4/ Introduction à la Programmation Objet avec le langage C++

4.1/ La conception orientée objet

Avant de concerner la programmation à proprement parler, l'orientation objet consiste, et a historiquement consisté, en une **méthode de conception** de logiciels.

En effet, les développeurs ont très tôt cherché des méthodologies visant à rendre plus efficaces les différentes étapes rencontrées lors de la production de logiciels.

Voici quelques une des principales questions qui se posent aux concepteurs de logiciels :

- Comment modéliser les situations réelles que l'on a décidé de gérer à l'aide de logiciels ?
- Comment, à l'instar d'autres domaines comme la mécanique ou l'électronique, produire des composants réutilisables ?
- Comment assurer la qualité et la conformité des logiciels ?

La conception orientée objet vise précisément à apporter des réponses aux trois problématiques exprimées par les questions ci-dessus.

4.2/ Le concept d'objet

a/ Le concept d'objet de la vie courante

C'est un concept familier de la vie courante. Un objet a une certaine existence matérielle (même si l'on peut imaginer des objets complètement abstraits) de laquelle découle des propriétés et des comportements. Un objet peut être plus ou moins abstrait ou concret, plus ou moins générique ou particulier.

Prenons l'exemple d'un objet voiture. Je peux parler de la voiture en général. Dans tous les cas j'aurai un objet comportant des roues, des portières avant, un volant. C'est là un cadre général pour parler d'une voiture mais cela ne définit pas encore une voiture particulière.

Si maintenant je m'intéresse à un modèle particulier de voiture, je saurai si elle a un levier de vitesses, des portes arrières, un système de climatisation, etc.

La voiture a aussi des propriétés et elle a un comportement qui résulte des actions effectuées sur elle par un utilisateur appelé conducteur. L'interaction entre l'utilisateur et la voiture est bien définie. Le concepteur du véhicule a prévu des dispositifs (pédales, levier de vitesse, divers boutons, poignées, etc) permettant à l'usager d'utiliser la voiture.

b/ Le concept d'objet en conception logicielle

La vision historique d'un programme informatique est celle d'un déroulement purement séquentiel d'instructions tel qu'il a lieu au niveau du microprocesseur. Si l'on considère un tel programme séquentiel, nous aurons une succession temporelle d'opérations de types très divers : les opérations d'entrée/sortie diverses (par exemple lecture du clavier, écriture dans un fichier) seront mélangées avec les opérations d'accès mémoire, de calculs sur les variables, etc.

C'est ce qui se passe avec la programmation traditionnelle en langage C et c'est pourquoi un grand programme écrit en C n'est pas toujours facile à comprendre, il réalise ses opération de manière « mélangée » même si une bonne découpe en sous-programmes permet de clarifier les choses.

Or, et cela dès le moment initial de la spécification de ce que doit faire le programme, nous avons besoin d'une vision plus fonctionnelle, d'une vision du **point de vue de l'utilisateur**. Au final, un programme va gérer des problématiques réelles en relation avec des utilisateurs. Nous avons donc besoin d'une représentation qui identifie, non pas des suites purement séquentielles d'instructions à effectuer, mais des **éléments à manipuler**.

Par exemple, si nous prenons le cas d'un logiciel de facturation, nous pourrons définir un certain nombre d'« **objets »** qui interagissent : des clients, des fournisseurs, des commandes, des articles, des factures etc.

La vision sous forme d'objets s'est révélée très efficace tant du point de vue de la spécification des systèmes logiciels que du point de vue de leur conception (description générale) et de leur implémentation (programmation).

c/ Le concept d'objet dans les programmes

Que les programmes soient plus ou moins concrets (développement de l'application de gestion d'une entreprise, de commande d'un robot) ou abstraits (programmation système, scientifique) la programmation orientée objets offre une meilleure découpe fonctionnelle des programmes, une plus grande clarté et une plus grande réutilisabilité d'éléments logiciels (on est passé de la création / utilisation de bibliothèques de fonctions à des bibliothèques d'objets et même au dialogue entre composants logiciels distribués).

4.3/ La Programmation orientée objet

La partie 4.1/ concernait la conception orientée objet. Nous nous intéressons maintenant à l'utilisation d'objets dans les programmes à proprement parler.

4.3.1/ Définition d'un objet

Un objet est une entité logicielle qui **associe des données et des traitements sur ces données**.

On accédera aux données de l'objet et à ses traitements par l’intermédiaire du nom de l'objet. Nous aurons alors déjà ainsi un regroupement fonctionnel.

Dans notre étude du langage C nous avons vu la notion de **structure**. La structure du langage C

permet d'associer des données de divers types mais qui sont relatives à un élément plus complexe bien déterminé.

Exemple :

Par exemple imaginons que nous nous intéressons au domaine du calcul matriciel. Une matrice est un ensemble de données numériques représenté sous forme de tableau(x) ayant une dimension et des tailles. Par exemple je peux avoir une matrice de dimension 2, dont la taille est 5x6. En programmation je devrai aussi définir la manière dont les données numériques seront stockées en mémoire. Je peux utiliser un tableau C à deux dimensions comme :

float tab[5][6] ;

ou je peux préférer un tableau simple :

float tab[5*6] ;

dans lequel je vais gérer les lignes sous forme d'offsets (chaque ligne est distante de 6 caractères de la ligne suivante.

En programmation objet je vais définir un type Matrice dont l'**interface** masquera les détails de l'**implémentation**. Ainsi le programmeur qui utilisera l'objet Matrice pourra utiliser les traitements (=fonctions) de l'objet Matrice sans se soucier de la manière dont celle-ci est réalisée.

Le mot **interface** est très important en programmation objet. En effet on veut séparer l'interface (comment l'objet sera utilisé) de l'**implémentation** (comment l'objet est réalisé « en interne »)

Si on reprend l'exemple de l'objet voiture vu en introduction, on sait bien qu'une voiture possède des éléments techniques comme le moteur, le système de direction, d'injection d'essence etc. Mais on veut assurer que le conducteur n'ait pas à manipuler directement ces éléments pour conduire la voiture. L'interface entre la voiture et le conducteur sera matérialisée par le volant, la clé du démarreur, les pédales etc. On peut très bien conduire la voiture sans savoir combien de cylindres a son moteur et même sans rien savoir des notions de cylindres et de pistons.

Revenons maintenant à l'exemple des matrices.

En langage C, nous pouvons représenter une matrice à l'aide d'une structure, comme suit :

```
typedef struct {
  int nb_lines;
  int nb_cols;
  float *stockage;
} Matrice ;
```

et nous pouvons manipuler des « objets » de type Matrice en créant des fonctions dont les signatures pourront être :

a/

Matrice * init_matrice(int lines, int cols, Matrice *m) ;

init_matrice prend en arguments le nombre de lignes et de colonnes voulues et un pointeur sur une variable, déjà existante, dont le type est la structure Matrice (mais dont aucun stockage n'a encore été réservé). La fonction va réserver l'espace mémoire nécessaire au stockage des nombres réels (type float), stockage qui sera accessible dans la structure grâce au pointeur stockage. En cas de succès, la fonction retournera le pointeur sur la matrice donnée en argument (le même pointeur donc que celui donné en troisième argument), elle retournera un pointeur NULL en cas d'erreur.

b/ void delete_matrice(matrice *m) ;

delete_matrice permet de « détruire » la matrice quand elle n'est plus utile. Le rôle de delete_matrice sera de libérer l'espace de stockage qui aura été réservé pour la matrice

c/ void aff_matrice(matrice *m) ;

Fonction pour afficher les valeurs stockées dns la matrice.

d/ float get_val(int ligne, int colonne, matrice *m) ;

cette fonction est utilisée pour lire un élément de la matrice pointée par m. elle renvoie la valeur de l'élément situé à la position ligne, colonne ou un code à définir en cas d'erreur.

e/ float set_val(int ligne, int colonne, matrice *m, float val) ;

cette fonction est utilisée pour affecter la position ligne, colonne de la matrice m avec la valeur val. La valeur val est renvoyée en cas de succès et en cas d'erreur un code à définir est renvoyé.

Dans cet exemple, nous avons donc une structure matrice et des fonctions pour manipuler les objets basés sur cette structure. Nous pouvons dès lors créer, manipuler, détruire des matrices sans toucher directement à la structure de données mais uniquement grâce à l'i**nterface** représentée par les fonctions.

4.3.2 / Les objets avec le langage C++

a/ L'élément class

Le langage C++ introduit l'élément **class** qui permet de regrouper des variables de différents types (tout comme l'élément struct du langage C) et des traitements sur ces variables.

Voici l'exemple de définition d'une classe permettant de décrire un élément géométrique en deux dimension :

On définit d'abord une classe « point » :

```
class Point {
        public :
        // Constructeur
        Point(int x, int y) {X = x, Y = y } ;

        // Constructeur de copie
        Point(const Point& unPoint) ;

        // Méthode pour afficher le point en couleur en donnant ses trois composantes rouge, vert,
```

```
        // bleu
        Set(int rouge, int vert, int bleu) ;

        // Données de la classe
        int X ;
        int Y ;
        Color Couleur;
}

class Forme {
        public:
        // Constructeur
        Forme(Point p) {Pref = p};

        // Méthodes
        virtual void Translation(int offsetX, int offsetY) {Pref.x += offsetX; Pref.y += offsetY ; }
        virtual void Rotation(int angle) {};
        virtual void Dessine() ;

        protected:
        Point Pref; //Point de référence définissant la position de la forme dans l'espace de travail
        Color Couleur ;
};
```

Les fonctions membres peuvent être définies à l'intérieur de la classe (comme ici les fonctions Translation() et Rotation(), cette dernière ne faisant rien pour cette classe de base) ou elles peuvent être seulement déclarées (comme ici la méthode Dessine() ou comme le constructeur de copie de la classe Point).

Une fonction membre peut être définie à l'extérieur de la déclaration de classe par la notation suivante :

```
type nomClasse::fonction_membre( arguments ) {
 ...
}
```

par exemple (si l'on ne l'avait pas déjà fait dans la définition de la classe) :

```
Forme::Translation(int offsetX, int offsetY)
```

```
{
  Pref.X += offsetX;
  Pref.Y += offsetY ;
}
```

Rq : Un constructeur (de même qu'un destructeur) ne renvoie jamais de valeur.

Notion de destructeur : De même qu'une classe a un constructeur (ou plusieurs, voir plus loin la notion de **surcharge**), elle peut avoir un destructeur. Dans le cas présent de la classe Forme, le destructeur serait déclaré par l'écriture suivante :

```
~Forme() ;
```

Un destructeur est utilisé par exemple pour libérer la mémoire réservée préalablement dans des méthodes de la classe ou pour fermer les fichiers que la classe à ouvert. Ici, dans la classe de base Forme, nous n'en avons pas besoin. Mais si nous créons une classe Matrice pour mettre en œuvre l'exemple vu plus haut qui est basé sur une structure contenant un pointeur sur une zone de stockage dynamique, alors nous aurons besoin d'un destructeur destiné à libérer l'espace réservé préalablement (dans le constructeur).

Ce destructeur pourra être défini ainsi :

```
Matrice::~Matrice()
{
  if (stockage!= NULL)
        delete stockage ; // c'est l'équivalent de l'instruction free(stockage) du langage C
}
```

Remarque : tout comme un constructeur, un destructeur ne renvoie pas de valeur.

Exercice : en reprenant l'exemple de la matrice, écrire la définition d'une classe matrice.

b/ Instanciation d'une classe

Les objets réellement créés et utilisés dans un programme à partir d'une classe sont des **instances** de la classe. ils ont une existence en mémoire au moment de l'exécution du programme.

Ex :

```
Point unPoint(5,7);
Point unAutre(unPoint); // un second objet point au même endroit
Forme *pt;
pt = new Forme(unPoint); /* Création dynamique, necessitera une instruction delete pt ; */
```

4.3.3/ Héritage et polymorphisme

Ce sont là deux notions essentielles de la programmation orientée objet.

a/ Héritage

La classe Forme vue dans l'exemple précédent est bien trop limitée pour être utile. Mais elle fournit une base de travail pour déclarer et définir des formes géométriques réelles. La classe Forme n'est qu'une classe de **base** à partir de laquelle nous allons créer de nouvelles classes représentant des formes géométriques.

Les classes créées à partir d'une classe de base sont appelées **classes dérivées**. Elles dérivent de la forme de base par la notion d'**héritage**.

Une classe dérivée hérite de toutes les données et méthodes publiques et protégées (protected) de sa classe de base.

Voici la notation pour définir une classe dérivée sur l'exemple d'une classe Rectangle dérivée de la classe de base Forme :

```
class Rectange : public Forme {
        // Constructeurs
        Rectangle(Point topLeft, Point bottomRight) {
                Pref = topLeft ;
                BottomRight = bottomRight ;
        }

        virtual void Translation(int offsetX, int offsetY) ;
        virtual void Rotation(int angle);
        virtual void Dessine() ;

        // Données
        Point BottomRight ;
 }
```

La classe Rectangle hérite donc de la donnée Pref (point de référence) défini dans la classe Forme. Cette donnée peut donc être utilisée sans être redéclarée dans la classe dérivée Rectangle.

La classe Rectangle hérite aussi des trois méthodes Translation(), Rotation() et Dessine() mais ses

fonctions de base ne sont pas suffisantes pour la classe Rectangle. C'est pourquoi la classe Rectangle **redéfinit** ces trois méthodes. Cette redéfinition est possible car les fonctions ont été déclarées **virtual** dans la classe de base.

Les trois fonctions n'ayant été que déclarées à l'intérieur de la définition de la classe, elles devront encore être définies à l'extérieur. Par exemple :

```
void Rectangle ::Translation(int offsetX, int offsetY)
 {
   Pref.X += offsetX ;
   Pref.Y += offsetY ;
   BottomRight.X += offsetX ;
   BottomRight.Y += offsetY ;
 }
```

De même il faudra définir les fonctions pour effectuer la rotation du rectangle et pour le dessiner.

b/ Polymorphisme

C'est dans le polymorphisme que réside l'une des grandes forces de la programmation orientée objet. Dans notre exemple des formes géométriques, supposons que j'ai un conteneur (un tableau) de diverses formes à translater puis à afficher.

A partir de la classe de base Forme ont été dérivées par exemple des classes Rectangle, Triangle et Cercle. Toutes ont redéfini adéquatement les méthodes Translation(int offsetX, int offsetY) et Dessine().

Supposons que j'ai déclaré un conteneur, c'est à dire ici un tableau pour y mettre des objets géométrique, c'est à dire des **instances** des classes Rectangle, Triangle et Cercle :

```
Forme * lesFormes[] ;
```

Comme on le voit, mon tableau est déclaré comme un tableau de pointeurs sur des objets de la classe de base Forme, qui est la classe la plus générique pour mes formes géométriques. C'est précisément cette généricité qui me permettra d'écrire un algorithme qui marchera ensuite quelles que soient les formes réellement présentes dans mon conteneur.

Supposons que mon conteneur ait ensuite été rempli de différentes instantes de formes par une fonction LireFormes() qui, par exemple, lit les définitions de formes dans un fichier.

```
nbFormes = lireFormes(lesFormes) ; // lit la description des formes et créé dynamiquement les
```

// instances des formes en mémoire

Maintenant pour effectuer une certaine translation sur toutes les formes du conteneur :

```
int i=0 ;
Forme *pt ;
while ( i < nbFormes) {
        pt = lesFormes[i] ;
        pt->Translation(x,y) ;
        i++ ;
}
```

Et pour afficher les formes :

```
i=0 ;
while ( i < nbFormes) {
        pt = lesFormes[i] ;
        pt->Dessine();
        i++ ;
}
```

Dans cet exemple de déplacement et d'affichage, l'algorithme travaille sur des éléments de type déclaré Forme, qui est le type de la classe de base. Or **à l'exécution** du programme, le tableau des instances de Forme contiendra des éléments de types dérivés comme Rectangle, Triangle ou Cercle. Et ce seront bien les méthodes Translation() et Dessine() **redéfinies par ces classes dérivées** qui seront invoquées.

C'est cela qui est appelé **polymorphisme**, cette possibilité de définir un programme travaillant sur des objets d'une classe de base mais qui s'appliquera ensuite, lors de l'**exécution**, à des éléments dérivés plus évolués ayant des méthodes spécialisées qui auront cependant gardé le même nom, et la même signature, que les méthodes correspondantes de la classe de base. Ces méthodes homonymes ont donc **plusieurs formes**.

4.3.4/ Autres spécificités du langage C++

a/ Surcharge d'opérateurs et de fonctions

Une des autres spécificités les plus intéressantes du C++ est la **surcharge d'opérateurs et de fonctions**.

Comme exemple de surcharge d'opérateurs, il y a les opérateurs d'Entrées / Sorties très pratiques du C++ que sont << (écriture dans un flux) et >> (lecture dans un flux). En effet, en C++ les Entrées / Sorties sont d'une écriture bien plus simples qu'avec les fonctions prinf et scanf du langage C car les

opérateurs << et >> s'adaptent spontanément aux types des arguments. Ces opérateurs sont **surchargés** :

par exemple :

```
int i ;
float f ;
char chaine[50] ;

cout << "Entrez une chaine de caractères" ;
cin >> chaine ;
cout << Entrez un entier ;
cin >> i ;
cout << Entrez un nombre réel ;
cin >> f ;
cout << "Vous avez entré la chaine :" << chaine << ", l'entier :" << i << "et le réel :" << f ;
```

Cela semble très naturel. En fait les opérateurs << et >> ont été définis pour travailler avec un flux comme membre de gauche et **différents types** comme membre de droite. L'opérateur renvoie le flux, ce qui permet d'enchaîner les opérations comme vu ci-dessus.

Il est possible pour le programmeur de redéfinir quasiment tous les opérateurs pour les adapter à différents types d'objets.

Il est aussi possible et courant de **surcharger les fonctions**, en particulier les méthodes de classe. Une méthode est surchargée quand elle est **définie plus d'une fois avec un nombre et/ou des types d'arguments différents.**

Par exemple :

Supposons que j'ai une classe Matrice qui peut être construite de différentes manières : soit la matrice sera initialement vide, soit elle sera lue dans le constructeur depuis un flux (un fichier), soit elle sera initialisée par recopie d'une autre matrice déjà existante.

J'aurai donc les trois constructeurs suivants pour ma classe Matrice :

```
void Matrice::Matrice() {…}      // matrice vide au départ
void Matrice::Matrice(int nbLines, int nbCols, ifstream &f) ;   // matrice nbLine * nbCols qui sera
                                                                // lue depuis le flux d'entrée f
void Matrice::Matrice(Matrice &m) ;     // la matrice sera initialisée à partir de la matrice m
```

On le voit, le constructeur de la classe Matrice est défini trois fois. Il porte le même nom lors de chaque **surcharge** mais le compilateur peut faire la différence entre les trois fonctions grâce au nombre et aux types des arguments.

b/ Il y a encore d'autres spécificités du langage C++ mais qui sortent du cadre de ce cours :

Les Templates qui sont des généralisations de traitements (des fonctions) s'appliquant sur différents types de classe, les classes n'étant pas explicités dans la définition de la fonction.

Le langage C++ permet de définir facilement (grâce aux Templates évoqués ci-dessus) des **Conteneurs** qui sont des collections dynamiques d'objets et que l'on manipule grâce à des **Itérateurs**.

Les opérateurs d'allocation de mémoire dynamique du langage C ont été remplacés :

new et **delete** remplacent malloc et free

```
int pi = new int;
char *pc = new char[15];
…..
delete pi;
delete [] pc;
```