Rappel : Notions d'objets et de classe

Un objet est une entité qui regroupe des données membre (les **attributs**) et des fonctions membre (les **méthodes**). Les méthodes sont des fonctions qui s'appliquent à l'objet et à ses attributs, elles représentent les fonctionnalités de l'objet (son comportement).

Une **classe est un moule d'objet**, elle décrit les attributs et les méthodes de toute une famille d'objets. Une classe définit en particulier un type d'objet. Les objets d'une même classe auront des attributs de même type, mais avec des valeurs différentes. Une classe est la description générale d'une famille d'objet et un objet est une instance de classe.

Par exemple, si une classe Voiture décrit une voiture avec comme attributs une couleur, un type de moteur et un nombre de chevaux, **une instance de cette classe, un objet**, désigne donc une voiture particulière, "la voiture de Monsieur Jean", qui est rouge et qui a un moteur diesel de 90 chevaux.

Déclaration d'une classe

La déclaration d'une classe s'inspire de la déclaration des structures en C. Voici la déclaration d'une classe A avec un attribut entier i et une méthode afficher qui affiche l'objet sur la sortie standard :

Attention au point-virgule à la fin de la déclaration de la classe

```
class A {

       public:
         int i;
         void afficher()
         {
              std::cout << "i = " << i << std::endl;
         }
};
```

Instanciation statique, accès aux membres (publics)

Pour déclarer une instance nommée a de la classe A, il suffit d'écrire :

```
A a;
```

L'accès aux attributs et aux méthodes (publiques) de l'objet s'effectue avec l'opérateur **.**

```
a.i = 10;      // Accès direct à l'attribut i de l'objet a (de classe A)
a.afficher(); // Appel de la fonction membre afficher de l'objet a
```

Une méthode d'instance (ou d'objet) a accès directement aux attributs de l'objet. Par exemple, dans la classe A précédente, la méthode afficher accède directement à l'attribut i de l'objet courant.

Méthodes en ligne, méthodes déportées

Il faut distinguer la **déclaration** d'une fonction (membre ou non) de sa **définition**.

La **déclaration** consiste à signaler à une portion de code que la fonction existe, elle est faite en générale dans les fichiers d'en-tête et elle consiste juste à donner le prototype de la fonction (qu'elle soit dans une classe, et il s'agit alors d'une méthode, ou à l'extérieur de toute déclaration de classe, et il s'agit alors d'une fonction globale).

La **définition** consiste à définir la fonction, c'est à dire à spécifier le corps de la méthode. Ceci ne doit apparaître qu'une seule fois dans tout le programme.

Lorsque la **définition** d'une méthode est faite lors sa **déclaration**, on parle de **méthode en ligne**, ce qui est le cas pour la méthode **afficher** de la **classe A** précédemment déclarée.

A l'inverse, lorsque la méthode est définie **séparemment** de sa déclaration (i.e. à l'extérieur de la déclaration de la classe), on parle de **méthode déportée** : cela est très courant lors de la compilation séparée. Cela permet de bien séparer la déclaration d'une classe de son implémentation (qui est transparente pour celui qui utilise la classe).

Réécrivons la classe A en déportant la définition de sa méthode afficher :

```
// généralement dans un fichier.h

#include <iostream>

// Déclaration de la classe A
class A {

public:
  int i;
  void afficher();
};


//généralement dans un fichier.cpp

#include "fichier.h"

// Définition de la méthode afficher de la classe A

void A::afficher()
{
  // Affichage de l'attribut i sur la sortie standard
  cout << "i = " << i << endl;
}
```

Pour spécifier qu'il s'agit de la méthode **afficher** de la classe A, il faut faire précéder le nom de la méthode par le nom de la classe, les deux identifiants étant séparés par **::**.

Ainsi **A::afficher()** désigne la méthode afficher de la classe A.

Visibilité des attributs et des méthodes

Pour se conformer au principe d'encapsulation vu dans le chapitre d'introduction, il est possible de masquer des attributs et des méthodes au sein d'un objet. On utilise pour cela les mots clés **private** et **public** : les membres (attributs ou méthodes) qui sont **private** ne sont accessibles que par les méthodes de l'objet lui-même et les méthodes des objets de la même classe, tandis que les membres public sont visibles par tous.

```
class B
{
       private:
         int attribut_prive;
         void methode_privee();

       public:
         int attribut_publique;
         void methode_publique();

};


int main()
{

         B b;                      // Déclaration d'un objet b de classe B
         b.attribut_publique = 0; // OK
         b.attribut_prive = 0;    // Erreur de compilation

         b.methode_publique();    // OK
         b.methode_privee();      // Erreur de compilation

         return 0;
}
```

Ainsi un attribut **public** pourra être modifié par tous et l'objet ne pourra finalement pas contrôler les modifications effectuées sur son propre attribut. Pour éviter ce manque de contrôle, il est préférable de toujours protéger les attributs et de mettre en place une méthode qui permet de récupérer la valeur d'un attribut et une méthode permettant de modifier la valeur d'un attribut (si l'on veut pouvoir modifier cet attribut bien sûr).

Ecrivons à nouveau la classe B :

```
// Déclaration de la classe B (dans un fichier .h)
class B
{

private:
  int attribut1;
  double attribut2;

public:

  int getAttribut1();  // Méthode qui renvoie la valeur de l'attribut1
  void setAttribut1(int v); // Méthode qui affecte la valeur passée en paramètre
                            // à l'attribut1
```

```
  double getAttribut2();  // Méthode qui renvoie la valeur de l'attribut2
  void setAttribut2(double v);  // Méthode qui affecte la valeur passée en
                                //paramètre à l'attribut2

};

// Corps des méthodes déportées (définition des méthodes dans un fichier .cpp)
int B::getAttribut1() {
  return attribut1;
}

void B::setAttribut1(int v) {
  attribut1 = v;
}

double B::getAttribut2() {
  return attribut2;
}

void B::setAttribut2(double v) {
  attribut2 = v;
}
```

Les attributs sont encapsulés : principe d' **encapsulation**

Construction et destruction des objets

Le constructeur

A la déclaration d'un objet, les attributs ne sont pas initialisés. La norme ANSI ne spécifie pas d'initialisation par défaut. Il faut donc initialiser les attributs de l'objet avant de pouvoir manipuler l'objet (comme en C, il ne faut pas manipuler une variable sans qu'elle soit initialisée). **Cette phase d'initialisation des attributs est effectuée par une méthode particulière : le constructeur.**
Le constructeur est une méthode appelée lors de la construction de l'objet.
Le constructeur est une méthode ayant le même nom que la classe et qui n'a pas de type de retour. Il peut prendre des paramètres, il faudra alors spécifier un paramètre lors de la déclaration d'un objet.

Une classe peut définir plusieurs constructeurs avec des paramètres différents (il y a **surcharge de méthode**).

```
// Déclaration de la classe Etudiant avec pour seul attribut la moyenne de l'etudiant
class Etudiant
{

  private:
       double moyenne;

  public:
       // Constructeur sans paramètre (constructeur par défaut)
       Etudiant()
       {
              moyenne = 0;
       }
```

```
        // Constructeur avec un paramètre
        Etudiant(double note)
        {
                moyenne = note;
        }

        // Méthodes d'accès à l'attribut moyenne
        double getMoyenne() { return moyenne; }
        void setMoyenne(double note) { moyenne = note; }

};

int main()
{
        // Déclaration d'un objet avec appel du constructeur par défaut
       Etudiant bob;      // La moyenne de l'étudiant est initialisé à 0

        // Déclaration d'un objet avec appel du constructeur avec un paramètre
        Etudiant john(10); // La moyenne de l'étudiant est initialisée à 10
        return 0;
}
```

Le destructeur

Tout comme il y a une méthode appelé à la construction d'un objet, le destructeur est une méthode appelée lorsque l'objet est détruit. Le destructeur doit être impérativement écrit pour libérer la mémoire allouée de façon dynamique pendant la vie d'un objet. Cependant ce n'est pas le destructeur qui libère la mémoire prise par l'objet lui-même : celle-ci sera libérée automatiquement après l'exécution du destructeur.

Le destructeur est désigné par le nom de la classe précédé de ~ , il n'a pas de paramètre, ni de type de retour :

```
class A
{
   private:
        ...
   public:
       A();   // Constructeur par défaut
       ~A();  // Destructeur
        ...
};
```

Durée de vie d'un objet

Le destructeur est appelé lorsque l'objet est détruit. Quand un objet est-il détruit ? Dans le cas d'une instanciation automatique (c'est à dire non dynamique), l'objet est détruit à la fin du bloc dans lequel il a été déclaré.

Dans la portion de code suivante :

```
int main()
{
        A a1;
        for(int i = 0; i<10; i++)
        {
          A a2;
          a2.afficher();
        } // Destruction de a2

        a1.afficher();
        return 0;


} // Destruction de a1
```

L'objet **a1** est détruit après la dernière instruction du **main**, et l'objet **a2** est détruit à la fin de chaque itération de la boucle for.

On verra dans la section suivante qu'un objet allouée de façon dynamique avec new n'est détruit qu'à l'appel explicite de l'opérateur **delete**.

Instanciation dynamique

L'allocation dynamique du C++ s'effectue avec les opérateurs **new** et **delete**, on doit absolument éviter d'utiliser les fonctions malloc et free qui sont propres au C.

L'allocation dynamique d'un objet permet de créer des objets à la volée, selon les besoins de l'exécution. Ces objets créés dynamiquement ne sont pas liés au bloc de programme dans lequel il a été déclaré : ils perdurent tant qu'ils n'ont pas été supprimés explicitement par l'utilisateur. Si l'on oublie de libérer des objets dynamiques, la quantité de mémoire allouée sera perdue pour le reste du programme.

L'allocation dynamique s'effectue avec l'opérateur **new** : l'objet est alloué puis initialisé par appel au constructeur. Tout comme pour l'instanciation automatique, il est possible de spécifier des paramètres au constructeur. L'opérateur **new** renvoie un pointeur sur l'objet nouvellement alloué.

```
// Allocation dynamique d'un entier
int * ptr_entier = new int;

// Allocation dynamique d'un objet de classe Etudiant
// Le constructeur par défaut est appelé
Etudiant * ptr_etudiant1 = new Etudiant;

// Allocation dynamique d'un objet de classe Etudiant
// Le constructeur avec un paramètre de type double est appelé
Etudiant * ptr_etudiant2 = new Etudiant(12.5);
```

On utilise l'opérateur **delete** pour la suppression d'un objet dynamique. On ne peut l'appeler que sur un pointeur pointant vers un objet dynamique. L'opérateur **delete** fait appel au destructeur de l'objet avant de libérer la mémoire en vue d'une nouvelle allocation. Pour supprimer les objets pointés par ptr_entier, ptr_etudiant1, ptr_etudiant2, alloués juste au dessus, il faut écrire tout simplement :

```
// Suppression de l'objet pointé par ptr_entier
delete ptr_entier;

// Suppression de l'objet pointé par ptr_etudiant1
// Le destructeur de l'objet (de classe Etudiant) est appelé
delete ptr_etudiant1;

// Suppression de l'objet pointé par ptr_etudiant2
// Le destructeur de l'objet (de classe Etudiant) est appelé
delete ptr_etudiant2;
```

Rappel

Dans l'expression suivante :

```
A * ptr;
```

ptr désigne un pointeur sur un objet de classe A. Dans cette déclaration, le pointeur n'est pas initialisé, et il n'y a pas de construction d'objet.

Attributs et méthodes de classe

Une classe est un type et chaque objet de la classe détient sa propre copie des données membres. Il peut être intéressant que des objets d'une même classe partagent certaines données. On peut utiliser à cette fin des variables globales. Cependant il est généralement plus judicieux et plus élégant que ces données soient déclarées comme faisant partie de la classe : ce sont les attributs de classe.

On utilise le mot clé **`static`** pour déclarer un attribut de classe.

Revenons sur la classe Etudiant précédemment déclarée. On désire compter le nombre d'instances de classe (i.e. le nombre d'objets de cette classe). Pour cela, on va déclarer un attribut de classe compteur à la classe Etudiant qui sera incrémenté à la création d'un objet et décrémenté à la destruction.

Cet attribut de classe doit pouvoir être récupéré depuis l'extérieur mais il ne faut surtout pas qu'on puisse le modifier : on le déclare donc en private et on définit une méthode d'accès getNbEtudiants() qui renvoie la valeur du compteur. Cette méthode ne doit pas être détenu par chaque instance de la classe Etudiant, c'est une méthode partagée par tous les objets de la classe Etudiant : c'est une méthode de classe, déclarée elle aussi avec le mot clé **`static`**.

```
// Déclaration de la classe Etudiant
	class Etudiant
	{
		private:
		 static int compteur; // Attribut de classe
		 double moyenne;      // Attribut d'instance
```

```
        public:

       // Méthode de classe qui renvoie le nombre d'instances de la classe
        static int getNbEtudiants();

        // Constructeur sans paramètre (constructeur par défaut)
         Etudiant() {
               moyenne = 0;
               compteur++; // Incrémenbtation du compteur d'instances
         }

        // Constructeur avec un paramètre
        Etudiant(double note) {
               moyenne = note;
               compteur++; // Incrémenbtation du compteur d'instances
        }

        // Destructeur
        ~Etudiant() {
               compteur--;
        }

        // Méthodes d'accès à l'attribut (d'objet) moyenne
        double getMoyenne() { return moyenne; }
        void setMoyenne(double note) { moyenne = note; }
};
```

Dans cette déclaration de la classe Etudiant, le membre statique compteur est seulement déclaré et non défini. Il faut que la définition (et en particulier son initialisation qui est ici essentielle) apparaissent quelque part dans le programme (en général dans le .cpp lié à la classe), et de façon unique :

```
// Définition de l'attribut de classe
int Etudiant::compteur = 0;

// Définition la méthode de classe
int Etudiant::getNbEtudiants() { return compteur; }
```

Pour accéder à la méthode de classe getEtudiant dans le programme, on la désignera par Etudiant::getNbEtudiants() :

```
int main()
{

    Etudiant bob, john, jenny;
    cout << Etudiant::getNbEtudiant() << " ";

    Etudiant * ptr_etudiant = new Etudiant;
    cout << Etudiant::getNbEtudiant() << " ";

    delete ptr_etudiant;
    cout << Etudiant::getNbEtudiant() << endl;
    return 0;
}
```

Ce petit programme affichera donc sur la sortie standard : **3 4 3**.

Attention, **une méthode `static` ne peut accéder qu'aux attributs `static` de la classe**.
Il faut bien distinguer membre d'instance (que l'on désigne souvent abusivement par membre) et membre de classe. Un membre est un attribut (une donnée membre) ou une méthode (une fonction membre).