IX Types complexes

Les structures

Elles permettent de représenter plusieurs variables contiguës de type différent que l'on peut adresser directement.

Exemple :

```
Struct  Individu
{
        char nom[20] ;
        char prenom[20] ;
        char sexe ;
        int age ;
} ;
```

Pour définir une variable de ce type il suffit de faire :

```
Individu i ;      ⇒ Allocation mémoire
i.nom ;
i.prenom ;
i.sexe ;
i.age ;
```

Pour accéder au différents éléments de la structure on passe par l'opérateur **.**

Exemple :

```
i.age = 30 ;     // affecte la valeur 30 à la variable age de
cout<<i.sexe ;          // Affiche le sexe de
```

Remarque :

- On peut initialiser une structure dés sa déclaration

  ```
  Individu i = { « Durand » , « Paul » , 'M' , 30 } ;
  ```

- La variable de type structure peuvent être affectées, transmises comme argument de fonctions et renvoyées comme résultat d'une fonction.

Exemple :

```
Individu i ;
Individu changement( Individu j )
{
        Indvidu Aux = i ;
        i = j ;
        return Aux ;
}

Individu ancien ;
ancien = Changement ( nouveau ) ;
```

Les unions

Une union est une structure dans laquelle tous les membres sont alloués à la même adresse

Exemple :

```
union Valeur
{
        char c[8] ;
        int v ;
} ;
```

Valeur b ; ⇒ allocation mémoire octets

Exemple :

```
Individu
{
        char nom[20] ;
        char prenom[20] ;
        char statut ;
        char sexe ;
        Valeur v ;
} ;
```

Individu i ;
On peut parler de i.v.c et de .v.j selon le cas

Union anonyme

```
struct Individu
{
        char nom[20] ;
        char statut ;
        union
        {
                char c[8] ;
                in j ;
        } ;
} ;
```

Les énumérations

Type dans lequel il est possible de stocker un ensemble de valeur spécifiés par l'utilisateur.

Exemple :

```
enum   Couleurs {rouge , vert, …. } ;
```

Couleurs c ;
c = rouge ;
Couleurs d ;
d = c ;
Intérêt :
- Lisibilité
- Contrôle

Inconvénients :
- C++ va associer un entier à chaque identificateur (0 au premier, 1 au second, etc ..)

enum CouleursBis { rouge = 10 ; vert = 20 ; … }
enum { printemps, été, automne , hiver } ; // Enumération directe

Exemple récapitulati :

```
struct Date
{
        int j ;
        Mois m ;
        int a ;
} ;
enum Mois { janvier , février ,…. }

struct SécuritéSocial
{
        Sexe s ;
        Date naissance ;
        int département ;
} ;
enum Sexe {masculin , f éminin } ;

struct Individu
{
        char nom[20] ;
        SécuritéSociale SS ;
        Cotisation c[10] ;
        Statut s ;
        Valeur v ;
} ;
enum Statut {vivant , décédé } ;

struct cotisation
{
        int numéro ;
        float montant ;
} ;

union Valeur
{
        Date décés ;

        int age ;
} ;
```

Les classes :

Ce sont les types de données les plus puissants et les plus complets pouvant être définit par le programmeur.
Elles permettent de définir des types abstraits de données enrichissant fortement le langage.
Par rapport aux structures les classes introduisent deux concepts supplémentaires, extrêmemen puissants :

- Les données membres sont encapsulées au sein de la classe (non accessible directement de l'extérieur)
- les fonctions peuvent être membre de la classe (elles agis sent uniquement sur les variables de ce type)

La déclaration d'une classe est voisine de celle d'une structure, il suffit de :

- Remplacer struct par class
- préciser quels sont les membres publique, privés en utilisant « public » ou « privés »
- Déclarer d'éventuelles fonctions membres rattachés à la classe.

Exemple :

```
class Individu
{
        private :                   // facultatif car par défau
                char sexe ;
                int age ;
        public :
                void initialise ( char , int     ;
                void affiche( ) ;
} ;
```

1. Comment est transmise la variable sur laquelle s'appliqueront les fonctions membres ?
2. Seules les déclarations des fonctions apparaissent (en -têtes) il faut définir les fonctions.

```
Void Individu ::Afficher( )
{
        cout<<sexe<< « de »<<age<< « ans » ;
}

void Indivi     ::Initialise( char c, int x )
{
        sexe = c ; age =    ;
}
```

Initialisation des fonctions membres d'une classe :

Individu i, j, k ; // Introduit 3 variables de type individu

L'appel d'une fonction membre est fait d'une manière semblable à l'accès d'une variable membre c.a.d par l'opérateur « . ».

i.Affiche() ;
j.Individu ('F', 30) ;

- Donc age = x ; de « Individu ::Initialise » placera dans le champ age la variable j la valeur reçue par x (30)

-Encapsulation des membres :

Par exemple dans le main( )
Si vous faites : i.age = 30 ; // Erreur de compilation
Sauf si age est une variable membre publique

- Une classe est une seule entité qui regroupe :
  - Une collection de données de type différent éventuellement, les détails de l'implémentation sont en général caché
  - Une collection de fonctions (en général accessible de l'extérieur) qui sert à manipuler les données membres.

Autres exemple

Implémentation du concept de « Date »

```
Classe Date
{
        int jour, mois, annee ;

        public :
                void init( int, int,int   ;
                int lire_jours() const ;
                int lire_mois() const ;
                int lire_annee() cons  ;
                void modifier_jours( int    ;
                void ajouter_mois( int    ;
                void afficher_annee() cons  ;
} ;

function_Name() const          // Fonctions qui ne modifie pas les variables de m  embres

void Date ::Init( int j, int mois , int annee )
{
        jour = j ;
        mois = m ;
        annee = a ;
}
```

```
int Date ::lire_jour( ) cons  ;
{
        return jour ;
}

void Date ::modifier_jours( int j )
{
        jour = j ;
}

void Date ::Afficher_annee( ) const ;
{
        cout<<annee ;
}

void Date ::Ajouter_mois( int m )
{
        mois = mois + m ;
}
```

NB :

- Les petites fonctions peuvent être définies à l'intérieur des classes (en même temps que la déclaration)
- L'utilisation des membres privés est interdit aux fonctions non membres de la classe

```
Void violation_dacces( Date &d )
{
        d.mois += 100 ;         // Erreur de compilation
}
```

il faut faire

```
Void violation_dacces( Date &d )
{
        d.ajouter.mois ( 100 )  ;
}
```

- Utilisation de la classe Date :

```
Date    d1, d2 ;

d1.init( 19, 11 , 78 ) ;
d2 = d1 ;
if( d2.lire_mois() < 5 )
        d2.ajouter_mois( 7 ) ;

d1.afficher_annee( ) ;
```

Pointeur :

```
int T[ N ] , i , Max ;
Max = T[ 0 ] ;

for( i = 1 ; i < N ; i++ )
        if( T[ i ] > Max )
                Max = T[ i ] ;

ind = 0 ;

for( i = 1 ; i < N ; i++ )
        if( T[ i ] > T[ ind ])
                ind = i ;
```

Variables pointeurs : Emplacement en mémoire localisation

Variables pointées : Contenu, Valeur

Déclaration d'un pointeu :

Type * Ident ;

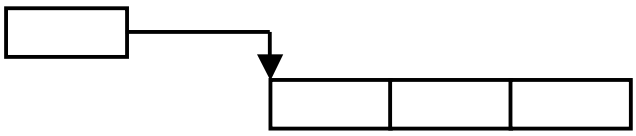
*Exemple : int * Ad ; // Réservation mémoire pour type entier*

Réservation mémoire :

Ident = new Type

Exemple : Ad = new Type ;

Ad



Utilisation :

* Ident = x ;

*Exemple : * Ad = 5 ;*

Libération mémoir :

delete(Ident) ;

Exemple : delete(Ad) ;

Tableaux : Arithmétique des pointeurs :

Ident[n] = x ;

⇐⇒ *(Ident + n) = x ;

* (p + i) ⇐⇒ p[i]

Zone mémoire qui est à l'adresse p + (I * Taille élément pointé)

Déclaration d'un tableau de taille variable :

```
int * Tab ;
Tab = new int[ n ] ;

delete [ ] Tab ;

int * p , * q ;
```

p – q → renvoi un nombre d'élément du type int entre les deux pointeurs

Attention :

```
int * p1 ;
int * p2 ;

p1 = new int ;
p2 = new int ;

*p1 = 5
*p2 = p1 ;
…
p2 = p1 ;
…
delete p1 ;
…
delete p2 ;
```

→ Erreur système

Les constructeurs

C++ propose un mécanisme spécifique pour initialiser les variables de type classe.

Le constructeur est une fonction membre particulière dont l'o bjectif explicite est d'initialiser les variables.

Il est clairement identifiable par le fait qu'il possède un nom identique à celui de la classe.

Exemple :

```
Class Date
{
        int jour, mois, annee ;

        public :
        Date( int, int, int ) ;          // Remplace   nit
        …
}
```

```
void Date ::Date( int j , int m , int a )
{
        jour = j ;
        mois = m ;
        annee = a ;
}
```

```
Date d1( 19, 11, 78 ) ;          // Crée d1 avec 3 champs initialisé

Date d1 ;        // Erreur de compilation
```

Il est permis de fournir plusieurs mode d'initialisation (➔ Définir plusieurs constructeurs)

Exemple :

```
Class Date
{
        int jour, mois, annee ;

        public :
        Date( ) ;
        Date( int ) ;
        Date( int, int, int ) ;          // Remplace init
        …
}
```

```
void Date ::Date()
{
        jour = mois = annee = 1 ;
}

void Date ::Date( int x )
{
        jour = mois = annee = x ;
}

void Date ::Date( int j , int m , int a )
{
        jour = j ;
        mois = m ;
        annee = a ;
}

Date d1 ;
Date d2( 10 ) ;
Date d3 ( 19 , 11 , 78 ) ;
```

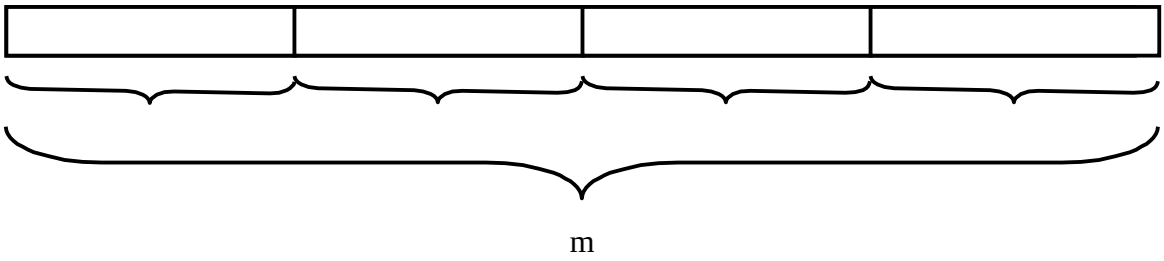
Remarque :

Les arguments des fonctions membres d'une clas se peuvent être du type de cette classe Supposons que nous souhaitions examiner la coïncidence de deux dates.

```
Class Date ( )
{
        bool Coincide ( Date ) ;
}

bool Date::Coincide( Date d )
{
        return ( d.jour == jour ) && ( d.mois  == mois ) && ( d.annee == annee ) ;
}

d1.Coincide( d2 ) ;
```

Tableaux à deux dimensions :

int T[m][n] ; ⇐⇒ int ** T ;

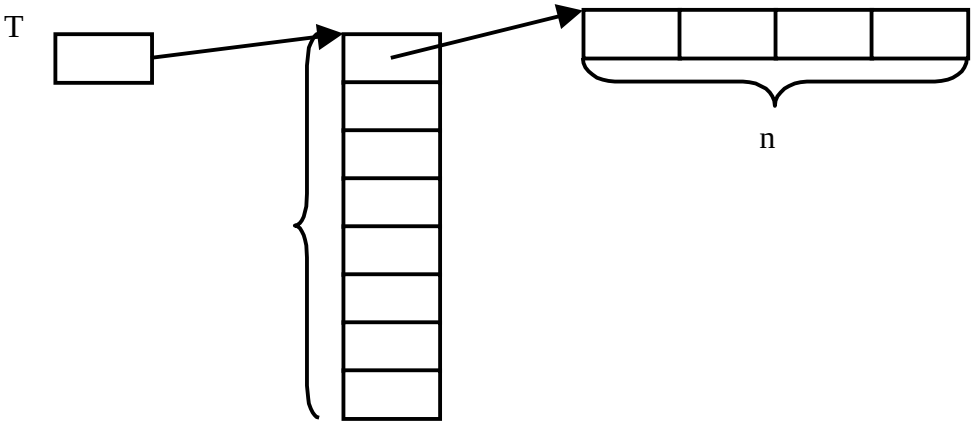


Fonction Toto (int ** T ou int T[][])
car T[x][y] = 0 ; pointe sur la mauvaise case
il faut faire fonction Toto - int T[][10]

Le compilateur connaît la taille des éléments du tableaux

Deuxième méthode :

Ce que je veux int ** T



T = new int *[5] ;

T[0] = new int[10] ;
…
T[5] = new int [10] ;

maintenant je peux utiliser la foncti
Toto(int T[][])
{
T[x][y] = 0 ;
}

Comment avoir une seule grand zone ?

```
T = new int *[5] ;

T[0] = new int[ 5 * 10 ] : ⇐⇒ int [5][10] ;
T[1] = T[0] + 10 ;
T[2] = T[1] + 10 ;
T[3] = T[2] + 10 ;
T[4] = T[3] + 10 ;
```

Le cas des classes :

```
Class Complexe
{
        private :
                double re, im ;
        public :
                Complexe()
                {
                        re = 0 ; im = 0 ;
                }

                Complexe( double r )
                {
                        re = r ; im = 0 ;
                }

                Complexe( double r, double i )
                {
                        re = r ; im = i ;
                }

                ~Complexe( ) ;
}
```

```
Complexe( double r = 0 , double i = 0 )
{
        re = r ;
        im = i ;
}
```

Exemples :

```
Complexe z ;
Complexe z = Complexe( 3, 4 ) ;
Complexe * z ;
z = new Complexe ;
z = new Complexe( 2, 3 ) ;

delete z ;
```

- Y a t'il un destructeur ?
- Appel du destructeur par défau

```
Complexe * z ;
z = new Complexe[ 5 ] ;
delete  []z ;
```

Class avec allocation dynamique :

```
Class MaxC
{
        Private :
                int nbLignes ;
                int *TabMax ;
        Public :
                MaxC( int = 10 ) ;
                Range( int, int = 0 ) ;
                int Lit( int ) ;
                ~MaxC( ) ;
                { delete []TabMax ; }
                MaxC( MaxC ) ;
```

Constructeur par recopie

```
}

MaxC ::MaxC( int n )
{
        nbLignes = n ;
        TabMax = new int[ n ] ;
}

MaxC::Range( int Ind, int Val )
{
        if( Ind >= nbLignes )
        …
        else TabMax[ Ind ] = Val ;
}

int MaxC::Lit( int Ind )
{
        if( Ind >= nbLignes )
        …
        else return tabMax[Ind] ;
}
```

dans le main() :

```
MaxC T = MaxC = 30 ;        ⇐⇒MaxC T( 30 ) ;
T.Range( 5, 100 ) ;
cout<< T.List(5) ;
```

Exemple :

```
Toto( MaxC Tb )
{
}

Toto( T )
```

Problème :

- Lorsque l'on appelle une fonction avec un paramètre par valeur il y a copie.
- Si l'on crée dans la fonction un objet MaxC et si on l'initie avec Tb les données (TabMax) peuvent être les mêmes.

Il faut donc créer un constructeur par recopie (voir la déclaration de la class)

```
MaxC ::MaxC( MaxC &T )
{
        TabMax = new int [ T.NbLignes ]
        nbLignes = T.nbLignes ;
        for( int i = 0 ; i < nbLignes ; i++)
                TabMax[ i ] = T.TabMax[ i ]
}
```

Le constructeur par recopie est utilisé dans :

- Initialisation de variable (MaxC Tb = Ta)
- Passe une variable « class » en argument d'une foncti
- Valeur de retour : Variable de type « class »

Allocation dynamique :

Class

- Constructeur → New → MaxC (int)
- Destructeur → delete → ~MaxC ( )
- Constructeur par recopie → MaxC(const MaxC &)

Affectation :

```
MaxC Ta = MaxC ( 30 ) ;
MaxC Tb = MaxC ( 20 ) ;

Ta.Range( 20, 100 ) ;
Tb = Ta ;       // Seulement copie de surface ⇒ on perd e tableau pointé par Tb
```

On doit redéfinir l'opérateur d'affectation
On doit « expliquer » au compilateur ce que l'on veut

Su -définition des opérateurs :

Pour tous les opérateurs standard sau **. → :: ?: sizeof**

- Soit par une fonction membre
- Soit par une fonction ami de la class (Friend)

On doit ajouter dans la classe :

Public :
MaxC operator = (const MaxC &)

Définition :

```
MaxC & MaxC:: operator = ( const MaxC & T )
{

        if( * This == T ) return ( * this ) ;
        delete [ ] TabMax ;
        nblignes = T.nblignes ;
        TabMax = new int[ nblignes ] ;
        for( int I = 0 ; I < nblignes ; I++ )
                TabMax[ I ] = T.TabMax[ I ] ;

        return ( * this ) ;
}
```

Contenu du pointeur This

This pointe sur la variable sur laquelle je suis en train de travailler

Pour éviter la recopie en renvoi un objet MaxC par référence

En utilisant les fonctions amies :

Dans la déclaration de la classe :

Public :

friend MaxC & operator = (MaxC &, const MaxC &)

la fonction définit après aura droit d'accès à toutes les variables de classe

```
MaxC & operator = ( MaxC  TG , const MaxC & TD )
{
        if( TG == TD ) return Tg ;
        delete [ ] TabMax ;
        TG.nblignes = TD.nblignes ;
        TG.TabMax = new int[ TG.nblignes ] ;
        for( int I = 0 ; I < nblignes ; I++ )
                TG.TabMax[ I ] = TD.TabMax[ I ] ;

        return ( TG ) ;
}
```

Il est possible de redéfinir les opérateurs d'égalité par exemple pour tester l'égalité de chaque case de TabMax dans notre étude.

On optimise notre procédure pour gérer le cas « TA = TA » ⇒ Voir au dessus en ver

Pour l'instant on est obligé de faire:

```
TA.Lit( 5 ) ;
```

J'aimerais faire : `TA[ 5 ] ;`

dans la classe on ajou e :

```
Public :
    int & operator[ ] ( int )
```

Déclaration :

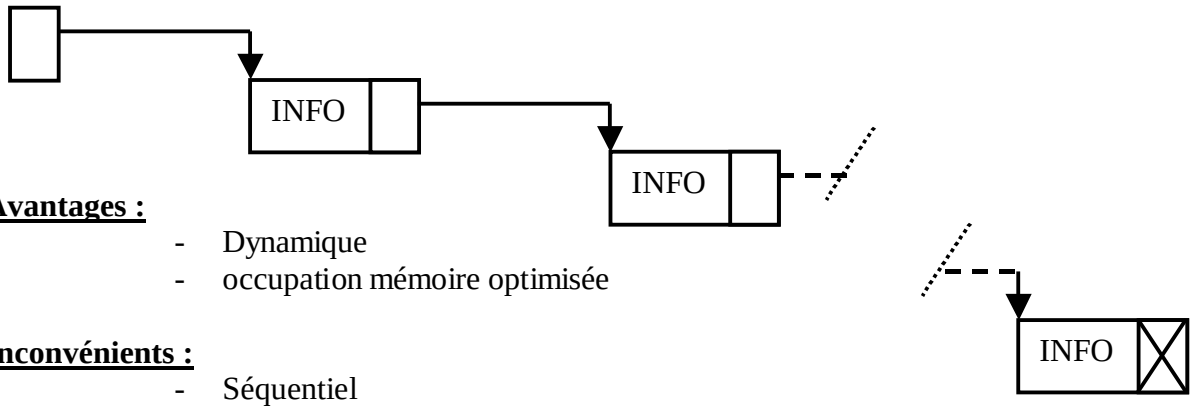
```
int & MaxC :: operator [ ] ( int ind )
{
    if( ind >=nblignes ) … return ( ) ;

    return TabMax[ Ind ] ;
}
```

Structure de données :

On souhaite créer une structure de données :

- Dynamique
- Sans taille fixe

: Liste chaînée



Avantages :

- Dynamique
- occupation mémoire optimisée

Inconvénients :

- Séquentiel

Liste vide : Un pointeur qui vaut NULL représenté 

Accès à la liste :

Pointeur qui permet d'accéder à la chaîne des données

Définition des classes :

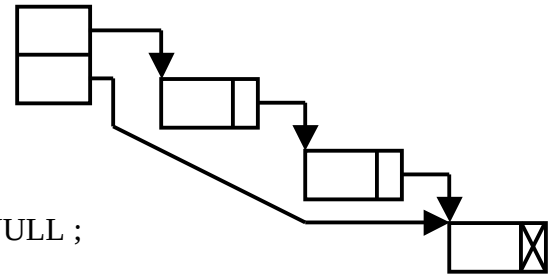
```
Class Maillon
{
        Private :
                TYPE Info ;
                Maillon * Sui  ;
                friend class Liste ;

        Public :
                Maillon( Const TYPE & )  ;
}
```

La classe liste est une classe amie : Toutes les fonctions de la classe liste pourront accéder aux fonctions et propriétés de Maillon

```
Maillon ::Maillon( const TYPE & Inf
{
        Info = Inf ;
        Suiv = NULL ;
}
```

Remarque :

```
Maillon m1, m  ;

m1 = m2 ;      // Cette écriture fonctionne car il n'y a pas d'allocation
               mémoire dans la Classe Maillon.
```



```
Class Liste
{
        Private :
                Maillon * Debu  ;
                Maillon * Fi  ;
        Public :
                Liste ( )
                {
                        Debut = Fin = NULL ;
                }
                ~Liste( ) ;
                int EstVide( )
                {
                        return( debut = = NULL ) ;
                }
                void FaireEntrer( Const & TYPE )
                TYPE FaireSortir( ) ;
                Void Affiche( ) ;
}
```

Rappel surdéfinition d'opérateurs :

Utilisation :

```
C = A.Addition ( B )  ⇐⇒ C = A.operator+( B) ;
                      ⇐⇒ C = A + B ;

Complexe Addition2( Complexe OG, Complexe OD )
{
        Complexe Res ;
        Res.IM = OG.IM + OD.IM ;
        Res.RE = OG.RE + OD.RE ;
        return Res ;
}

C = Addition2( A, B ) ;
```

Opérateur binaire :

OG ⊕ OD ⊕ opérateur binaire

→ Opérateur – fonction membre
OG.operator⊕ (OD)

→ Opérateur – fonction amie
operator ⊕ (OG, OD)

Opérateur unaire :

→ Opérateur – fonction membre
OP.operator ⊕ ( )

→ Opérateur – fonction amie
operator ⊕ (OP)

Fonction amie de la class e Toto ⇒ fonctions qui accèdent aux variables privées de Toto

Class Titi amie de la class Toto

```
Class Toto
{
        friend class Titi ;
}
```

Toutes les fonctions membres de Titi sont “amies” de la classe Tot

Suite Liste chaînées :

Lorsque l’on a fait des fonctions avec l’objet en paramètre, il faut créer un constructeur par recopie .

```
void List   ::FaireEntrer( int inf )
{
        Maillon * NM ;
        Nm = new maillon( inf
        if( debut == NULL )  // Liste vide
        {
                debut = NM ;
                fin = NM ;
        }
        else
        {
                (*fin).suiv = NM        // fin→suiv = NM ;
                fin = NM ;
        }
}
```

```
int Liste ::FaireSortir( )
{
        Maillon * p, int Res ;
        if( debut == NULL )
        {
                cout<<"Problème"<<endl ;
                return –1 ;
        }
        res = debut→info ;
        p = debut ;
        debut = debut→suiv ;
        delete  p ;
        if ( debut == NULL ) fin = NULL ;
        return Res ;
}

Liste::Affiche()
{
        Maillon *P = debut

        while( p )
        {
                cout<<p→info<<  « _ » ;
                p = p → suiv ;
        }
        cout<<end  ;
}

Liste ::~Liste()
{
        int Tmp ;
        while( ! estVide() )
                Tmp = FaireSortir()  ;
}
```

On ajoute le constructeur par recopie :

```
public :
        Liste( Liste & )          // Par référence car sinon il y aura appel infini.

Liste ::Liste( Liste & L )
{
        debut = NULL ;
        fin = NULL ;

        Maillon * P = L.debu  ;
        while( P )
        {
                FaireEntrer( p→info ) ;
                p = p→suiv ;
        }
}
```

```
int main()
{
        Liste L ;

        for( int i = 5 ; i < 5 ; i++ )      L.FaireEntrer( i ) ;
        L.Affiche() ;
        Toto( L ) ;      // Action : Appel du constructeur par recopie
                         // on crée une liste chaînée identique à la liste L.

        L.Affiche() ;

}

void Toto( Liste Tmp )         //  Appel du constructeur par recopie
{
        while( !Tmp.ListeVide() )
                cout<<Tmp.FaireSortir() <<"_" ;
        cout<<end  ;
}       // Appel du destructeur de Tmp

void Toto2( Liste &Tmp )
{
        while( !Tmp.ListeVide() )
                cout<<Tmp.FaireSortir() <<"_" ;
        cout<<end  ;
}
```

Avec Toto2, Tmp pointe sur la liste passée en paramètre. Le constructeur par recopie n'est pas exécuté.