Classes imbriquées

En C++, il est possible de créer une classe par une relation d'appartenance.

Exemple : une voiture a un moteur, a des roues ...

```
class Moteur { /* ... */ };
class Roue { /* ... */ };

class Voiture
{
   public:
   // ....
   private:
      Moteur m_moteur;
      Roue m_roue[4];
      // ....
};
```

Affectation et Initialisation

Le langage C++ (de même que le C) fait la différence entre l'**initialisation** et l'**affectation**.

- l'*affectation* consiste à modifier la valeur d'une variable (et peut avoir lieu plusieurs fois),

- l'*initialisation* est une opération qui n'a lieu qu'une fois immédiatement après que l'espace mémoire de la variable ait été alloué. Cette opération consiste à donner une valeur initiale à l'objet ainsi crée.

Classes et fonctions amies (1)

Dans la définition d'une classe il est possible de designer des fonctions (ou des classes) à qui on laisse un libre accès à ses membres privés et protégés. Ce n'est pas une infraction aux règles d'*encapsulation* car les *déclarations* de toutes les fonctions (ou classes) amies se font à l'intérieur de la classe en question.

Exemple d'une fonction amie:

```
class Nombre
{
   friend int manipule_nbre(); // fonction amie
public :
   int getnbre();
   // ...
private :
    int m_nbre;
};

int manipule_nbre (Nombre n)
{
   return 3*(n.m_nbre + 1); // accès à m_nbre permis
}
```

Classes et fonctions amies (2)

Une classe amie ou une fonction amie d'une classe *X* peuvent accéder aux parties privées et protégées de X.

Exemple de classe amie:

```
class Window; // déclaration de la classe Window


class Screen
{
   friend class Window;
public:
   //...
private:
   //...
};
```

Toutes les fonctions membres de la classe *Window* peuvent accéder à tous les membres (publics et non publics) de la classe *Screen*.

Pointeur *this*

Toute méthode d'une classe X a un paramètre caché: le pointeur *this*. Celui-ci contient l'adresse de l'objet qui l'a appelé. Il est implicitement déclaré comme (*pour une variable*): X * **const this**,

et comme (*pour un objet constant*): **const** X * **const this**,

Il est initialisé avec l'adresse de l'objet sur lequel la méthode est appelée. Il peut être explicitement utilisé:

```
class X
{
public:
   int f1()
   {
      return this->i; // utilisation de this
   }
   // idem que : int f1() { return i; }
private:
   int i;
   // ...
};
```

Liste d'initialisation d'un constructeur

```
class Y { /* ... */ }; // la classe Y
class X // la classe X
{
public:
   X (int a, int b, Y y); // le constructeur de X
   ~X(); // le destructeur de X
   // ....
private:
   const int _x;
   Y _y;
   int _z;
};
X :: X(int a, int b, Y y) // le constructeur de X
{
   _x = a; // ERREUR: l'affectation à une constante est interdite
   _z = b; // OK : affectation et initialisation de l'objet membre _z
}
```

Question: Comment initialiser la donnée membre constante _x et appeler le constructeur de la classe Y ?

Réponse: Utiliser la liste d'initialisation.

Liste d'initialisation

La phase d'initialisation de l'objet peut utiliser une liste d'initialisation qui est spécifiée dans la définition du constructeur.

Syntaxe:

```
nom_classe::nom_constructeur( args ... ) : liste_d'initialisation
{
   // corps du constructeur
}
```

Exemple:

```
X :: X(int a, int b, Y y) : _x( a ) , _z( b ), _y( y )
{
   // rien d'autre à faire
}
```

- L'expression *_x(a)* permet d'initialiser la donnée membre *_x* avec la valeur du paramètre *a*.
- L'expression *_y(y)* permet d'initialiser la donnée membre *_y* par un appel au constructeur (avec l'argument *y*) de la classe Y.

Allocation de la mémoire (1)

Le C++ met à la disposition du programmeur deux opérateurs ***new*** et ***delete*** pour remplacer respectivement les fonctions ***malloc*** et ***free*** (bien qu'il soit toujours possible de les utiliser).

L'opérateur *new*

L'opérateur ***new*** réserve l'espace mémoire qu'on lui demande et l'initialise. Il retourne soit l'adresse de début de la zone mémoire allouée, soit NULL si l'opération a échoué.

```
int *ptr1, *ptr2, *ptr3;
struct date { int jour, mois, an; };
ptr1 = new int; // allocation dynamique pour un entier
ptr2 = new int [10]; // allocation pour un tableau de 10 entiers
ptr3 = new int (10); // allocation pour un entier avec initialisation
date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};
ptr4 = new date; // allocation dynamique pour une structure
ptr5 = new date[10]; // allocation pour un tableau de 10 structures
ptr6 = new date(d); // allocation pour une structure avec initialisation
```

Libération de la mémoire (2)

L'opérateur *delete*

L'opérateur ***delete*** libère l'espace mémoire alloué par ***new*** à un seul objet, tandis que l'opérateur ***delete [ ]*** libère l'espace mémoire alloué à un tableau d'objets.

// libération d'un entier: **delete** ptr1;

// libération d'un tableau d'entiers: **delete** [] ptr2;

L'application de l'opérateur ***delete*** à un pointeur nul est légale et n'entraîne aucune conséquence fâcheuse.

À chaque instruction ***new*** doit correspondre une instruction ***delete***. Il est important de libérer l'espace mémoire dès que celui-ci n'est plus nécessaire. La mémoire allouée en cours de l'exécution du programme sera libérée automatiquement à la fin de l'exécution.

Variables références

En plus des variables normales et des pointeurs, le C++ offre les *variables références*. Une variable référence permet de créer une variable qui est un "synonyme" d'une autre. Une variable référence doit être initialisée et le type de l'objet initial doit être le même que l'objet référence.

```
int i;
int & ir = i; // ir est une référence à i
int * ptr;
i = 1;
cout << "i = " << i << " ir = " << ir << endl;
// affichage de: i = 1 ir = 1
ir = 2;
cout << "i = " << i << " ir = " << ir << endl;
// affichage de: i = 2 ir = 2
ptr = &ir;
*ptr = 3;
cout << "i= " << i << " ir= " << ir << endl;
// affichage de: i = 3 ir = 3
```

Intérêt: passage des paramètres à une fonction par référence

Passage des paramètres par référence

En plus du passage par valeur et adresse, le C++ définit le passage par référence. Lorsque l'on passe à une fonction un paramètre par référence, cette fonction reçoit un "synonyme" du paramètre réel. Toute modification du paramètre référence est répercutée sur le paramètre réel.

Exemple:

```
void echange( int & n1, int & n2)
{
   int temp = n1; n1 = n2; n2 = temp;
}

void main( )
{
  int i = 2, j = 3;
  cout << "i= " << i << " j= " << j << endl;
  // affichage de : i = 2  j = 3
  echange(i, j);
  cout << "i = " << i << " j= " << j << endl;
  // affichage de : i = 3  j = 2
}
```

Surcharge des fonctions (1)

Une fonction est définie par:

- son nom,
- sa liste typée de paramètres formels,
- le type de la valeur qu'elle retourne.

Mais seuls les deux premiers critères sont discriminants. On dit qu'ils constituent la **signature** de la fonction. On peut utiliser cette propriété pour donner un même nom à des fonctions qui ont des paramètres différents !

Le compilateur sélectionnera la fonction à appeler en fonction du type et du nombre des arguments qui figurent dans l'appel de la fonction.

Surcharge des fonctions (2)

Exemple:

```
int somme( int n1, int n2)
{ return n1 + n2; }

int somme( int n1, int n2, int n3)
{ return n1 + n2 + n3; }

double somme( double n1, double n2)
{ return n1 + n2; }


void main()

{
  cout << "1 + 2 = " << somme(1, 2) << endl;
  cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
  cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;
}
```

Le choix de la fonction à appeler se fait à la compilation. L'appel d'une fonction surchargée procure des performances identiques à un appel d’une fonction "classique".

Surcharge des fonctions (3)

Exemples (*qui ne marchent pas*):

```
int somme1 (int n1, int n2) {return n1 + n2;}
int somme1 (const int n1, const int n2) {return n1 + n2;}
  // Erreur: la liste de paramètres dans les déclarations des deux
  // fonctions n'est pas assez divergente pour les différencier.

int somme2 (int n1, int n2) {return n1 + n2;}
int somme2 (int & n1, int & n2) {return n1 + n2;}
  // Erreur: la liste de paramètres dans les déclarations des deux
  // fonctions n'est pas assez divergente pour les différencier.

int somme3 (int n1, int n2) {return n1 + n2;}
double somme3 (int n1, int n2) {return (double) n1 + n2;}
  // Erreur: seul le type des paramètres permet de faire la
  // distinction entre les fonctions et non pas la valeur retournée.

int somme4 (int n1, int n2) {return n1 + n2;}
int somme4 (int n1, int n2 = 8) {return n1 + n2;}
  // Erreur: la liste de paramètres dans les déclarations des deux
  // fonctions n'est pas assez divergente pour les différencier.
```

Surcharge des opérateurs (1)

Le concepteur d'une classe doit fournir à l'utilisateur de celle-ci toute une serie d'opérateurs agissant sur les objets de la classe. Ceci permet une syntaxe intuitive de la classe. Par exemple, il est plus intuitif et plus clair d'additionner deux matrices en surchargeant l'opérateur d'addition et en écrivant: *result = m0 + m1*, que d'écrire *matrice_add(result, m0, m1)*.

Règles d'utilisation:

- il faut veiller à respecter l'esprit de l'opérateur. Il faut surcharger les opérateurs de façon qu'ils fassent des opérations identiques à celles que font les opérateurs avec les types prédéfinis.
- la plupart des opérateurs sont surchargeables.
- les opérateurs suivants ne sont pas surchargeables: **:: . sizeof .* ?:**
- il n'est pas possible pour un opérateur de:
    - changer sa priorité,
    - changer son associativité,
    - changer sa pluralité (unaire, binaire, ternaire).

Surcharge des opérateurs (2)

Quand l'opérateur + (par exemple) est appelé, le compilateur génère un appel à la fonction ***operator+***.

Ainsi, l'instruction ***a = b + c;*** est équivalente aux instructions:

a = **operator+**(b, c); // pour une fonction globale

a = b.**operator+**(c); // pour une fonction membre

Les opérateurs **= () [] -> new delete** ne peuvent être surchargés que comme des fonctions membres.

Surcharge par une fonction globale

Cette façon de procéder est plus adaptée à la surcharge des opérateurs binaires. En effet, elle permet d'appliquer des conversions implicites au premier membre de l'expression.

Surcharge des opérateurs (3)

```
// Surcharge par une fonction globale
class Nombre {
  friend Nombre operator+ (const Nombre &, const Nombre &);
  public:
    Nombre(int n = 0) { _nbre = n; }
  private:
    int _nbre;
};
Nombre operator+(const Nombre &nbr1, const Nombre &nbr2) {
  Nombre n;
  n._nbre = nbr1._nbre + nbr2._nbre;
  return n;
}
void main () {
  Nombre n1(10);
  n1 + 20;  // OK appel à :  operator+( n1, Nombre(20) );
  30 + n1;  // OK appel à :  operator+( Nombre(30) , n1 );
}
```