Introduction à la programmation orientée objet en C

par Aymeric Lesert (Site personnel de Aymeric Lesert)

Date de publication : 2003

Dernière mise à jour :

Ce document présente brièvement les mécanismes du langage C qui sont sollicités pour parvenir à la Programmation Orientée Objet (POO).
Il constitue donc une introduction à la Programmation Orientée Objet en C.

Votre avis et vos suggestions sur cet article nous intéressent !
Alors après votre lecture, n'hésitez pas :

I - Les mécanismes du langage C utilisés

Les « pointeurs » ont une réputation de mal-aimé. Dés qu'on en parle, on pense :

- complexités
- difficultés à résoudre les bugs
- débordement de mémoire (coredump)

Or, ce n'est ni plus ni moins qu'une adresse mémoire. Vous déplacez simplement votre référentiel de la variable à son adresse (un peu comme si vous ne vous adressez pas à un interlocuteur par nom mais par son lieu d'habitation). Dans notre cas, les pointeurs doivent devenir vos alliés et vous devez apprendre à les maîtriser. L'autre objectif de ce document est de montrer comment avec quelques règles simples, il est possible de devenir copain-copain avec eux.

I-A - Les pointeurs

Un pointeur est une adresse mémoire (on parle aussi de référence). Il est très fréquemment utilisé avec des allocations dynamiques (malloc, free ou realloc). Lors de la déclaration d'une variable, nous la matérialisons par une étoile. (ex : int i; i contient un entier et int *i; contient une référence (une adresse) sur une variable contenant un entier).
Il est utilisé dans le passage de paramètres par adresse (la valeur du paramètre est susceptible d'évoluer).
Exemple :

```
int *Ptr; /* Ptr est un pointeur sur un entier */
```

```
t_Cellule *Courant; /* Courant est un pointeur sur un objet de type t_Cellule */
```

```
char *Car; /
* Car est un pointeur sur un caractère ou un pointeur sur le premier caractère d'une chaîne de caractères. */
```

```
t_Individu **Individu; /* Ceci est un pointeur sur un pointeur de type t_Individu */
```

Dans le dernier exemple, cela revient à décrire un individu non pas par son nom, ni par son adresse mais par sa ville. Ceci est utilisé quand l'individu doit changer d'adresse.

I-B - Les flèches

Les flèches constituent un raccourci dans l'utilisation du pointeur.
Courant->Suivant est équivalent à (*Courant).Suivant. Le parenthésage est très important, il indique l'ordre dans lequel il faut lire.
Remarques :

- (*Courant).Suivant : on accède à la propriété « suivant » de l'objet à l'adresse Courant
- *Courant.Suivant : on accède à l'objet adressé par l'information « Courant.Suivant »

I-C - Les pointeurs de fonctions

En C, il n'existe pas de procédures au sens strict du terme. Nous assimilons une procédure à une fonction qui retourne un objet non défini (void).
Les fonctions, qui ne sont pas seulement du code, sont aussi des adresses statiques (en interne). Elles indiquent l'adresse dans le segment de code du début de son corps. Pour disposer de fonctions « dynamiques » (comme une fonction variable), il existe un dispositif qui est le pointeur de fonction :

void (*Ecrire)(void) : Ecrire est un pointeur sur une fonction qui n'accepte aucun paramètre et qui ne retourne rien. Ecrire est donc une variable contenant l'adresse d'une fonction.
L'affectation de cette variable se fait comme sur l'exemple ci-dessous :

```
void Ecrire_fonction(void)
{
  printf("Bonjour") ;
  return ;
}

void Traitement(void)
{

  /* ... */

  Ecrire = Ecrire_fonction;
  /* ou */
  Ecrire = (void (*)(void))Ecrire_fonction;
  /
* caste Ecrire_fonction en un pointeur sur une fonction ne prenant pas de paramètres et ne retournant rien */

  /* ... */

}
```

L'utilisation de la variable s'utilise comme une fonction normale :

```
Ecrire();
```

I-D - Un cast (changement de type)

C'est un mécanisme qui permet de convertir de manière explicite le type d'une valeur en un autre type.
Exemple :

```
int i ;
float f=10.546 ;

i=(int)f ; /* converti le réel f en entier */
```

I-E - Le cast (changement de type) des pointeurs

Le cast de pointeurs permet de modifier le type de l'objet référencé par une adresse.
Exemple :

```
t_Individu *Individu ;
t_Fonctionnaire *Fonctionnaire ;

Individu=(t_Individu *)Fonctionnaire ; /* un fonctionnaire est un individu */
```

I-F - Les types énumérés

Cela permet (au contrainte de #define) de définir des constantes et de les regrouper sous un même type.
Exemple :

```
typedef enum { VRAI=1, FAUX=0, TRUE=1, FALSE=0} e_Booleen ;
```

I-G - Les types structurés

Un type structuré permet de regrouper au sein d'une même entité un ensemble de propriétés cohérentes (ex : nom, prénom, civilité pour un individu). Ils sont utilisés aussi pour la construction de listes chaînées, d'arbres, ...

```
typedef struct
{
  e_Civilite Civilite;
  char Nom[40];
  char Prénom[40];

} t_Individu ;
```

II - L'utilisation des mécanismes pour la Programmation Orientée Objet

II-A - Modélisation d'une classe

Pour faire simple, une classe est un type structuré comprenant des propriétés de type « pointeur de fonction » et d'autres types plus « classiques ». Il est nécessaire de ranger les propriétés selon l'ordre suivant :

- Le destructeur (pointeur de fonction)
- Les méthodes publiques (pointeurs de fonction)
- Les méthodes protégées (pointeurs de fonction)
- Les méthodes privées (pointeurs de fonction)
- Les attributs (autres types)

Pour compléter le dispositif, il faut disposer de fonctions qui permettent d'initialiser cette structure et d'associer les pointeurs de fonction (adresse de fonction dynamique) avec les fonctions implémentées (adresse de fonction statique) : les constructeurs.

II-B - Les constructeurs

Ces fonctions devront :

- Allouer une zone mémoire suffisante pour stocker la structure matérialisant la classe
- Affecter les méthodes et le destructeur
- Initialiser les attributs
- Retourner une référence sur une nouvelle instance de la classe

De manière pratique, je les nomme comme suit : Instancier_<nom classe>_<nature> ;
Exemple :

```
t_Stylo *Instancier_stylo_rouge() ;
t_Stylo *Instancier_stylo_copie(t_Stylo *) ;
```

II-C - Le this

En C++ ou en Java, l'objet « this » est la référence de l'instance d'une classe utilisant une méthode. Il est souvent utilisé de manière implicite. Cette information, en C, est référencée par le premier argument des méthodes (pointeur sur la structure à laquelle appartient la méthode) et doit être utilisée de manière explicite.
Exemple :

```
e_Couleur Lire_couleur_stylo(t_Stylo *this) ;
void Ecrire_couleur_stylo(t_Stylo *this,e_Couleur Couleur) ;
```

II-D - Le destructeur

C'est une fonction qui permet de libérer la zone mémoire allouée par les constructeurs. Sa particularité est de devoir positionner la référence de l'instance à NULL après sa destruction. C'est pour cette raison qu'il est nécessaire de passer la référence de la référence d'une instance.
Exemple :

```
void Liberer_stylo(t_Stylo **this) ;
```

II-E - Les méthodes

Dans notre modèle, il est très compliqué de mettre en œuvre une séparation entre les méthodes privées, protégées et publiques. J'ai donc fait des choix de simplification. La classe des méthodes est matérialisée par des commentaires (Cf. Exemple ci-dessous).
L'implémentation des méthodes s'effectue dans des fonctions statiques.
Exemple :

```
e_Couleur Lire_couleur_stylo(t_Stylo *this)
{
  return(this->Couleur) ;
}

void Ecrire_couleur_stylo(t_Stylo *this,e_Couleur Couleur)
{
  this->Couleur = Couleur;
  return ;
}

t_Stylo Instancier_stylo()
{

  /* ... */

  this->Lire_couleur = Lire_couleur_stylo ;
  this->Ecrire_couleur = Ecrire_couleur_stylo ;

  /* ... */

}
```

II-F - Conseils et avantages

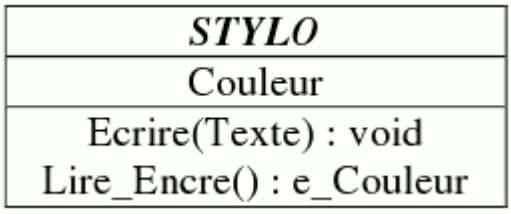
Je vous conseille vivement de décrire 2 fichiers par classe ou par ensemble de classes :

- Un fichier header (exemple : stylo.h) : contenant la structure matérialisant la classe et définissant le prototype du(des) constructeur(s)
- Un fichier source (exemple : stylo.c) : contenant l'implémentation des méthodes et des constructeurs

Dans votre fichier source, toutes les implémentations de vos fonctions peuvent commencer par le mot clé « static ». Cela permet de masquer (et d'interdire) l'accès à ces fonctions sans passer par la classe.

II-G - Exemple complet

II-G-1 - La classe Stylo



II-G-2 - Définition du type en C dans le header (stylo.h)

```
typedef enum { ROUGE, VERT, BLEU }e_Couleur ;
/* ---DESCRIPTION DE LA CLASSE STYLO ---*/
typedef struct t_Stylo

{
  /* le destructeur */
  void (*Liberer)(struct t_Stylo **this) ;

  /* les méthodes publiques */
  void (*Ecrire)(struct t_Stylo *this,char *i_Texte);
  e_Couleur (*Lire_encre)(struct t_Stylo *this) ;

  /* une propriété privée */

  e_Couleur Couleur;
} t_Stylo;
extern t_Stylo *Instancier_stylo(e_Couleur);
```

II-G-3 - Définition de la classe en C dans le fichier source C (stylo.c)

```
/* ---DESCRIPTION DES FONCTIONS ASSOCIEES AUX METHODES ---*/

/* le destructeur */

static void Liberer_stylo(t_Stylo **this)
{
  AL_FREE(*this);
  *this=(t_Stylo *)NULL;
  return;
}

/* Les méthodes publiques */

static void Ecrire_stylo(t_Stylo *this,char *i_Texte)
{
  Change_couleur(this->Couleur);
  printf("Texte : <%s>\n",i_Texte);
  return ;
}

static e_Couleur Lire_encre_stylo(t_Stylo *this)
{
  return(this->Couleur);
}

/* le constructeur */

t_Stylo *Instancier_stylo(e_Couleur i_Couleur)
{

  /* allocation d'une instance */

  t_Stylo *this=(t_Stylo *)AL_MALLOC(sizeof(t_Stylo)) ;
  if (this==(t_Stylo *)NULL)
    return((t_Stylo *)NULL) ;

  /* affectation des méthodes publiques */

  this->Ecrire=Ecrire_stylo ;

  this->Lire_encre=Lire_encre_stylo;

  this->Liberer=Liberer_stylo;

  /* initialisation des propriétés */

  this->Couleur=i_Couleur;

  /* retour de l'objet instancié */
```

```
  return(this);
}
```

II-G-4 - Utilisation dans un programme

```
void ma_fonction()

{

  t_Stylo *Stylo_Bleu=Instancier_stylo (Bleu);

  /* ... */

  Stylo_Bleu->Ecrire(Stylo,"Coucou");

  /* ... */

  Stylo_Bleu->Liberer(&Stylo);

  return;
}
```