Concepts et Implémentation du Modèle Objet : 1ère partie

Encapsulation, héritage et polymorphisme

Implémentation des principes de base du modèle objet[1]

La Programmation Orientée Objet (POO) repose sur trois grands principes : l'encapsulation, l'héritage et le polymorphisme

Dans ce TP, il faut essayer de comprendre et de respecter ces principes en proposant pour chacun des solutions d'implémentation en langage C. Les premiers compilateurs C++ étaient des « cfront » frontaux qui généraient du C. Sur un exemple simple d'objet graphique avec cercles et rectangles, vous écrirez un programme principal permettant de tester les solutions d'implémentations qui suivent étape par étape, ces 3 grands principes.

1. L'encapsulation

Exigences du modèle objet

L'encapsulation est orientée selon deux axes :

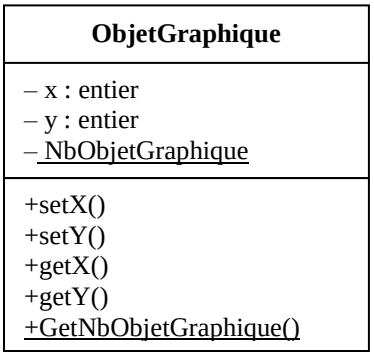
- regrouper les données et le code associé
- protéger les données

La protection des données est plus du ressort du Parser que du générateur de code, donc ne sera pas traitée ici ; il ne s'agit en effet que de contrôler que l'accès aux variables est valide selon les droits associés au contexte. Nous ferons comme si les attributs étaient privés, en définissant pour chacun un accesseur et un mutateur « public ».

Le traitement des messages en C++ passe par l'appel explicite des méthodes, et les délégations de message ne sont pas transparentes, donc ne seront pas non plus traitées ici.

Exemple d'utilisation pour ce TP

On se propose d'utiliser ce principe sur la classe suivante :



[1]Tutorial conçu à partir du cours d'Objet avancé de D.Hill (+ rédaction de Pierre Chatelier - F2 maintenant Docteur en informatique).

Solution proposée et à implémenter

Le regroupement des données et fonctions en C pose problème. Pour les variables, il est aisé de les ranger dans une structure, mais que faire des méthodes ? Il faut passer par des pointeurs de fonction, mais il n'est pas envisageable d'avoir des copies de tous ces pointeurs pour *chaque* instance de la classe ; ce n'est même pas de l'optimisation, c'est presque une exigence du modèle.

A une classe d'objet est donc associée une méta-classe, partagée par toutes les instances de cette classe, et qui regroupe les méthodes, les attributs de classe et les méthodes de classe.

La structure proposée pour l'objet n'a donc plus besoin de contenir les pointeurs de fonctions des méthodes, mais un simple pointeur vers sa méta-classe.

La méta-classe est instanciée une et une seule fois pour toute utilisation de la classe.

Implémentation correspondante en langage C

Classe ObjetGraphique

```
struct  ObjetGraphique
{
        struct MetaObjetGraphique* myClass ;
        int x ;
        int y ;
} ;
```

Classe MetaObjetGraphique

```
struct MetaObjetGraphique
{
   /* méthodes d'instances */
   void (*setX)(int, this[1]) ;
   void (*setY)(int, this[1]) ;
   int  (*getX)(this[1]) ;
   int  (*getY)(this[1]) ;

   /* attributs de classe */
   int NbObjetGraphique ;

   /* méthodes de classe */
   int (*GetNbObjetGraphique)(void)

   /* constructeur  de la classe */
   void(*ConstructeurObjetGraphique)(this[1])
} ;


1 :
NB « this » n'existe pas en C, c'est une simplification d'écriture dans ce tutorial. Il faudra
coder :

struct ObjetGraphique* this

On peut aussi utiliser un typedef dans la déclaration des structures pour simplifier
l'écriture et éviter de trainer des 'struct' à chaque déclaration.
```

Remarquons :

- l'apparition systématique du pointeur *this*, qui permet à la méthode de connaître l'objet sur lequel elle travaille.
- l'apparition de la méthode *ConstructeurObjetGraphique()*, qui est appelée à chaque instanciation de la classe ObjetGraphique, et qui a le rôle du constructeur et qui incrémentera le nombre d'objets graphiques (attribut de classe). Elle permettrait notamment de donner des valeurs par défaut à *x* et *y*, par exemple, si on les passait en paramètres, mais surtout d'initialiser le pointeur *ObjetGraphique.myClass* à l'adresse de l'objet qui instancie la méta-classe. On appellera cet objet global LeMetaObjetGraphique.
- la nécessité d'un constructeur pour la méta-classe. Ce constructeur est une fonction globale, qui permettra d'initialiser LeMetaObjetGraphique.NbObjetGraphique à 0 et les pointeurs de fonctions de LeMetaObjetGraphique aux bonnes adresses.

Le programme principal doit mettre en place tous les « métaobjets » et initialiser leurs structures pour que l'on puisse ensuite manipuler les instances. On pourra éventuellement mettre en place des fonctions pour regrouper le travail à faire pour construire les les « métaobjets » de chaque « classe ».

Dans cette implémentation, on cherchera un codage fonctionnel sans se soucier des différentes optimisations possibles.

Exemple de code de méthodes d'instances

On note le passage du 'this' pour ces méthodes. On donne ci-dessous à titre d'indication le code de *getX(…)* et *setX(…)*, qui sont similaires à ceux des autres accesseurs/mutateurs. Pour le code ci-dessous, on peut définir avec un typedef que *struct ObjectGraphique* se nomme *ObjetGraphique* pour éviter une écriture un peu lourde du code.

```
int getX(struct ObjetGraphique* this)
{
	return this->x;
}

void setX(int inX, struct ObjetGraphique* this)
{
	this->x = inX;
}
```

2. L'héritage

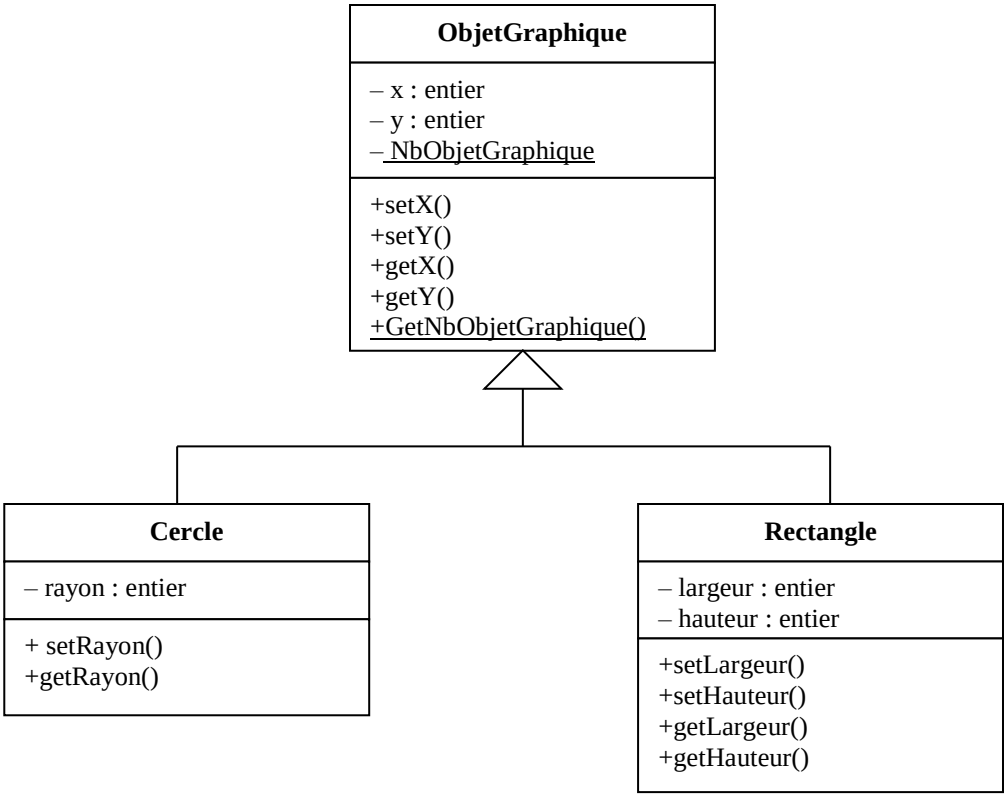
Exigences du modèle objet

Une classe dérivée contient au moins tous les attributs/méthodes de sa classe mère, et on peut en spécifier de nouveaux, qui resteront inconnus pour cette classe mère.

Nous ne nous soucierons pas des types de dérivation (public/private/protected) qui ne sont pas vraiment du ressort du générateur de code.

Exemple d'utilisation

On se propose d'utiliser ce principe sur les classes suivantes :



Remarque : nous verrons dans la prochaine partie qu'ici, nous choisissons que (x,y) représente pour le Cercle son centre, et pour le Rectangle son coin supérieur gauche.

Solution proposée pour l'implémentation en TP

On utilise une structure pour la classe dérivée, agrégeant une structure de la classe mère. La méta-classe dérivée contient un pointeur vers la méta-classe mère. (On évoquera dans l'implémentation un problème de redondance d'information).
Il s'agit d'héritage *statique*.

Implémentation correspondante en C

Classe Cercle

```
struct  Cercle
{
   struct ObjetGraphique   superClasse ;
   struct MetaCercle     * myClass ;
   int                     rayon ;
} ;
```

Classe Rectangle

```
struct  Rectangle
{
   struct ObjetGraphique   superClasse ;
   struct MetaRectangle  * myClass ;
   int                     largeur ;
   int                     hauteur ;
} ;
```

Classe MetaCercle

```
struct MetaCercle
{
   struct MetaObjetGraphique * superMetaClasse ;

   /* méthodes d'instances */
   void (*setRayon)( int, this[1]) ;
   int  (*getRayon)(this[1]) ;

   /* constructeur  de la classe */
   void (*ConstructeurCercle)(this[1])
} ;

1 :struct Cercle* this
```

Classe MetaRectangle

```
struct MetaRectangle
{
   struct MetaObjetGraphique * superMetaClasse ;

   /* méthodes d'instances */
   void (*setLargeur)(int, this[1]) ;
   void (*setHauteur)(int, this[1]) ;
   int  (*getLargeur)(this[1]) ;
   int  (*getHauteur)(this[1]) ;

   /* constructeur  de la classe */
   void (*ConstructeurRectangle)(this[1])
} ;

1 :struct Rectangle* this
```

Remarquons :

- La superclasse est placée comme premier champ de la structure de la classe dérivée, cela permettra de faire des *cast* de type **`(ObjetGraphique*) &Cercle`**, qui marcheront.
- Avec cette technique, il y a redondance d'information. En effet, le cercle pointe à la fois sur sa méta-classe via son champ *myClass*, et sur la méta-classe de sa classe mère via son champ *superClasse.myClass*. Or, on fait aussi pointer MetaCercle vers MetaObjetGraphique.

On pourrait empêcher cela au prix d'une optimisation non triviale à base de pointeurs (void*) que l'on casterait. Un seul pointeur pourrait donc tantôt servir à pointer la Méta-classe de la classe mère, tantôt vers la Méta-classe de la classe dérivée. Le jeu n'en vaut pas la chandelle ici, car on ne cherche pas à optimiser à tout prix, mais au contraire à voir les concepts et les chaînages de façon explicite.

Pourquoi fait-on pointer MetaCercle vers MetaObjetGraphique ? Si comme pour les objets on stockait directement la superMetaClasse, on aurait une redondance d'information rapidement importante dans toutes les sous-classe. Le pointeur permet à la fois, parce que cela permet de conserver le lien d'héritage, et ensuite la raison suivante montre l'intérêt d'avoir cette information :

Quand on construit un Cercle, il faut construire "son" ObjetGraphique via le constructeur correspondant. Comment a-t-il connaissance de la fonction à appeler ? En fait, on pourrait écrire directement l'appel à la méthode, car le compilateur la connaît à ce moment là. Nous avons préféré la technique plus lente, certes, mais plus "automatique", de passer par la hiérarchie de classe, et d'appeler alors :

`myClass->superMetaClasse->ConstructeurObjetGraphique(…)`

Le code généré permet ainsi, par exemple, de ne pas avoir à exporter la fonction *ConstructeurObjetGraphique(…)* de `ObjetGraphique.c` vers `Cercle.c` pour qu'il la connaisse.

Exemple de code

Pour indication, voici le code correspondant utilisé dans notre constructeur du cercle:

```
void ConstruireCercle(struct Cercle* this)
{
	printf("Cercle::ConstruireCercle\n");
	this->myClass = &LeMetaCercle;

	/* Ci on a besoin du & car la superclasse embarquée dans une sous-classe  */
	/* est une structure complète et pas un pointeur comme dans la metaclasse */
	this->myClass->superMetaClasse->ConstructeurObjetGraphique(&this->superClasse);

	printf("Construction partie propre au Cercle\n");
	/* Initialisation rayon par exemple */

	printf("Cercle::ConstruireCercle effectué\n");
}
```

3. Le polymorphisme

Exigences du modèle objet

Le polymorphisme est un outil très puissant qui se caractérise par la cohabitation de fonctions de même noms mais dont l'appel dépend du contexte et est résolu par le compilateur lui-même, voire même lors du run-time.

Nous ne traiterons que des méthodes virtuelles, mais pas de la surcharge, qui elle peut simplement être résolue par le Parser.

Le comportement que l'on veut obtenir pour les méthodes virtuelles est le suivant:

- On veut pouvoir, sans la redéfinir, utiliser une méthode virtuelle de la classe mère
- On veut pouvoir redéfinir cette méthode et que ce soit la bonne qui soit appelée:

Traduction de ces exigences:

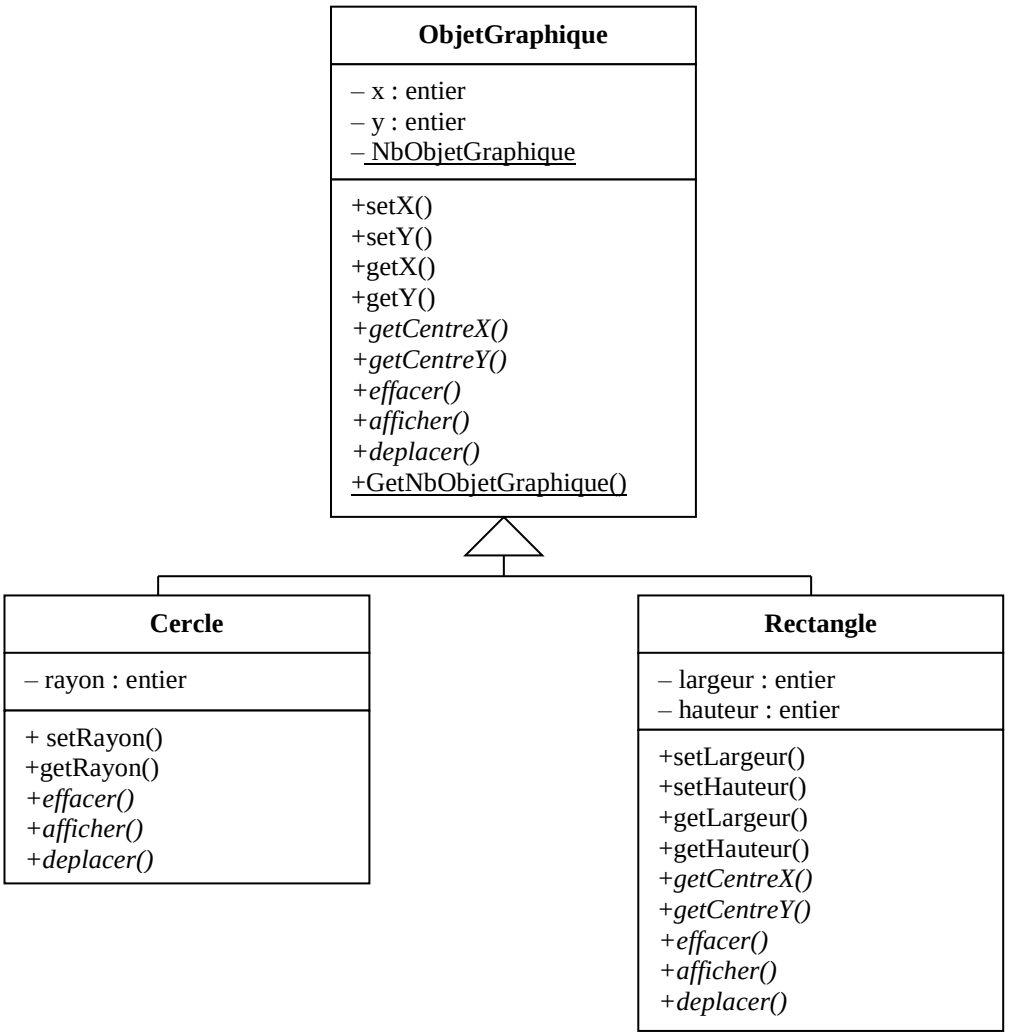
Soit C_M un objet de classe mère, pC_M un pointeur vers C_M, C_D un objet dérivé, pC_D un pointeur vers cet objet; soit f une méthode virtuelle de C_M, non redéfinie pour C_D, et g une méthode virtuelle de C_M, redéfinie pour C_D

Alors on veut que :

-C_M.f(), pC_M->f() soient valides et appellent f
-C_D.f(), pC_D->f() soient valides et appellent f
-C_M.g(), pC_M->g() soient valides et appellent C_M::g()
-C_D.g(), pC_D->g() soient valide et appellent C_D::g()
-si pC_M reçoit pC_D (ce qui est correct en modèle objet), pC_M->g() appelle C_D::g()

Exemple d'utilisation

On se propose d'utiliser ce principe sur les classes suivantes :



- getCentreX() et getCentreY() sont redéfinies pour le rectangle, pour lequel x et y ne représentant que le coin supérieur gauche, il faut tenir compte de la largeur et de la hauteur pour trouver le centre de la figure.
- effacer(), afficher() et deplacer() sont virtuelles pures pour ObjetGraphique, et définies pour Cercle et Rectangle.

Solution proposée pour le TP

Le fait qu'un pointeur vers ObjetGraphique puisse appeler Cercle::effacer() si ce pointeur contient l'adresse d'un Cercle implique que d'une façon ou d'une autre, un ObjetGraphique est renseigné sur son type réel. Nous nous proposons donc de rajouter un attribut *type* à cette classe, pouvant contenir une constante caractérisant soit un Cercle, soit un Rectangle, soit un ObjetGraphique non dérivé.

Un autre problème se pose : selon le type d'objet graphique, plusieurs fonctions *effacer()* sont possibles. Or, la Méta-classe n'en contient *a priori* qu'une.

Il faut donc rajouter quelque chose aux solutions utilisées jusqu'à maintenant.

On pourrait choisir de mettre le pointeur de fonction dans l'objet et non dans sa méta-classe. La construction permettrait d'initialiser convenablement ce pointeur. Mais ceci trahit la conception à laquelle on s'est tenu jusqu'alors.

Une autre façon de faire consisterait à utiliser un tableau de fonctions *effacer()* dans la méta-classe, et choisir la bonne selon l'objet appelant; c'est le principe de table des méthodes virtuelles. Appelons celle-ci *TVMeffacer*

Un autre choix est alors à faire : prenons la méthode *effacer()*. On peut :

- soit disposer d'un pointeur de fonction dans la classe *MetaCercle*, référençant une case du tableau *MetaObjetGraphique.TVMeffacer*, sachant que dans *MetaRectangle*, un pointeur similaire référence une autre case
- soit mettre plutôt un pointeur de fonction *effacer(this)* directement dans *MetaObjetGraphique*, et que cette fonction choisisse ensuite la case idoine de la TVM pour appeler le code d'effaçage, selon le type d'objet qu'on lui passe *via* `this`.

De toutes manières, cette dernière façon de faire est bien pratique pour traiter le cas **`pObjetGraphique->effacer()`**.

Comme elle est en outre plus économique en place, elle nous semble meilleure à tous points de vue. La perte en rapidité peut même être réduite avec une étape supplémentaire : traiter cette fonction *effacer(this)* qui choisit la bonne case de la TVM comme une fonction *inline*.)

Sinon, sans tenir compte de la place, si on se permet de rajouter des pointeurs *effacer* dans les méta-classes dérivées, alors pourquoi ne pas directement les faire pointer la méthode réelle "*effacerCercle()*" ou "*effacerRectangle()*" ? On limiterait l'usage de la TVM au seul traitement du cas **`pObjetGraphique->effacer()`**.

On remarquera aussi que le remplissage de la TVM dépend du fait que la fonction soit virtuelle pure ou non; en effet, pour une fonction virtuelle pure, aucune méthode réelle n'est associée à la classe mère qui peut juste avoir un pointeur NULL que l'on retrouve dans les spécifications de code C++ par exemple : « `virtual void fonctionVirtuelle() = 0` ».

Implémentation correspondante en C

```
/* deux dérivations possibles */
#define NBCLASSES 2
typedef enum OG_t {CERCLE=0, RECTANGLE=1, OG=2} OG_t;
```

Classe ObjetGraphique

```
struct  ObjetGraphique
{
  struct MetaObjetGraphique * myClass ;
  int                         x ;
  int                         y ;
  OG t                        type;
} ;
```

Classe MetaObjetGraphique

```
struct MetaObjetGraphique
{
       /* méthodes d'instances */
       void (*setX)(int, this[1]) ;
       void (*setY)(int, this[1]) ;
       int (*getX)(this[1]) ;
       int (*getY)(this[1]) ;

       /* TVMs */
       void (*TVMeffacer[NBCLASSES])(this[1]);
       void (*TVMafficher[NBCLASSES])(this[1]);
       void (*TVMdeplacer[NBCLASSES])(this[1]);
       int (*TVMgetCentreX[NBCLASSES])(this[1]);
       int (*TVMgetCentreY[NBCLASSES])(this[1]);

       /* pointeurs pour appeler les méthodes */
       void (*effacer)(this[1]);
       void (*afficher)(this[1]);
       void (*deplacer)(this[1]);
       int (*getCentreX)(this[1]);
       int (*getCentreY)(this[1]);

       /* attributs de classe */
       int NbObjetGraphique ;

       /* méthodes de classe */
       void (*GetNbObjetGraphique)(void)

       /* constructeur  de la classe */
       void (*ConstructeurObjetGraphique)(this[1])
} ;

1 :struct ObjetGraphique* this
```

Classe Cercle

```
struct  Cercle
{
  struct ObjetGraphique  superClasse ;
  struct MetaCercle    * myClass ;
  int                    rayon ;
} ;
```

Classe Rectangle

```
struct  Rectangle
{
   struct ObjetGraphique superClasse ;
   struct MetaRectangle* myClass ;
   int                   largeur ;
   int                   hauteur ;
} ;
```

Classe MetaCercle

```
struct MetaCercle
{
   struct MetaObjetGraphique* superMetaClasse ;

   /* méthodes d'instances */
   void (*setRayon)(this[1]) ;
   int (*getRayon)(this[1]) ;

   /* constructeur  de la classe */
   void (*ConstructeurCercle)(this[1])
} ;

1 :struct Cercle* this
```

Classe MetaRectangle

```
struct MetaRectangle
{
   struct MetaRectangle* superMetaClasse ;

   /* méthodes d'instances */
   void (*setLargeur)(int, this[1]) ;
   void (*setHauteur)(int, this[1]) ;
   int (*getLargeur)(this[1]) ;
   int (*getHauteur)(this[1]) ;

   /* constructeur  de la classe */
   void (* ConstructeurRectangle)(this[1])
};
1 :struct Rectangle* this
```

Remarquons :

- l'introduction des méthodes virtuelles et des TVM ne modifie pas les classes Cercle et Rectangle.
- les méthodes *afficher(), effacer(), deplacer()* pour le cercle sont écrites quelque part sous des noms comme *afficherCercle(), effacerCercle(), deplacerCercle()*, de même qu'il existe *afficherRectangle(), effacerRectangle(), deplacerRectangle()*. C'est d'ailleurs vers ces fonctions que pointent les cases des TVM. Or, devant être rangées dans des tableaux, elles ont le même prototype. Donc, elles ne font référence qu'à des ObjetGraphique. Donc, encore, un *cast* est nécessaire dans leur code.
- Les cases de TVM sont initialisées avec ces fonctions spécifiques après la création du MetaObjetGraphique.

Exemple de code avec la méthode *afficher()*, similaire avec *effacer()* et *deplacer()*:

```
void afficherCercle(structObjetGraphique* this)
{
   struct Cercle* thisCercle = (struct Cercle*) this;
   printf("On affiche le cercle de rayon %d\n", thisCercle->rayon);
}
```

LeMetaObjetGraphique.TVMafficher[CERCLE] contient l'adresse de *afficherCercle()*.

LeMetaObjetGraphique.afficher pointe sur la fonction ci-dessous qui en fonction du type d'objet qui sert d'index dans la TVM, va appeler la bonne fonction :

```
void afficher(struct objetGraphique* this)
{
   return LeMetaObjetGraphique.TVMafficher[this->type](this);
}
```

Comme on l'a fait remarquer plus haut, pour gagner en rapidité, on peut très bien imaginer une étape supplémentaire durant laquelle le générateur de code mette le code ci-dessus *inline* à la compilation. Cette optimisation se réalise au niveau des compilateurs et pas dans ce tutoriel, c'est une optimisation non négligeable.

Les initialisations de pointeurs mentionnées précédemment sont faites dans le constructeur de M*etaObjetGraphique*.

Le constructeur du Cercle devient

```
void ConstruireCercle(struct Cercle* this)
{
	printf("Cercle::ConstruireCercle\n");
	this->myClass = &LeMetaCercle;
	this->myClass->superMetaClasse->ConstruireObjetGraphique(&this->superClasse);
	this->superClasse.type = CERCLE;

	printf("Construction partie propre au Cercle\n");
	/* Initialisation rayon par exemple */

	printf("Cercle::ConstruireCercle effectué\n");
}
```

Et enfin, pour traiter le cas de *getCentreX()* et *getCentreY()* non redéfinies pour le Cercle, on a juste à faire en sorte que :

```
TVMgetCentreX[OG] == TVMgetCentreX[CERCLE] == &getCentreXOG
TVMgetCentreX[RECTANGLE] == &getCentreXRectangle
```

où *getCentreXOG()* et *getCentreXRectangle()* sont des fonctions écrites dans le code pour répondre correctement aux appels.

Objectif pratique de ce tutorial

Comprendre les concepts du modèle objet en implémentant en 3 étapes un modèle d'objet graphique simple et fonctionnel. Ecrire les codes des initialisations, des constructeurs, des classes et des métaclasses ainsi qu'un petit programme principal fonctionnel testant : déclaration et instanciation, un exemple d'héritage (avec Cercle et Rectangle), puis un dernier exemple mettant en œuvre un exemple de polymorphisme.