Cours: Programmation C++

Chapitre 3 : Les constructeurs et les destructeurs

Réalisé par:
Dr. Sakka Rouis Taoufik
https://github.com/srtaoufik/coursCpp

1

I. Initialisation d'objets

Il est possible d'attribuer des valeurs initiales aux attributs d'un objet à travers une liste de valeurs spécifiées entre deux accolades et séparées par des virgules (comme pour les tableaux en C).

Exemple :

```
class Point2D {
public:
   int x;
   int y;  };
class Point3D {
public:
   int x;
   int y;
   int z;
};
```

```
Point2D P1={5,6};
Point3D P2={5,6,9};
Point3D P3={4,,9} ;
// le deuxième attribut est initialisé à 0.
Point2D T[3] = {1,5,6,8,9,10};
Point2D T[3] = {{1,5},{6,8},{9,10}};
```

II. Limites de l'initialisation avec les listes de valeurs

- Il faut que tous les membres de l'objet à initialiser soient publics.
- Il faut que l'objet soit d'une classe de base et qu'il n'ait pas de membres virtuels.
- Certaines opérations d'initialisation sont complexes et nécessitent plusieurs étapes comme le montre l'exemple suivant

Exemple :

```
class TabEntiers {
public :
  int Nb;
  int* T;      };
```

Pour pouvoir initialiser le membre T, ce dernier doit auparavant être alloué.

II. Limites de l'initialisation avec les listes de valeurs

Solution 1 : Passage par une méthode d'initialisation

\+ Possibilité de réaliser les opérations d'allocation pour les membres dynamiques.

\+ Possibilité d'accéder aux membres privés et protégés.

```
class TabEntiers {
 int Nb;
  int* T;
public :
  void Init(int Ni) {
     Nb=Ni;
     T=new int [Nb];   }
  void Init(int* Ti, int Ni) {
     Nb=Ni;
     T=new int [Nb];
     for(int k=0;k<Nb;k++)
        T[k]=Ti[k];   }
};
```

```
void main() {
  TabEntiers TE;
  TE.init(5);
}
```

II. Limites de l'initialisation avec les listes de valeurs

Solution 1 : Passage par une méthode d'initialisation

Inconvénients :

- Pour pouvoir utiliser l'objet, il est nécessaire d'appeler à chaque fois, d'une manière explicite, la méthode Init. Cet appel explicite peut engendrer des erreurs surtout dans les programmes de grande taille où le risque d'oubli devient élevé.
- L'utilisation de Init ne peut pas être considérée comme étant une initialisation au vrai sens technique du terme car une initialisation se fait généralement dans la même instruction que la déclaration.

II. Limites de l'initialisation avec les listes de valeurs

Solution 2 : Utilisation des constructeurs du C++

Le C++ offre une solution de création et d'initialisation d'objets qui permet de remédier à tous les problèmes susmentionnés (initialisation au vrai sens du terme, appel implicite, initialisation des membres dynamiques avec allocation de mémoire,…).

Définition

Un constructeur est une méthode particulière d'une classe qui s'exécute automatiquement d'une manière implicite, lors de la création d'un objet de cette classe.

II. Limites de l'initialisation avec les listes de valeurs

Solution 2 : Utilisation des constructeurs du C++

Rôle de base: Un constructeur d'une classe assure la réservation de l'espace mémoire nécessaire à la création de tout objet de cette classe. L'espace dont on parle ici désigne l'espace de base nécessaire à la création de l'objet. Il ne comprend pas les espaces dynamiques associés aux membres dynamiques éventuels de la classe.

Pour la classe *TabEntiers* définie précédemment, l'espace de base nécessaire à la création d'un objet est égale à la somme des tailles de *Nb* et du pointeur *T*. Il ne comprend pas la taille de la zone mémoire pointée par *T*.

II. Limites de l'initialisation avec les listes de valeurs

Solution 2 : Utilisation des constructeurs du C++

Exploitation usuelle des constructeurs : Les constructeurs sont généralement exploités par les programmeurs pour réaliser les opérations suivantes :

- Initialisation des attributs de la classe (publics, privés ou protégés).
- Allocation des espaces mémoires nécessaires aux membres dynamiques de la classe.

III. Déclaration et définition d'un constructeur

- Un constructeur est une méthode de la classe.
- Un constructeur doit porter le même nom que celui de sa classe.
- Un constructeur peut admettre 0, un ou plusieurs paramètres.
- Un constructeur ne retourne aucune valeur. (le fait de ne pas mettre un type de retour devant le constructeur ne signifie pas que ce dernier retourne par défaut un entier (*int*) comme c'est le cas pour les méthodes classiques).
- Une classe peut comporter un ou plusieurs constructeurs. Dans le cas d'existence de plusieurs constructeurs, ces derniers doivent respecter les règles de surcharge des méthodes.
- Comme toute méthode un constructeur peut être défini à l'intérieur de la classe ou à l'extérieur.

III. Déclaration et définition d'un constructeur

Exemple:

```
class Time {
 int Hour;
 int Minute;
 int Second;
 public :
  Time (int H, int M, int S) {
     Hour = H;
     Minute = M;
     Second = S;
 }
```

```
void Afficher ( ){
  cout<<"L'heure est : ";
  cout<<Hour<<':'<<Minute<<':'
<<Second<<endl;
}
};
```

III. Déclaration et définition d'un constructeur

Rq.

• Les spécificateurs d'accès public, private et protected s'appliquent aux constructeurs de la même manière que pour n'importe quelle méthode de classe.

• Pour pouvoir appeler un constructeur et par suite créer un objet, ce constructeur doit être obligatoirement public.

IV. Appel d'un constructeur

Toute construction d'un objet d'une classe doit être accompagnée obligatoirement par un appel à son constructeur.

1. Cas d'un objet de type auto

La création d'un objet de type auto peut se faire à travers un appel explicite ou implicite du constructeur.

Appel explicite :

Le constructeur est appelé par son nom comme une méthode classique.

NomClasse NomObjet = NomClasse(<paramètres effectifs>);

Time T = Time (16, 30, 25);

Appel implicite :

NomClasse NomObjet(<paramètres effectifs>);

Time T (16, 30, 25) ;

IV. Appel d'un constructeur

2. Cas d'un objet dynamique

Pour la création dynamique d'objets, seul l'appel explicite du constructeur est possible.

NomClasse* ptrObjet = new NomClasse(<paramètres effectifs>);

Time* T = new Time(16,30,25);

Exemple :

```
Time T1(16,30,25);
Time* T2 = new Time(17,44,59);
T1.Affiche();
T2->Affiche();
```


V. La surcharge des constructeurs

• Il est possible de définir plusieurs constructeurs pour une même classe. Ces constructeurs sont alors dits surchargés.

• Les constructeurs d'une même classe doivent respecter les règles de surcharge des fonctions en C++. En d'autres mots, ils doivent avoir des signatures différentes.

V. La surcharge des constructeurs

```
class Time {
 int Hour;
 int Minute;
 int Second;
public :
Time(int H,int M, int S)  {
 Hour = H;
 Minute = M;
 Second = S;
}
Time() {
 Hour = 0;
 Minute =0;
 Second = 0;
}
```

```
void Saisir() {
 cout<<"Donner l'heure : "
 cin>>Hour>>Minute>>Second;
}
void Afficher() {
 cout<<"L'heure est : ";
 cout<<Hour<<':'<<Minute<<':'<<Second;
 cout<<endl;
}
};
void main() {
 Time T1;
 T1.Saisir();
 Time T2(16,15,30);
 T1.Afficher();
 T2.Afficher();
}
```

15

V. La surcharge des constructeurs

Rq1 : Si une classe dispose de plusieurs constructeurs alors au moment de la création d'un objet à partir de cette classe il y aura appel à un parmi ces constructeurs. Dans l'exemple précèdent, la création de l'objet T1 est faite à l'aide du constructeur sans arguments. Cet objet est non initialisé. Son contenu sera par la suite saisi avec la méthode Saisir. La création de l'objet T2 est faite à l'aide du constructeur paramétré. Son contenu est initialisé à 16h15mn30s.

Rq2 : Tout objet ne peut être créé qu'à travers un appel à un constructeur. Par conséquent, ce sont les constructeurs définis dans une classe qui déterminent la manière avec laquelle peuvent être instanciés les objets à partir de cette classe. A titre d'exemple, si le constructeur sans arguments Time() n'était pas défini dans la classe Time alors l'instanciation Time T1; serait syntaxiquement incorrecte (erreur de compilation).

16

V. La surcharge des constructeurs

Rq3 : Il est toujours recommandé de définir plusieurs constructeurs dans une classe et ce pour offrir aux utilisateurs de cette dernière plus de scénarios possibles et une plus grande souplesse concernant l'instanciation des objets.

VI. Les destructeurs

Un destructeur joue le rôle symétrique du constructeur. Il est automatiquement appelé pour libérer l'espace mémoire de base nécessaire à l'existence même de l'objet (espace occupé par les attributs). Cette libération est implicite. Elle est effectuée par tout destructeur. Il est généralement recommandé d'exploiter le destructeur pour faire :

- La libération d'éventuels espaces mémoires dynamiques associés aux attributs de l'objet. Contrairement à la libération de l'espace de base, cette libération ne s'effectue pas d'une manière automatique. Elle doit être faite d'une manière explicite par le programmeur.
- La fermeture d'éventuels fichiers utilisés par l'objet.
- L'enregistrement des données, etc.

VII. Déclaration et définition d'un destructeur

• Un destructeur porte le même nom que celui de la classe mais précédé d'un tilde ~ (pour le distinguer du constructeur).
• Un destructeur ne retourne aucune valeur.
• Un destructeur ne prend jamais de paramètres.
• Un destructeur est une méthode de la classe. Toutes les règles qui s'appliquent aux méthodes (portée, qualification d'accès, déclaration, définition) s'appliquent donc également au destructeur.
• Une classe ne peut disposer que d'un seul destructeur. Il est toujours non paramétré.

VII. Déclaration et définition d'un destructeur

Exemple 1 :

```
class Time {
        int Hour;
        int Minute;
        int Second;
public :
        Time (){ .... }
        Time (int H, int M, int S){ .... }
        Time (const char* str){ .... }
        ~Time(){  } // Destructeur vide
        Afficher(){ .... }
} ;
```

RQ. La classe *Time* possède des attributs de type *auto*. Donc la libération d'un objet de cette classe ne demande aucun traitement supplémentaire particulier. Par conséquent son destructeur est défini vide.

VIII. Appel d'un destructeur

1. Appel explicite :

NomObjet.~NomClasse(); Ou **PointeurObjet -> ~NomClasse();**
Cet appel reste toutefois rare d'utilisation.

2. Appel implicite :

Un destructeur est implicitement appelé lorsque la durée de vie de l'objet est écoulée. Ceci est le cas :

• Pour un objet local de la classe mémoire auto, lorsque le domaine de validité de l'objet (bloc d'instructions dans lequel il est déclaré) est quitté.

• Pour un objet global ou local de la classe mémoire "*static*" lorsque le programme se termine.

• Pour un objet dynamique créé par *new* lorsque l'opérateur *delete* est appliqué au pointeur qui le désigne

VIII. Appel d'un destructeur

Exemple : L'exemple suivant montre les moments d'appel du constructeur et du destructeur pour les objets de type auto et dynamique.

```
class X {
  char NomObjet[20];
public :
  X(char* Nom) {
  strcpy(NomObjet, Nom);
  cout<<"Exécution du constructeur de l'objet : "<<NomObjet<<endl;
}
~X() {
   cout<<"Exécution du destructeur de l'objet : "<<NomObjet<<endl;}
};
void main() {
  X x1("x1");
  X* x2 = new X("x2");
  delete x2;
}
```

VIII. Appel d'un destructeur

Exemple : L'exemple suivant montre les moments d'appel du constructeur et du destructeur pour les objets de type auto et dynamique.

```
class X {
  char NomObjet[20];
public :
   X(char* Nom) {
   strcpy(NomObjet, Nom);
   cout<<"Exécution du constructeur de l'objet : "<<NomObjet<<endl;
}
~X() {
   cout<<"Exécution du destructeur de l'objet : "<<NomObjet<<endl;}
};
void main() {
  X x1("x1");
  X* x2 = new X("x2");
  delete x2;
}
```

```
Exécution du constructeur de l'objet : x1
Exécution du constructeur de l'objet : x2
Exécution du destructeur de l'objet : x2
Exécution du destructeur de l'objet : x1
```


IIX. Destructeur par défaut

Si aucun destructeur n'a été explicitement défini pour une classe, alors le compilateur en génère un par défaut, public. Ce destructeur réalise par défaut la libération de l'espace mémoire de base (des attributs). Il est par conséquent toujours défini vide de la manière suivante : **~NomClasse(){}**

Remarque :

Le destructeur généré par défaut convient généralement aux classes simples (Exemple : la classe *Time*). Toutefois, les classes complexes qui demandent des traitements supplémentaires lors de leurs libérations nécessitent une définition explicite et personnalisée de leur destructeur comme le montre le paragraphe suivant.

IX. Exercices d'application

Exercice 1

1 - Proposer une spécification et une implémentation d'une classe *Date* comportant trois attributs *jour*, *mois* et *année*. La classe doit comporter :
- Deux constructeurs, un sans paramètres et l'autre permettant d'initialiser une date à travers trois paramètres de type entier.
- Un destructeur.
- Deux méthodes permettant la saisie et l'affichage d'une date.
- Une méthode de comparaison de deux dates. Proposer deux versions pour cette méthode :
    - o Une version qui retourne la date la plus récente.
    - o Une version qui retourne tout juste un entier indiquant la date la plus récente.

2 - Proposer une fonction principale montrant les possibilités d'utilisation de cette classe.

IX. Exercices d'application

Exercice 2

1 - Proposer une spécification et une implémentation d'une classe *Personne* comportant trois attributs : *Nom*, *Prenom* et *Date_de_naissance*. La classe doit comporter :
- Des constructeurs permettant de créer de différentes manières un objet de type *Personne*.
- Un destructeur.
- Deux méthodes de saisie et d'affichage des informations concernant une *Personne*.

2 - Proposer une fonction principale montrant les possibilités d'utilisation de cette classe.