Notions de base 2 : Constructeurs, destructeurs, fonctions membres et amies

Les exercices sont présentés à l'intérieur de chaque partie dans un ordre croissant de difficulté. Comme vous le constaterez très souvent en Programmation, un problème peut être résolu de différentes manières. Il peut donc plusieurs solutions à un exercice. Cependant, les meilleures solutions sont souvent les plus simples, et définitivement celles qui suivent la logique du cours et des travaux dirigés.

Les exercices marqués d'une étoile (*) sont réservés aux plus avancés sur les notions de base.

Objectifs

— Savoir définir des constructeurs et destructeur dans une classe
— Comprendre le concept de surdéfinition
— Comprendre le fonctionnement et l'intérêt des fonctions en ligne
— Savoir définir une fonctions amie

1 Constructeurs et destructeurs

EXERCICE 1 :

Créer une classe Personne représentée par un nom, un prénom et un âge. Le nom et le prénom seront représentés sous forme de tableau de 20 caractères et l'âge par un entier. On veut pouvoir créer des objets de type Personne soit en spécifiant le nom, le prénom et l'âge, soit en ne spécifiant rien, soit en spécifiant un objet Personne préexistant (constructeur de copie). Définir les constructeurs et destructeur associés. Afin de pouvoir tester la validité de votre classe, implémenter une fonction d'affichage : *void affiche() const.*

Dans un autre fichier. Faire une fonction `main` permettant de tester votre classe. Ce programme de test créera un tableau de Personne avec différentes initialisations, affichera l'ensemble des objets Personne à l'aide de la fonction *affiche()* et détruira le tableau.

EXERCICE 2 :

Modifier la classe Personne en remplaçant les tableaux par des tableaux dynamiques. Modifier les constructeurs afin qu'ils allouent de la mémoire aux tableaux lors de la création d'un objet et le destructeur afin qu'il libère la mémoire allouée.

*EXERCICE 3 :

Écrivez une classe nommée pile_entier permettant de gérer une pile d'entiers. Ces derniers seront conservés dans un tableau d'entiers alloué dynamiquement. La classe comportera les fonctions membres suivantes :

- *pile_ entier (int n)* : constructeur allouant dynamiquement un emplacement de n entiers,
- *pile_ entier ()* : constructeur allouant dynamiquement un emplacement de 20 entiers,
- *pile_ entier (pile_ entier &pile const)* : constructeur de copie créant une copie de la pile d'entier *pile*,
- *pile_ entier ()* : destructeur
- *void empile (int p)* : empile l'entier p sur la pile,
- *int depile ()* : fournit la valeur de l'entier en haut de la pile et le supprime de la pile,
- *int pleine ()* : renvoie 1 si la pile est pleine (nombre d'entiers dans la pile = taille du tableau) et 0 sinon,
- *int vide ()* : renvoie 1 si la pile est vide et 0 sinon.

2 Mot-clés const et static

EXERCICE 4 :

Quelles différences il-y-t-il entre les deux constantes MAX1 et MAX2 définies de la façon suivante ?

- #define MAX1 100
- static const int MAX2 = 100

Essayer d'accéder à l'adresse (pointeur) de MAX1 (int *p1=&MAX1) et à l'adresse de MAX2 (int *p2=&MAX2). Que se passe-t-il ?

EXERCICE 5 :

1. Écrire un fonction affichant un entier : void affiche(const int& n). Que signifie le const dans le prototype de la fonction affiche ? Que se passe-t-il s'il ont tente de modifier l'entier n à l'intérieur de la fonction affiche ?
2. Rajouter dans la classe Personne une fonction membre d'affichage : "void affiche() const". Que se passe-t-il ont tente de modifier l'objet courant à l'intérieur de la fonction membre affiche (par exemple en utilisant this->age =0) ?

*EXERCICE 6 :

Définir un pointeur sur un entier, nommé p (int *p), un pointeur sur un entier constant, nommé q (const int* q), un pointeur constant sur un entier, nommé r (int* const r). Faites des allocations de mémoire avec l'opérateur new. Que constatez-vous ? Définir un tableau de dix pointeurs sur des entiers. Faites une allocation de l'ensemble et une désallocation de mémoire. Vérifiez l'état de la mémoire. Définir une référence sur un entier, nommée s (int& s) puis une référence constante vers un entier, nommée t (const int& t). Essayez toutes les affectations entre p, q, r, s et t, et expliquez les résultats obtenus.

3 Fonctions en ligne

EXERCICE 7 :

Quel est la différence entre les deux "fonctions" suivantes ?

- #define copie1(source,dest) source=dest ;
- inline void copie2(int source, int dest) { source=dest ; }

Est-ce-que la copie est réalisée dans les deux cas ? Essayer d'appeler les deux "fonctions" avec des rationnels (de type double). Que se-passe-t-il ? Peut-on définir des fonctions récursives de cette façon ?

4 Fonctions membres et surdéfinition

EXERCICE 8 :

On souhaite donner le même nom à trois fonctions : "somme". La première additionne deux entiers (type int), la deuxième deux réels (type float) et la troisième deux tableaux de dix entiers. Donner le protoype de ces fonctions. Que se passe-t-il lorsqu'un appel est fait avec comme arguments deux short. Est-il possible de créer une fonction somme prenant 3 paramètres de même type ? Est-il possible de définir une fonction somme ayant deux paramètres de types différents (par exemple un int et un float) ?

EXERCICE 9 :

Ecrivez une classe Vecteur3D comportant :

- trois données membre de type double x,y,z (privées)
- deux fonctions membres d'affichage :
  - *void affiche()* affichant le vecteur.
  - *void affiche(const char* string)* affichant string avant l'affichage du vecteur.
- deux constructeurs
  - l'un sans argument, initialisant chaque composante à 0.
  - l'autre, avec trois arguments correspondant aux coordonnées du vecteur.

Modifiez ensuite le code pour que les constructeurs soit des fonctions en ligne (inline).

EXERCICE 10 :

Rajouter les fonctions suivantes à la classe Vecteur3D :

- Des fonctions permettant d'accéder au coordonnées d'un vecteur :
  - *int abscisse()* pour x,
  - *int ordonnée()* pour y et
  - *int cote()* pour z.
- Des fonctions permettant de modifier au coordonnées d'un vecteur :
  - *void fixer_abscisse(int nouvelle_abscisse)* pour x,
  - *void fixer_ordonnée(int nouvelle_ordonnée)* pour y et
  - *void fixer_cote(int nouvelle_ordonnée)* pour z.
- *bool coincide(Vecteur3D v)* qui renvoie true si v et l'objet courant ont les mêmes coordonnées, false sinon.

*EXERCICE 11 :

Rajouter les fonctions suivantes à la classe Vecteur3D :

- double produit_scalaire(Vecteur3D v) qui calcule le produit scalaire de v avec l'objet courant.
- Vecteur3D somme(Vecteur3D v) qui calcule le vecteur somme de v avec l'objet courant.

5 Fonctions amies

EXERCICE 12 :

Changer la fonction *coincide* le l'exercice 10 pour quelle soit symétrique. Utiliser le prototype suivant : *friend int coincide(Vecteur3D v1, Vecteur3D v2)*. Que signifie le mot-clé friend ?

*EXERCICE 13 :

On veut réaliser le produit d'une matrice avec un vecteur. Les classes Matrice et Vecteur sont définies de la manière suivante :

```
#define TAILLE 3

class Vecteur
{
  private:
    double vect[TAILLE];
  public:
    Vecteur(double t[TAILLE]){
    for(i=0;i<TAILLE;i++)
      vect[i]=t[i];
    }
}

class Matrice
{
  private:
    double mat[TAILLE][TAILLE];
  public:
    Matrice(double t[TAILLE][TAILLE]){
      int i,j;
      for(i=0;i<TAILLE;i++)
        for(j=0;j<TAILLE;j++)
          mat[i][j]=t[i][j];
    }
}
```

- Pourquoi les constructeurs sont-il implémentés dans la classe ? Quelle est la différence avec des constructeurs implémentés à l'extérieur de la classe ?
- Où peut-on déclarer la fonction *Vecteur produit(Matrice mat, Vecteur vect)* faisant le produit de mat par vect ? Comment faire pour que la fonction *produit* puisse accéder au données membres des objets mat et vect ?
- On souhaite implémenter une fonction *Vecteur produit(Vecteur vect)* dans la classe matrice. Comment faire pour que la fonction produit puisse accéder au données membres de l'objet vect ?