COURS et TP DE LANGAGE C++

Chapitre 15

L'héritage

Joëlle MAILLEFERT

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

IUT de CACHAN

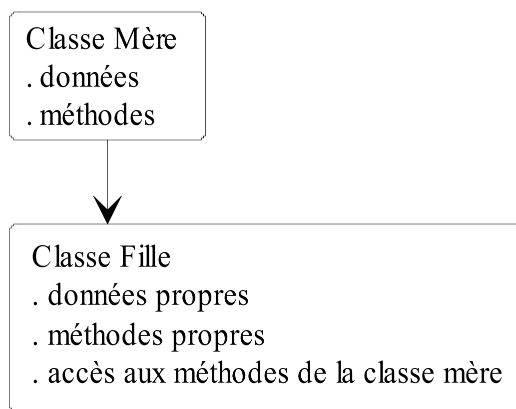
Département GEII 2

CHAPITRE 15

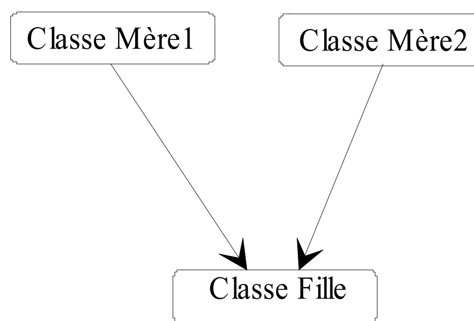
L'HERITAGE

La P.O.O. permet de définir de nouvelles classes (classes filles) dérivées de classes de base (classes mères), avec de nouvelles potentialités. Ceci permettra à l'utilisateur, à partir d'une bibliothèque de classes donnée, de développer ses propres classes munies de fonctionnalités propres à l'application.
On dit qu'une classe fille DERIVE d'une ou de plusieurs classes mères.

Héritage simple:



Héritage multiple:



La classe fille n'a pas accès aux données (privées) de la classe mère.

I- DERIVATION DES FONCTIONS MEMBRES

Exemple (à tester) et exercice XV-1:
L'exemple ci-dessous illustre les mécanismes de base :

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

    class vecteur  // classe mère
   {

   private:
      float x,y;
   public:
      void initialise(float,float);
      void  homothetie(float);
      void affiche();
   };
```

```
 void vecteur::initialise(float abs =0.,float ord = 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";
}
 // classe fille
 class vecteur3:public vecteur
{
private :
   float z;
public:
   void initialise3(float,float,float);
   void  homothetie3(float);
    void hauteur(float ha){z = ha;}
   void affiche3();
};

 void vecteur3::initialise3(float abs=0.,float ord=0.,float haut=0.)
{  // appel de la fonction membre de la classe vecteur
   initialise(abs,ord); z = haut;
}

 void vecteur3:: homothetie3(float val)
{  // appel de la fonction membre de la classe vecteur
   homothetie(val);  z = z*val;
}

 void vecteur3::affiche3()
{  // appel de la fonction membre de la classe vecteur
   affiche();cout<<"z = "<<z<<"\n";
}

 void main()
{
   vecteur3 v, w;
   v.initialise3(5,4,3);v.affiche3();  // fonctions de la fille
   w.initialise(8,2);w.hauteur(7);w.affiche(); // fonctions de la mère
   cout<<"*******************************\n";
   w.affiche3();w.homothetie3(6);w.affiche3(); //fonctions de la fille
   getch();
}
```

L’exemple ci-dessus présente une syntaxe assez lourde et très dangereuse. Il serait plus simple, pour l’utilisateur, de donner aux fonctions membres de la classe fille, le même nom que dans la classe mère, lorsque celles-ci jouent le même rôle (ici fonctions **initialise** et **homothétie**).
Ceci est possible en utilisant la propriété de ***redéfinition*** des fonctions membres.

Exemple (à tester) et exercice XV-2:

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

    class vecteur  // classe mère
   {
   private:
      float x,y;
   public: void initialise(float,float);
      void  homothetie(float);
      void affiche();
   };

    void vecteur::initialise(float abs =0.,float ord = 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";
   }

    class vecteur3:public vecteur // classe fille
   {
   private :
      float z;
   public:
      void initialise(float,float,float);
      void  homothetie(float);
       void hauteur(float ha)
      {
         z = ha;
      }
      void affiche();
   };
```

```
 void vecteur3::initialise(float abs=0.,float ord=0.,float haut=0.)
{  // appel de la fonction membre de la classe vecteur
   vecteur::initialise(abs,ord);
   z = haut;
}

 void vecteur3:: homothetie(float val)
{  // appel de la fonction membre de la classe vecteur
   vecteur::homothetie(val);
   z = z*val;
}

 void vecteur3::affiche()
{  // appel de la fonction membre de la classe vecteur
   vecteur::affiche();
   cout<<"z = "<<z<<"\n";
}

 void main()
{
   vecteur3 v, w;
   v.initialise(5,4,3); v.affiche();
   w.initialise(8,2); w.hauteur(7);
   w.affiche();
   cout<<"******************************\n";
   w.affiche();
   w.homothetie(6); w.affiche();
   getch();
}
```

Exercice XV-3:

A partir de l’exemple précédent, créer un projet. La classe mère sera considérée comme une bibliothèque. Définir un fichier **mere.h** contenant les lignes suivantes :

```
 class vecteur  // classe mère
{
private:
   float x,y;
public: void initialise(float,float);
   void  homothetie(float);
   void affiche();
};
```

Le programme utilisateur contiendra la définition de la classe fille, et le programme principal.

Exercice XV-4:

Dans le programme principal précédent, mettre en œuvre des pointeurs de **vecteur.**

Remarque :
L'amitié n'est pas transmissible: une fonction amie de la classe mère ne sera amie que de la classe fille que si elle a été déclarée amie dans la classe fille.

II- DERIVATION DES CONSTRUCTEURS ET DU DESTRUCTEUR

On suppose la situation suivante :

class A
{ ...
public :
A (......) ; // constructeur
~A ( ) ;   // destructeur
....
} ;

class B : public A
{ ...
public :
B (......) ; // constructeur
~B( ) ;   // destructeur
....
} ;

Si on déclare un objet B, seront exécutés
- Le constructeur de A, puis le constructeur de B,
- Le destructeur de B, puis le destructeur de A.

Exemple (à tester) et exercice XV-5:

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

    class vecteur  // classe mère
   {
   private:
      float x,y;
   public:
      vecteur(); // constructeur
      void affiche();
      ~vecteur();      // destructeur
   };

    vecteur::vecteur()
   {
      x=1; y=2; cout<<"Constructeur de la classe mère\n";
   }
    void vecteur::affiche()
   {
      cout<<"x = "<<x<<"  y = "<<y<<"\n";
   }
    vecteur::~vecteur()
   {
      cout<<"Destructeur de la classe mère\n";
   }
```

```
 class vecteur3 : public vecteur // classe fille
{
private :
   float z;
public:
   vecteur3();  // Constructeur
   void affiche();
   ~vecteur3();
};

 vecteur3::vecteur3()
{
   z = 3;
   cout<<"Constructeur de la classe fille\n";
}

 void vecteur3::affiche()
{
   vecteur::affiche();
   cout<<"z = "<<z<<"\n";
}

 vecteur3::~vecteur3()
{
   cout<<"Destructeur de la classe fille\n";
}

 void main()
{
   vecteur3 v; v.affiche();
   getch();
}
```

Lorsque il faut passer des paramètres aux constructeurs, on a la possibilité de spécifier au compilateur vers lequel des 2 constructeurs, les paramètres sont destinés :

Exemple (à tester) et exercice XV-6:
Modifier le programme principal, pour tester les différentes possibilités de passage d'arguments par défaut.

```
#include <iostream.h>
#include <conio.h>
// Héritage simple
    class vecteur  // classe mère
   {
   private:
      float x,y;
   public:
      vecteur(float,float); // constructeur
      void affiche();
      ~vecteur();  // destructeur
   };

    vecteur::vecteur(float abs=1, float ord=2)
   {
      x=abs;y=ord; cout<<"Constructeur  de la classe mère\n";
   }
__
    void vecteur::affiche()
   {
      cout<<"x = "<<x<<"  y = "<<y<<"\n";
   }

    vecteur::~vecteur()
   {
      cout<<"Destructeur de la classe mère\n";
   }

    class vecteur3:public vecteur    // classe fille
   {
   private:
      float z;
   public:
      vecteur3(float, float, float); // Constructeur
      void affiche();
      ~vecteur3();
   };
```

```
vecteur3::vecteur3(float abs=3, float ord=4, float haut=5)
:vecteur(abs,ord)
{// les 2 1ers paramètres sont pour le constructeur de la classe mère
  z = haut; cout<<"Constructeur fille\n";
}

void vecteur3::affiche()
{
  vecteur::affiche(); cout<<"z = "<<z<<"\n";
}

vecteur3::~vecteur3()
{
  cout<<"Destructeur de la classe fille\n";
}

void main()
{
  vecteur u; vecteur3 v, w(7,8,9);
  u.affiche();v.affiche(); w.affiche();
  getch();
}
```

Cas du constructeur par recopie (***)

Rappel : Le constructeur par recopie est appelé dans 2 cas :
- Initialisation d'un objet par un objet de même type :

  vecteur a (3,2) ;
  vecteur b = a ;
- Lorsqu'une fonction retourne un objet par valeur :

  vecteur a, b ;
  b = a.symetrique() ;

Dans le cas de l'héritage, on doit définir un constructeur par recopie pour la classe fille, qui appelle le constructeur par recopie de la classe mère.

Exemple (à tester) et exercice XV-7:

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

 // classe mère

    class vecteur
   {
   private:
      float x,y;
   public:
      vecteur(float,float); // constructeur
      vecteur(vecteur &);   // constructeur par recopie
      void affiche();
      ~vecteur();           // destructeur
   };

    vecteur::vecteur(float abs=1, float ord=2)
   {
      x=abs;
      y=ord;
      cout<<"Constructeur de la classe mère\n";
   }

    vecteur::vecteur(vecteur &v)
   {
      x=v.x;
      y=v.y;
      cout<<"Constructeur par recopie de la classe mère\n";
   }

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

    vecteur::~vecteur()
   {
      cout<<"Destructeur de la classe mère\n";
   }
```

```
 class vecteur3:public vecteur // classe fille
{
private:
   float z;
public:
   vecteur3(float, float, float);  // Constructeur
   vecteur3(vecteur3 &);  // Constructeur par recopie
   void affiche();
   ~vecteur3();
};

 vecteur3::vecteur3(float abs=3, float ord=4, float haut=5)
 :vecteur(abs,ord)
{
   z = haut; cout<<"Constructeur de la classe fille\n";
}

 vecteur3::vecteur3(vecteur3 &v)
 :vecteur(v)// appel du constructeur par recopie de la classe vecteur
{
   z = v.z; cout<<"Constructeur par recopie de la classe fille\n";
}

 void vecteur3::affiche()
{
   vecteur::affiche(); cout<<"z = "<<z<<"\n";
}

 vecteur3::~vecteur3()
{
   cout<<"Destructeur de la classe fille\n";
}

 void main()
{
   vecteur3 v(5,6,7); vecteur3 w = v;
   v.affiche(); w.affiche();
   getch();
}
```

III- LES MEMBRES PROTEGES

On peut donner à certaines données d'une classe mère le statut « protégé ».
Dans ce cas, les fonctions membres, et les fonctions amies de la classe fille auront accès aux données de la classe mère :

```
 class vecteur  // classe mère
{
protected:
   float x,y;
public:
   vecteur(float,float); // constructeur
   vecteur(vecteur &);   // constructeur par recopie
   void affiche();
   ~vecteur();           // destructeur
};

 void vecteur3::affiche()
{
   cout<< "x = "<<x<<"  y= "<<y<< "  z = "<<z<<"\n";
}
```

La fonction **affiche** de la classe **vecteur3** a accès aux données **x** et **y** de la classe **vecteur**. Cette possibilité viole le principe d'encapsulation des données, on l'utilise pour simplifier le code généré.

IV- EXERCICES RECAPITULATIFS

On dérive la classe **chaîne** de l'exercice V-10:

```
 class chaine
{
private:
   int longueur; char *adr;
public:
   chaine();
   chaine(char *);
   chaine(chaine &); //constructeurs
   ~chaine();
   void operator=(chaine &);
   int operator==(chaine);
   chaine &operator+(chaine);
   char &operator[](int);
   void affiche();
};
```

La classe dérivée se nomme **chaine_T**.

```
 class chaine_T : public chaine
{
   int Type ;
   float Val ;
public :
   // .......
};
```

Type prendra 2 valeurs : 0 ou 1.
1 si la chaîne désigne un nombre, par exemple « 456 » ou « 456.8 », exploitable par **atof** la valeur retournée sera Val.
0 dans les autres cas, par exemple « BONJOUR » ou « XFLR6 ».

Exercice XV-8: Prévoir pour chaine_T
- un constructeur de prototype **chaine_T() ;** qui initialise les 3 nombres à 0.
- un constructeur de prototype **chaine_T(char *) ;** qui initialise les 3 nombres à 0 ainsi que la chaîne de caractères.
- une fonction membre de prototype **void affiche()** qui appelle la fonction **affiche** de **chaine** et qui affiche les valeurs des 3 nombres.

Exercice XV-9 (***): Prévoir un constructeur par recopie pour chaine_T , qui initialise les 3 nombres à 0.

Exercice XV-10: Déclarer « protected » la donnée **adr** de la classe **chaîne.** Ecrire une fonction membre pour chaine_T de prototype **void calcul()** qui donne les bonnes valeurs à Type, Val.

V- CONVERSIONS DE TYPE ENTRE CLASSES MERE ET FILLE

Règle : La conversion d'un objet de la classe fille en un objet de la classe mère est implicite.
La conversion d'un objet de la classe mère en un objet de la classe fille est interdite.

Autrement dit :

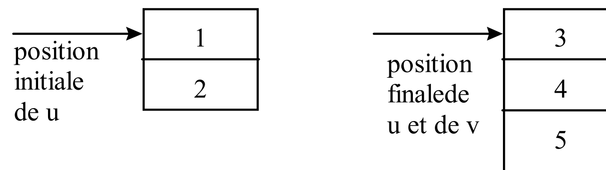
```
vecteur u ;
vecteur3  v;
u = v ;// est autorisée : conversion fictive de v en un vecteur
v = u; // est interdite
```

Exercice XV-11 : Tester ceci avec le programme XV-6

Avec des pointeurs :

```
vecteur *u ;
// réservation de mémoire, le constructeur de vecteur est exécuté
u = new vecteur ;
vecteur3  *v ;
// réservation de mémoire, le constructeur de vecteur3 est exécuté
v = new vecteur3 ;
u = v ; // autorisé, u vient pointer sur la même adresse que v
v = u;  // interdit
delete u ;
delete v ;
```

On obtient la configuration mémoire suivante :



Exemple (à tester) et exercice XV-12 :
Reprendre l'exemple XV-11 avec le programme principal suivant :

```
 void main()
{
   vecteur3 *u = new vecteur3;
   u->affiche();
   vecteur *v = new vecteur;
   v->affiche();
   v = u;  v->affiche();
   delete v ; delete u ; getch();
}
```

Conclusion : quelle est la fonction **affiche** exécutée lors du 2ème appel à **v->affiche()** ?
Le compilateur C++ a-t-il « compris » que v ne pointait plus sur un **vecteur** mais sur un **vecteur3** ?
Grâce à la notion de fonction virtuelle on pourra, pendant l'exécution du programme, tenir compte du type de l'objet pointé, indépendamment de la déclaration initiale.

VI- SURCHARGE DE L'OPERATEUR D'AFFECTATION (***)

Rappels:
- Le C++ définit l'opérateur = pour exprimer l'affectation.
- Il est impératif de le surcharger via une fonction membre, lorsque la classe contient des données de type pointeur pour allouer dynamiquement la mémoire.

A est la classe mère, B est la classe fille, on exécute les instructions suivantes :

B x, y ;
y = x ;

Dans ce cas:
a) Si ni A, ni B n'ont surchargé l'opérateur =, le mécanisme d'affectation par défaut est mis en œuvre.
b) Si = est surchargé dans A mais pas dans B, la surcharge est mise en œuvre pour les données A de B, le mécanisme d'affectation par défaut est mis en œuvre pour les données propres à B.
c) Si = est surchargé dans B, cette surcharge doit prendre en charge la TOTALITE des données (celles de A et celles de B).

Exemple (à tester) et exercice XV-13 :

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

    class vecteur
   {
   private:
      float x,y;
   public:
      vecteur(float,float);
      void affiche();
      void operator=(vecteur &); // surcharge de l'opérateur =
   };

    vecteur::vecteur(float abs=1, float ord=2)
   {
      x=abs; y=ord; cout<<"Constructeur de la classe mère\n";
   }

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

```
 void vecteur::operator=(vecteur &v)
{
   cout<<"opérateur affectation de la classe mère\n";
   x = v.x; y = v.y;
}

 class vecteur3 : public vecteur // classe fille
{
private:
   float z;
public:
   vecteur3(float, float, float); // Constructeur
   void operator=(vecteur3 &);    // surcharge de l'opérateur =
   void affiche();
};

 void vecteur3::operator=(vecteur3 &v)
{
   cout<<"operateur affectation de la classe fille\n";
   vecteur *u, *w;
   u = this; w = &v; *u = *w; z = v.z;
}

 vecteur3::vecteur3(float abs=3, float ord=4, float haut=5)
 :vecteur(abs,ord)
{// les 2 1ers paramètres sont pour le constructeur de la classe mère
   z = haut;
   cout<<"Constructeur fille\n";
}

 void vecteur3::affiche()
{
   vecteur::affiche();
   cout<<"z = "<<z<<"\n";
}

 void main()
{
   vecteur3 v1(6,7,8),v2;
   v2 = v1;
   v2.affiche(); getch();
}
```

VII- LES FONCTIONS VIRTUELLES

Les fonctions membres **de la classe mère** peuvent être déclarées ***virtual***.
Dans ce cas, on résout le problème invoqué dans le §IV.

Exemple (à tester) et exercice XV-14 :

Reprendre l'exercice XV-12 en déclarant la fonction **affiche**, **virtual** :

```
 class vecteur  // classe mère
{
private:
   float x,y;
public:
   vecteur(float,float); // constructeur
   virtual void affiche();
   ~vecteur();           // destructeur
};
```

Quelle est la fonction **affiche** exécutée lors du 2ème appel à **v->affiche()** ?
Le programme a-t-il « compris » que v ne pointait plus sur un **vecteur** mais sur un **vecteur3** ?

Ici, le choix de la fonction à exécuter ne s'est pas fait lors de la compilation, mais ***dynamiquement*** lors de l'exécution du programme.

Exemple (à tester) et exercice XV-15 :

Modifier les 2 classes de l'exercice précédent comme ci-dessous :

```
 class vecteur  // classe mère
{
private:
   float x,y;
public:
   vecteur(float,float); // constructeur
   virtual void affiche();
    void message() {cout<<"Message du vecteur\n";}
   ~vecteur();  // destructeur
};

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

 class vecteur3 : public vecteur // classe fille
{
private:
   float z;
public:
   vecteur3(float, float, float);  // Constructeur
   void affiche();
    void message(){cout<<"Message du vecteur3\n";}
   ~vecteur3();
};

 void vecteur3::affiche()
{
   message();
   vecteur::affiche();
   cout<<"z = "<<z<<"\n";
}
```

Remarques :
- Un constructeur ne peut pas être virtuel,
- Un destructeur peut-être virtuel,
- La déclaration d'une fonction membre virtuelle dans la classe mère, sera comprise par toutes les classes descendantes (sur toutes les générations).

Exercice XV-16 :

Expérimenter le principe des fonctions virtuelles avec le destructeur de la classe **vecteur** et conclure.

VIII- CORRIGE DES EXERCICES

Exercice XV-3: Le projet se nomme exvii_3, et contient les fichiers exvii_3.cpp et mere.cpp ou bien mere.obj.

Fichier exvii_3.cpp:

```
#include <iostream.h>
#include <conio.h>
#include "c:\bc5\cours_cpp\teach_cp\chap7\mere.h"

// Construction d'un projet à partir d'une classe mère disponible

    class vecteur3 : public vecteur // classe fille
   {
   private:
      float z;
   public:
      void initialise(float,float,float);
      void  homothetie(float);
       void hauteur(float ha){z = ha;}
      void affiche();
   };

    void vecteur3::initialise(float abs=0.,float ord=0.,float haut=0.)
   {  // fonction membre de la classe vecteur
      vecteur::initialise(abs,ord); z = haut;
   }

    void vecteur3:: homothetie(float val)
   {  // fonction membre de la classe vecteur
      vecteur:: homothetie(val); z = z*val;
   }

    void vecteur3::affiche()
   {  // fonction membre de la classe vecteur
      vecteur::affiche(); cout<<"z = "<<z<<"\n";
   }

    void main()
   {
      vecteur3 v, w;
      v.initialise(5,4,3); v.affiche();
      w.initialise(8,2);   w.hauteur(7);
      w.affiche();
      cout<<"*******************************\n";
      w.affiche();
      w.homothetie(6); w.affiche();
   }
```

Fichier mere.cpp:

```
#include <iostream.h>
#include <conio.h>
#include "c:\bc5\cours_cpp\teach_cp\chap7\mere.h"

    void vecteur::initialise(float abs =0.,float ord = 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";
   }
```

Exercice XV-4 :

Programme principal :

```
 void main()
{
   vecteur3 v, *w;
   w = new vecteur3;
   v.initialise(5, 4, 3);v.affiche();
   w->initialise(8,2);w->hauteur(7);
   w->affiche();
   cout<<"*******************************\n";
   w->affiche();
   w->homothetie(6);w->affiche();
   delete w;
}
```

Exercice XV-8 : Seuls la classe **chaine_T** et le programme principal sont listés

```
 class chaine_T : public chaine
{
   int Type;
   float Val ;
public:
   chaine_T(); // constructeurs
   chaine_T(char *);
   void affiche();
};

// dans les 2 cas le constructeur correspondant de chaine est appelé
 chaine_T::chaine_T() : chaine()
{
   Type=0; Val=0;
}
_
 chaine_T::chaine_T(char *texte) : chaine(texte)
{
   Type=0; Val=0;
}

 void chaine_T::affiche()
{
   chaine::affiche();
   cout<<"Type= "<<Type<<"  Val= "<<Val<<"\n";
}
```

```
 void main()
{
   chaine a("Bonjour ");
   chaine_T b("Coucou "),c;
   cout<<"a: "; a.affiche();
   cout<<"b: "; b.affiche();
   cout<<"c: "; c.affiche();
   getch();
}
```

<u>Exercice XV-9 :</u> Seules les modifications a été listées

```
 class chaine_T : public chaine
{
private:
   int Type ;
   float Val ;
public:
   chaine_T(); // constructeurs
   chaine_T(char *);
   chaine_T(chaine_T &ch); // constructeur par recopie
   void affiche();
};
```

puis :

```
//constructeur par recopie
 chaine_T::chaine_T(chaine_T &ch) : chaine(ch)
{  // il appelle le constructeur par recopie de chaine
   Type=0; Val=0;
}

 void main()
{
   chaine_T b("Coucou ");
   chaine_T c = b;
   cout<<"b: "; b.affiche();
   cout<<"c: "; c.affiche();
   getch();
}
```

Exercice XV-10 : Seules les modifications ont été listées

```
void chaine_T::calcul()
{
   Val = atof(adr);     // possible car donnée "protected"
   if(Val!=0)Type = 1;
}
```

puis :

```
void main()
{
   chaine_T b("Coucou "),c("123"), d("45.9"), e("XFLR6");
   b.calcul(); c.calcul(); d.calcul(); e.calcul();
   b.affiche();c.affiche();d.affiche();e.affiche();
   getch();
}
```

Exercice XV-11 : Seules les modifications ont été listées

```
void main()
{
   vecteur u;
   vecteur3 v(7,8,9);
   u.affiche();
   v.affiche();
   u=v;
   u.affiche();
   getch();
}
```