//constructeur 1
                point(int);//constructeur 2
                 point(int,int);//constructeur 3
                void afficher();} ;
point ::point()
        {x=0; y=0;}
point::point(int abs)
        {x=abs; y=abs;}
```

Constructeurs

```
point :: point(int abs , int ord)
            {x=abs; y=ord;}
void point :: afficher()
            {cout<<«" je suis en " <<x<< "  "  <<y<<endl;}
void main(){
            Point a, b(3,1);
            a.afficher();
            b.afficher();
            b.afficher();
            point c(3,12);
            c.afficher();
            getch();
            }
```

Destructeurs

Le destructeur est une fonction membre systématiquement exécutée à la fin de la vie d'un objet .
On ne peut pas passer de paramètres par le destructeur.

```
#include<conio.h>
#include<iostream.h>
class point
        {
        int x, y;
        public: point(int , int );
                void afficher();
                void deplacer(int, int);
                ~point();
        } ;
```

Destructeurs

```
point :: point(int abs , int ord)// initialisation par défaut grâce au constructeur,
            {x=abs; y=ord;}// ici  paramètre à passer
void point :: deplacer(int dx, int dy)
            {x=x+dx; y=y+dy;}
void point :: afficher()
            {cout<<« je suis en " <<x<< "  "  <<y<<endl;}
point::~point()
            {cout<< " destruction du point x="<<x <<" y=" <<y<<endl; }
void  tester()
            {point u(3,7);
            u.afficher();
            }
```

Destructeurs

```
void main(){
        point a(20,10);
        a.afficher();
        tester();
        point b(7,5);
        b.afficher();
        getch();
        }
```

Constructeur par recopie

La classe liste contient un membre privé de type pointeur. Le constructeur lui alloue dynamiquement de la place. Que se passe- t-il lors d'une initialisation de type a(5); liste b=a;

```
#include<iostream.h>
#include<conio.h>
class liste
        {int taille;
        float *adr;
        public : liste(int);
                ~ liste();};
Liste::liste(int val)
        {taille=val;
        adr=new float [taille]; cout<< " constructeur ";
        Cout<< " adresse de l'objet " <<this;
        Cout<< " adresse de la liste " <<adr<<"\n";}
```

Constructeur par recopie

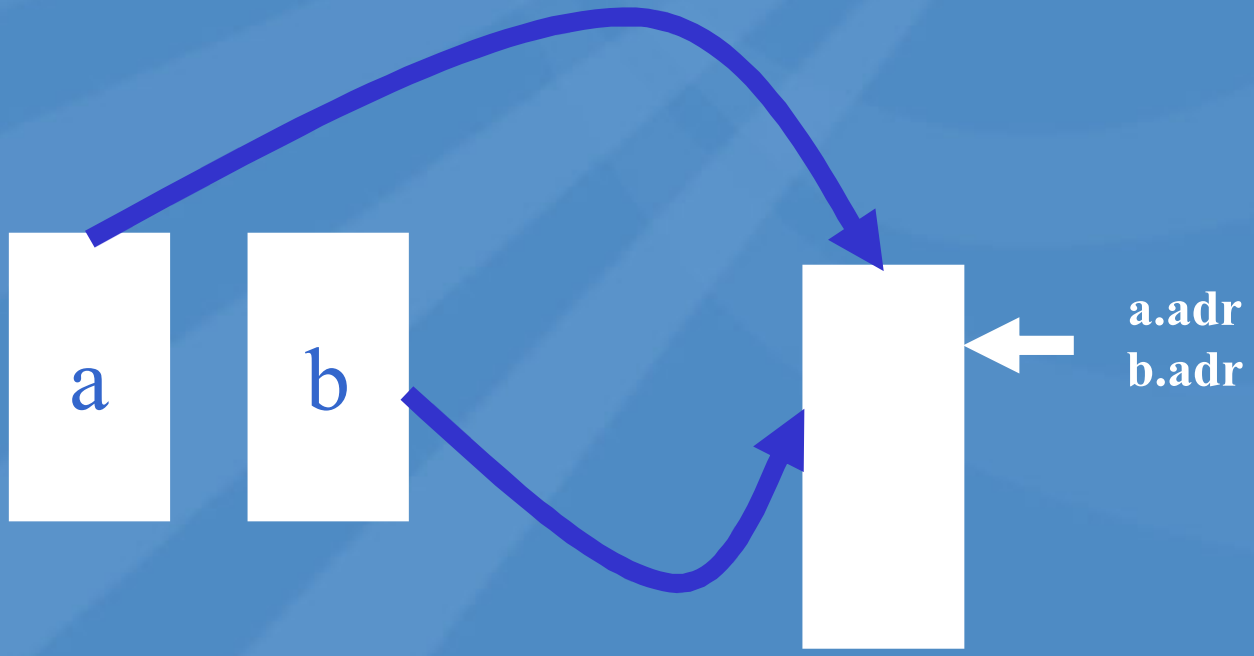
```
Liste::~liste()
        {cout<< " destructeur adresse de l'objet " <<this;
        Cout<< " adresse de la liste " <<adr<<\n;
        Delete adr;}
Void main()
        {cout<< " debut de main() " <<\n;
        Liste a(5);
        Liste b=a;
        Cout<< " fin de main " <<\n;
        getch();
        }
```

Dans ce programme, sont exécutés:

- Le constructeur pour a uniquement;
- Le destructeur pour a et b.

Constructeur par recopie

Le compilateur affecte des emplacements mémoires différents pour a et b. Par contre les pointeurs a.adr et b.adr pointent sur la même adresse mémoire. La réservation de place dans la mémoire ne s'est pas exécuté correctement.



Constructeur par recopie

Pour résoudre ce problème, on doit ajouter un constructeur de prototype liste(liste &) appelé **constructeur par recopie**, ce constructeur sera appelé lors de l'exécution de liste b=a.

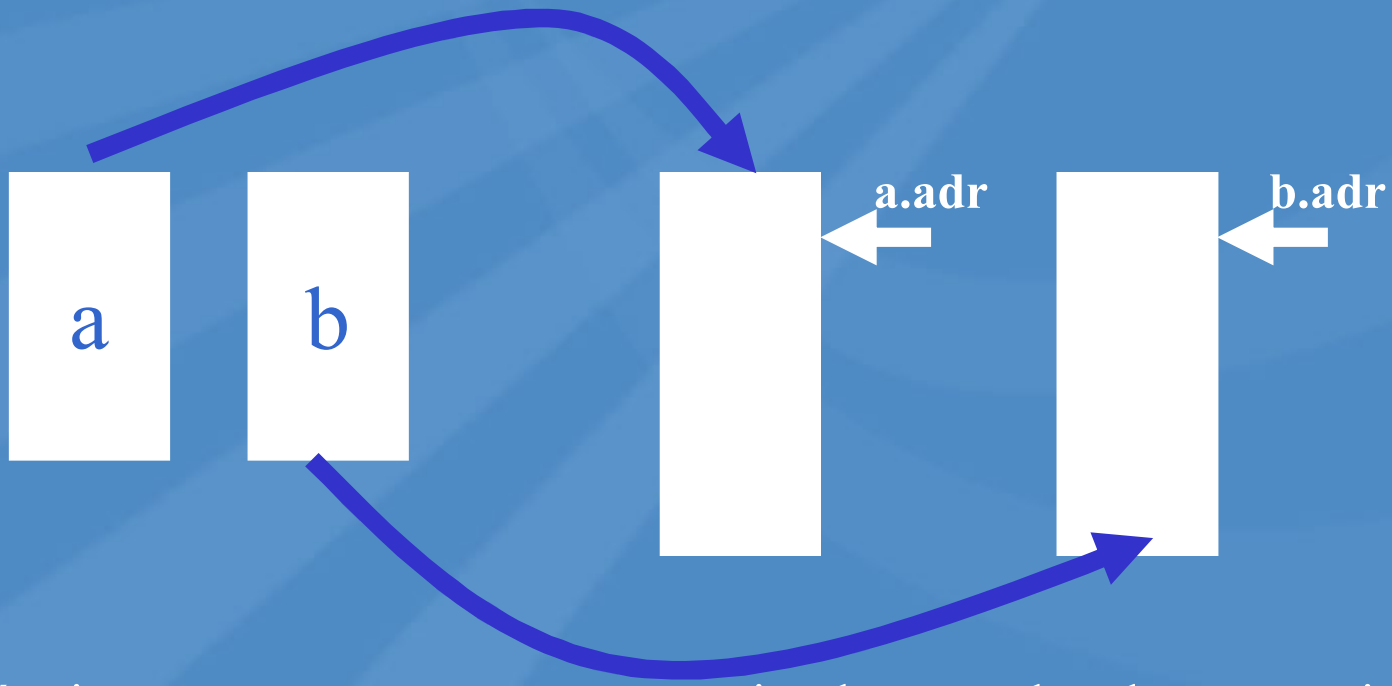
```
#include<iostream.h>
#include<conio.h>
class liste
        {int taille;
        float *adr;
        public : liste(int);
                 liste (liste &);
                ~ liste();};
liste::liste(int val)
        {taille=val; adr=new float [taille]; cout<< " constructeur ";
        cout<< " adresse de l'objet " <<this;
        cout<< " adresse de la liste " <<adr<< "\n";}
```

Constructeur par recopie

```
liste::liste(liste & lis)//passage par référence obligatoire
        {taille=lis.taille;adr=new float [taille];
        for(int i=0;i<taille,i++)adr[i]=lis.adr[i];
        cout<< " \nconstructeur par recopie ";
        cout<<"adresse de l'objet"<<this;
        cout<<"adresse de la liste"<<adr<<\n;}
liste::~liste()
        {cout<<"destructeur adresse de l'objet"<<this;
        cout<<"adresse de la liste " <<adr<<"\n";
        delete adr;}
Void main()
        {liste a(5);
        liste b=a;
        getch();}
```

Constructeur par recopie

Ici, toutes les réservations de place en mémoire ont été correctement réalisées.



Il faut prévoir un constructeur par recopie, lorsque la classe contient des données dynamiques. Lorsque, le compilateur ne trouve pas ce constructeur, aucun erreur n'est générée.

Exercices

Exercice 12 :

Ajouter à la classe liste les fonctions suivantes

- Saisir() :saisir au clavier les composantes de la liste;
- Afficher() :permet d'afficher les composantes de la liste;
- Opposer() : qui retourne la liste des coordonnées opposées.

Exercice 13:

Ecrire une classe pile_entier permettant de gérer une pile d'entiers, selon le modèle suivant:

Class pile_entier

{int *pile, taille, hauteur;//*pointeur de pile, taille maximum, hauteur actuelle*

public :

pile_entier(int);//*constructeur, taille de la pile, 20 par défaut, hauteur à 0*

//alloue dynamiquement de la place mémoire.

~pile_entier();//*destructeur*

Constructeur par recopie

pile_entier(pile_entier &);//*constructeur par recopie*
void empiler(int);//*ajouter un élément*
int depiler();//*retourner la valeur entière en haut de la pile , la hauteur diminue*
//de 1 unité
int pleine();//retourne 1 si la pile est pleine, 0 sinon
int vide();//retourne 1 si la pile est vide, 0 sinon
Mettre en œuvre cette classe dans main(). Le programme principale doit contenir les déclarations suivantes:

```
void main()
        {pile_entier a,b(18);//pile automatique
        pile_entier *pe;//pile dynamique
        ….
        }
```

Tableaux d’objets

Les tableaux d’objets se manipulent comme les tableaux classiques du langage C++. Avec la classe point déjà étudiée, on pourra par exemple déclarer:
point courbe [100];//déclaration d’un tableau de 100 points
La notation courbe[i]. afficher() a un sens .
La classe point doit obligatoirement posséder un constructeur sans argument(ou avec des argument par défaut). Le constructeur est exécuté à chaque élément du tableau.

Tableaux d'objets

La notation suivante est admise:

```
class point
{int x, y;
public  : point (int a=0;int b=0)
                    {x=abs; y=ord;}};
void main()
{point courbe[5]={7, 4, 2};
```

On obtiendra les résultats suivantes:

	x	y
courbe[0]	7	0
courbe[1]	4	0
courbe[2]	2	0
courbe[3]	0	0
courbe[4]	0	0

Tableaux d’objets

On pourra de la même façon créer un tableau dynamiquement:
Point *courbe =new point[10];
On doit utiliser la notation suivante delele[] courbe pour détruire ce tableau dynamique. Le destructeur sera alors exécuté pour chaque élément du tableau.

Fonctions amies

Grâce aux fonctions amies, on pourra accéder aux membres privés d'une classe, autrement que par le biais de ses fonctions membres.

Il existe plusieurs situations d'amitié:

- Une fonction indépendante est amie d'une ou de plusieurs classes;
- Une ou plusieurs fonctions membres d'une classe sont amies d'une autre classe.

Fonctions amies

```
#include<iostream.h>
#include<conio.h>
class point
        { int x,y;
        public: point(int abs=0; int ord =0){x=abs; y=ord;}
                friend int coincide(point , point);
        };
int coincide(point p, point q)
        {if((p.x==q.x)&&(p.y==q.y))
        return 1;
        else return 0;
        }
```

Fonctions amies

```
void main()
        {point a(4,0), b(4),c;
        if(coincide(a,b)) cout<< " a coincide avec b " <<endl;
        else cout<< " a est different de b " <<endl;
        if(coincide(a,c)) cout<< " a coincide avec c " <<endl;
        else cout<< " a est different de c " <<endl;
        getch();
        }
```

Dans cet exemple, la fonction coincide est une fonction amie de la classe point. C'est une fonction ordinaire qui peut manipuler les membres privés de la classe point.

Fonctions amies

Autres situations d'amitiés:

```
class A
        {
        ……..
        friend int B:: fm_de_B (char, A);
        };
class B
        {……..
         int fm_de_B (char , A);
        };
int B::fm_de_B(char c, A a)
        {……}
```

Fonctions amies

Dans cette situation , la fonction fm_de_b est une fonction membre de la classe B, a accès aux membres privés de la classe A.
Si toutes les fonctions membres de la classe B étaient amies de la classe A. On déclarerait directement dans la partie publique de la classe A:
friend Class B.

Fonctions amies

```
class A
        {
        ……..
        friend voif f_anonyme (A,B);
        };
class B
        {……..
         friend void f_anonyme( A,B);
        };
void f _anonyme(A a, B b)
        {……}
```

Dans cette situation, le fonction f_anonyme a accès aux membres privés des classes A et B.

Surdéfinition des opérateurs

- Le langage C++ autorise l'utilisateur à étendre la signification d'opérateurs tels que l'additions (+), la soustraction (-), la division (/), la multiplication (*), le ET logique (&),…
- Il faut se limiter aux opérateurs existants:+, -, /, *, /, =,++, --,new, delete, [, ^, ->, &, |, ^, <<, >>, ||, %...;
- Les règles d'associativité et de priorité dont maintenues.

Surdéfinition des opérateurs

Exemple de la surdéfinition de l'opérateur +:

```
#include<iostream.h>
#include<conio.h>
class vecteur
        { flot  x, y;
        public:   vecteur(float, float);
                  void afficher();
                  vecteur operator + (vecteur);// surdéfinition de l'opérateur +
        };//on passe un paramètre vecteur et la fonction return un vecteur
vecteur::vecteur(float abs=0, float ord=0)
        {x=abs, y=ord;}
void point :: afficher()
        {cout<<« x= " <<x<< « y=  " <<y<<endl;}
```

Surdéfinition des opérateurs

```
vecteur vecteur::operator +(vecteur v)
        {vecteur res;
        res.x=v.x+x;
        res.y=v.y+y;
        return res;
        }
void main()
        {vecteur a(2,6), b(4,8), c, d, e,f;
        c=a+b; c.afficher();
        d=a.operator+(b); d.afficher();
        e=b.operator+(a); e.afficher();
        f=a+b; f.afficher();
        getch();
        }
```

Exercices

Ce code permet de surdéfinir l'opérateur somme (+) qui permettra d »écrire dans un programme

```
vecteur v1, v2, v3;
v3-=v1+v2;
```

Exercice 14:

Ajouter une fonction membre de prototype float operator *(vecteur) permettant de créer l'opérateur « produit scalaire », c'est-à- dire de donner une signification à l'opération suivante:

```
vecteur v1, v2;
float prod_scal;
prod_scal=v1*v2;
```

Exercices

Exercice 15:

Ajouter une fonction memebre de protitype vecteur operator *(float) permettant de donner une signification au produit d'un réel et d'un vecteur selon le modèle suivant:

```
vecteur v1, v2;
float h
v2=v1 * h; // homothethie
```

Exercice 16 :

Reprendre la classe pile_entier et remplacer la fonction membre « empiler »par l'opérateur < et la fonction membre « depiler » par l'opérateur >.

p < n ajoute la valeur n sur la pile p.

p > n supprime la valeur du haut de la pile p et la place dans n.

Exercices

Exercice 17 :

Ajouter à la classe liste la surdéfinition de l'opérateur [], de sorte que la notation a[i] ait un sens et retourne l'élément d'emplacement i de la liste a.
Utiliser ce nouvel opérateur dans la fonction afficher et saisir. On créera donc une fonction membre de prototype float &liste::operator[](int i)

Exercice 18:

Définir une classe chaîne permettant de créer et de manipuler une chaîne de caractères:

Données: longueur de la chaîne (entier);
adresse d'une zone allouée dynamiquement.

Méthodes: constructeur chaine() initialise une chaine vide;
constructeur chaine(char*) initialise avec la chaine passée en argument;
constructeur par recopie chaine(chaine&)
opérateurs affectation(=), comparaison(==), concaténation(+), accès à caractère de rand donné([]).

Structures de données dynamiques

Une structure de données dynamique évolue au cours du temps.
Exemple de la pile: utiliser un tableau oblige à borner la taille de la pile.
On préfère allouer de l'espace mémoire en fonction des besoins :

- Ajouter un élément à la pile correspondra à allouer (avec new) de la mémoire,…
- Supprimer reviendra à désallouer (avec delete) la case contenant l'objet.

Le contenu de la pile sera donc éclaté dans la mémoire. Pour ne pas perdre d'éléments, on indique à chaque élément où se trouve le prochain élément. Cela correspond à la notion de liste chaînée

Structures de données dynamiques

Listes chaînées

```
Class element{
Int val;
Public : element *suivant;
};
```

Le champ suivant contient l'adresse du prochain élément de la liste. On utilise l'adresse du premier élément de la liste...