Support de cours

Programmation Orientée Objet en Langage C++

Mohamed EL WAFIQ

Version 2023

Avant-propos

Ce cours a été développé et amélioré au fil des années afin de donner un support résumé et simplifié aux étudiants ayant un bon niveau en programmation structurée en langage de programmation C.

Structure du cours

Ce support de cours est structuré pédagogiquement selon une progression simpliste en expliquant les concepts, en donnant des exemples d'application et en proposant des exercices de renforcement dont les solutions sont présentées à titre concis.

Compilation des programmes sources

Tous les programmes sources de ce cours sont compilés sous Dev-C++, Version 4.9.9.2, en incluant les fichiers entête correspondants.

Table des matières

Chapitre 0 : Spécificités du Langage C++

C++ dispose d'un certain nombre de spécificités qui ne sont pas obligatoirement relatives à la programmation orientée objet.

1. Commentaire en fin de ligne

Exemple

```
//ceci est un commentaire.
int  a; // déclaration de a.
```

2. Emplacement des déclarations

La déclaration des variables doit s'effectuer avant leur utilisation n'importe où dans un bloc ou dans une fonction.

Exemple

```
main()
{
     int  a=7,  b;
     b=a;
     float x, y;
     x=2*a+y;
}
```

Exemple

```
for(int i=0; i<7; i++)
```

3. Notion de référence

3.1. Transmission des arguments par valeur

Exemple

```
main()
{
     void echange(int,int);
     int a=2, b=5;
     cout<<"Avant appel : "<<a<<" - "<<b<<" \n";
     echange(a,b);
     cout<<"Après appel : "<<a<<" - "<<b<<" \n";
}
//-----------------------------------------------------
void echange(int m,int n)
{
     int z; z=m; m=n; n=z;
}
```

Exécution

```
Avant appel : 2 - 5
Apres appel : 2 - 5
```

3.2. Transmission des arguments par adresse

```
main()
{
     void echange(int *,int *);
     int a=2, b=5;
     cout<<"Avant appel : "<<a<<" - "<<b<< " \n ";
     echange(&a,&b);
     cout<<"Après appel : "<<a<<" - "<<b<< " \n ";
}
//-----------------------------------------------------------
void echange(int *x,int *y)
{
     int z; z=*x; *x=*y; *y=z;
}
```

Exécution

```
Avant appel : 2 - 5
Apres appel : 5 - 2
```

3.3. Transmission des arguments par référence

En C++, la transmission par référence est une forme simplifiée de la transmission par adresse.

```
main()
{
     void echange(int &,int &);
     int a=2, b=5;
     cout<<"Avant appel : "<<a<<" - "<<b<<"\n ";
     echange(a,b);
     cout<<"Apres appel : "<<a<<" - "<<b<<"\n ";
}
//-----------------------------------------------------------
void echange (int & x,int & y)
{
     int z; z=x; x=y; y=z;
}
```

4. Les arguments par défaut

En C, il est indispensable que le nombre d'arguments passés correspond au nombre d'arguments déclarés. C++ peut ne pas respecter cette règle.

Exemple 1

```
main()
{
      int m=1, n=5;
      void f(int,int=7);
      f(3,5);
      f(4);
}
//-----------------------------------------------
void f(int a,int b)
{
      cout<<"Valeur 1 : "<<a<<" - Valeur 2 : "<<b<<"\n";
}
```

Exécution

```
Valeur 1 : 3 - Valeur 2 : 5
Valeur 1 : 4 - Valeur 2 : 7
```

NB :

- La fonction **f** est déclarée avec un deuxième argument initialisé par défaut par la valeur **7**.
- Pendant l'appel de la fonction **f**, si le deuxième argument n'est pas précisé, il est alors égal à la valeur 7.
- Un appel du genre **f()** sera rejeté par le compilateur.

Exemple2

```
main()
{
      void f(int=33,int=77);
      f(99,55);
      f(44);
      f();
}
//---------------------------------
void f(int a,int b)
{
      cout<<"Valeur 1 : "<<a<<" - Valeur 2 : "<<a<<"\n";
}
```

Exécution

```
Valeur 1 : 99 - Valeur 2 : 55
Valeur 1 : 44 - Valeur 2 : 77
Valeur 1 : 33 - Valeur 2 : 77
```

Remarque

Lorsqu'une déclaration prévoit des valeurs par défaut, les arguments concernés doivent obligatoirement être les derniers de la liste.

Exemple

La déclaration : **int f (int=2, int, int=7)** ; est interdite, car un appel du genre **f(3, 4);** peut être interprété comme : **f(2, 3, 4)**; ou **f(3, 4, 7)**;

5. Surdéfinition de fonction (overloading : surcharge)

On parle de surdéfinition ou de surcharge lorsqu'un même symbole possède plusieurs significations différentes, le choix de l'une des significations se faisant en fonction du contexte. Pour pouvoir employer plusieurs fonctions du même nom, il faut un critère permettant de choisir la bonne fonction. En C++, ce choix est basé sur le type des arguments.

Exemple

```
void f(int x)
{ cout<<"fonction numéro 1 : "<<x<<"\n"; }
void f(double x)
{ cout<<"fonction numéro 2 : "<<x<<"\n"; }
main()
{
     int a=2; double b=5.7;
     f(a); f(b); f('A');
}
```

Exécution

```
Fonction numero 1 : 2
Fonction numero 2 : 5.7
Fonction numero 1 : 65
```

La valeur **65** représente le code ASCII de la lettre **A**.

Remarque

Le C++ a choisi donc la bonne fonction en fonction de ses arguments.

Cas 1

```
void f(int); //f1
void f(double); //f2
char c;
float b;
```

- **f(c);** appellera la fonction **f1** après conversion de la valeur de **c** en **int**.
- **f(b);** appellera la fonction **f2** après conversion de **b** en **double**.

Cas 2

```
void f(char *); //f1
void f(void *); //f2
char      *p1;
double    *p2;
```

- **f(p1);** appellera la fonction **f1**.
- **f(p2);** appellera la fonction **f2** après conversion de **p2** en **void ***.

Cas 3

```
void f(int, double) ; // f1
void f(double, int) ; // f2
int  a, b;
double x;
char c ;
```

- **f(a, x);** appellera la fonction **f1**.
- **f(c, x);** appellera la fonction **f1** après conversion de **c** en **int**.
- **f(a, b);** conduira à une erreur de compilation (convertir **a** en **double** ou **b** en **double**).

Cas 4

```
void f(int a=0 ; double c=0) ; // f1
void f(double y=0 ; int b=0) ; //f2
int  m;
double z ;
```

- **f(m, z);** appellera la fonction **f1**.
- **f(z, m);** appellera la fonction **f2**.
- **f (m);** appellera la fonction **f1**.
- **f(z);** appellera la fonction **f2**.
- **f();** conduira à une erreur de compilation.

6. Les opérateurs **new** et **delete**

En langage C, la gestion dynamique de la mémoire a été assurée par les fonctions **malloc(…)** et **free(…)**. En C++, elle est assurée par les opérateurs **new** et **delete**.

6.1. L'opérateur **new**

Exemple

```
int  *p;
p=new int;
ou
int  *p=new int; // Allocation dynamique d'un entier.
int  *p2=new int[5];  // Allocation dynamique de 5 entiers.

char *t;
t=new char[30];
ou
char *t=new char [30];
```

Remarque

new fournit comme résultat :

- Un pointeur sur l'emplacement correspondant, lorsque l'allocation réussie.
- Un pointeur **NULL** dans le cas contraire.

6.2. L'opérateur **delete**

delete possède deux syntaxes :
- delete p ;
- delete [ ]p ;

Où **p** est une variable devant avoir comme valeur un pointeur sur un emplacement alloué par **new**.

7. Utilisation des fonctions **en ligne** (**inline**)

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

Exemple

```
inline double norme(double vec[3])
{
     for(int i=0,double s=0; i<3; i++) s=s+vec[i]*vec[i];
     return sqrt(s);
}
//-----------------------------------------------------------
main()
{
     double V1[3], V2[3];
     for (int i=0; i<3; i++) { V1[i]=i;  V2[i]=i*3; }
     cout<<"Norme de V1 est : "<<norme(v1);
     cout<<" - Norme de V2 est : "<<norme(v2);
}
```

Commentaire

- La fonction **norme** a pour but de calculer la norme d'un vecteur passé comme argument.
- La présence du mot **inline** demande au compilateur de traiter la fonction norme d'une manière différente d'une fonction ordinaire, à chaque appel de **norme**, il devra incorporer au sein du programme, les instructions correspondantes (en langage machine). Le mécanisme habituel de gestion de l'appel est de retour n'existe plus, ce qui réalise une économie de temps.

Remarque

Une fonction **en ligne** doit être définie dans le même fichier source que celui utilisé. Elle ne peut être compilée séparément.

Chapitre 1 : Notions de Classe et Objet

Une classe est la généralisation de la notion de type défini par l'utilisateur, dont lequel se trouvent associées à la fois des données (données membres) et des méthodes (fonctions membres).
En programmation orientée objet pure, les données sont encapsulées et leur accès ne peut se faire que par le biais des méthodes.

1. Exemple du type classe

```
class  point // déclaration de la classe point
{
     int x;
     int y;
     public :
       void initialise(int,int);
       void affiche();
       void deplace(int,int);
};
//----Définition des fonctions membres-----
void point::initialise(int a,int b)
{ x=a; y=b; }
//----------------------------------------
void point::affiche()
{ cout<<"on est à : "<<x<<" - "<<y<<endl; }
//----------------------------------------
void point::deplace(int a,int b)
{ x=x+a; y=y+b; }
```

Commentaire

Le mot clé **'public'** précise que tout ce qui le suit (données membres ou fonctions membres) sera public, le reste étant privé. Ainsi :

- **x** et **y** sont deux membre privés.
- '**initialise**', '**deplace**' et '**affiche'** sont trois fonctions membres **publics**.
- La classe '**point**' ci-dessus peut être définie et utilisée comme dans l'exemple suivant :

Suite du programme (exemple du type classe)

```
.....
main()
{
     point p1, p2, p3;
     p1.initialise(1,3); p1.affiche();
     p1.deplace(2,4);    p1.affiche();
     p2.initialise(5,5); p2.affiche();
}
```

- On dit que **p1** et **p2** sont des **instances** de la classe '**point**' ou encore que se sont des **objets** de type 'point'.
- Tous les membres données de '**point**' sont **privés** ou **encapsulés.** Ainsi, on ne peut pas accéder à **x** ou à **y**.
- Toutefois, les membres données ou les fonctions membres peuvent être privées en utilisant le mot clé '**private**' ou publiques en utilisant le mot clé '**public**'.

Exemple

```
class   C
{
     public :
          ……. ;
          ……. ;
     private :
          ……. ;
          ……. ;
};
```

2. Affectation d'objets

Elle correspond à une recopie de valeurs des membres donnés (publics ou privés).
Ainsi, avec les déclarations :

```
class point
{          int x, y;
     public :
          …… ;
};
point p1, p2;
```

L'affectation p2=p1 ; provoquera la recopie des valeurs x et y de p1 dans les membres correspondants de p2.

3. Notion de constructeur et de destructeur

3.1. Introduction

Un constructeur est une fonction qui sera appelée automatiquement après la création d'un objet. Ceci aura lieu quelque soit la classe d'allocation de l'objet : statique, automatique ou dynamique.
- De la même façon, un objet pourra posséder un destructeur, il s'agit également d'une fonction membre qui est appelée automatiquement au moment de la destruction de l'objet correspondant.
- Par convention, le constructeur se reconnaît à ce qu'il porte le même nom que la classe. Quand au destructeur, il porte le même nom que la classe précédé du symbole **~.**

Exemple

Reprendre l'exemple utilisant la classe '**point**' définie avec son constructeur.

```
class point
{
     int x ;
     int y ;
  public :
     point(int,int);
     void affiche();
     void déplace();
};
//-------------------------------------------------------
point::point(int a , int b)
{ x=a; y=b; }
//-------------------------------------------------------
void point::affiche()
{ cout<<"on est à : "<<x<<" - "<<y<<"\n" ;}
//-------------------------------------------------------
void point::deplace (int a,  int b)
{ x=x+a; y=y+b; }
//--------------------------------
main()
{
     point p1(1,7), p2(2,5);
     p1.affiche();       p2.affiche();
     p1.deplace(-1,5);   p1.affiche();
     p2.deplace(3,3);    p2.afficxhe();
}
```

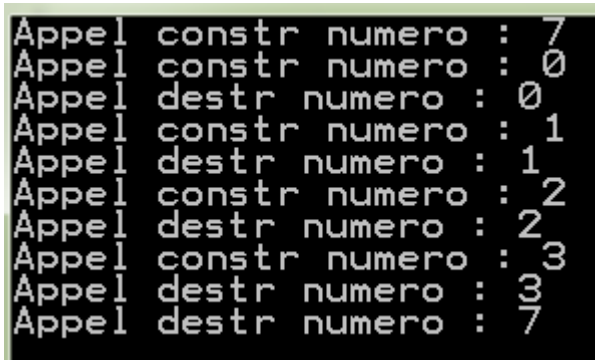
3.2. Construction et destruction des objets

Suivre la trace des objets automatiques créés dans le programme suivant en affichant les moments de leur construction et leur destruction.

```
class demo
{
     int num ;
  public :
     demo(int);
     ~demo();
};
//------------------
demo::demo(int n)
{
     num=n;
     cout<<"Appel constr numéro : "<<num<<endl;
}
//-----------------------------------------------
demo::~demo()
{
     cout<<"Appel destr numéro : "<<num<<endl;
}
//---------------------------------------
```

```
main()
{
      void f(int);
      demo obj(7);
      for (int i=0; i<4; i++)
      f(i);
}
//---------------------------------------
void  f(int m)
{ demo obj(m) ; }
```

Exécution



3.3. Rôle de constructeur et de destructeur

Le rôle d'un constructeur ou d'un destructeur peut dépasser celui d'initialisation ou de libération.

Exemple

Construction d'un tableau de 10 valeurs entières prises au hasard, tel que l'argument passé au constructeur est la valeur entière maximum à ne pas dépasser.

Ajouter une fonction membre appelée : 'affiche' pour afficher le contenu du tableau.

```
class tableau
{
       int t[10];
     public :
       tableau();
       void affiche();
};
//---------------------
tableau::tableau(int max)
{
     for(int i=0;i<10;i++) t[i]=i*2;
}
//-------------------------------------------------
void tableau::affiche()
{
   for(int i=0;i<10;i++) cout<<t[i]<<endl;
}
```

Règles

- Un constructeur peut ou non comporter quelques arguments.
- par définition, un constructeur ne renvoie pas de valeur et la présence de void (dans ce cas précis) est une erreur.
- Un destructeur, par définition, ne peut pas disposer d'arguments et ne renvoie pas de valeur.

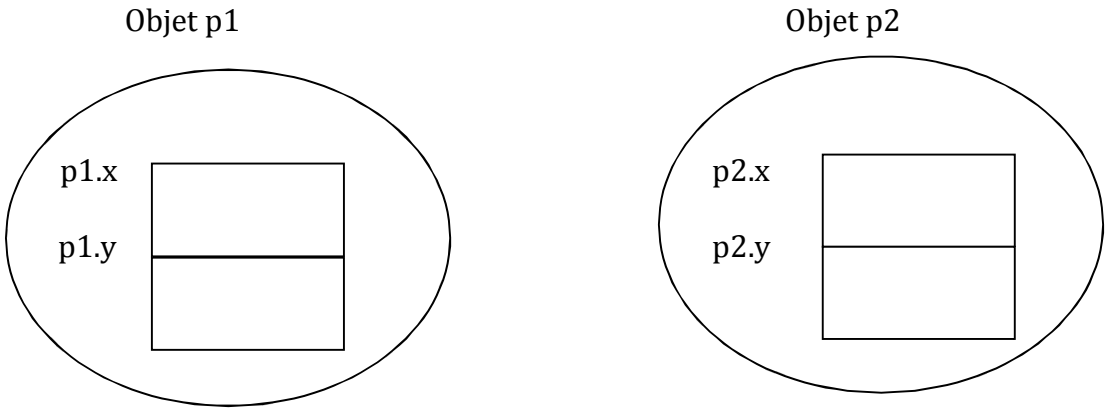
4. Membres statiques

Lorsqu'on crée différents objets d'une même classe, chaque objet possède ces propres données membres.

Exemple

```
calss point
{
      int x ;
      int y ;
      ……..
      ……..
      ……..
};
```

- Une déclaration telle que : `point p1, p2; provoquera :`

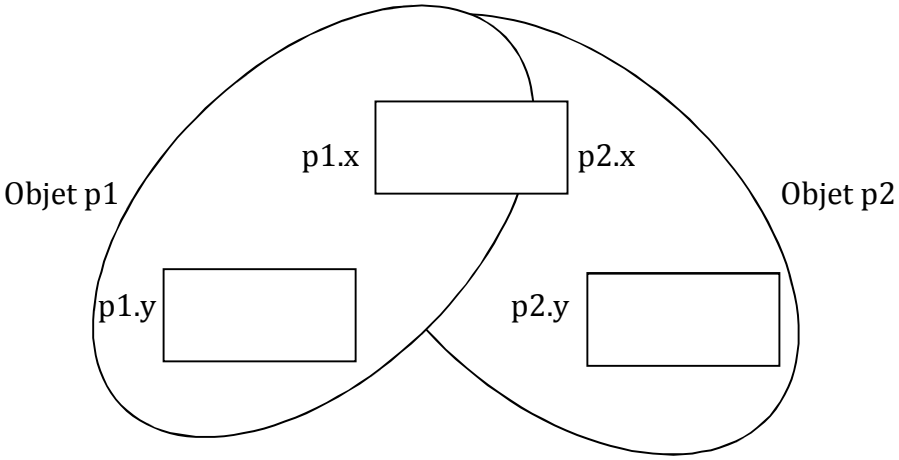


- Pour pouvoir partager des données entre objets de la même classe, on les déclare statiques.

Exemple

```
class     point
{     static     int x;
                 int y;
      ……..
      ……
      ……..
};
```

Une déclaration telle que : `point p1, p2; provoquera :`



p1.x est identique à p2.x mais p1.y ≠ p2.y.

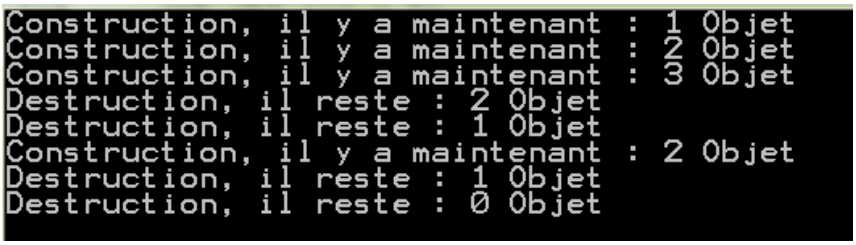
Commentaire

- Les membres statiques existent en un seul exemplaire quelque soit le nombre d'objets de la classe correspondante, et même si aucun objet de la même classe n'a été créé.
- Les membres statiques sont toujours initialisés par 0.

Exemple

```
class cpt_obj
{
       static int c;
     public :
       cpt_obj();
       ~cpt_obj();
};
//----------------------
int  cpt_obj::c=0;
//----------------------
cpt_obj()::cpt_obj()
{cout<<"Construction, il y a maintenant : "<<++c<<" Objet\n";}

//-------------------------------------------------------------
cpt_obj::~cpt_obj()
{cout<<"Destruction, il reste : " <<--c<<" Objet\n";}
//-------------------------------------------------------
main()
{
     void f();
     cpt_obj O1; f(); cpt_obj O2;
}
//-------------
void f(){ cpt_obj a, b; }
```

Exécution



Exercice 1

Ecrire un programme permettant de créer des objets ayant chacun :

- un tableau de 5 éléments de type entier en tant que donnée ;
- une fonction pour remplir le tableau, une fonction pour trier le tableau et une fonction pour afficher le contenu du tableau en tant que méthodes.

Solution (sous Dev-C++ 4.9.9.2)

```
class  tableau
{
       int t[5];
     public :
       tableau(){ for(int i=0;i<5;i++) t[i]=0; }
       void remplir();
       void afficher();
       void trier();
};
void tableau::remplir()
{
     cout<<"Veuillez remlir le tableau avec 5 entiers :"<<endl;
     for(int i=0;i<5;i++) cin>>t[i];
}
void tableau::afficher()
{
     for(int i=0;i<5;i++) cout<<t[i]<<"\t"; cout<<endl;
}
void tableau::trier()
{    int a;
     for(int i=0;i<4;i++)
          for(int j=i+1;j<5;j++)
               if(t[i]<t[j]){ a=t[i]; t[i]=t[j]; t[j]=a; }
}
//-------------------------------------------------------
main()
{
     tableau t1;
     t1.remplir();
     cout<<"Avant tri : "<<endl; t1.afficher();
     t1.trier();
     cout<<"Apres tri : "<<endl; t1.afficher();
     getch();
}
```

Exécution

```
Veuillez remplir le tableau avec 5 entiers :
12
55
34
67
24
Avant tri :
12      55      34      67      24
Apres tri :
67      55      34      24      12
```

Exercice 2

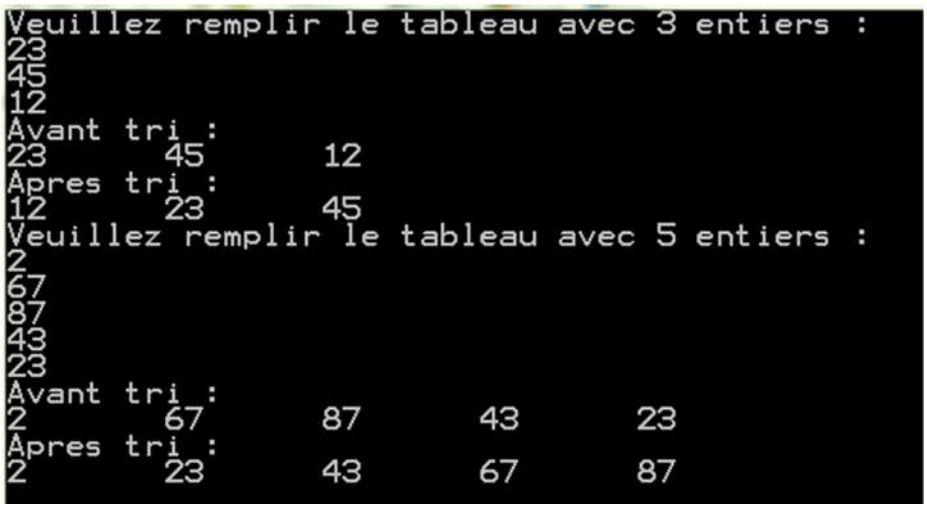
Reprendre le même programme en remplaçant le tableau de 5 éléments par un tableau dynamique de 'ne' éléments et instancier des objets ayant des tableaux dynamiques de différentes tailles.

Solution (sous Dev-C++ 4.9.9.2)

```
class  tableau
{
          int *t;
          int ne;
        public :
          tableau(int n)
          { ne=n; t=new int[ne]; for(int i=0;i<ne;i++) t[i]=0;}
          void remplir();
          void afficher();
          void trier();
          ~tableau(){ delete []t;}
};

//----------------------------------------
void tableau::remplir()
{ cout<<"Veuillez remplir le tableau avec "<<ne<<" entiers
  :"<<endl;
  for(int i=0;i<ne;i++) cin>>t[i];
}

//----------------------------------------
void tableau::afficher()
{ for(int i=0;i<ne;i++) cout<<t[i]<<"\t"; cout<<endl; }

//------------------------------------------------------
void tableau::trier()
{    int a;
     for(int i=0;i<ne-1;i++)
             for(int j=i+1;j<ne;j++)
                     if(t[i]>t[j]){a=t[i]; t[i]=t[j]; t[j]=a;}
}

//----------------------------
```

```
main()
{
     tableau t1(3);
     t1.remplir();
     cout<<"Avant tri : "<<endl; t1.afficher();
     t1.trier();
     cout<<"Apres tri : "<<endl; t1.afficher();
     //-----------------------------------------------------
     tableau t2(5);
     t2.remplir();
     cout<<"Avant tri : "<<endl; t2.afficher();
     t2.trier();
     cout<<"Apres tri : "<<endl; t2.afficher();
     getch();
}
```

Exécution



Chapitre 2 : Propriétés des fonctions membres

1. Surdéfinition des fonctions membres

La surdéfinition des fonctions s'applique également aux fonctions membres d'une classe, y compris au constructeur (mais pas au destructeur puisqu'il ne possède pas d'arguments).

Exemple

Surdéfinir les fonctions membres 'point' et 'affiche' de la classe 'point'.

```
class point
{
       int x, y;
     public :
       point();
       point(int);
       point(int,int);
       void affiche();
       void affiche(char *);
};
//--------------
point::point()
{ x=0; y=0; }
//-----------------------
point::point(int a)
{ x=y=a; }
//-----------------------
point::point(int a,int b)
{ x=a; y=b; }
//-------------------------
void point::affiche()
{
  cout<<"on est a : "<<x<<" - "<<y<<endl;
}
//-----------------------------------------
void point::affiche(char *t)
{
  cout<<t;
  affiche();
}
//---------------------------
main()
{
     point p1;           p1.affiche();
     point p2(7);        p2.affiche();
     point p3(44,52);    p3.affiche();
     p3.affiche("Troisième point : ");
}
```

Exécution

2. Arguments par défaut

Les fonctions membres, et les fonctions ordinaires, peuvent disposer d'arguments par défaut.

Exemple

Modifier l'exemple précédent de telle façon à ce qu'on remplace les deux fonctions 'affiche' par une seule, à un argument de type chaîne : le texte à afficher avant le message : "on est à : x – y ". Sa valeur par défaut est la chaîne de caractères vide.

3. Les fonctions membres en ligne

Pour rendre 'en ligne' une fonction membre, on l'a défini dans la déclaration de la classe même (au lieu de la déclarer dans la classe et de la définir ailleurs). Le mot clé 'inline' n'est plus utilisé.

Exemple

Reprendre le dernier exemple en définissant les trois constructeurs en ligne.

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

//----------------------------
void point::affiche(char *t)
{
     cout<<t<<"on est à : "<<x<<"-"<<y<<endl;
}
//-----------------------------------------------
main()
{
     ……. ;
     …… ;
}
```

4. Cas des objets transmis en argument d'une fonction membre

Une fonction membre peut recevoir un ou plusieurs arguments du type de sa classe.

Exemple

```
class point
{
       int x, y;
     public :
       point(int a,int b){ x=a; y=b; }
       int coincide(point);
};
//----------------------------
int  point::coincide(point O)
{
     if(x==O.x && y==O.y) return 1;
     return 0;
}
//-----------
main()
{
     point p1(5,7), p2(5,7), p3(6,6);
     cout<<"p1 et p2 : "<<p1.coincide(p2)<<endl;
     cout<<"p2 et p3 : "<<p3.coincide(p2)<<endl;
     cout<<"p1 et p3 : "<<p1.coincide(p3)<<endl;
}
```

Exécution

5. Mode de transmission des objets, arguments d'une fonction

Dans l'exemple ci-dessus, le mode de transmission utilisé, était par valeur.

5.1. Transmission de l'adresse d'un objet

Reprendre le programme précédent en faisant un passage d'objets par adresse.

```
class point
{
       int x, y;
     public :
       point (int a=0,int b=0){ x=a; y=b; }
       int coincide(point *);
};

//----------------------------
```

```
int point::coincide(point *O)
{
     if(x==O->x) && (y==O->y) return 1;
     return 0;
}
//-----------------------------------------------

main()
{
     point p1, p2(57), p3(57,0);
     cout<<"p1 et p2 : "<<p1.coincide(&p2)<<endl;
     cout<<"p2 et p3 : "<<p3.coincide(&p2)<<endl;
     cout<<"p1 et p3 : "<<p1.coincide(&p3)<<endl;
}
```

5.2. Transmission par référence

L'emploi des références permet de mettre en place une transmission par adresse, sans avoir à prendre en charge soi même la gestion.

Exemple

Reprendre le dernier exemple en adaptant la fonction 'coincide' de telle façon à ce que son argument soit transmis par référence.

```
class     point
{
       int x,y ;
     public :
       point(int a=0,int b=0){ x=a; y=b; }
       int coincide(point &);
};
//--------------------------------
int point::coincide(point & O)
{
     return (x==O.x && y==O.y) ? 1 : 0;
}
//----------------------------------------
main()
{
     point p1, p2(57), p3(57,0);
     cout<<"p1 et p2 : "<<p1.coincide(p2)<<endl;
     cout<<"p2 et p3 : "<<p3.coincide(p2)<<endl;
     cout<<"p1 et p3 : "<<p1.coincide(p3)<<endl;
}
```

Exécution

6. Autoréférence : le mot clé 'this'

Le mot clé 'this' utilisé uniquement au sein d'une fonction membre désigne un pointeur sur l'objet l'ayant appelé.

Exemple

```
class point
{
        int x, y;
     public :
        point(int a=0,int b=0){ x=a; y=b; }
        void affiche(){ cout<<"Point : "<<x<<" - "<<y<<"de
                  l'objet dont l'adresse est : "<<this<<endl; }
};
//-------------------------------------------------------------
main()
{
     point p1, p2(5,3), p3(6,70);
     p1.affiche(); p2.affiche(); p3.affiche();
}
```

Exécution

```
Point : 0 - 0 de l'objet dont l'adresse est : 0x22ff40
Point : 5 - 3 de l'objet dont l'adresse est : 0x22ff38
Point : 6 - 7 de l'objet dont l'adresse est : 0x22ff30
```

Chapitre 3 : Construction, destruction et initialisation des objets

1. Objets automatiques et statiques

1.1. Durée de vie d'allocation mémoire

Les objets automatiques sont créés par une déclaration :
- Dans une fonction.
- Dans un bloc.

Les objets statiques créés par une déclaration :
- En dehors de toute fonction.
- Dans une fonction, mais 'static'.

Remarque

Les objets statiques sont créés avant le début de l'exécution de la fonction 'main' et ils sont détruits après la fin de son exécution.

1.2. Appel des constructeurs et des destructeurs

Exemple

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

- point p1(2, 7) ; est une déclaration correcte.
- point p2 ; et point p3(17) ; sont des déclarations incorrectes.

Remarque

- Le constructeur est appelé après la création d'un objet.
- Le destructeur est appelé avant la destruction d'un objet.

2. Les objets temporaires

Lorsqu'une classe dispose d'un constructeur, ce dernier peut être appelé explicitement ; dans ce cas, il y a alors création d'un objet temporaire.

Exemple

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

Supposons que 'p' est un objet de la classe 'point' avec les paramètres (1,1) :

```
point  p(1,1);
```

On peut écrire une affectation telle que :

```
p=point(2,7);
```

L'évaluation de l'expression point(2,7) ; entraîne :
- La création d'un objet temporaire de type 'point' (qui n'a pas de nom).
- L'appel du constructeur 'point', pour cet objet temporaire, avec transmission des arguments spécifiés.
- La recopie de cet objet temporaire dans l'objet p.

Exercice

Ecrire un programme permettant de créer un objet automatique dans la fonction 'main', deux autres objets temporaires affectés à cet objet tout en affichant le moment de leur création et leurs adresses.

```
……….
……….
main()
{
     point     p(1,1);
     p=point(5,2); p.affiche();
     p=point(7,3); p.affiche();
}
```

3. Les objets dynamiques

Exemple

```
……….
main()
{
     point  *p;
     p=new point(7,2);
     p->affiche(); ou (*p).affiche();
     p=new point(8,4);
     p->affiche();
     delete p;
}
```

4. Tableaux d'objets

Un tableau d'objet est déclaré sous la forme :

Exemple

```
point courbe[3];
```

5. Objets d'objets

Une classe peut contenir des membres données de type quelconque y compris le type 'class'.

Exemple

Ecrire un programme permettant de définir une classe appelée 'point_coul' à partir de la classe 'point' définie précédemment et ayant comme données : x, y et couleur.

```
class point
{
       int x,y;
     public :
       point (int a=0,int b=0)
         {
            x=a; y=b;
            cout<<"Construction du point : "<<x<<" - "
            <<y<<endl;
         }
     ~point()
         {
            cout<<"Destruction du point : "<<x<<" - "
            <<y<<endl;
         }
};
//---------------------------------------
class point_coul
{
       point p;
       int couleur;
     public :
       point_coul(int,int,int);
     ~point_coul()
         {
           cout<<"Destruction du point colore En couleur : "
           <<couleur<<endl; getch();
         }
};
point_coul::point_coul(int a,int b,int c):p(a,b)
{
     couleur=c;
     cout<<"Construction de point_coul en couleur : "
     <<couleur<<endl;
}

//---------------------------
main()
{
     point_coul pc(3,7,8);
}
```

- L'entête de point_coul spécifie, après les deux points (:), la liste des arguments qui seront transmis au constructeur 'point'.
- Les constructeurs seront appelés dans l'ordre suivant 'point', 'point_coul'.
- S'il existe des destructeurs, ils seront appelés dans l'ordre inverse.

Exécution

```
Construction de point : 3 - 7
Construction de point colore En couleur : 8
Destruction de point colore En couleur : 8
Destruction de point : 3 - 7
```

6. Initialisation d'un objet lors de sa déclaration

En plus d'une éventuelle initialisation par défaut (réservée aux variables statiques), une variable peut être initialisée explicitement lors de sa déclaration.

Exemple

```
int  n=2;
int  m=3*n-7;
int  t[3]={5,12,43};
```

Remarque

La partie suivant le signe '=' porte le nom d'initialiseur, il ne s'agit pas d'un opérateur d'affectation.

C++ garantie l'appel d'un constructeur pour un objet créé par une déclaration sans initialisation (ou par 'new'). Mais, il est également possible d'associer un initialiseur à la déclaration d'un objet.

6.1. Un premier exemple

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

- point p1(3) ; est une déclaration ordinaire d'un objet 'p1' aux coordonnées 3 et 0.
- point p2=7 ; entraîne :
    - La création d'un objet appelé 'p2'.
    - L'appel du constructeur auquel on transmet en argument la valeur de l'initialiseur '7'.

En fin, les deux déclarations :
'point p1(3) ;' et 'point p2=7 ;' sont équivalentes.

6.2. Constructeur par recopie

L'initialiseur d'un objet peut être d'un type quelconque, en particulier, il peut s'agir du type de l'objet lui-même.

Exemple

```
point p1(33);
```

Il est possible de déclarer un nouvel objet 'p3' tel que :

```
point  p3=p1;  // équivalente à : point   p3(p1);
```

Cette situation est traitée par C++, selon qu'il existe un constructeur ou il n'existe pas de constructeur correspondant à ce cas.

a. Il n'existe pas de constructeur approprié

Cela signifie que, dans la classe 'point', il n'existe aucun constructeur à un seul argument de type 'point'. Dans ce cas, C++ considère que l'on souhaite initialiser l'objet 'p3' après sa déclaration avec les valeurs de l'objet 'p1'. Le compilateur va donc mettre en place une recopie de ces valeurs. (Analogue à celle qui est mise en place lors d'une affectation entre objets de même type).
Ce cas est le seul ou C++ accepte qu'il n'existe pas de constructeur.
Une déclaration telle que : 'point p(x);' sera rejetée si 'x' n'est pas de type 'point'.

b. Il existe un constructeur approprié

Cela signifie qu'il doit exister un constructeur de la forme : 'point (point &);'.
Dans ce cas, ce constructeur est appelé de manière habituelle, après la création de l'objet, sans aucune recopie.

Remarque

C++ impose au constructeur en question que son unique argument soit transmis par référence.
La forme 'point (point) ;' serait rejetée par le compilateur.

6.3. Exemple d'utilisation du constructeur par recopie

6.3.1. Traitement par défaut

```
class tableau
{
       int     ne;
       double *p;
     public :
       tableau(int n)
       {
          p=new double[ne=n];
          cout<<"Constructeur ordinaire : "
          <<this<<"avec son tableau : " <<p<<endl;
       }
```

```
        ~tableau()
        {
          delete p;
          cout<<"Destruction objet : "<<this
          <<"avec son tableau : "<<p<<endl;
        }
};
//---------------------------------------
main()
{
     tableau t1(3);
     tableau t2=t1; // équivalente à 'tableau    t2(t1);'
}
```

Exécution

```
Constructeur ordinaire : 0x22ff20 avec son tableau : 0x8d10f0
Destruction objet : 0x22ff10 avec son tableau : 0x8d10f0
Destruction objet : 0x22ff20 avec son tableau : 0x8d10f0
```

Commentaire

La déclaration : '`tableau t1 ;`' a créé un nouvel objet sans faire appel au constructeur, dans lequel on a recopié les valeurs des membres 'ne' et 'p' de l'objet t1.
A la fin de l'exécution de 'main', il y a appel du destructeur pour 't1' et pour 't2', pour libérer le même emplacement.

6.3.2. Définition d'un constructeur par recopie

On peut éviter le problème posé ci-dessus de telle façon à ce que la déclaration '`tableau t2=t1;`' conduise à créer un nouvel objet de type '`tableau`' avec non seulement ses membres données 'ne' et 'p' mais également son propre tableau dynamique.

Pour ce faire, on définit un constructeur par recopie de la forme :

```
tableau (tableau &);
```

Programme (avec remplissage du tableau dynamique)

```
class tableau
{
        int    ne;
        double *p;
      public :
        tableau(int n)
        {
          p=new double[ne=n];
          cout<<"Constructeur ordinaire : "<<this<<
          " avec son tableau dynamique : "<<p<<endl;
          for(int i=0;i<ne;i++) p[i]=(i+1)*10;
        }
      tableau(tableau & t)
```

```
        {
            ne=t.ne; p=new double[ne];
            cout<<"Constructeur par recopie : "<<this<<
            " avec son tableau dynamique : "<<p<<endl;
            for(int i=0;i<ne;i++) p[i]=t.p[i];
        }
         ~tableau()
        {
          delete []p;
          cout<<"Destruction objet : "<<this<<
          " avec son tableau : dynamique"<<p<<endl;getch();
        }
};
//---------------------------------------
main()
{
     tableau t1(3);
     tableau t2=t1; // équivalente à : tableau t2(t1);
     getch();
}
```

Exécution

```
Constructeur ordinaire : 0x22ff20 avec son tableau dynamique : 0x5d10f0
Constructeur par recopie : 0x22ff10 avec son tableau dynamique : 0x5d1110
Destruction objet : 0x22ff10 avec son tableau : dynamique0x5d1110
Destruction objet : 0x22ff20 avec son tableau : dynamique0x5d10f0
```

Chapitre 4 : Les fonctions amies

En principe, l'encapsulation interdit à une fonction membre d'une classe ou toute fonction d'accéder à des données privées d'une autre classe.

Mais grâce à la notion d'amitié entre fonction et classe, il est possible, lors de la définition de cette classe d'y déclarer une ou plusieurs fonctions (extérieurs de la classe) amies de cette classe.

Une telle déclaration d'amitié les autorise alors à accéder aux données privées, au même titre que n'importe quelle fonction membre.

Il existe plusieurs situations d'amitiés :

1. Fonction indépendante, amie d'une classe.
2. Fonction membre d'une classe, amie d'une autre classe.
3. Fonction amie de plusieurs classes.
4. Toutes les fonctions membres d'une classe, amies d'une autre classe.

1. Exemple de fonction indépendante amie d'une classe

Pour déclarer une fonction amie d'une classe, il suffit de la déclarer dans cette classe en la précédent par le mot clé **'friend'**.

```
class point
{
       int x,y;
     public :
       point(int a=0,int b=0) {x=a; y=b;}
       friend  int  coincide(point,point);
};
//----------------------------------------
int  coincide(point p1,point p2)
{
     if(p1.x==p2.x && p1.y==p2.y) return 1; else return 0;
}
//-------------------------
main()
{
     point o1(15,2), o2(15,2), o3(13,25);
     if(coincide(o1,o2)) cout<<"les objets coïncident\n";
     else cout<<"les objets sont différents\n";

     if(coincide(o1,o3)) cout<<"les objets coïncident\n";
     else cout<<"les objets sont différents\n";
}
```

Exécution

```
les objets o1 et o2 coincident
les objets o1 et o3 sont differents
```

Remarques

-L'emplacement de la déclaration d'amitié dans la classe est quelconque.
-Généralement, une fonction amie d'une classe possédera 1 ou plusieurs arguments ou une valeur de retour du type de cette classe.

2. Les différentes situations d'amitié

2.1. Fonction membre d'une classe, amie d'une autre classe

Syntaxe

```
class     B;
class     A
{
        ……….
     public :
        friend  int  B::f(int,A);
};
//-----------------------------------------------
class     B
{
        ……….
     public:
        int f(int,A);
};
//-------------------------------------------------------
int  B::f(int,A)
{
     ……….
     ……….
}
```

Commentaire

1. la fonction membre 'f' de la classe 'B' peut accéder aux membres privées de n'importe quel objet de la classe 'A'.
2. Pour compiler correctement la déclaration de la classe 'A', contenant une déclaration d'une fonction amie, il faut :
- Définir 'B' avant 'A'.
- Déclarer 'A' avant 'B'.

2.2. Fonction amie de plusieurs classes

Toute fonction membre ou indépendante, peut être amie de plusieurs classes.

Syntaxe

```
class     A
{
        ……….
     public :
        friend  void f(A,B);
};//-------------------------------
class     B
{
        ……….
     public :
        friend  void f(A,B);
};//---------------------------
void f(A,B)
{
     ……..
}
```

2.3. Toutes les fonctions d'une classe sont amies d'une autre classe

Au lieu de faire autant de déclarations de fonctions amies qu'il y a de fonctions membres, on peut résumer toutes ces déclarations en une seule.

Exemple

`'friend  class B;'` déclarée dans la classe 'A' signifie que toutes les fonctions membres de la classe 'B' sont amies de la classe 'A'.

Remarque

Pour compiler la déclaration de la classe 'A', il suffit de la faire précéder de : 'class B ;'
Ce type de fonction évite d'avoir à déclarer, les entêtes des fonctions concernées par l'amitié.

Exercice

Ecrire un programme permettant de réaliser le produit d'une matrice par un vecteur à l'aide d'une fonction indépendante appelée 'produit' amie des deux classes 'matrice' et 'vecteur'.

La classe 'vecteur' possède :
- comme données : un vecteur de 3 éléments entiers.
- comme fonctions membres :
  - Un constructeur à 3 valeurs entiers.
  - Une fonction 'affiche' pour afficher le contenu du tableau (vecteur).

La classe 'matrice' possède :
- comme donnée : une matrice de 9 éléments (3x3).
- comme fonction membre : un constructeur ayant une matrice (3x3) comme paramètre.

La fonction 'produit' retourne normalement un objet de type 'vecteur' résultat du produit d'une matrice par un vecteur.

Programme

```
class vecteur;
class matrice
{
    int m[3][3];
  public :
    matrice(int ma[3][3])
    {
      for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
          m[i][j]=ma[i][j];
    }
    friend vecteur produit(matrice ma,vecteur ve);
};//-----------------------------------------------
class vecteur
{
    int v[3];
  public :
    vecteur(int a=0,int b=0,int c=0){v[0]=a; v[1]=b; v[2]=c;}
    void affiche() {for(int i=0;i<3;i++) cout<<v[i]<<"\t";}
    friend vecteur produit(matrice ma,vecteur ve);
};//-------------------------------------------------------------
vecteur produit(matrice ma,vecteur ve)
{
    int i,j;
    vecteur vect;
    for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
          vect.v[i]+=ma.m[i][j]*ve.v[j];
    return vect;
}//----------------------------------------
main()
{
      vecteur v1(1,2,3);
      int mat[3][3]={1,2,3,4,5,6,7,8,9};
      matrice m1(mat);
      vecteur resultat;
      resultat=produit(m1,v1);
      resultat.affiche();
      getch();
}
```

Exécution

```
14      32      50
```