C avancé : abstraction pointeur
G. Subrenat, Y. Bertrand, N. Courilleau, D. Meneveaux
Licence Informatique L3 - 2021-2022
UFR SFA - Université de Poitiers

Abstraction pointeur

Table des matières

1 Résumé et cadre de travail

Nous nous plaçons dans l'optique suivante :

- Il y a 10 [1] sortes de développeurs :
  - *Nous* : concepteurs du module et programmeurs expérimentés,
  - *Client* : utilisateur du module créé par *nous*, et qui est un programmeur moyen, voire non obéissant : il ne suffit pas de lui donner des directives d'utilisation, il faut l'empêcher de faire une manipulation erronée.
- *Nous* voulons écrire un module (couple *.h*/*.c*) ce qui est l'équivalent d'une classe au sens objet du terme.
- *Nous* voulons que le *client* ne puisse pas manipuler directement la structure de données. Autrement dit seul le module y a accès.
- *Nous* voulons qu'il ne puisse pas la voir (i.e. que le code soit inaccessible).

Rappel : lors d'une conception, on écrit en premier lieu le *.h* sans penser à la structure de données et sans penser au code des fonctions [2]. Cela permet :

- de se concentrer sur les fonctionnalités sans être bridé par d'éventuelles difficultés de programmation,
- à d'autres programmeurs d'utiliser le module avant qu'il soit implémenté [3].

2 Code initial

Voici le code classique d'un tel module :

Jeu.h

```h
#ifndef JEU_H
#define JEU_H

// il ne faut pas utiliser la structure de données
// il faut obligatoirement passer par les méthodes
struct JeuP
{
    char *titre;
    int prix;
};
typedef struct JeuP Jeu;

Jeu creer(const char *titre, int prix);
void detruire(Jeu *self);

void setPrix(Jeu *self, int prix);
int getPrix(const Jeu *self);

void setTitre(Jeu *self, const char *titre);
const char * getTitre(const Jeu *self);

void affiche(const Jeu *self);

#endif
```

Jeu.c (extraits)

```c
void detruire(Jeu *self)
{
    free(self->titre);
    self->titre = NULL;
}

void setPrix(Jeu *self, int prix)
{
    assert(prix >= 0);
    self->prix = prix;
}

void setTitre(Jeu *self, const char *titre)
{
    assert(strcmp(titre, "") != 0);

    // auto-affectation
    if (titre == self->titre)
        return;

    free(self->titre);
    self->titre = strdup(titre);
}
```

Le but est donc que le *client* n'utilise que les méthodes pour manipuler un objet et donc n'ait pas accès à la structure de données.

Or malheureusement c'est le cas, et on ne peut pas l'empêcher de faire un tel accès.

1. 10 en binaire !
2. Directive malgré tout très théorique et difficilement compatible avec le développement d'un module complexe.
3. C'est à dire écrire du code utilisant le module en possédant uniquement son .h.

main.c (ok)

```c
void normal()
{
    Jeu j = creer("Life is strange", 25);
    setPrix(&j, 19);
    setTitre(&j, "Life is Strange!");
    affiche(&j);
    detruire(&j);
}
```

main.c (mauvais)

```c
void erreur()
{
    Jeu j;
    j.prix = -12;
    j.titre = "";
    affiche(&j);
    // appeler detruire conduit à un crash mémoire
}
```

Dans le code erroné, on dénote plusieurs anomalies :

- prix négatif
- titre avec une chaîne vide
- et surtout le titre n'est pas mis avec une allocation dynamique, et un appel ultérieur à *setTitre* ou *detruire* conduira à un crash mémoire.

Un commentaire, dans le *.h*, indiquant qu'il ne faut pas utiliser les membres de la structure est insuffisant : tôt ou tard le *client* le fera et de bonne foi.

3 Principe de résolution

3.1 Première approche

Toute définition de type faite dans un *.c* est locale à celui-ci. C'est ce que nous allons faire : nous déportons la définition du type dans *Jeu.c* (c'est le point clé de l'abstraction pointeur).

Le code devient alors :

Jeu.h

```c
#ifndef JEU_H
#define JEU_H

// la structure n'est plus ici

Jeu creer(const char *titre, int prix);
void detruire(Jeu *self);

void setPrix(Jeu *self, int prix);
int getPrix(const Jeu *self);

void setTitre(Jeu *self, const char *titre);
const char * getTitre(const Jeu *self);

void affiche(const Jeu *self);

#endif
```

Jeu.c (début)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "Jeu.h"

struct JeuP
{
    char *titre;
    int prix;
};
typedef struct JeuP Jeu;

Jeu creer(const char *titre, int prix)
{
    assert(strcmp(titre, "") != 0);
```

Dans le fichier *main.c* on ne peut plus accéder aux membres de la structure car elle n'est plus connue ; c'est le but recherché.

Malheureusement le code correct ne compile plus : il y a une erreur dès le *.h* et par conséquent dans le *main.c*.

Pour rappel, voici le code de *main.c* :

main.c

```c
#include "Jeu.h"

void normal()
{
    Jeu j = creer("Life is strange", 25);
    setPrix(&j, 19);
```

Et le résultat de la compilation :

```
$ gcc -Wall -Wextra -pedantic -std=c99 -c main.c

In file included from main.c:4:0:
Jeu.h:6:1: error: unknown type name 'Jeu'
Jeu creer(const char *titre, int prix);

main.c: In function 'normal':
main.c:8:5: error: unknown type name 'Jeu'
Jeu j = creer("Life is strange", 25);
```

En effet le type *Jeu*, pour *main.c*, n'est défini nulle part et le compilateur joue son rôle en le signalant.

Note : le fichier *Jeu.c* ne compile pas non plus, mais il suffirait de mettre la ligne "`#include "Jeu.h"`" après la définition de la structure. Outre le fait que ce n'est pas élégant (l'ordre et la place des directives d'inclusion ne devraient pas avoir d'importance), cela ne résout pas le problème de la compilation du fichier *main.c*.

Il faut donc affiner ce mécanisme qui est cependant un bon départ.

3.2 Seconde approche

Le problème est donc que le type n'est pas connu. Il existe, en C, un mécanisme de déclaration anticipée pour les types : on indique que le type existe sans le définir [4].

Jeu.h

```
1  #ifndef JEU_H
2  #define JEU_H
3
4  // déclaration anticipée
5  struct JeuP;
6  typedef struct JeuP Jeu;
7
8  Jeu creer(const char *titre, int prix);
9  void detruire(Jeu *self);
10
11 void setPrix(Jeu *self, int prix);
12 int getPrix(const Jeu *self);
13
14 void setTitre(Jeu *self, const char *titre);
15 const char * getTitre(const Jeu *self);
16
17 void affiche(const Jeu *self);
18
19 #endif
```

Jeu.c (début)

```
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <assert.h>
8
9  #include "Jeu.h"
10
11 // les membres de cette structure ne sont
12 // accessibles que localement
13 struct JeuP
14 {
15     char *titre;
16     int prix;
17 };
18
19 Jeu creer(const char *titre, int prix)
20 {
21     assert(strcmp(titre, "") != 0);
22     assert(prix >= 0);
```

Malheureusement il y a encore un problème de compilation.

Pour rappel, voici le code de *main.c* :

main.c

```
4  #include "Jeu.h"
5
6  void normal()
7  {
8      Jeu j = creer("Life is strange", 25);
9      setPrix(&j, 19);
```

```
$ gcc -Wall -Wextra -pedantic -std=c99 -c main.c

main.c: In function 'normal':
main.c:8:5: error: variable 'j' has initializer but incomplete type
Jeu j = creer("Life is strange", 25);

main.c:8:5: error: invalid use of incomplete typedef 'Jeu'
main.c:8:9: error: storage size of 'j' isn't known
Jeu j = creer("Life is strange", 25);
```

4. En C++, la déclaration anticipée d'une classe est fréquente, souvent pour raccourcir les temps de compilation.

Les erreurs sont plus précises et plus intéressantes :

- le type est incomplet
- l'espace mémoire nécessaire pour le type est inconnu

Le fait que le type est incomplet n'est pas un réel problème : c'est ce que l'on voulait.
En revanche la non-connaissance de la taille du type est rédhibitoire : lorsqu'on déclare une variable dans une fonction, le compilateur a besoin de la taille de cette variable pour allouer la bonne taille sur la pile d'appels.

Il reste donc un dernier défi : comment définir un type incomplet dont on connaît la taille ?
Cette question est un non-sens, mais dans la section suivante nous allons légèrement la reformuler et profiter d'une permissivité du C pour arriver à nos fins.

4 Résolution

4.1 Première approche

Le compilateur ne permet pas définir une variable dont le type est incomplet, car, comme nous l'avons vu, il en a besoin pour réserver la mémoire sur la pile d'appels.

En revanche il autorise de manipuler des pointeurs sur des types qu'il ne connaît pas.

Pourquoi ? Parce que tous les pointeurs (*void **, *int **, *struct Image **, ...) occupent la même taille en mémoire (8 octets sur mon architecture).

L'utilisation de tels pointeurs est autorisée ... tant que l'on accède pas à l'objet pointé. Et c'est exactement ce que nous voulons.

Le revers de la médaille est le suivant : comme on manipule désormais des pointeurs sur notre type *Jeu*, il faudra allouer l'espace pour la structure (et la désallouer en fin de vie). Ce sera le rôle de la fonction de construction, et donc ce sera complètement transparent pour le *client*.

Jeu.h

```c
#ifndef JEU_H
#define JEU_H

// déclaration anticipée
struct JeuP;
typedef struct JeuP Jeu;

Jeu * creer(const char *titre, int prix);
void detruire(Jeu *self);

void setPrix(Jeu *self, int prix);
int getPrix(const Jeu *self);

void setTitre(Jeu *self, const char *titre);
const char * getTitre(const Jeu *self);

void affiche(const Jeu *self);

#endif
```

main.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "Jeu.h"

void normal()
{
    Jeu *j = creer("Life is strange", 25);
    setPrix(j, 19);
    setTitre(j, "Life is Strange!");
    affiche(j);
    detruire(j);
}

int main()
{
    normal();

    return EXIT_SUCCESS;
}
```

On remarque que pour *Jeu.h* la seule différence avec la version précédente est que la méthode *creer* renvoie désormais un pointeur sur un *Jeu*.

Le fichier *main.c* s'adapte en déclarant un pointeur sur un *Jeu*. Et lors des appels aux méthodes, l'écriture est un peu plus simple car la variable est déjà un pointeur.

Pour *Jeu.c* il faut adapter le constructeur qui a une allocation en plus, et le desctructeur qui a une désallocation supplémentaire.

Jeu.c

```c
struct JeuP
{
    char *titre;
    int prix;
};

Jeu * creer(const char *titre, int prix)
{
    assert(strcmp(titre, "") != 0);
    assert(prix >= 0);

    Jeu *self = malloc(sizeof(Jeu));

    // ou strdup
    self->titre = malloc((strlen(titre)+1) * sizeof(char));
    strcpy(self->titre, titre);
    self->prix = prix;

    return self;
}

void detruire(Jeu *self)
{
    free(self->titre);
    free(self);
}
```

4.2 Solution finale

Nous avons une solution pleinement opérationnelle. Nous allons juste améliorer la lisibilité pour que le *client* n'ait même plus besoin de savoir qu'il manipule un pointeur.

Pour cela le type *Jeu* sera directement un pointeur sur la structure. Ainsi le signe "*" va disparaître de tous les prototypes.

En outre, le destructeur est légèrement amélioré pour que le pointeur de type *Jeu* soit mis à *NULL* après la désallocation ; pour cela il faut passer, en paramètre, un pointeur sur le pointeur.

Jeu.h

```c
#ifndef JEU_H
#define JEU_H

struct JeuP;
typedef struct JeuP * Jeu;
typedef const struct JeuP * const_Jeu;

Jeu creer(const char *titre, int prix);
void detruire(Jeu *self);

void setPrix(Jeu self, int prix);
int getPrix(const_Jeu self);

void setTitre(Jeu self, const char *titre);
const char * getTitre(const_Jeu self);

void affiche(const_Jeu self);

#endif
```

main.c

```c
#include <stdio.h>
#include <stdlib.h>

#include "Jeu.h"

void normal()
{
    Jeu j = creer("Life is strange", 25);
    setPrix(j, 19);
    setTitre(j, "Life is Strange!");
    affiche(j);
    detruire(&j);
}

int main()
{
    normal();

    return EXIT_SUCCESS;
}
```

Jeu.c

```c
struct JeuP
{
    char *titre;
    int prix;
};

Jeu creer(const char *titre, int prix)
{
    assert(strcmp(titre, "") != 0);
    assert(prix >= 0);

    Jeu self = malloc(sizeof(struct JeuP));

    // ou strdup
    self->titre = malloc((strlen(titre)+1) * sizeof(char));
    strcpy(self->titre, titre);
    self->prix = prix;

    return self;
}

void detruire(Jeu *self)
{
    free((*self)->titre);
    free(*self);
    *self = NULL;
}
```

Enfin on note un point technique avec l'apparition du type *const_Jeu* :

```c
typedef const struct JeuP * const_Jeu;
```

En effet lorsque qu'on passe un *Jeu* en paramètre à une fonction qui ne le modifie pas, il faut le préciser avec le mot-clé *const*. Malheureusement le type *"const Jeu"* indique que le pointeur est constant et non pas l'objet pointé comme on le souhaiterait, d'où la nécessité du type supplémentaire.

5 Bilan

D'un point de vue "génie logiciel", l'abstraction pointeur est idéale :

- on ne peut pas accéder aux membres de la structure (qui sont de fait *private*)
- on ne peut pas voir le code encapsulé (il suffit de fournir le *.o* du module au lieu du *.c*)
- une modification du code interne du module n'implique pas une recompilation de tous les fichiers l'utilisant, mais seulement du code source du module

Le principe de l'abstraction pointeur s'appuie une permissivité du C : on peut manipuler des pointeurs sur des types inconnus. Tout cela parce que tous les pointeurs ont la même taille en mémoire. C'est intellectuellement peu satisfaisant.

Il y a deux inconvénients :

- un surcoût mémoire : il y a un pointeur supplémentaire pour chaque objet instancié (8 octets sur mon architecture).
- un surcoût temporel : la création d'un objet nécessite une allocation mémoire qui est coûteuse en temps système [5] ; de même pour la désallocation.

Problème mémoire : ce problème est relatif au type de module :

- S'il s'agit d'un module d'*Image*, le prix d'un pointeur supplémentaire est négligeable.
- S'il s'agit d'un module de *Pixel* (codé sur 3 caractères) alors l'espace mémoire est triplé, ce qui signifierait qu'une image occuperait trois fois plus d'espace mémoire. Dans ce cas le surcoût est prohibitif.

Problème temps : ce problème est relatif au type de module :

- S'il s'agit d'un module de *Liste*, le nombre d'opérations que l'on fait, pour une seule création, rend le surcoût temporel négligeable.
- S'il s'agit d'un module de *Vecteur* (mathématique), alors il se peut que dans une boucle on ait besoin d'un *Vecteur* temporaire. Dans un tel cas il peut y avoir un cycle de constructions/destructions important induisant un surcoût prohibitif.

Enfin soyons clairs (et honnêtes !) :

- Si on veut faire de l'objet, autant prendre un langage objets.
- Mais c'est un excellent exercice pour comprendre les pointeurs en C.

5. il peut être intéressant dans un tel cas de réécrire son mécanisme d'allocation