Université de Monastir

Cours: Programmation C++

Chapitre 4 : Les types de constructeurs en C++

Réalisé par:
Dr. Sakka Rouis Taoufik

https://github.com/srtaoufik/coursCpp

1

I. Introduction

Il existe en C++ trois types de constructeurs qui sont un peu particuliers. Leur particularité provient du fait qu'ils peuvent être appelés automatiquement d'une manière parfois cachée par le compilateur. Ces constructeurs sont :

- le constructeur par défaut,
- le constructeur de recopie,
- le constructeur de transtypage.

2

II. Constructeur par défaut

Définition: On appelle constructeur par défaut tout constructeur qui peut être appelé sans lui passer d'arguments.

Exemple :

```
#include <iostream.h>
#include <stdlib.h>
#include <iomanip.h>
class Time {
    int Hour;
    int Minute;
    int Second;
public :
   Time() {
      Hour = 12; Minute = 0; Second = 0;
   }
   Time(int H, int M, int S){
      Hour = H; Minute = M; Second = S;
   }
void Affiche() {
  cout<<"L'heure est : ";
  cout<<setw(2)<<setfill('0')<<Hour<<':'
 <<setw(2)<<setfill('0')<<Minute<<':'
 <<setw(2)<<setfill('0')<<Second<<endl;
 }
};
…
Time T;
// T contient
//Hour =12, Minute=0, Second=0;
```


III. Constructeur de recopie (par copie)

Définition

• Un constructeur de recopie est un constructeur qui permet de créer un objet et de l'initialiser avec le contenu d'un autre objet de la même classe.

• L'utilisation des constructeurs de recopie revêt d'une grande importance surtout avec les objets qui possèdent des attributs **dynamiques**.

III. Constructeur de recopie (par copie)

Syntaxe simplifiée :

NomClasse(<const> NomClasse){....}

Exemple :

```
class Time { ...
public :
Time(const Time& T){...} // Constructeur de recopie
...
};
```

Rq. Un constructeur de recopie peut avoir d'autres paramètres supplémentaires. Ces derniers doivent obligatoirement avoir des valeurs par défaut.

• Un constructeur de recopie est toujours appelé avec un seul argument (le premier argument). L'appel des arguments supplémentaires éventuels se fait d'une manière implicite.

5

Chapitre 4 : Les types de constructeurs en C++

III. Constructeur de recopie (par copie)

Pourquoi un premier paramètre de type référence ?

La question à laquelle répond cette remarque est la suivante : pourquoi passe-t-on au constructeur de recopie une référence à l'objet à copier et non une copie de cet objet ? En d'autres mots, pourquoi utilise-t-on un passage par référence là où le passage par valeur semble suffisant?

6

III. Constructeur de recopie (par copie)

Justification du passage par référence :

Considérons la classe *Time* définie précédemment. On ajoute à cette classe un constructeur de recopie qui prend un paramètre *Tm* de type *Time*.

```
class Time { ...
public :
Time(){...}
Time(Time Tm){...} // On suppose que c'est possible
...
};
```


III. Constructeur de recopie (par copie)

Justification du passage par référence :

Soient *f* une fonction qui prend un paramètre *Tf* de type *Time* et *X* une instance de *Time* :

```
void f (Time Tf){...}
Time X;
```

L'appel de *f* avec le paramètre effectif *X* engendre la copie de *X* dans le paramètre formel *Tf*. Cette copie est réalisée avec le constructeur de recopie de *Time*.

```
f(X) // Tf=X ⇔ Tf(X)
//Appel au constructeur de recopie de Time et par suite :
f(X) ⇔ f(Time(X)) // instanciation du paramètre formel à l'aide du
// constructeur de recopie.
```

III. Constructeur de recopie (par copie)

Justification du passage par référence :

Le passage de *X* au constructeur *Time* se fait par valeur. Ce constructeur va travailler donc sur une copie de *X*. Cette copie sera créée par un autre appel au constructeur de recopie. Ce dernier travaillant lui-même sur une copie de *X*, il va faire lui aussi un appel à lui-même. Cet appel récursif du constructeur de recopie va se poursuivre à l'infini.

Time(X) // Tm=X ⇔ Tm(X)

et par suite Time(X) devient Time(Time(X))

et f(X) devient f(Time(Time(Time ……)))

III. Constructeur de recopie (par copie)

Justification du passage par référence :

Pour résoudre ce problème il suffit d'éviter l'appel implicite du constructeur de recopie qui est engendré par le passage par valeur et de remplacer ce dernier par un passage par référence.

Ainsi avec Time(Time& Tm), f(X)se réduit seulement à f(Time(X))

car l'affectation du paramètre effectif au paramètre formel (Time& Tm=Time(X))

représente une copie de références et non une copie d'objets.

III. Constructeur de recopie (par copie)

Pourquoi la recommandation const

L'utilisation d'une référence constante comme premier paramètre dans les constructeurs de recopie est une recommandation.

L'intérêt de cette recommandation réside dans la protection de l'objet passé comme argument contre toute modification qui peut survenir par erreur dans le constructeur surtout que son passage se fait par référence (on ne travaille pas sur une copie de l'objet mais directement sur ce dernier).

III. Constructeur de recopie (par copie)

Rq. Constructeur de recopie par défaut

En cas d'absence d'un constructeur de recopie explicitement défini, le compilateur en génère automatiquement un par défaut. Ce constructeur effectue une copie champ par champ du contenu des deux objets.

Limites du constructeur de recopie par défaut

Exemple 1 :

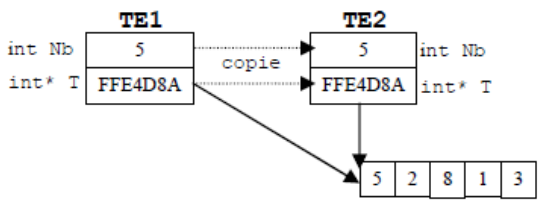
```
class TabEntiers {
public :
   int Nb;
   int* T;
  TabEntiers(int Ni){...}
  TabEntiers(int* Ti,int Ni){...}
  void Saisir(){...}
  void Afficher(){...}
};
```

```
void main() {
  int Ti[4]={5,2,8,1,3};
 TabEntiers TE1(Ti,4);
 TabEntiers TE2(TE1);
 TE2.Afficher();
}
```

III. Constructeur de recopie (par copie)

La classe *TabEntiers* ne dispose pas d'un constructeur de recopie explicitement définie. Par suite c'est le constructeur de recopie par défaut qui est utilisé dans l'instruction : TabEntiers **TE2**(TE1);

➔ Ce constructeur effectue une copie champ par champ du contenu de *TE1* dans *TE2*. De ce fait, pour le membre dynamique *T*, c'est une copie d'adresses qui est effectuée entre *TE1.T* et *TE2.T*.

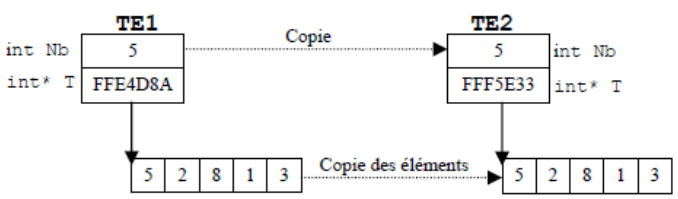


13

III. Constructeur de recopie (par copie)

Solution: il faut définir explicitement le constructeur de recopie de la manière suivante :

```
TabEntiers::TabEntiers(const TabEntiers& Tc) {
   Nb=Tc.Nb;
  T=new int [Nb];
  for(int k=0;k<Nb;k++)
    T[k]=Tc.T[k];
}
```



14

III. Constructeur de recopie (par copie)

Lieux d'appel d'un constructeur de recopie d'objet

Un constructeur de recopie est appelé lors :

• de la création d'un objet à l'aide d'un autre objet de la même classe. **Exemple:**

Point P1; // constructeur par défaut

Point P2=P1; Point P3=Point(P1); // constructeur par recopie

• de la transmission de la valeur d'un paramètre effectif (de type objet) à un paramètre formel suite à l'appel d'une fonction.

• du retour d'une valeur de type objet par une fonction. En effet la fonction crée un objet temporaire sans nom à l'aide du constructeur de recopie et ce, à partir de l'objet passé à *return* et c'est cet objet temporaire qui est affecté à la variable recevant la valeur de retour de la fonction.

15

Chapitre 4 : Les types de constructeurs en C++

III. Constructeur de recopie (par copie)

Lieux d'appel d'un constructeur de recopie d'objet

```
class NoData {
public :
  NoData(){cout<<"constructeur par défaut";}
  NoData(const NoData& ND){cout<<"constructeur de recopie";}
};
void f1(NoData D){ }
void f2(NoData& D){ }
NoData f3() {  NoData D;  ...return D;    }

void main() {
 NoData X;// Constructeur par défaut
 f1(X); // Constructeur de recopie (en raison du passage par valeur)
 f2(X); // Rien n'est affiché
 X=f3(); // Constructeur par défaut (pour l'objet local D)
         // Constructeur de recopie (pour l'objet renvoyé)
}
```

16

III. Constructeur de recopie (par copie)

Lieux d'appel d'un constructeur de recopie d'objet

Question :

Le prototype suivant pour *f3* : NoData& **f3**() permet-il d'aboutir à un résultat correct même en l'absence d'un constructeur de recopie explicite ?

Réponse :

Le constructeur de recopie n'est pas appelé mais on obtient une référence sur un objet qui n'existe plus en mémoire puisqu'il est local à la fonction.

IV. Les constructeurs de transtypage

Un constructeur de transtypage, appelé également constructeur de conversion, est un constructeur qui permet de créer et d'initialiser un objet d'un type T1 à partir d'une donnée (objet ou non) d'un autre type T2 différent de T1.

Caractéristiques du constructeur de transtypage

• Un constructeur de transtypage ne peut être appelé qu'avec un seul paramètre : comme c'est le cas du constructeur de recopie, les paramètres qui suivraient éventuellement le premier paramètre formel doivent avoir obligatoirement des valeurs par défaut.

• Le premier paramètre du constructeur de transtypage contient la donnée à convertir. Ce paramètre peut être de n'importe quel type (préfini, struct, class, etc.) sauf le type de la classe du constructeur.

IV. Les constructeurs de transtypage

Syntaxe générale :

NomClasse(<const> Type1 Param1<,Type2 Param2=DefVal2,..,Type_n Param_n=DefVa_n>);

Exemple :

```
class Time {
  int Hour;
  int Minute;
  int Second;
public :
  Time(){Hour = 12; Minute = 0; Second = 0;}
  Time(int H,int M, int S){Hour = H; Minute = M; Second = S;}
  Time(const char* str) {
    cout<<"Appel du constructeur de transtypage";
    Hour = 10*(*str-'0') + (*(str+1)-'0');
    Minute = 10*(*(str+3)-'0') + (*(str+4)-'0');
    Second = 10*(*(str+6)-'0') + (*(str+7)-'0');
   }
  void Afficher(){...}
};
```


IV. Les constructeurs de transtypage

Syntaxe générale :

Le constructeur Time(const char* str) peut être considéré comme étant un constructeur de transtypage puisqu'il répond aux deux caractéristiques de ce type de constructeur à savoir un appel à l'aide d'un seul paramètre en plus le paramètre possède un type différent de Time(const char*).

IV. Les constructeurs de transtypage

Champs d'utilisation des constructeurs de transtypage

Utilisation explicite : Un constructeur de transtypage peut être explicitement appelé par le programmeur à chaque fois que ce dernier a besoin de travailler avec un objet d'un certain type et qu'il dispose d'une donnée d'un autre type.

Exemple

Time X = Time("18:30:12"); ou également Time X("18:30:12");

Utilisation implicite : Un constructeur de transtypage d'un type T1 (objet ou non) vers un type objet T2 différent de T1 est implicitement appelé par le compilateur :

• Lors de la transmission d'un paramètre effectif de type T1 à une fonction ayant un paramètre formel de type T2.

• Dans une opération d'affectation ayant un membre de droite (objet, variable ou expression) de type T1 et un membre de gauche de type objet T2.

X. Les listes d'initialisation des constructeurs

Une liste d'initialisation de constructeurs représente une alternative syntaxique pour l'initialisation des attributs d'une classe.

➔ Au lieu de faire cette initialisation dans le corps du constructeur, il devient possible grâce à cette alternative de la faire tout juste devant l'entête du constructeur à l'aide de la syntaxe suivante :

NomClasse(arguments) : A1(V1), A2(V2), …, An(Vn)

Où :

• NomClasse(arguments) représente un appel au constructeur,

• Ai : représente les attributs de NomClasse à initialiser

• Vi : valeur initiale affectée à l'attribut Ai.

X. Les listes d'initialisation des constructeurs

Exemple 1 :

```
// Définition du constructeur de la classe Time
Time():Hour(12),Minute(30),Second(45) {   }
```

Exemple 2 :

```
// Définition du constructeur de la classe TabEntiers
TabEntiers (int N, int* Tab) : Nb(N),T(new int[N]) {
   for(int i = 0;i<Nb;i++)
     T[i] = Tab[i];
}
```


X. Les listes d'initialisation des constructeurs

Initialisation des attributs constants et références

Pour les attributs constants et références, les listes d'initialisation ne constituent pas une simple alternative d'écriture de code mais représentent plutôt le seul moyen permettant de leurs affecter leurs valeurs initiales.

Considérons la classe *ConstRef* qui comporte les trois attributs suivants :

x (entier), *r* (référence à un entier) et *y* (constante entière).

```
class ConstRef {
 int x;
int& r = x ; // ERROR
const int y = 5; // ERROR
… … …
};
```

X. Les listes d'initialisation des constructeurs

Initialisation des attributs constants et références

Le fait que *r* et *y* soient respectivement une référence et une constante rend leurs initialisations obligatoires. Toutefois leurs initialisations telles que réalisées dans la classe *ConstRef* sont interdites. En effet, la définition d'une classe constitue une définition de type sans aucune instanciation en mémoire.

➔ Au moment de la définition de *ConstRef* aucun attribut n'a une existence physique en mémoire. Par suite, initialiser r avec la référence d'une variable *x* qui n'existe pas ou également affecter 5 à une constante *y* qui n'existe pas en mémoire représentent des opérations interdites.

X. Les listes d'initialisation des constructeurs

Initialisation des attributs constants et références

L'utilisation d'un constructeur défini comme suit est également interdite :

```
ConstRef::ConstRef( ){
  x =3; // OK
  r = x; // ERROR
  y = 5; // ERROR
}
```

En effet, l'instruction r=x; représente une affectation de la valeur 3 à la variable référencée par *r*. Or *r* ne référence aucune variable car elle n'a pas été initialisée. Par ailleurs, l'instruction y=5; représente une affectation de la valeur 5 à la constante *y* et non une initialisation. Or les opérations d'affectations ne peuvent pas être appliquées aux constantes.

X. Les listes d'initialisation des constructeurs

Initialisation des attributs constants et références

Le seul moyen en C++ permettant d'initialiser de tels membres consiste dans l'utilisation des listes d'initialisation.

Exemple :

```
class ConstRef {
  int x;
  int& r ;
  const int y ;
  ConstRef( ) : r(x),y(5) {
    x=3;
   }
};
```


X. Les listes d'initialisation des constructeurs

Initialisation d'objets membres

Les listes d'initialisation sont également utilisées pour initialiser les membres qui sont de type classe (membres objet) essentiellement lorsque ces derniers possèdent des constructeurs paramétrés.

L'initialisation du membre x comme suit : x(55) est possible parce qu'un constructeur ayant le prototype X (int i) est défini pour la classe X.

Exemple :

```
class X {
  int a;
public :
  X(int i) {
    a=i;
    cout<<"Constructeur de X\n";
  }
};
```

```
class Y {
  int b;
  X x;
public :
  Y():b(5),x(55) {
    cout<<"Constructeur de Y\n";
   }
};
```

X. Les listes d'initialisation des constructeurs

Initialisation d'objets membres

Rq. En tenant compte de la définition de la classe X, la définition suivante du constructeur de Y n'est pas correcte :

```
Y() {
   b=5;
   x=X(55);
   cout<<"Constructeur de Y";
}
```

En effet, l'instruction x=X(55); ne constitue pas syntaxiquement une initialisation au vrai sens du terme. Elle est plutôt considérée comme étant une première affectation à x du contenu de l'objet temporaire X(55). Cela suppose que x a été créé auparavant à l'aide d'un constructeur par défaut chose qui n'est pas possible faute de ce type de constructeur dans la définition de la classe X.

XI. Exercices d'application

Soit la classe ***Vector*** dont la spécification est donnée comme suit :

```
class Vector {
  const int Size;
  int* Elements;
public :
  Vector(int S);
  Vector(int S, int* Elts);
  Vector(const Vector& V);
  ~Vector();
  void Show();
  void Set(); // Effectue la saisie de tous les éléments du vecteur
  void Set(int Index, int Value);
  int Get(int Index);
};
```

1- Donner les définitions des méthodes de la classe ***Vector***.

2- Donner une fonction principale qui montre

XI. Exercices d'application

Soit la classe ***Matrix*** dont la spécification est donnée comme suit :

```
class Matrix  {
 const int NbLines;
 const int NbColumns;
 int** Elements;
public :
  Matrix (int L, int C);
  Matrix (int L, int C, int** Elts);
  Matrix(const Matrix& M);
  ~Matrix();
  void Set(int LineIndex, int ColumnIndex, int Value);
  int Get(int LineIndex, int ColumnIndex);
  void Show(); // Affiche le contenu de la matrice
  void Set(); // effectue la saisie de tous les éléments
};
```

1- Donner les définitions des méthodes de la classe ***Matrix***.

2- Donner une fonction principale qui montre l'instanciation de la classe ***Matrix*** et l'utilisation de ses différentes méthodes.

XI. Exercices d'application

1- Proposer une fonction ***Multiply*** externe et amie aux deux classes ***Vector*** et ***Matrix*** qui effectue le produit d'une matrice et d'un vecteur.

2- Transformer la fonction amie ***Multiply*** de façon à ce qu'elle devienne une méthode de la classe ***Matrix.*** Indiquer comment faut-il modifier l'organisation du code pour pouvoir compiler avec succès.