



# COURS et TP DE LANGAGE C++

Chapitre 12

## Initialisation, construction, destruction des objets

Joëlle MAILLEFERT

joelle.maillefert@iut-cachan.u-psud.fr

IUT de CACHAN

Département GEII 2

# CHAPITRE 12 (***)

# INITIALISATION, CONSTRUCTION, DESTRUCTION DES OBJETS

Dans ce chapitre, on va chercher à mettre en évidence les cas pour lesquels le compilateur cherche à exécuter un constructeur, et quel est ce constructeur et, d'une façon plus générale, on étudiera les mécanismes de construction et de destruction.

## *I- CONSTRUCTION ET DESTRUCTION DES OBJETS AUTOMATIQUES*

Rappel: Une variable locale est appelée encore « automatique », si elle n'est pas précédée du mot « static ». Elle n'est alors pas initialisée et sa portée (ou durée de vie) est limitée au bloc où elle est déclarée.

Exemple et exercice XII-1:
Exécuter le programme suivant et étudier soigneusement à quel moment sont créés puis détruits les objets déclarés. Noter l'écran d'exécution obtenu.

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

    class point
   {
   private:
      int x,y;
   public:
      point(int abs,int ord);
      ~point();
   };

    point::point(int abs,int ord)
   {
      x = abs; y = ord;
      cout<<"Construction du point "<< x <<"  "<< y <<"\n";
   }

    point::~point()
   {cout<<"Destruction du point "<< x <<" "<< y <<"\n";}

    void test()
   {cout<<"Début de test()\n";point u(3,7);cout<<"Fin de test()\n";}

    void main()
   {
      cout<<"Début de main()\n";point a(1,4); test();
      point b(5,10);
      for(int i=0;i<3;i++)point(7+i,12+i);
      cout<<"Fin de main()\n"; getch();
   }
```

## *II- CONSTRUCTION ET DESTRUCTION DES OBJETS STATIQUES*

Exemple et exercice XII-2: Même étude avec le programme suivant:

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

    class point
   {
   private :
      int x,y;
   public:
      point(int abs,int ord);
      ~point();
   };

    point::point(int abs,int ord)
   {
      x = abs; y = ord;
      cout<<"Construction du point "<< x <<"  "<< y <<"\n";
   }

    point::~point()
   {
      cout<<"Destruction du point "<< x <<" "<< y <<"\n";
   }

    void test()
   {
      cout<<"Début de test()\n";
      static point u(3,7); cout<<"Fin de test()\n";
   }

    void main()
   {
      cout<<"Début de main()\n";
      point a(1,4);
      test();
      point b(5,10);
      cout<<"Fin de main()\n";
      getch() ;
   }
```

## *III- CONSTRUCTION ET DESTRUCTION DES OBJETS GLOBAUX*

Exemple et exercice XII-3: Même étude avec le programme suivant

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

    class point
   {
   private:
      int x,y;
   public:
      point(int abs,int ord);
      ~point();
   };

    point::point(int abs,int ord)
   {
      x = abs; y = ord;
      cout<<"Construction du point "<<x<<"  "<<y<<"\n";
   }

    point::~point()
   {
      cout<<"Destruction du point "<<x<<" "<<y<<"\n";
   }

   point a(1,4);  // variable globale

    void main()
   {
      cout<<"Début de main()\n";
      point b(5,10);
      cout<<"Fin de main()\n";
      getch();
   }
```

## *IV- CONSTRUCTION ET DESTRUCTION DES OBJETS DYNAMIQUES*

Exemple et exercice XII-4: Même étude avec le programme suivant

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

    class point
   {
   private:
      int x,y;
   public:
      point(int abs,int ord);
      ~point();
   };

    point::point(int abs,int ord)
   {
      x = abs; y = ord;
      cout<<"Construction du point "<< x <<" "<< y <<"\n";
   }

    point::~point()
   {
      cout<<"Destruction du point "<< x <<" "<< y <<"\n";
   }

    void main()
   {
      cout<<"Début de main()\n";
      point *adr;
      adr = new point(3,7); // réservation de place en mémoire
      delete adr;  // libération de la place
      cout<<"Fin de main()\n";
      getch();
   }
```

Exécuter à nouveau le programme en mettant en commentaires l'instruction « delete adr ».

Donc, dans le cas d'un objet dynamique, le constructeur est exécuté au moment de la réservation de place mémoire (« new »), le destructeur est exécuté lors de la libération de cette place (« delete »).

## *V- INITIALISATION DES OBJETS*

Le langage C++ autorise l'initialisation des variables dès leur déclaration:
Par exemple: **int i = 2;**

Cette initialisation est possible, et de façon plus large, avec les objets:
Par exemple: **point a(5,6); // constructeur avec arguments**

Et même: **point b = a;**

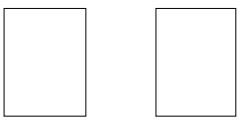
Que se passe-t-il alors à la création du point b ? En particulier, quel constructeur est-il exécuté?

Exemple et exercice XII-5: Tester l'exemple suivant, noter l'écran d'exécution obtenu et conclure

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

    class point
   {
   private :
      int x,y;
   public:
      point(int abs,int ord);
      ~point();}
   ;

    point::point(int abs,int ord)
   {
      x = abs; y = ord;
      cout<<"Construction du point "<< x <<" "<< y;
      cout<<" Son adresse: "<< this <<"\n";
   }

    point::~point()
   {
      cout<<"Destruction du point "<<x<<" "<<y;
      cout<<" Son adresse:"<<this<<"\n";
   }

    void main()
   {
      cout<<"Début de main()\n";
      point a(3,7);
      point b=a;
      cout<<"Fin de main()\n";
      clrscr();
   }
```

Sont donc exécutés ici:
- le constructeur pour a UNIQUEMENT
- le destructeur pour a ET pour b

Le compilateur affecte correctement des emplacements-mémoire différents pour a et b:

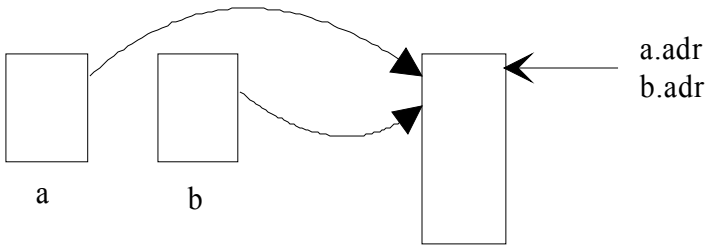


Exemple et exercice XII-6:
Dans l'exemple ci-dessous, la classe **liste** contient un membre privé de type pointeur. Le constructeur lui alloue dynamiquement de la place. Que se passe-t-il lors d'une initialisation de type: **liste a(3);**
**liste b = a;**

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

    class liste
   {
   private :
      int taille;
      float *adr;
   public:
      liste(int t);
      ~liste();
   };

    liste::liste(int t)
   {
      taille = t;adr = new float[taille]; cout<<"Construction";
      cout<<" Adresse de l'objet:"<<this;
      cout<<" Adresse de liste:"<< adr <<"\n";
   }

    liste::~liste()
   {
      cout<<"Destruction Adresse de l'objet:"<<this;
      cout<<" Adresse de liste:"<< adr <<"\n"; delete adr;
   }

    void main()
   {
      cout<<"Début de main()\n";
      liste a(3);
      liste b=a;
      cout<<"Fin de main()\n"; getch();
   }
```

Comme précédemment, sont exécutés ici:
- le constructeur pour a UNIQUEMENT
- le destructeur pour a ET pour b

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



Exercice XII-7:

Ecrire une fonction membre **void saisie()** permettant de saisir au clavier les composantes d'une liste et une fonction membre **void affiche()** permettant de les afficher sur l'écran. Les mettre en oeuvre dans **void main()** en mettant en évidence le défaut vu dans l'exercice IV-6.

*L'étude de ces différents exemples montre que, lorsque le compilateur ne trouve pas de constructeur approprié, il exécute un constructeur par défaut,invisible du programmeur, dont la fonction est de copier les données non allouées dynamiquement .*

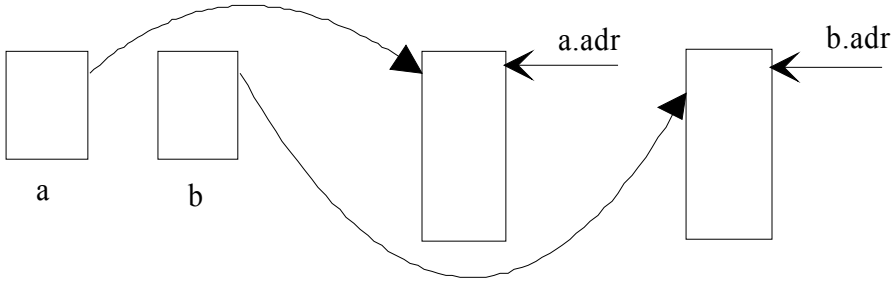
Exemple et exercice XII-8:

On va maintenant ajouter un constructeur de prototype **liste(liste &)** appelé encore « constructeur par recopie ». Ce constructeur sera appelé lors de l'exécution de **liste b=a;**

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

    class liste
   {
   private:
      int taille;
      float *adr;
   public:
      liste(int t);
      liste(liste &v);
      ~liste();
   };

    liste::liste(int t)
   {
      taille = t;adr = new float[taille];
      cout<<"\nConstruction"; cout<<"\nAdresse de l'objet:"<<this;
      cout<<"  Adresse de liste:"<<adr<<"\n";
   }

    liste::liste(liste &v)  // passage par référence obligatoire
   {
      taille = v.taille; adr = new float[taille];
      for(int i=0;i<taille;i++)adr[i] = v.adr[i];
      cout<<"\nConstructeur par recopie";
      cout<<"  Adresse de l'objet:"<<this;
      cout<<"  Adresse de liste:"<<adr<<"\n";
   }

    liste::~liste()
   {
      cout<<"\nDestruction Adresse de l'objet:"<<this;
      cout<<"  Adresse de liste:"<<adr<<"\n"; delete adr;
   }
```

```
 void main()
{
   cout<<"Debut de main()\n";
   liste a(3);
   liste b=a;
   cout<<"\nFin de main()\n";getch() ;
}
```

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




Exercice XII-9:

Reprendre l'exercice IV-7, et montrer qu'avec le « constructeur par recopie », on a résolu le problème rencontré.

## *VI- CONCLUSION*

*Il faut prévoir un « constructeur par recopie » lorsque la classe contient des données dynamiques.*
*Lorsque le compilateur ne trouve pas ce constructeur, aucune erreur n'est générée, par contre, le programme ne fonctionne pas.*

## *VII - ROLE DU CONSTRUCTEUR LORSQU'UNE FONCTION RETOURNE UN OBJET*

On va étudier maintenant une autre application du « constructeur par recopie ».

Exemple et exercice XII-10:

On reprend la fonction membre **point symetrique()** étudiée dans l'exercice III-11. Cette fonction retourne donc un objet.
Tester le programme suivant et étudier avec précision à quel moment les constructeurs et le destructeur sont exécutés.

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

    class point
   {
   private:
      int x,y;
   public:
      point(int abs,int ord);
      point(point &); // constructeur par recopie
      point symetrique();
       void affiche()
      { // Fonction inline
         cout<<"x="<< x <<" y="<< y <<"\n";
      }
      ~point();
   };

    point::point(int abs=0,int ord=0)
   {
      x = abs; y = ord; cout<<"Construction du point "<< x <<"  "<< y;
      cout<<" d'adresse "<< this <<"\n";
   }

    point::point(point &pt)
   {
      x = pt.x; y = pt.y;
      cout<<"Construction par recopie du point "<< x <<" "<< y;
      cout<<" d'adresse "<< this <<"\n";
   }

    point point::symetrique()
   {
      point res; res.x = -x; res.y = -y;
      return res;
   }

    point::~point()
   {
      cout<<"Destruction du point "<<x<<" "<<y;
      cout<<" d'adresse "<< this <<"\n";
   }
```

```
 void main()
{
   cout<<"Début de main()\n";
   point a(1,4), b;
   cout<<"Avant appel à symetrique\n";
   b = a.symetrique();
   b.affiche();
   cout<<"Après appel à symetrique et fin de main()\n"; getch() ;
}
```

Il y a donc création d'un objet temporaire, au moment de la transmission de la valeur de « res » à « b ». Le constructeur par recopie et le destructeur sont exécutés.
Il faut insister sur le fait que la présence du construteur par recopie n'était pas obligatoire ici: l'exercice III-1 a fonctionné correctement ! et se rappeler ce qui a été mentionné plus haut:

*Lorsqu'un constructeur approprié existe, il est exécuté. S'il n'existe pas, aucune erreur n'est générée.*
*Il faut toujours prévoir un constructeur par recopie lorsque l'objet contient une partie dynamique.*

Tester éventuellement le programme IV-10, en supprimant le constructeur par recopie.

Exemple et exercice XII-11:

On a écrit ici, pour la classe **liste** étudiée précédemment, une fonction membre de prototype **liste oppose()** qui retourne la liste de coordonnées opposées.
Exécuter ce programme et conclure.

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

 class liste
{
private :
   int taille;
   float *adr;
public:
   liste(int t);
   liste(liste &v);
   void saisie();
   void affiche();
   liste oppose();
   ~liste();
};
```

```
 liste::liste(int t)
{
   taille = t; adr = new float[taille];
   cout<<"Construction";
   cout<<" Adresse de l'objet:"<<this;
   cout<<" Adresse de liste:"<<adr<<"\n";
}

 liste::liste(liste &v)  // passage par référence obligatoire
{
   taille = v.taille;
   adr = new float[taille];
   for(int i=0;i<taille;i++)adr[i]=v.adr[i];
   cout<<"Constructeur par recopie";
   cout<<" Adresse de l'objet:"<< this;
   cout<<" Adresse de liste:"<< adr <<"\n";
}

 liste::~liste()
{
   cout<<"Destruction Adresse de l'objet:"<< this;
   cout<<" Adresse de liste:"<< adr <<"\n"; delete adr;
}

 void liste::saisie()
{
   for(int i=0;i<taille;i++)
   {cout<<"Entrer un nombre:"; cin>>*(adr+i);}
}

 void liste::affiche()
{
   for(int i=0;i<taille;i++)cout<<*(adr+i)<<" ";
   cout<<"adresse de l'objet: "<< this;
   cout<<" adresse de liste: "<< adr <<"\n";
}

 liste liste::oppose()
{
   liste res(taille);
   for(int i=0;i<taille;i++)res.adr[i] = - adr[i];
   for(i=0;i<taille;i++)cout<<res.adr[i]<<" ";
   cout<<"\n";
   return res;
}

 void main()
{
   cout<<"Début de main()\n";
   liste a(3), b(3);
   a.saisie();     a.affiche();
   b = a.oppose(); b.affiche();
   cout<<"Fin de main()\n"; getch();
}
```

Solution et exercice XII-12:

On constate donc que l'objet local **res** de la fonction **oppose()** est détruit AVANT que la transmission de valeur ait été faite. Ainsi, la libération de place mémoire a lieu trop tôt. Ré-écrire la fonction **oppose()** en effectuant le retour par référence (cf chapitre 3) et conclure sur le rôle du retour par référence.

## *VIII- EXERCICES RECAPITULATIFS*

Exercice XII-13:

Ecrire une classe **pile_entier** permettant de gérer une pile d'entiers, selon le modèle ci-dessous.

```
 class  pile_entier
{
private:
 // pointeur de pile, taille maximum, hauteur courante
   int *pile,taille,hauteur;
public:
// constructeur : alloue dynamiquement de la mémoire
// taille de la pile(20 par défaut), initialise la hauteur à 0
   pile_entier(int n);
   ~pile_entier();     // destructeur
   void empile(int p); // ajoute un élément
   int depile(); // retourne la valeur de l'entier en haut de la pile
                 // la hauteur  diminue d'une 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 oeuvre cette classe dans main(). Le programme principal doit contenir les déclarations suivantes:

```
 void main()
{
   pile_entier a,b(15); // pile automatique
   pile_entier *adp;    // pile dynamique
}
```

Exercice XII-14:

Ajouter un constructeur par recopie et le mettre en oeuvre.

## *IX- LES 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].affiche()** a un sens.
La classe courbe étant un tableau contenant des objets de la classe **point** doit dans ce cas, OBLIGATOIREMENT posséder un **constructeur** sans argument (ou avec tous les arguments avec valeur par défaut). *Le constructeur est exécuté pour chaque élément du tableau.*
La notation suivante est admise:

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

 void main()
{
   point courbe[5] = {7,4,2};
}
```

On obtiendra les résultats suivants:

| | x | y |
|---|---|---|
| **courbe[0]** | **7** | **0** |
| **courbe[1]** | **4** | **0** |
| **courbe[2]** | **2** | **0** |
| **courbe[3]** | **0** | **0** |
| **courbe[4]** | **0** | **0** |

On pourra de la même façon créer un tableau dynamiquement:

**point *adcourbe = new point[20];**

et utiliser les notations ci-dessus. Pour détruire ce tableau, on écrira **delete []adcourbe;**
Le destructeur sera alors exécuté pour chaque élément du tableau.

Exercice XII-15:

Reprendre par exemple l'exercice III-8 (classe **vecteur**), et mettre en oeuvre dans le programme principal un tableau de vecteurs.

## X - CORRIGE DES EXERCICES

Exercice XII-7:

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

    class liste
   {
   private:
      int taille;
      float *adr;
   public:
      liste(int t);
      void saisie();
      void affiche();
      ~liste();
   };

    liste::liste(int t)
   {
      taille = t;adr = new float[taille];cout<<"Construction";
      cout<<" Adresse de l'objet:"<<this;
      cout<<" Adresse de liste:"<<adr<<"\n";
   }

    liste::~liste()
   {
      cout<<"Destruction Adresse de l'objet:"<<this;
      cout<<" Adresse de liste:"<<adr<<"\n";
      delete adr;
   }

    void liste::saisie()
   {
      for(int i=0;i<taille;i++)
      {cout<<"Entrer un nombre:"; cin>>*(adr+i);}
   }

    void liste::affiche()
   {
      for(int i=0;i<taille;i++)cout<<*(adr+i)<<" ";
      cout<<"\n";
   }

    void main()
   {
      cout<<"Debut de main()\n";
      liste a(3);
      liste b=a;
      a.saisie();a.affiche();
      b.saisie();b.affiche();a.affiche();
      cout<<"Fin de main()\n"; getch() ;
   }
```

Exercice XII-9:

Même programme qu'au IV-7, en ajoutant le « constructeur par recopie » du IV-8.

Exercice XII-12:

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

    class liste
   {
   private:
      int taille;
      float *adr;
   public:
      liste(int t);
      liste(liste &v);
      void saisie();
      void affiche();
      liste &oppose();
      ~liste();
   };

    liste::liste(int t)
   {
      taille = t; adr = new float[taille];
      cout<<"Construction";
      cout<<" Adresse de l'objet:"<<this;
      cout<<" Adresse de liste:"<<adr<<"\n";
   }

    liste::liste(liste &v)  // passage par référence obligatoire
   {
      taille = v.taille;
      adr = new float[taille];
      for(int i=0;i<taille;i++)adr[i]=v.adr[i];
      cout<<"Constructeur par recopie";
      cout<<" Adresse de l'objet:"<<this;
      cout<<" Adresse de liste:"<<adr<<"\n";
   }

    liste::~liste()
   {
      cout<<"Destruction Adresse de l'objet:"<<this;
      cout<<" Adresse de liste:"<<adr<<"\n";
      delete adr;
   }

    void liste::saisie()
   {
      for(int i=0;i<taille;i++)
      {cout<<"Entrer un nombre:";cin>> *(adr+i) ;}
   }
```

```
 void liste::affiche()
{
   for(int i=0;i<taille;i++)cout<<*(adr+i)<<" ";
   cout<<"Adresse de l'objet: "<<this;
   cout<<" Adresse de liste: "<<adr<<"\n";
}

 liste &liste::oppose()
{
   static liste res(taille);
   for(int i=0;i<taille;i++)*(res.adr+i) = - *(adr+i);
   for(i=0;i<taille;i++)cout<<*(res.adr+i);
   cout<<"\n";
   return res;
}

 void main()
{
   cout<<"Début de main()\n";
   liste a(3),b(3);
   a.saisie();a.affiche();
   b = a.oppose();b.affiche();
   cout<<"Fin de main()\n";
   getch();
}
```

Exercice XII-13:

```
#include <iostream.h>  // Gestion d'une pile d'entiers
#include <conio.h>

 class  pile_entier
{
private :
   int *pile;
   int taille;
   int hauteur;
public:
   pile_entier(int n); // constructeur, taille de la pile
   ~pile_entier();     // destructeur
   void empile(int p); // ajoute un élément
   int depile();       // dépile un élément
   int pleine();       // 1 si vrai 0 sinon
   int vide();         // 1 si vrai 0 sinon
};
```

```
 pile_entier::pile_entier(int n=20)  // taille par défaut: 20
{
   taille = n;
   pile = new int[taille];  // taille de la pile
   hauteur = 0;
   cout<<"On a fabrique une pile de "<<taille<<" éléments\n";
}

 pile_entier::~pile_entier()
{
   delete pile; // libère la mémoire
}

 void pile_entier::empile(int p)
{
   *(pile+hauteur) = p; hauteur++;
}

 int pile_entier::depile()
{
   hauteur--;
   int res = *(pile+hauteur);
   return res;
}

 int pile_entier::pleine()
{
   if(hauteur==taille)
      return 1;
   return 0;
}

 int pile_entier::vide()
{
   if(hauteur==0)
      return 1;
   return 0;
}

 void main()
{
   pile_entier a, b(15);   // pile automatique
   a.empile(8);
   if(a.vide()==1) cout<<"a vide\n";
   else cout<<"a non vide\n";

   pile_entier *adp;          // pile dynamique
   adp = new pile_entier(5);  // pointeur sur pile de 5 entiers
   for(int i=0;adp->pleine()!=1;i++) adp->empile(10*i);
   cout<<"\nContenu de la pile dynamique:\n";
   for(int i=0;i<5;i++)
      if(adp->vide()!=1) cout<<adp->depile()<<"\n";
   getch();
}
```

Exercice XII-14:

```
#include <iostream.h>  // constructeur par recopie
#include <conio.h>

    class  pile_entier
   {
   private :
      int *pile,taille,hauteur;
   public:
      pile_entier(int n); // constructeur, taille de la pile
      pile_entier(pile_entier &p);  // constructeur par recopie
      ~pile_entier(); // destructeur
      void empile(int p); // ajoute un élément
      int depile();       // dépile un élément
      int pleine();       // 1 si vrai   0 sinon
      int vide();         // 1 si vrai  0 sinon
   };

    pile_entier::pile_entier(int n=20)  // taille par défaut: 20
   {
      taille = n;
      pile = new int[taille];  // taille de la pile
      hauteur = 0;
      cout<<"On a fabrique une pile de "<<taille<<" éléments\n";
      cout<<"Adresse de la pile: "<<pile<<" ; de l'objet: "<<this<<"\n";
   }

    pile_entier::pile_entier(pile_entier &p)
   {
      taille = p.taille; hauteur = p.hauteur;
      pile=new int[taille];
      for(int i=0;i<hauteur;i++)*(pile+i) = p.pile[i];
      cout<<"On a fabriqué une pile de "<<taille<<" éléments\n";
      cout<<"Adresse de la pile: "<<pile<<" ; de l'objet: "<<this<<"\n";
   }

    pile_entier::~pile_entier()
   {
      delete pile; // libère la mémoire
   }

    void pile_entier::empile(int p)
   {
      *(pile+hauteur) = p; hauteur++;
   }

    int pile_entier::depile()
   {
      hauteur--;
      int res=*(pile+hauteur);
      return res;
   }
```

```
 int pile_entier::pleine()
{
   if(hauteur==taille)
      return 1;
   else
      return 0;
}

 int pile_entier::vide()
{
   if(hauteur==0)
      return 1;
   else
      return 0;
}

 void main()
{
   cout<<"Pile a:\n";pile_entier a(10);
   for(int i=0;a.pleine()!=1;i++)a.empile(2*i);
   cout<<"Pile b:\n";
   pile_entier b = a;
   while(b.vide()!=1)cout<<b.depile()<<"  "; getch();
}
```

Exercice XII-15:

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

// Tableau de vecteurs

 class vecteur
{
private:
   float x,y;
public: vecteur(float abs,float ord);
   void  homothetie(float val);
   void affiche();
   float det(vecteur w);
};

 vecteur::vecteur(float abs =5.0,float ord = 3.0)
{
   x = abs;
   y = ord;
}

 void vecteur::homothetie(float val)
{
   x = x*val;
   y = y*val;
}
```

```
 void vecteur::affiche()
{
   cout<<"x = "<< x <<"  y = "<< y <<"\n";
}

 float vecteur::det(vecteur w)
{
   float res = x * w.y - y * w.x;
   return res;
}

 void main()
{
   vecteur v[4]={17,9},*u;
   u = new vecteur[3];  // tableau de 3 vecteurs
   for(int i=0;i<4;i++)
   {
      v[i].affiche();
   }
   v[2].homothetie(3);
   v[2].affiche();

   cout <<"Determinant de (u1,v0) = "<<v[0].det(u[1])<<"\n";
   cout <<"Determinant de (v2,u2) = "<<u[2].det(v[2])<<"\n";

   delete []u; // noter les crochets

   getch();
}
```