Programmation Orientée Objet en C

Introduction

Qu'est-ce que la programmation orienté objet ? Wikipédia nous donne la définition suivante :

> Il consiste en la définition et l'interaction de briques logicielles appelées *objets* ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. - Source

Plus simplement il s'agit de programmer avec différents **objets** qui seront des **instances** de **classes**. Cependant le langage C est pas définition un langage fonctionnel et non pas orienté objet.

On peut contourner cette difficulté en implémentant les principes relevés par Axel T. Schreiner dans son livre "Object-Oriented Programming with ANSI-C". On peut retrouver un comportement similaire à de la **POO** (programmation orientée objet) à l'aide des structures en C et de quelques astuces fournies par l'auteur.

On va se baser sur un fonctionnement similaire au Java et au Python, c'est à dire que tout est un `Objet` et que toute classe possède la classe `Objet` comme parent.

Les classes `Objet` et `Class` fournissent par défaut les **méthodes** suivantes qui pourront être appliquées à n'importe quel objet :

- `new(const void *_class, ...)` : permet de créer une nouvelle instance de classe avec ses arguments
- `delete(const void *_self)` : permet de libérer l'espace mémoire occupée par une instance de classe
- `differ(const void *a, const void *b)` : retourne vrai si les deux objets sont différents
- `puto(const void *_self, FILE * fp)` : affiche dans le descripteur de fichier (ou console) une représentation minimale de l'objet (sa classe et son adresse en mémoire)
- `cast(const void *_class, const void *_self)` : permet de "caster" un objet dans le type (classe) voulu. Retourne une erreur en cas d'échec.

Comment créer une classe

Quelques classes ont déjà été implémentées par mes soins telles que les `List` ou `Dictionary` (fonctionnement expliqué plus loin). Pour créer une nouvelle classe il est nécessaire de suivre une certaine nomenclature expliquée par l'auteur. Imaginons que l'on veuille créer une classe `Point` (exemple). Il faudra **créer deux classes** : `PointClass` qui sera une méta-classe contenant les méthodes de classe et `Point` qui sera une classe objet contenant les attributs de la classe.

Il sera de plus nécessaire de créer **3 fichiers** pour respecter la nomenclature ::

- `Point.r` : **fichier de représentation** (d'un point) : on y retrouve la structure de la classe et de la méta-classe ainsi que le attributs et les méthodes de la classe
- `Point.h` : **fichier d'en-têtes** contenant une référence vers les classes ainsi que les références vers les méthodes de classe

- `Point.c` : **fichier contenant le code** actuel des méthodes de classe, du constructeur et du destructeur

Ainsi le fichier de représentation contient la représentation de la classe et la méta-classe :

```
#include "Object.r"

struct Point {
    const struct Object _;              /* Point : Object */
    int x, y;                           /* coordonnées */
};

struct PointClass {
    const struct Class _;               /* PointClass : MétaClasse */
    void (*draw) (const void *self);
};
```

On continue avec le fichier d'en-têtes :

```
#include "Object.h"

extern const void * const Point(void);          /* new(Point, x, y); */
extern const void * const PointClass(void);     /* Métaclasse PointClass */

void draw (const void *self);                   /* Méthode de classe pour
afficher un point à l'écran */
void move (void *point, int dx, int dy);        /* Méthode statique pour
déplacer un point */
```

Et finalement avec le fichier contenant le code en plusieurs parties. On commence par le code nécessaire au fonctionnement de la classe :

```
#include "Point.r"
#include "Point.h"

/******************************************************************************/
/*                              CLASSE POINT                                  */
/******************************************************************************/

/* Constructeur avec arguments */
static void *Point_ctor (void *_self, va_list *app) {
    struct Point *self = super_ctor(Point(), _self, app);

    // On récupère les arguments dynamiques
    self->x = va_arg(*app, int);
    self->y = va_arg(*app, int);
    return self;
}

/* Destructeur */
static void *Point_dtor (void *_self) {
    struct Point *self = cast(Point(), _self);
    // rien d'autre à libérer on retourne la référence à l'objet
    // pour être libéré avec free()
    return self;
}
```

```
25
26  /* Méthode de classe */
27  static void Point_draw (const void *_self) {
28      const struct Point *self = _self;
29      printf("\".\" at %d,%d\n", self->x, self->y);
30  }
31
32  /* Override d'une méthode de classe parente */
33  static void Point_puto(const void * _self, FILE * fp) {
34      const struct Point *self = _self;
35      fprintf(fp, "Point (x, y): (%d, %d)\n", self->x, self->y);
36  }
```

On observe que le destructeur de la classe est assez naïf et cela est normal. En effet les attributs de la classe `Point` sont gérés par le système. Cependant si on avait du allouer dynamiquement de l'espace (`malloc` / `calloc`) c'est ici qu'on aurait libéré l'espace.

On continue avec le code de la méta-classe :

```
1   /*****************************************************************************/
2   /*                            MÉTACLASSE POINTCLASS                          */
3   /*****************************************************************************/
4
5   /* Méthode de classe */
6   void draw (const void *_self) {
7       const struct PointClass *class = classOf(_self);
8
9       assert(class->draw); // on vérifie que l'objet possède bien la fonction
10      class->draw(_self); // on appelle la fonction de l'objet
11  }
12
13  /* Méthode statique */
14  void move (void *point, int dx, int dy) {
15      struct Point *self = _self;
16      self->x += dx, self->y += dy;
17  }
18
19  /* Constructeur */
20  static void * PointClass_ctor (void * _self, va_list * app) {
21      // on appelle le constructeur du parent (appel en chaîne jusqu'à objet)
22      struct PointClass * self = super_ctor(PointClass(), _self, app);
23      typedef void (*voidf) ();
24      voidf selector;
25  #ifdef va_copy
26      va_list ap; va_copy(ap, *app);
27  #else
28      va_list ap = *app;
29  #endif
30
31      while ((selector = va_arg(ap, voidf))) {
32          voidf method = va_arg(ap, voidf);
33
34          // on ajoute ici les méthodes de classe
35          // 1 if par méthode
36          if (selector == (voidf) draw)
37              *(voidf *) &self->draw = method;
38      }
```

```
#ifdef va_copy
    va_end(ap);
#endif

    return self;
}
```

> Le `#ifdef` est une instruction qui sera donnée au préprocesseur (Wikipédia). Ainsi si la macro `va_copy` est définie par le compilateur alors elle sera utilisée. Si ça n'est pas le cas on propose une alternative.

On termine avec l'initialisation de la classe dans le fichier (auto-initialisation par appel à la référence) :

```
/******************************************************************************/
/*                     INITIALISATION POINT,POINTCLASS                        */
/******************************************************************************/
static const void *_Point, *_PointClass; // références internes

// référence externe dans le projet
const void *const PointClass(void) {
    return _PointClass ?
            _PointClass : (_PointClass = new(Class(), "PointClass",
            Class(), sizeof(struct PointClass),
            ctor, PointClass_ctor,  // constructeur de classe
            (void *) 0));
}

// référence externe dans le projet
const void *const Point(void) {
    return _Point ?
            _Point : (_Point = new(PointClass(), "Point",
            Object(), sizeof(struct Point),
            ctor, Point_ctor,       // contructeur de classe
            draw, Point_draw,       // méthodes de classe (obligatoire)
            (void *) 0));
}
```

Avec ces 3 fichiers on a maintenant une classe complète qui peut être appelée dans le projet. Voici un code permettant d'utiliser le point dans un projet :

```
void *pA = new(Point(), 0, 0); // création du Point(0, 0)

puto(pA, stdout);               // affichage à l'écran
draw(pA);                       // utilisation de la méthode de classe

move(pA, 5, 2);                 // utilisation d'une méthode statique
draw(pA);
puto(pA, stdout);

delete(pA);                     // suppression de l'objet
```

Les types non-natifs implémentés

Dans le cadre du projet EVEEX (**E**ncodeur **V**ideo **E**NSTA Bretagne **EX**périmental), le prototype en python étant réalisé, l'étape d'après est le passage à un prototype en C. Pour cela on va essayer de "traduire" le code (réduction du temps d'itération). C'est pourquoi nous avons besoin d'objets non natifs tels que les listes ou encore les dictionnaires.

Les listes

La classe `List` représente un tableau d'`Object` contenu dans un espace mémoire donné. Il s'agit d'un tableau à taille dynamique c'est à dire que lorsque le tableau est plein, on agrandit l'espace alloué lors d'un ajout suivant d'élément.

Voici les méthodes associées à cette classe :

- `new(List(), unsigned size)` : permet de créer une liste de taille `size` (si précisé) ou de taille minimale (32) si la taille n'est pas précisée
- `addFirst(void *_self, const void *element)` : ajoute en début de liste l'élément `element`. Si le tableau est plein, double sa taille.
- `addLast(void *_self, const void *element)` : ajoute en fin de liste l'élément `element`. Semblable à `addFirst`
- `count(const void *self)` : compte le nombre d'éléments présents dans la liste
- `lookAt(const void *_self, unsigned n)` : retourne l'élément à la position `n` dans la liste
- `takeFirst(void *_self)` : retire le premier élément de la liste et le retourne
- `takeLast(void *_self)` : retire le dernier élément de la liste et le retourne
- `indexOf(const void *_self, const void *element)` : retourne la position (indice) de la première instance de `element` dans la liste

Avec ces quelques méthodes on peut créer une liste et la manipuler. Voici un exemple succin de son utilisation :

```
1  void *list = new(List());
2
3  void *pB = new(Point(), 1, 1);
4  void *pC = new(Point(), 2, 9);
5
6  addLast(list, pB);
7  addLast(list, pC);
8
9  printf("Nb éléments dans liste : %d\n", count(list));
10
11 printf("Affichage du premier élément : ");
12 puto(lookAt(list, 0), stdout);
13
14 delete(pB);
15 delete(pC);
16 delete(list);
```

Les dictionnaires

La classe `Dictionary` représente un dictionnaire. Contrairement à une liste où les objets sont définis par leur position, les objets sont définis à partir d'une **clé**. Ainsi pour accéder à une valeur contenue dans un dictionnaire on passe par sa clé. Il s'agit encore d'une structure à taille dynamique c'est à dire que lorsque le dictionnaire est plein, on agrandit l'espace alloué lors d'un

ajout suivant d'élément.

Voici les méthodes associées à cette classe :

- `new(Dictionary(), unsigned size)` : permet de créer dictionnaire de taille `size` (si précisé) ou de taille minimale (32) si la taille n'est pas précisée
- `set(void *_self, const char *key, const void *value)` : permet de mettre l'objet `value` à la position `key` dans le dictionnaire. Retourne l'objet ainsi crée
- `get(const void *_self, const char *key)` : retourne l'élément du dictionnaire stocké avec la clé `key`
- `size(const void)` : retourne le nombre d'élément dans le dictionnaire

Avec ces quelques méthodes on peut créer un dictionnaire et le manipuler. Voici un exemple succin de son utilisation :

```
1  void *dict = new(Dictionary());
2
3  void *pD = new(Point(), 1, 5);
4  void *pE = new(Point(), 5, 2);
5
6  set(dict, "D", pD);
7  set(dict, "E", pE);
8  set(dict, "D2", pD);
9
10 void *pD2 = get(dict, "D");
11 if (!differ(pD, pD2)) {
12     puts("pD == pD2");
13 }
14 printf("Nb éléments dans le dico : %d\n", size(dict));
15
16 delete(pD);
17 delete(pE);
18 delete(dict);
```

Les pixels

Le pixel est une couche d'abstraction supplémentaire d'une image. Toujours à 4 composantes : RGB ou YUV avec une dernière couche de transparence. Leur utilisation est assez directe. Les méthodes présentées ci-dessous sont équivalentes pour RGB et YUV :

- `new(RGBPixel(), int r, int g, int b, int a)` : permet de créer un `Pixel` en définissant ses 3 composantes rgb

On peut alors créer un pixel et le manipuler :

```
void *rgb = new(RGBPixel(), 15, 255, 16);
void *yuv = new(YUVPixel(), 252, 15, 15);

puto(rgb, stdout);
puto(yuv, stdout);

((struct RGBPixel *) rgb)->r = 5;
puto(rgb, stdout);

delete(rgb);
delete(yuv);

```

Les images

La classe image est une couche d'abstraction de la bibliothèque stb_image. Elle permet de créer et de manipuler avec aisance une image en C en s'occupant tout seul de la partie gestion de la mémoire. Il s'agit d'une structure assez employée en C mais adaptée ici à l'utilisation des classes pour le projet.

Voici les méthodes associées à cette classe :

- `new(Image(), int width, int height, int channels, int init_with_zeros)` : permet de créer une instance d'image de taille `width` x `height` avec un nombre de canaux `channels` (1 pour du noir et blanc, 3 pour du RGB, 4 avec de la transparence) et permet d'initialiser la mémoire allouée avec des zéros (ou non)
- `loadImg(const char *filename)` : permet de charger une image sur le disque et retourne un objet `Image` avec les données de ce fichier
- `saveImg(const struct Image *self, const char *filename)` : permet de sauvegarder une image sur le disque à partir des données contenues dans l'image `self`
- `toGray(const struct Image *self)` : retourne une nouvelle image qui est une copie en noir et blanc de l'image `self`
- `toSepia(const struct Image *self)` : retourne une nouvelle image qui est une copie avec le filtre sepia de l'image `self`

Avec ces quelques méthodes on peut créer un dictionnaire et le manipuler. Voici un exemple succin de son utilisation :

```
struct Image *img, *gray, *sepia;

img = (struct Image *) loadImg("assets/image_res_low.jpg");
printf("Image loaded: ");
puto(img, stdout);

gray = (struct Image *) toGray(img);
sepia = (struct Image *) toSepia(img);

saveImg(gray, "assets/gray_res.jpg");
printf("Saving gray img to disk\n");

saveImg(sepia, "assets/sepia_res.jpg");
printf("Saving sepia img to disk\n");

```

```
delete(gray);
delete(sepia);
delete(img);
```

Les Bitstreams

Le `Bitstream` est une structure de données qui nous utilisons pour communiquer sur le réseau. Ainsi le client enverra un `Bitstream` de données au serveur. Pour l'instant son implémentation reste naïve :

- `new(Bitstream(), int frameid, enum MessageTypes type, int size, char *data)` : permet de créer un Bitstream de type `type` associé à la frame n° `frame_id` avec une taille de donnée de `size` et enfin avec les données `data`

On peut alors créer et manipuler des Bitstreams :

```
int frameid = 5;
enum MessageTypes type = HEADER_MSG;
char msg[] = "Hello world !!!!";
unsigned long size = strlen(msg);

void *stream = new(Bitstream(), frameid, type, size, msg);

puto(stream, stdout);

delete(stream);
```

Points d'amélioration

> A considérer comme des points optionnels de travail

- Faire en sorte que `List` et `Dictionary` soient classes héritées d'une classe `Countable` afin qu'ils aient une fonction `count` commune
- Implémenter une fonction de retrait d'objet du dictionnaire
- Implémenter une fonction permettant de récupérer les clés/valeurs présentes dans un dictionnaire
- Implémenter une fonction permettant de dire si un objet est déjà présent dans une liste
- Implémenter les fonctions manquantes au `Bitstream`
- Intégrer les `Pixel` aux `Image` pour ajouter une nouvelle couche d'abstraction