Notions avancées du C++ (1)

Jusqu'à présent, nous avons vu les mécanismes de base de la programmation orientée objet en C++. Nous allons maintenant nous attarder sur les principes nécessaire à la création d'objets complexes ou composites : les objets ayant des attributs objets, et les objets qui ont des attributs dynamiques (c'est-à-dire dont on ne connait pas la taille ou la quantité en mémoire).

Ces objets aux attributs dynamiques demandent une attention particulière dans leur conception et il faut toujours réfléchir aux rôles des constructeurs, destructeurs et opérateur d'affectation.

Il faut toujours se poser les questions suivantes :

- où et quand l'allocation dynamique est-elle effectuée?
- Où et quand les objets dynamiques sont-ils détruits ?
- Que se passe-t-il lors de la copie et de l'affectation ?

Les objets membres

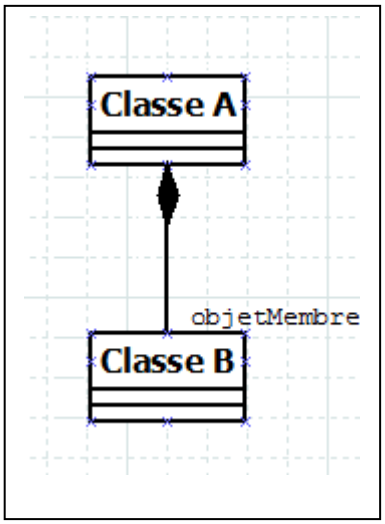
On désigne par objet membre, un objet qui est attribut d'un autre objet : (attention, il ne s'agit pas d'allocation dynamique)

```
class B
{
       private:

        A objetMembre; // Objet de classe A, membre d'instance
                       //de la classe B

        ...

};
```



Les deux classes **A** et **B** sont reliées par une **relation de composition**.

Lors de la construction d'un **objet composite**, le constructeur va appeler (implicitement) le constructeur par défaut des attributs (dans l'ordre de leur déclaration) avant son exécution. Si l'on veut que le constructeur appelle les constructeurs des attributs avec des paramètres, il faut le spécifier explicitement dans la définition du constructeur de la façon suivante :

```
class A {

                    private:
                      B attribut1;
                      C attribut2;
                      D attribut3;

                    public:
                      A(int param1, double param2);
             };
```

```
// Définition déportée du constructeur
// Il spécifie les appels aux constructeurs des attributs

A::A(int param1, double param2): attribut1(param1), attribut2(param2)
{ ... }
```

Lorsque l'on va construire un objet **a** de classe **A** par l'instruction : `A a(12, 13.5);`

le constructeur construit d'abord l'**attribut1** en appelant le **constructeur de la classe B** avec en paramètre un entier de valeur **12**, ensuite il construit **attribut2** en appelant **le constructeur de la classe C** avec un double de valeur **13.5** en paramètre, puis il construit **attribut3** de **classe D en appelant son constructeur par défaut**, tout cela avant d'exécuter ses propres instructions.

L'objet **a** que l'on vient de déclarer sera détruit implicitement à la fin du bloc dans lequel il a été déclaré.

Le destructeur de l'objet va automatiquement appeler les destructeurs des attributs. L'ordre d'exécution des destructeurs est inverse de l'ordre d'exécution des constructeurs : le destructeur de l'objet **a** va d'abord s'exécuter puis va automatiquement appeler le **destructeur de attribut3**, puis **celui de attribut2** et enfin **celui de attribut1**. Tout cela est automatique : l'utilisateur ne doit pas appeler les destructeurs lui-même car il s'agit d'objets membres non dynamiques.

Les attributs dynamiques

Jusqu'à présent, nous avons vu que les objets d'une classe donnée peuvent avoir comme attributs des instances des types de base (int, double...), des tableaux de taille fixe, ou des objets.
Ils peuvent aussi avoir des attributs dynamiques qui seront alloués soit à la construction de l'objet soit pendant son utilisation (sa "vie"). Ce genre d'objets aux attributs dynamiques est très fréquemment utilisé en C++.

Revenons sur la classe **Etudiant** vue dans le chapitre précédent. On veut maintenant que chaque **Etudiant** ait un nom, désigné par une chaine de caractères.
Cependant, on ne sait pas à l'avance quelle est la taille du nom de tous les objets : la chaine de caractères aura une taille différente pour chaque instance.

Pour mettre en place un tel attribut dynamique, on va déclarer un pointeur sur une chaîne de caractères comme attribut des objets de la classe **Etudiant** et l'allocation et la libération de l'espace mémoire nécessaire à la chaine de caractères vont être gérées par l'objet lui-même.

Voici la classe **Etudiant** avec une gestion dynamique du nom :

```
class Etudiant {

                    private:

                      char * nom;  // Pointeur sur une chaine de caratères
                      double moyenne;

                    public:

                      // Constructeur par défaut
                      Etudiant();
```

```
        // Constructeur qui initialise le nom de l'etudiant
        Etudiant(char *);

        // getter et setter pour l'attribut moyenne
        double getMoyenne() { return moyenne; }
        void setMoyenne(double m) { moyenne = m; }

        // getter et setter pour l'attribut nom
        char * getNom() { return nom; }
        void setNom(char * name);

        // Destructeur
        ~Etudiant();
      };


// Constructeur par défaut
// Le pointeur nom est initialisé au pointeur nul, la moyenne à 0
Etudiant::Etudiant() : nom(0), moyenne(0.0) {}

// Constructeur avec le nom en paramètre
// Le constructeur alloue un tableau de caractère de la taille du nom
// puis copie la chaine passée en argument
// La moyenne est initialisée à zéro
Etudiant::Etudiant(char * name) : moyenne(0.0) {
  if(name) {
       nom = new char[strlen(name)+1];
       strcpy(nom, name);
  }
}

// Methode pour changer le nom de l'etudiant.
// Si l'etudiant a un nom, on supprime l'ancien nom
// et on alloue une nouvelle chaine de la bonne taille
void Etudiant::setNom(char * name) {
  // On appelle delete [] pour supprimer un tableau dynamique
  if(nom) delete [] nom;

  // Allocation d'une nouvelle chaine
  if(name) {
       nom = new char[strlen(name)+1];
       strcpy(nom, name);
  }
}

// Destructeur
// Il est indispensable ici pour libérer la chaine de caractère
Etudiant::~Etudiant() {
  delete [] nom;
}
```

Il ne faut pas oublier de supprimer les attributs alloués de façon dynamique : **cela doit être fait a priori dans le destructeur.**

Le mot clé const

Les constantes nommées

Le mot clé **const** peut être ajouté à la déclaration d'un objet pour en faire une constante plutôt qu'une variable. Une constante ne pouvant être modifiée, elle doit donc être initialisée lors de sa définition.

```
const int largeurMax = 640;         // Entier constant
const int sizeMax[] = {640, 480};   // Tableau constant
const Point origine(0, 0);          // Objet constant de classe Point
```

Pointeurs et objets constants

Lorsqu'on manipule des pointeurs et des objets, il faut distinguer les pointeurs sur des objets constants et les pointeurs constants sur des objets.

Pour déclarer un pointeur sur un objet constant, le mot clé **const** modifie le type de l'objet pointé (il se place avant).

Dans ce cas-là les modifications sur l'objet sont interdites, mais il est possible de changer la valeur du pointeur.

```
const double pi = 3.14159; // Déclaration d'un double constant
const double * ptr1 = 0;   // Pointeur (non constant) sur un
                           // double constant initialisé au pointeur nul

ptr1  = &pi;               // OK
*ptr1 = 12;                // Erreur (tentative de modification de
                           // l'objet pointé)

const double e = 2.71828;  // Déclaration d'un autre double constant

ptr1 = &e;                 // OK (modification de la valeur du pointeur)
```

Pour déclarer **un pointeur constant**, on utilise ***const**. Il faut l'initialiser lors de sa définition. Il ne sera pas possible de modifier la valeur du pointeur mais on pourra modifier l'objet pointé.

```
// Déclaration de deux double
double x = 0.0, y = 0.0;
// Déclaration d'un pointeur constant initialisé avec l'adresse de x
double *const ptr2 = &x;


*ptr2 = pi;    // OK
// (modification de l'objet pointé, x prend la valeur de pi

ptr2 = &y;     // Erreur
//(tentative de modification de la valeur du pointeur)
```

Méthodes constantes

Lorsqu'un objet est déclaré constant, il n'est pas possible de le modifier. Par conséquent, on ne peut appeler que les méthodes qui ne modifient pas l'objet courant. De telles méthodes doivent être déclarées en ajoutant le mot clé **const** après leur déclaration.

```
class Point {

        private:
          double x;
          double y;

        public:

          Point(double a, double b);

          // Renvoie la valeur des attributs :
          //l'objet courant n'est pas modifié
          double getX() const;
          double getY() const;

          // Assigne une valeur aux attributs : l'objet est modifié
          void setX(double absisse);
          void setY(double ordonnee);
};

// Constructeur de la classe Point avec initialisation
// à la construction des attributs
Point::Point(double a, double b) : x(a), y(b) {}

// Methode constante qui renvoie l'abscisse du point
double Point::getX() const {
        return x;
}

// Methode constante qui renvoie l'ordonnee du point
double Point::getY() const {
          return y;
}

        ...

int main() {

        const Point origine(0, 0);
        double x = origine.getX(); // OK
        origne.setX(0.5);          // Erreur de compilation
}
```

Arguments constants

Lors du passage d'arguments à une fonction (ou une méthode), un argument non constant peut-être passé à une fonction qui attend un argument constant : l'objet passé en argument sera alors constant dans la fonction. Attention l'inverse est faux !!

```
// Fonction qui prend en argument un pointeur sur un objet constant
void f(const A * a) {
```

```
  ...
}

// Fonction qui prend en argument un pointeur sur un objet de classe A
void g(A * a) {
  ...
}

int main() {

  const A a1; // Un objet constant de classe A
  A a2;       // Un objet non constant de classe A

  f(&a1);     // OK
  f(&a2);     // OK

  g(&a1);     // Erreur
  g(&a2);     // OK
}
```

Exemple : nous pouvons réécrire désormais la déclaration de la classe Etudiant de la section précédente pour contrôler le cas des objets constant (le corps des méthodes est inchangé) :

```
class Etudiant {

       private:

        // Pointeur sur une chaine de caratères
        char * nom;

        double moyenne

       public:

        // Constructeur par défaut
        Etudiant();

       // Constructeur qui initialise le nom de l'etudiant
       // On ne modifiera pas la chaine passée en argument
       Etudiant(const char *);

       // getter et setter pour l'attribut moyenne

       // getMoyenne ne modifie pas l'objet courant
        double getMoyenne() const;

       // Le passage d'argument est fait par valeur
       // donc le mot clé const est inutile ici
       void setMoyenne(double m);


        // getter et setter pour l'attribut nom

       // getNom ne modifie pas l'objet courant
       // et renvoie un pointeur sur la chaine déclarée constante
       // pour ne pas la modifier
       const char * getNom() const;
```

```
        // setNom prend en argument un pointeur sur une
        // chaine de caractères constante (on ne la modifie pas,
        // on ne fait que la copier)
        void setNom(const char * name);

        // Destructeur
        ~Etudiant();
};
```

Les références

Définition

La référence est une façon de renommer un objet. C'est une sorte d'alias vers un objet.

X & désigne une **référence sur un objet de type X**.

Dans la portion de code suivante :

```
A a;        // Un objet a de classe A
A & b = a; // Une référence b qui fait référence au même objet a
```

a et **b** désigne le même objet, on peut modifier l'objet **a** en **manipulant indifféremment a ou b**.

Exemple :

```
int compteur = 0;
double & ref = compteur;
ref++; // Incrément le compteur
```

Lorsqu'on déclare directement une référence (comme dans les 2 exemples précédents), il faut qu'elle soit initialisée lors de sa déclaration. (On ne peut pas écrire directement `A & b; ... b=a;`).

Cependant, les références ne sont quasiment jamais utilisée dans ce cas là : on utilise principalement les références pour le passage d'arguments ou pour la valeur de retour des fonctions (ou méthodes).

Le passage d'arguments

On distingue en C++, trois types de passage d'arguments dans les fonctions : le passage par valeur, le passage par adresse et le passage par référence.

Lors du **passage par valeur**, les arguments sont copiés dans des objets temporaires qui sont créés (avec un appel au constructeur de recopie, voir section suivante) lors du passage d'arguments et qui sont détruit à la fin de l'exécution de la fonction : toutes les modifications effectuées sur ces objets temporaires seront donc perdues à la fin de la fonction et n'auront aucune incidence sur l'objet passé en argument lors de l'appel de la fonction.

```
void incremente(int i) { i++; }

int main() {
  int j = 12;
  incremente(j);
  cout << j << endl; // Affiche 12
}
```

Lors du **passage par adresse**, on manipule des pointeurs (c'est le passage d'arguments classique du C lorsqu'on veut modifier les arguments) : les arguments sont des pointeurs qui contiennent l'adresse en mémoire des objets que l'on veut manipuler. Il n'y a qu'une copie de pointeur lors du passage par adresse.

```
void incremente(int * i) { (*i)++; }

int main() {
  int j = 12;
  incremente(&j);
  cout << j << endl; // Affiche 13
}
```

Le C++ permet le **passage par référence** : la fonction appelée manipule directement l'objet qui lui est passé par référence (cela correspond au `var` du langage Pascal). Il n'y a pas de copie ni de construction d'objet temporaire lors le passage par référence.

```
void incremente(int & i) { i++; }

int main() {
  int j = 12;
  incremente(j);
  cout << j << endl; // Affiche 13
}
```

Lorsqu'on manipule de gros objets (des matrices, des images...), il est particulièrement intéressant d'effectuer des passages par référence. Si l'on veut s'assurer que l'objet que l'on passe en paramètre ne sera pas modifié dans la fonction, il faut rajouter le mot clé **`const`** : l'objet sera alors considéré comme constant dans la fonction.

```
void f(const A & a);
```