Programmation Orientée Objet avec de l'ANSI-C

> Prise de notes réalisée par
> Lecture du livre [Object-Oriented Programming With ANSI-C - RIT CS]
> Le 24 novembre 2020

Même si l'ANSI-C est un langage de programmation fonctionnel, il est possible d'imiter le fonctionnement de langages de programmation orienté objet en implémentant ce qui donne leur particularité à ces langages.

Types de données abstraites et dissimulation de l'information

Types de données abstraites

Appeler un type de donnée "abstrait" veut dire que l'on ne révèle pas sa représentation à l'utilisateur. Il y a donc des étapes que l'on va lui cacher (des opérations, des tests, etc...). Avec un type de donnée abstraite on sépare l'implémentation de son utilisation ce qui rend le code final beaucoup plus accessible. On peut alors décomposer un large système en plein de petits modules.

Exemple avec le type `Set`. Ce type de donnée est abstraite, on déclare ce que l'on peut faire avec un set :

```
#ifndef SET_H
#define SET_H

extern const void *Set;

void *add (void *set, const void *element);
void *find (const void *set, const void *element);
void *drop (void *set, const void *element);
int contains (const void *set, const void *element);

#endif
```

Le header est complet mais est-ce qu'il est utile ? On peut travailler maintenant avec des set grâce à `add()`, `find()` ou `drop()`. On a l'exemple d'une fonction d'aide qui convertit le résultat de `find` dans une valeur de vérité.

On a utilisé ici un pointeur `void *` pour deux raisons : il est impossible de découvrir ce à quoi un set ressemble, et parce que cela nous permet de passer ce que l'on veut aux fonctions.

Gestion de la mémoire

Comment est-ce que l'on peut créer un `Set` ? Il s'agit d'un pointeur, pas un type définit avec `typedef`. On va utiliser des pointeurs pour référer au `Set` et à ses éléments :

```
void *new(const void *type, ...);
void delete(void *item);
```

La fonction `new()` accepte un descripteur comme `Set` et possiblement plus d'argument pour l'initialisation de la commande. `delete()` accepte un pointeur produit par `new()` et le recycle. Concrètement ce sont juste des "wrappers" de `calloc()` et `free()`.

De même si l'on veut stocker quelque chose d'intéressant dans le `Set` il va falloir un autre type abstrait de données. Ici `Objet` :

```
extern const void *Objet;
int differ(const void *a, const void *b);
```

La fonction `differ()` peut comparer des objets en retournant vrai si ils sont égaux et faux sinon. Il s'agit d'un "wrapper" de la fonctionnalité `strcmp()` pour certaines paires d'objets on retournera un nombre négatifs et d'autres un positif pour donner le concept d'ordre.

Ainsi le code suivant prend du sens :

```
void *s = new(Set);
void *a = add(s, new(Object));
void *b = add(s, new(Object));
void *c = new(Object);

contains(s, a) && contains(s, b); // vrai pour les deux
contains(s, c); // sera faux
contains(s, drop(s, a)); // retournera faux

differ(a, add(s, a)); // retournera faux les deux objets étant identiques

delete(drop(s, b)); // on supprime les références aux objets
delete(drop(s, c));
```

Implémentation - Set

La fonction `new()` est simple à comprendre. Il s'agit de la même pour les `Set` et les `Object`. On utilise un tableau `heap[]` pour contenir les valeurs :

```
#if !defined MANY || MANY < 1
#define MANY 10
#endif

static int head[MANY];

void *new(const void *type, ...) {
    int *p;     /* &heap[1..] */
    for (p = heap + 1; p < heap + MANY; ++p)
        if (!*p) // si p est non null -> on est à la fin du tab initialisé
            break;

    assert(p < heap + MANY); // on vérifie qu'on a le bon nombre d'éléments
                             // peut se déclencher si on est à court de
mémoire
    *p = MANY;
```

```
    return p;
}
```

Avant qu'un objet soit ajouté à un `Set`, on le laisse contenir une valeur d'index impossible pour que `new()` ne puisse pas le retrouver et qu'on ne le confonde pas comme la valeur d'un membre d'un `Set`.

La fonction `delete()` doit faire attention aux pointeurs null. Un élément de `heap[]` est recyclé en le mettant à zéro :

```
void delete(void *_item) {
    int *item = _item;
    if (item) {
        // on s'assure que l'objet appartienne bien au set
        assert(item > heap && item < heap + MANY);
        *item = 0; // on le recycle
    }
}
```

Ici on utilise une nomenclature pour gérer les pointeurs génériques : un underscore en préfixe et utilisation uniquement dans les arguments de la fonction.

Un `Set` est représenté par ses objets : chaque élément pointe vers le `Set`. Si un élément contient `MANY` alors il peut être ajouté au `Set` sinon il est déjà dedans car on ne permet pas aux `Object` d'être dans deux `Set` différents :

```
void *add(void *_set, const void *_element) {
    int *set = _set;
    const int *element = _element;

    // on ne veut traiter qu'avec des pointeurs dans heap[]
    assert(set > heap && set < heap + MANY);
    // on veut que set n'appartienne pas à un autre set (logique)
    assert(*set == MANY);
    assert(element > heap && element < heap + MANY);

    // si l'élément n'appartient à aucun set on l'ajoute
    if (*element == MANY)
        *(int *) element = set - heap;
    else
        assert(*element == set - heap);

    return (void *) element;
}
```

Les autres fonctions sont transparentes. `find()` cherche uniquement si ses éléments contiennent le bon index du `Set` :

```
void *find(const void *_set, const void *_element) {
    const int *set = _set;
    const int *element = _element;

    assert(set > heap && set < heap + MANY);
    assert(*set ==MANY);
    assert(element > heap && element < heap + MANY);
    assert(*element);

    return *element == set - heap ? (void *) element : 0;
}
```

Et `contains()` convertit le résultat de `find()` dans une valeur de vérité :

```
int contains(const void *_set, const void *_element) {
    return find(_set, _element) != 0;
}
```

La fonction `drop()` peut s'appuyer sur `find()` pour vérifier que l'élément à retirer appartient bien au `Set`. Dans ce cas on marque l'objet avec `MANY` :

```
void *drop(void *_set, const void *_element) {
    int *element = find(_set, _element);

    // si l'élément appartenait bien au Set
    if (element)
        *element = MANY;
    return element;
}
```

Cette implémentation permet de se passer de la fonction `differ()` cependant notre application en dépend, il faut donc l'implémenter (une simple comparaison de pointeurs ici est suffisante) :

```
int differ(const void *a, const void *b) {
    return a != b;
}
```

Il ne reste plus qu'à rendre le compilateur heureux en définissant les descripteurs `Set` et `Object` :

```
const void *Set;
const void *Object;
```

Autre implémentation - Sac

On conserve l'interface de `Set` mais cette fois-ci on utilise l'allocation dynamique et les objets deviennent des structures :

```
struct Set { unsigned count; };
struct Object { unsigned count; struct Set *in; };
```

La variable `count` garde la trace du nombre d'éléments dans un `Set`. Pour un élément `count` enregistre le nombre de fois que cet élément à été ajouté au `Set`. Si on décrémente cette variable à chaque appel de `drop()` et que l'on ne le retire que lorsque cette valeur vaut 0 alors on a un Sac.

On doit initialiser les objets dans `new()`, on doit donc pouvoir connaitre leur taille :

```
static const size_t _Set = sizeof(structSet);
static const size_t _Object = sizeof(struct Object);
const void *Set = &_Set;
const void *Object = &_Object;
```

Ce qui rend la fonction `new()` nettement plus simple :

```
void *new(const void *type, ...) {
    const size_t size = *(const size_t *) type;
    void *p = calloc(1, size);
    asser(p); // on s'assure que le pointeur existe bien
    return p;
}
```

De même `delete()` transmet directement son argument à la fonction `free()`. La fonction `add()` doit croire ses arguments et incrémente le nombre de ses enregistrements :

```
void *add(void* _set, const void*_element) {
    struct Set *set = _set;
    struct Object *element = (void *) _element;

    assert(set); // on vérifie que le set existe
    assert(element);// on vérifie que l'élément existe

    if (!element->in) // si l'élément n'appartenait à aucun set
        element->in = set;
    else
        assert(element-> in == set);

    ++element->count, ++set->count;
    return element;
}
```

La fonction `find()` vérifie toujours si l'élément pointe vers le bon set (`contains()` ne change pas) :

```
void *find(const void *_set, const void *_element) {
    const struct Object *element = _element;

    assert(_set);
    assert(element);

    return element->in == _set ? (void *) element : 0;
}
```

Si `drop()` trouve l'élément dans le `Set` il décrémente sa référence et le nombre d'éléments dans le `Set`. Si sa référence tombe à zéro il est retiré du `Set` :

```
void *drop(void *_set, const void *_element) {
    struct Set * set = _set;
    struct Object * element = find(set, _element);

    // si l'élément appartient toujours au set
    if (element) {
        if(--element->count == 0)
            element->in = 0; // on retirer la référence au Set
        --set->count;
    }
    return element;
}
```

On peut maintenant fournir une fonction `count()` qui retourne le nombre d'éléments dans un set :

```
unsigned count(const void *_set) {
    const struct Set *set = _set;
    assert(set); // on s'assure que le Set existe
    return set->count;
}
```

Bien sur ça serait plus simple que l'application ait accès au composant `.count` directement mais on insiste sur le fait de ne pas révéler la représentation de la donnée. Ce n'est rien par rapport au danger d'avoir une application écrivant sur une donnée critique.

Liaison dynamique et fonctions génériques

Constructeurs et destructeurs

On va implémenter un type "chaîne de caractères" que l'on va ensuite mettre dans un `Set`. La fonction `new()` sait quel type d'objet elle doit créer mais la fonction `delete()` devient une autre histoire. La façon évidente de l'implémenter est de donner le lien vers la façon de détruire l'objet, dans son type pour conserver la structure de donnée abstraite.

```
struct type {
    size_t size;                /* Taille de l'objet */
    void (* dtor) (void *);     /* destructeur */
}

struct String {
    char *text;                 /* Chaine de caractères dynamique */
    const void *destroy;        /* Localisation du destructeur */
}

struct Set {
    ... information ...
    const void *destroy;        /* Localisation du destructeur */
}
```

L'initialisation fait partie du boulot de `new()` et les différents types exigent différentes fonctions d'initialisation avec probablement des arguments supplémentaires :

```
new(Set);                       /* init un Set */
new(String, "text");            /* init une string */
```

Il nous faut donc une nouvelle fonction de la même manière que le destructeur : c'est ce qu'on va appeler le constructeur. Comme le constructeur et le destructeur ne sont pas spécifiques au type et ne changent pas, on les passent tous les deux à `new()`.

> Bien noter que ce n'est pas le rôle du constructeur et du destructeur de gérer directement la mémoire. Pour cela il faudra bien faire appel aux fonctions `delete()` et `new()`.

Méthodes, messages, classes et objets

La fonction `delete()` doit dans tous les cas pouvoir être capable de trouver le destructeur qu'importe le type d'objet. On peut ainsi créer une structure `Class` avec le pointeur vers le destructeur toujours en haut. De la même manière on va ainsi définir des pointeurs vers d'autres fonctions que l'on va considérer indispensable comme le constructeur, le clonage (`clone()`) et la comparaison (`differ()`).

```
struct Class {
    size_t size;
    void * (* ctor) (void *self, va_list *app);         /* Constructeur */
    void * (* dtor) (void *self);                       /* Destructeur */
    void * (* clone) (const void *self);                /* Clonage */
    int (* differ) (const void *self, const void *b);   /* Comparaison */
};
```

On peut se rendre compte qu'il suffit de faire passer cette structure à tous nos nouveaux types pour régler ces problèmes.

```
struct String {
    const void *class;  /* Doit être en premier */
    char *text;
};

struct Set {
    const void *class;  /* Doit être en premier */
    ... information ...
};
```

En regardant la liste des fonctions que l'on vient de créer, on se rend compte qu'elles vont fonctionner pour n'importe quel type d'objet. Il n'y a que le constructeur qui sera un peu délicat. On appelle ces fonctions les **méthodes** des objets. Appeler une méthode c'est mettre un terme sur un message et nous avons marqué l'*objet receveur* avec le paramètre `self`.

Beaucoup d'objets vont partager le même descripteur de type, ils auront besoin de la même quantité de mémoire et les mêmes méthodes pourront s'appliquer à eux. On appelle ce descripteur de type une **classe**. Un seul objet deviendra alors une **instance** de la classe.

Sélecteurs, liaison dynamique et polymorphismes

Qui s'occupe de faire la messagerie ? Le constructeur est appelé par `new()` pour une nouvelle zone mémoire qui est quasiment non-initialisée :

```
void *new(const void *_class, ...) {
```

```
    const struct Class *class = _class;
    void *p = calloc(1, class->size);

    assert(p); // on vérifie que la mémoire à bien été allouée
    *(const struct Class **)p = class; // quelque soit l'objet on lui donne sa classe

    // si la classe possède un constructeur
    if (class->ctor) {
        va_list ap; // liste variable d'arguments
        va_start(ap, _class); // init de la liste variable d'arguments
        p = class->ctor(p, &ap); // on appelle le constructeur de la classe
        va_end(ap); // on termine son utilisation
    }
    return p; // on retourne l'instance de classe ainsi crée
}
```

L'existence du pointeur vers `struct Class` au début d'un objet est du coup extrêmement important. C'est pour cela qu'on initialise le pointeur correspondant dans `new()`.

La fonction `delete()` suppose que chaque objet (i.e. chaque pointeur non null) pointe vers un descripteur de type pour appeler le destructeur. Ici `self` joue le rôle de `p` précédemment.

```
void delete(void *self) {
    const struct Class **cp = self;

    // si l'objet existe et qu'il possède un destructeur
    if (self && *cp && (*cp)->dtor)
        self = (*cp)->dtor(self); // on appelle le destructeur
    free(self); // on libère enfin l'objet
}
```

Le destructeur répare ici les bêtises qui peuvent être fait par le constructeur. Si un objet ne veut pas être supprimé, le destructeur peut alors simplement retourner un pointeur null.

Les autres méthodes stockées dans le descripteur de type fonctionnent d'une manière similaire en acceptant un paramètre `self`, il suffit de router la méthode vers son descripteur :

```
int differ(const void *self, const void *b) {
    const struct Class *const *cp = self;

    // on s'assure que l'objet existe bien et qu'il possède une fct de comparaison
    assert(self && *cp && (*cp)->differ);
    return (*cp)->differ(self, b); // on appelle la méthode de la classe
}
```

On appelle alors `differ()` une **fonction sélecteur**. C'est un exemple de fonction polymorphique, i.e. une fonction qui accepte des arguments de différents types et agit différemment selon leur type. Quand `differ()` est implémentée dans toutes les classes elle devient une **fonction générique** et s'applique partout.

Des méthodes peuvent être polymorphiques sans pour autant avoir une liaison dynamique. Exemple avec la fonction `sizeOf()` qui retourne la taille de n'importe quel objet :

```c
size_t sizeOf(const void *self) {
    const struct Class *const *cp = self;
    assert(self && *cp); // on s'assure que l'objet existe ainsi que son contenu
    return (*cp)->size; // on retourne la taille de l'objet
}
```

Tous les objets portant leur descripteur peuvent utiliser cette fonction. Attention cependant :

```c
void *s = new(String, "text");
assert(sizeof s != sizeOf(s)); // différence de résultat
```

`sizeof` est un opérateur en C qui est évalué au moment de la compilation et qui retourne le nombre de bytes requis par ses variables. `sizeOf()` est une fonction polymorphique qui retourne au moment de l'exécution le nombre de bytes de l'objet.

Une application

Même si on a pas encore implémenté les strings, on va écrire un programme de test.

```c
/* String.h */
extern const void *String;
```

Nos méthodes sont communes à tous les objets. On ajoute alors leurs déclarations dans le fichier de gestion de mémoire :

```c
/* new.h */
void *clone(const void *self);                      /* Fct sélecteur dérivée de struct Class */
int differ(const void *self, const void *b);        /* Fct sélecteur dérivée de struct Class */
size_t sizeOf(const void *self);
```

On a alors l'application suivante :

```c
#include "String.h"
#include "new.h"

int main() {
    void *a = new(String, "a"), *aa = clone(a);
    void *b = new(String, "b"); // création de 2 strings + 1 clone

    printf("sizeOf(a) == %u\n"; sizeOf(a));
    if (differ(a, b)) // 2 textes différents donnent 2 strings différentes
        puts("ok");
    if (differ(a, aa)) // on vérifie qu'une copie est égale mais pas identique
        puts("differ?");
    if (a == aa)
        puts("clone?");

    // résultat : sizeOf(a) == 8 \n ok

    delete(a), delete(aa), delete(b);
```

```
    return 0;
}
```

Une implémentation - String

On implémente les `String` en écrivant les méthodes qui vont entrer dans le descripteur. Le constructeur récupère le texte passé dans `new()` et garde une copie dynamique dans la structure. La variable `class` étant déjà initialisée dans `new()`.

```
struct String {
    const void *class;  /* Doit être en premier */
    char *text;
};

static void *String_ctor(void *_self, va_list *app) {
    struct String *self = _self; // on récupère les arguments
    const char *text = va_arg(*app, const char *);

    self->text = malloc(strlen(text) + 1); // allocation de mémoire pour la string
    assert(self->text); // on vérifie qu'il n'y a pas eu d'erreur
    strcpy(self->text, text); // copie de la chaine dans la variable
    return self; // on retourne l'instant de variable ainsi crée
}
```

Le destructeur libère la mémoire contrôlée par la `String`. Comme `delete()` est appelée si et seulement si `self` est non null on a pas besoin de vérifier :

```
static void *String_dtor(void *_self) {
    struct String *self = _self;
    free(self->text), self->text = 0; // on libère l'espace alloué
    return self; // on retourne le pointeur sur l'objet pour le libérer dans delete()
}
```

La fonction `String_clone()` va créer une copie de `String`. Plus tard l'original et la copie seront passé à `delete()` donc on doit faire une copie dynamique. C'est plus facile en appelant `new()` :

```
static void *String_clone(const void *_self) {
    const struct String *self = _self;
    return new(String, self->text); // crée une nouvelle instance dynamiquement
}
```

`String_differ()` est fausse si les deux objets ne sont pas du même type ou que leur contenu est différent :

```c
static int String_differ(const void *_self, const void *b) {
    const struct String *self = _self;
    const struct String *b = _b;

    if (self == b) // si il s'agit du même objet c'est forcément vrai
        return 0;
    if (!b || b->class != String) // types différents donc f
        return 1;
    return strcmp(self->text, b->text); // on compare leur contenu
}
```

Toutes ces méthodes sont statiques parce qu'elles ne doivent être appelée que par `new()`, `delete()` ou bien par les sélecteurs. Alors on peut définir la classe `String` :

```c
#include "new.r" // header privé contenant la struct Class

static const struct Class _String = {
    sizeof(struct String),
    String_ctor, String_dtor,
    String_clone, String_differ
};
const void *String = &_String;
```