

Méthode pour écrire du C Orienté Objet

Cédric Cellier

rixed@happyleptic.org

pour relecture

1. Pourquoi de la POO en C ?

Le point de départ de l'analyse objet, c'est l'idée que les motifs comportementaux sont plus important que les algorithmes. L'analyse objet s'occupe donc de classer ces motifs en familles et de mettre en relief leurs relations - par exemple un motif *formulaire* que l'on peut *remplir* ou *valider*, et qui utilise une *liste* à laquelle on peut *ajouter* ou *retirer* des éléments pour contenir une série de *boîtes à cocher* que l'on peut *cocher* ou *décocher*). Succèdent ainsi aux organigrammes d'autres type de schémas, faisant apparaître les motifs, leurs comportements possibles ainsi que les relations qu'ils entretiennent, ou bien la vie d'une instance donnée de tel motif dans ses relations aux autres, etc...

Le choix de ce que l'on considère être un motif est généralement dicté par ce qui constitue une bonne unité d'information pour le problème traité, ainsi que par l'ambition de pouvoir réutiliser certains de ces composants dans un autre problème ; véritable esprit derrière la machine, la *réutilisabilité* est en effet devenu le principal objectif.

La programmation objet consiste à transposer cette analyse dans le processus de développement, en créant des types de données correspondant à chaque famille de motif comportemental, de telle façon qu'aux relations entre ces motifs correspondent des relations similaires entre ces types, puis en définissant des fonctions pour exécuter chacun des comportements spécifiques à chaque motif.

Le sujet de cet article est d'exposer une méthode de programmation qui permet de satisfaire ces exigences en langage C, en tenant compte des performances.

Si les avantages de la POO sont largement reconnus, je m'attend par contre à rencontrer de l'étonnement quant au choix du langage C pour réaliser un programme analysé en termes d'objets. Le C++ ou l'Objective C, par exemple, ne sont-ils pas plus adaptés à la mise en oeuvre de la conception objet, ayant été inventés, à partir du C, pour remplir précisément cette fonction ? On associe tellement la programmation objet aux langages dits "objets" qu'on pourra se demander s'il est simplement *possible* de pratiquer la POO en C.

Clarifions tout de suite ce dernier point : que certains langages de programmation soient dit "objets" traduit simplement l'existence de mots clefs supplémentaires qui facilitent l'écriture (et la lecture) des programmes. La *Programmation Orientée Objet* (POO) est une méthode de programmation qui ne dépend pas, contrairement à ce que l'on pense souvent et enseigne parfois, de l'existence ou pas de ces mots-clés. La question de sa mise en oeuvre peut donc être légitimement posée dans tous les langages.

Évoluer en langage C nous obligera à implémenter nous même les mécanismes cachés derrière ces mots clefs des langages objets, supposés familiers au lecteur, nous offrant donc de comprendre comment ils fonctionnent.

Avant de continuer, énonçons donc quelques définitions adaptées au C¹ :

objet

La norme du C définit un objet comme une zone de mémoire (formée de bytes contigus) contenant une valeur d'un certain type. Cette définition est parfaitement adaptée à la POO, nous n'y changerons donc rien.

type

Désigne n'importe quel type de base, ou un type élaboré à l'aide des mots-clés struct, enum, union ou typedef. Nous n'ajouterons rien à cette définition.

type complètement défini, type incomplètement défini

En C, un type est dit complètement défini si le compilateur a connaissance de son codage interne. Un type incomplet, ou incomplètement défini, en revanche, est créé chaque fois qu'une structure ou une union est déclaré sans être définie, ou si sa définition inclue un type incomplet.

Le seul usage légal d'un type incomplet réside dans la construction d'un autre type (incomplet lui aussi) avec un typedef, ou dans la construction d'un type pointeur vers ce type. C'est dans ce dernier cas que se cache toute l'astuce : un pointeur vers un type incomplet constitue un type complet.

Par exemple :

```
struct incompl;  
  
union incomp2 {  
    int a;  
    struct incompl b;
```

```
};  
  
struct complet {  
    int b;  
    union incomp2 *ptr;  
};
```

créé deux types incomplètement définis (struct incomp1 et union incomp2), et un type complètement défini (struct complet). Les deux premiers resteront incomplets tant que le compilateur ne rencontrera pas la définition (complète) de struct incomp1.

type abstrait (ADT)

Terme consacré par l'usage pour désigner un type implémentant un motif destiné à être largement réemployable. Le type est généralement incomplet au moment où on s'en sert, interdisant d'accéder à sa valeur, mais exportant les *méthodes* qui implémentent son comportement.

On ne distinguera pas systématiquement type et type abstrait ; le but de la POO est d'ailleurs de confondre ces deux notions. L'acronyme provient de l'anglais Abstract Data Type.

méthode

Fonction implémentant le comportement d'un type abstrait. Les méthodes ont accès au codage interne de leur type (en d'autres termes : le type auquel elles s'appliquent sont complètement définis lorsqu'elles sont compilées).

accesseur

Méthode d'un type renvoyant une partie de sa valeur (souvent un simple membre de sa structure privée) plutôt que d'implémenter un comportement. Ce sont les "fameux" `get_x()` et `get_y()` du type coordonnée, qui ne font en réalité qu'exposer le codage interne du type.

classe, instance

Désignent en C++ respectivement ce que nous avons appelé ici "type abstrait" et "objet". Nous ne nous serviront pas de ces termes, préférant la simplicité des termes "type" et "objet" du C.

membre, fonction membre

En C, un membre est un élément (nommé) d'une structure ou d'une union. Ce terme nous convient parfaitement. Nous ne parlerons en revanche pas de fonction membre, terme laissant à penser que la fonction elle-même ou un pointeur vers elle est incorporée dans la structure, ce qui n'est jamais le cas. Une fonction membre du C++ est ici simplement une méthode.

constructeur

Méthode qui initialise un objet donné en fonction de ses paramètres ou de valeurs par défaut.

destructeur

Méthode qui libère les ressources éventuellement allouées lors de l'initialisation de cet objet.

allocateur

Méthode qui alloue de la mémoire pour un objet, avant d'appeler (éventuellement) le constructeur, et qui retourne l'adresse de l'objet construit.

libérateur

Méthode qui appelle éventuellement le destructeur puis libère la mémoire allouée par l'allocateur.

module ou unité de compilation

L'unité de compilation est le bloc de code qui sera compilé en une fois indépendamment des autres blocs de codes qui peuvent être plus tard rassemblés par l'éditeur de lien pour former le programme ou la bibliothèque.

espace de nom

Les espaces de nom bien pratiques du C++ n'ont pas leur équivalent en C, mais on peut les simuler en préfixant tous les symboles publics (les noms de variables globales ou de fonctions qui ne sont pas déclarées static, ainsi que les types ou constantes déclarées dans les éventuels fichiers de déclarations).

2. Qu'est-ce qu'un type ?

Les définitions varient d'un auteur à l'autre, mais nous devons au moins distinguer trois concepts : *type*, *valeur* et *variable*.

Certains auteurs, faisant du type le type idéal du monde réel, ajoutent le concept de représentation pour désigner le ou les codages informatiques des valeurs d'un type². Pour alléger le texte, choisissons quant à nous des définitions qui soient proches du C, donc relativement "bas niveau", quitte à perdre en généralité. Désignons donc par type la même chose qu'en C, c'est à dire un type de base ou bien un ensemble de types de bases agglomérés dans une structure ou une union, associé à un nom (et on l'espère à une signification). Ce codage induit donc un ensemble de valeurs possibles. Ajoutons qu'un type renseigne aussi sur les valeurs permises de cet ensemble. En dernier lieu, un type participe à certaines opérations, en tant que membre ou que résultat. Chaque type connaît au minimum l'opération qui consiste à affecter une valeur à une variable. L'opérateur C dénoté "=" réalise toujours cette affectation mais certains types peuvent fournir des fonctions d'affectations plus élaborées.

La variable, c'est l'objet qui contient une valeur d'un type donné, typiquement en RAM, et qui possède généralement un nom. Cette valeur sera amenée à changer (ou pas) mais ni son emplacement ni son type ne changeront (un des objectifs de la POO consiste cependant à rendre possible l'utilisation de variables dont on ne connaît pas le type exact, et qui peuvent prendre l'apparence de plusieurs types). Pour être valide, une variable doit contenir une valeur permise pour son type. Cette protection ne pouvant être assurée, pour les types élaborés, que par le programme, on prendra donc soin, en mode DEBUG, de vérifier à chaque occurrence d'une variable qu'elle est valide.

Par exemple, voici le type abstrait Address³ :

```
typedef struct {
    unsigned street_nbr; // doit etre > 0
    char *street;        // peut valoir "" mais pas NULL
    char zip_code[5];    // padding à gauche avec des '0'
    City *city;          // peut valoir NULL, dans ce cas considérer place_known_as
    char *place_known_as; // Ne doit pas être NULL si city est NULL
} Address;
```

Et des variables de ce type :

```
// l'adresse n'existe pas, mais elle est valide !
Adresse chez_moi = { 71, "rue de Clignancourt", "75016", &paris, NULL };
Adresse ailleurs; // non définie, donc pas encore vraiment une variable pour nous
```

Vous noterez l'usage du typedef pour faire disparaître le mot-clé struct ainsi que la majuscule aux noms de type, bien pratique pour distinguer le type City du membre *city*. On notera systématiquement les noms des types abstraits en majuscule sans underscore (à la java), et on réservera cette notation à cette catégorie d'identificateurs.

Dans la pratique, on déclarera dans le fichier de déclarations publiques l'existence du type Address de la façon suivante :

```
typedef struct Address Address;
```

et il suffira donc de définir struct Address dans le module de compilation de ce type.

Ajoutons une fonction pour vérifier la validité d'une Address :

```
int Address_is_valid(const Address *this) {
    assert(this);
    if (0==this->street_nbr ||
        !this->street ||
        (!this->city && !this->lieu->dit)
    ) return 0;
    for (int j=0; j<5; j++) {
        if (!isdigit(this->zip_code[j])) return 0;
    }
    return 1;
}
```

Remarquez que l'on passe un pointeur sur l'objet de type Address plutôt qu'une copie de l'objet pour des raisons de performances. On indique cependant (au compilateur autant qu'au programmeur qui utilisera cette fonction) que l'Address ne sera pas modifiée par le mot-clé const. On utilisera systématiquement le premier argument d'une fonction qui travaille sur un type pour passer ce pointeur sur le type, et on prefixera également le nom de la fonction par le nom du type, afin de ne pas donner le même nom à des symboles différents. Remarquez que le C99 impose aux compilateurs de considérer au minimum les 31 premiers caractères des noms pour les différencier, ce qui laisse une certaine marge.

Une telle fonction, accédant aux codage interne d'un objet pour en révéler des informations (ici, l'information sur la validité de l'objet) sont appelées *méthodes* de ce type. Toutes les méthodes d'un type sont généralement implémentées dans une seule unité de compilation, incluant la définition du type.

3. Construction, Destruction

Cacher les détails d'implémentation d'un type abstrait, c'est d'abord cacher le codage interne de ce type. L'utilisateur du type abstrait n'a donc plus la facilité de l'opérateur d'initialisation "=" pour initialiser les variables de ce type. Il n'est même pas certain a priori que le "=" de l'affectation, qui peut aussi être utilisé lors de l'initialisation d'une variable pour lui donner la valeur d'une autre variable déjà initialisé, soit fonctionnel. En effet, rien n'oblige le type abstrait d'être réalisé par une structure, rien n'oblige non plus cette variable à être toujours valide après affectation (l'objet peut contenir une ressource qui n'est pas partageable, tel un pointeur vers un buffer ou bien un descripteur de fichier), et surtout le compilateur ne connaît pas nécessairement lui non plus, au moment de compiler ce module, le codage interne du type abstrait (qui peut avoir été déclaré mais non complètement défini). C'est d'ailleurs le meilleur moyen de

s'assurer que l'utilisateur ne va pas commettre l'imprudence d'accéder directement aux membres d'une structure que de ne pas la définir publiquement.

L'utilisateur d'un type abstrait devra donc se référer à sa documentation (dans son fichier de déclarations publiques) pour savoir comment le créer. En passant, insistons lourdement sur l'importance de signifier également dans ce fichier, dans le cas où le type est complètement défini pour l'utilisateur, si l'opération d'affectation fonctionne ou pas.

Nous voilà donc privé de l'opérateur d'initialisation "=", de l'opérateur d'affectation "=", obligé de nous référer à la documentation pour savoir comment initialiser une variable. Cela peut sembler une perte de temps mais il n'en est rien : il aurait de toute façon fallu chercher dans les fichiers de déclaration la définition du type (procédé hautement récursif) pour pouvoir l'initialiser directement ; quand à l'opérateur de copie, son utilisation est de toute façon à proscrire pour les types de donnée dont on n'a pas complètement saisi le fonctionnement (comment savoir que l'objet référencé compte ses références sans étudier soigneusement le code ?) Autant d'investissement qu'il faudra réitérer à chaque changement de version de la librairie utilisée. Dans la pratique, l'argument du temps perdu par le programmeur n'est donc pas valide.

Il faut donc définir et documenter au moins deux fonctions : l'une construisant la variable grâce à tous les paramètres nécessaires, l'autre la détruisant. Vous remarquerez le subtil glissement de notion d'initialisation (qui peut se faire n'importe quand) aux concepts de création/destructions qui laisse penser que la variable ne peut être créée qu'une seule fois puis détruite, et qu'elle n'a d'existence qu'entre ces deux moments. Techniquement rien n'empêche à la même variable d'être construite puis détruite puis reconstruite plusieurs fois, mais la séquence construction puis destruction doit généralement être respectée.

Il peut être utile ici d'insister sur un point : la construction d'un objet est l'équivalent non pas de la déclaration d'une variable classique, mais de son initialisation. L'objet doit donc déjà exister, non initialisé, quelquepart en mémoire.

Ces fonctions seront toujours nommées `Type_construct` et `Type_destruct`, où `Type` est à remplacer par le préfixe en vigueur, et leur premier argument est, comme toujours, l'adresse de l'objet du type `Type` que l'on souhaite construire ou détruire. Bien souvent `Type_destruct` ne prendra d'ailleurs pas d'autres arguments. `Type_construct` par contre peut avoir besoin de beaucoup d'autres arguments.

Le bon déroulement de ces deux fonctions n'étant pas garanti, il faut renvoyer un code d'erreur (qui peut souvent être ignoré dans le cas du destructeur). J'ai choisi par habitude de faire renvoyer `NULL` au constructeur en cas d'erreur et l'adresse de l'objet en cas de succès, et de faire renvoyer `(int)0` en cas d'échec du destructeur, et `(int)1` en cas de succès du destructeur. On peut évidemment faire d'autre choix, l'essentiel étant de le documenter et de s'y tenir.

Il peut sembler arbitraire de retourner l'adresse de la variable construite, plutôt qu'un code d'erreur entier. Le but est le suivant : permettre aux constructeurs d'allouer eux mêmes l'espace mémoire pour la variable dans le cas où l'utilisateur passe une adresse valant `NULL` au constructeur, ceci afin de ne pas

avoir un constructeur et un allocateur distincts (l'allocateur est alors une simple macro ou une fonction inline) ce qui est plus efficace lorsque la liste de paramètre à faire suivre au constructeur est longue.

Illustrons ceci avec les méthodes de construction/destruction du type Address :

```
Address *Address_construct(Address *this, unsigned street_nbr, char *street,
                           char *zip_code, City *city, char *place_known_as) {
    assert(this); // pas d'allocation automatique pour ce type.
    this->street_nbr = street_nbr;
    this->street = street;
    memcpy(this->zip_code, zip_code, sizeof(this->zip_code));
    this->city = city;
    this->place_known_as = place_known_as;
    if (!Address_is_valid(this)) return NULL;
    City_reference(city);
    return this;
}

int Address_destruct(Address *this) {
    assert(this);
    assert(Address_is_valid(this));
    if (this->city) City_dereference(this->city);
    return 1;
}
```

Plusieurs remarques :

- Le pointeur vers l'objet auquel on applique la méthode s'appelle `this` - d'autres préfèrent `self`, l'un et l'autre étant des mots clés réservés respectivement en C++ et en Objective-C ;
- On assume (`assert`) que `this` est non nul. Nous ne sommes donc pas, pour simplifier, dans le cas où le constructeur fait l'allocation lui même. Rappelez vous : l'objet doit déjà exister, non initialisé, quelquepart en mémoire. La macro `assert` est employée plutôt qu'un test d'erreur car nous estimons ici que `this` ne peut pas valoir `NULL` suite aux conditions du déroulement du programme (entrées, environnement...) mais uniquement à cause d'un bug du programme lui même ;
- Nous avons imaginé que le type `City` utilise un compteur de référence, afin d'avoir une ressource à libérer dans le destructeur, et de pouvoir illustrer le rôle du destructeur (qui ne libère pas la mémoire consommée par l'objet). Mais il n'est pas rare qu'un destructeur ne fasse rien, si ce n'est peut-être appeler d'autres destructeurs qui eux mêmes ne font rien d'autre ;

Rien n'interdit de proposer plusieurs constructeurs pour un type. Par exemple, `Address` pourrait aussi se construire grâce à une fonction de géocodage (en lui fournissant longitude et latitude), etc... On peut aussi créer un objet de type `Address` en recopiant un autre objet de ce type :

```
Address *Address_dup(Address *this, Address *src);
```

que l'on aurait pu nommer également `Address_copy` (mais `Address_dup` évoque d'avantage la duplication de ressource que la simple copie).

Nous avons dit tout à l'heure que le programmeur ne perdait pas son temps à utiliser un constructeur plutôt qu'une initialisation directe, grâce à l'indépendance acquise vis à vis de l'implémentation. Par contre, le temps et la place perdu par le programme, eux, sont réels. Avec une initialisation directe d'une variable globale ou de portée locale mais statique, l'initialisation de la variable est faite une fois pour toute par le compilateur ou l'éditeur de liens, et le programme arrive à la vie avec ces données déjà initialisées. Avec un constructeur, au contraire, le programme commence sa tâche par un laborieux travail d'initialisations. La dernière section de cet article expose des techniques pour circonvier ce problème.

Si le type abstrait est incomplètement défini pour l'utilisateur (ce qui doit être le cas général), on se trouve dans l'impossibilité de déclarer une variable de ce type ; le constructeur, qui suppose que la variable est déjà déclarée (et qui requiert d'ailleurs son adresse) ne nous est donc d'aucune utilité pour le moment.

4. Allocations

Lorsqu'un type abstrait est incomplètement défini, il ne vous aura pas échappé une contrainte supplémentaire : on ne peut pas déclarer une variable locale de ce type.

Si on connaissait la taille de l'objet, on pourrait au moins allouer un bloc mémoire et utiliser son constructeur dessus. Ce bloc mémoire pourrait être rendu par `malloc` ou toute autre librairie de gestion de mémoire, ou bien pourrait même être pris sur le tas en le déclarant par exemple comme ceci :

```
char addy_c[Address_sizeof()];  
Address * const addy = (Address *)addy_c;
```

Puisqu'en C on peut désormais allouer des tableaux de taille non constante (`Address_sizeof` étant au pire une fonction, donc évalué à l'exécution).

Vous noterez qu'il est évidemment impossible de déclarer de la sorte des variables globales.

Connaissant la taille, il est par contre possible d'appeler `malloc` ou toute autre fonction allouant de la mémoire. Puisqu'il faut lancer le constructeur juste après, on fournit généralement une fonction (ou une macro) d'allocation qui fait les deux, de même signature que le constructeur sauf qu'elle ne prend pas `this` en premier argument, et renvoi un `Type *` ou `NULL` en cas d'erreur. Cette méthode étant le pendant du `new` du C++, on la dénommera `Type_new`.

De même, on fournit généralement la fonction qui détruit l'objet et libère la mémoire réservée par l'allocateur. On note ce libérateur d'après le mot-clé C++ `del` : `Type_del`

Ex :

```
Address *Address_new(.....) {
    Adresse *this = malloc(sizeof(*this));
    if (!Address_construct(this, ...)) {
        free(this);
        this = NULL;
    }
    return this;
}

int Address_del(Address *this) {
    assert(this);
    int ret = Address_destruct(this);
    free(this);
    return ret;
}
```

La méthode `new` peut être une simple macro définie dans l'entête public et appelant le constructeur d'`Address` avec le paramètre `this` valant `NULL`, à condition de modifier la méthode `Address_construct` pour qu'elle alloue de la place pour `this` dans ce cas.

L'inconvénient d'appeler directement `malloc` et `free` c'est que l'utilisateur peut préférer utiliser d'autres fonctions de réservation / libération de mémoire, ne serait-ce que pour tirer profit du fait que de nombreux blocs de même taille vont être alloués (alors que `malloc` traite chaque allocation en toute généralité).

On prendra donc soin de ne pas appeler `malloc` mais `obj_malloc` avec une signature un peu plus riche :

```
void *obj_malloc(const char *file, const char *line, const char *object,
unsigned size);
```

```
void *obj_free(const char *file, const char *line, const char *object, void
*ptr);
```

Appelés comme ceci :

```
#define obj_malloc(f,l,o,s) obj_malloc(__FILE__, __LINE__, OBJECT, s)
#define obj_free(f,l,o,p) obj_free(__FILE__, __LINE__, OBJECT, p)
```

avec la "constante" `OBJECT` redéfinie dans chaque module avec le nom de l'objet. Les fonctions `obj_malloc_` et `obj_free_` doivent être exportées par l'utilisateur.

Il y a beaucoup à dire et à faire au sujet des fuites mémoires, mais ce n'est pas notre sujet. Citons simplement la règle d'or : il faut s'efforcer de détruire et libérer les objets dans la fonction⁴ où ils ont été alloués et construit, sauf dans le cas où une ressource est alloué par un constructeur - elle doit alors être détruite par le destructeur. Les fuites mémoires deviennent alors des erreurs triviales.

Parfaits pour les grands objets isolés, ces allocateurs ne sont cependant pas suffisants. En effet, on considère souvent plutôt des ensembles d'objets que des objets individuels, les ensembles en question pouvant être de type différent selon la manière dont on veut retrouver des éléments et parcourir cet ensemble : liste, arbre, hash...

Ces ensembles sont donc eux mêmes des types de données abstraits, dont les méthodes incluent la recherche d'élément, l'ajout ou la suppression d'élément, l'information sur le nombre d'éléments contenus, et l'allocation d'un élément vierge au sein de l'ensemble⁵.

C'est donc plus spécifiquement cette méthode d'allocation d'un ensemble qui à besoin d'utiliser `malloc` (ou `obj_malloc` ou autre), la plupart des autres objets étant construit dans une zone allouée dans un ensemble.

5. Méthodes

L'un des aspect essentiel du découpage en types abstraits consiste à choisir les méthodes du type. Cette question est plus critique que le choix d'un codage interne, qui lui peut se modifier à tout moment sans trop d'impacts. Cette question relève cependant davantage de l'analyse que de la réalisation.

Nous n'insisterons donc que sur un point : il faut à tout prix éviter d'implémenter des accesseurs : cela ne sert en effet à rien de chercher à s'abstraire du codage interne d'un type si on le réintroduit avec des méthodes `Type_get_tel_membre()`. Il faut au contraire chercher les opérations que l'on désire réaliser avec ce type avant de se demander de quoi il seront constitués (d'où le concept de motifs comportementaux). Commencer par offrir des accesseurs (`Type_get_machin()`) traduit généralement que l'analyse objet n'a pas été poussée assez loin.

Pour ce qui est de la réalisation pratique des méthodes, l'essentiel à déjà été dit :

- préfixer le nom de la méthode du nom du type auquel elle s'applique ;
- passer en premier paramètre un pointeur vers l'objet de ce type, appelé `this` ;
- commencer par `assert(this && Type_is_valid(this))` ;
- documenter et déclarer la méthode dans un (unique) fichier de déclarations publiques pour ce type ;
- implémenter toutes les méthodes d'un type dans une (unique) unité de compilation incluant la définition complète du type (et éventuellement la définition complète d'autres types, à vos risques et périls). Déclarer `static` toutes les autres fonctions de cette unité de compilation ;

6. Relations

On distingue couramment trois variétés de relations entre motifs :

- l'*utilisation* (relation *USES A*), qui indique qu'un type en utilise un autre - par exemple, un formulaire peut utiliser une fenêtre pour s'afficher ;
- la *composition* (relations *HAS A*), qui indique qu'un type est constitué d'un autre - par exemple, un formulaire est composé d'une liste ;
- la *spécialisation* (relation *IS A*), qui indique qu'un type est une variante d'un autre - par exemple, une boîte à cocher est une spécialisation d'une question ;

La première relation entre deux types, la relation *USES A*, s'implémente classiquement par un simple pointeur vers l'objet utilisé, et éventuellement un compteur de références vers cet objet. Inutile dans ce cas que l'objet utilisé soit complètement défini. Dans nos exemples, le type *Form* *utilise* le type *Window*.

Typiquement, l'objet utilisé préexiste à la construction de l'objet "utilisant". Il suffit donc de passer son adresse au constructeur pour que celui-ci puisse la mémoriser. Il ne faut donc pas détruire l'objet utilisé dans le destructeur de l'objet qui l'utilise (règle d'or de la gestion des ressources : l'objet qui alloue désalloue). Si l'objet utilisé possède un compteur de référence (ou équivalent), il faut juste l'incrémenter à la construction et le décrémenter dans le destructeur comme cela à été fait pour l'objet *city* dans l'implémentation de *Address_construct* donnée en exemple.

La seconde relation entre deux types est la relation *HAS A*, rendue par l'inclusion d'un objet du type possédé dans la structure du type possédant. Il faut alors que le type possédé soit complètement défini pour le constructeur du type possédant. Rien n'interdit d'utiliser une relation *USES A* exclusive pour rendre le même rapport, mais l'inclusion directe d'une structure dans une autre est généralement plus efficace puisqu'elle n'implique pas d'allocation/libération mémoire distincte, et supprime un niveau d'indirection.

Ainsi, le type *Person* qui *utilise* le type *Address* se construit et ce détruit ainsi :

```
--- person.h ---

#include "gender.h"

typedef struct Person Person;

/* This is a documentation for Person
 * blablabla
 * You cannot use "=" to copy it.
 */
Person *Person_construct(Person *this, char *name, char *first_name, Gender *gender,
                        unsigned street_nbr, char *street,
                        char *zip_code, City *city, char *place_known_as);
```

```
--- person_i.h ---

#include "person.h" // mon header public
#include "address_i.h" // header privé de Address

struct Person {
    char *name, *first_name;
    Gender *gender; // USES_A
    Address address; // HAS_A
};

--- person.c ---

#include "person_i.h"

Person *Person_construct(Person *this, char *name, char *first_name, Gender *gender,
                        unsigned street_nbr, char *street,
                        char *zip_code, City *city, char *place_known_as) {
    assert(this);
    if (!gender) return NULL;
    this->name = name;
    this->first_name = first_name;
    this->gender = gender;
    if (!Address_construct(&this->address,
                          street_nbr, street, zip_code, city, place_known_as)) {
        return NULL;
    }
    return this;
}

int Person_destruct(Person *this) {
    assert(this);
    return Address_destruct(&this->address);
}
```

En cas de référence, il aurait fallu appeler l'allocateur du type utilisé (ici, `Address_new`) dans le constructeur du type utilisant (ici, `Person_construct`) en vérifiant bien sur le code de retour, et il aurait fallu appeler son libérateur (`Address_del`) dans le destructeur (`Person_destruct`) :

```
--- person_i.h ---

#include "person.h" // mon header public
#include "address_i.h" // header privé de Address

struct Person {
    char *name, *first_name;
    Gender *gender; // USES_A
    Address *address; // HAS_A, par reference
};

--- person.c ---
```

```

#include "person_i.h"

Person *Person_construct(Person *this, char *name, char *first_name, Gender *gender,
                        unsigned street_nbr, char *street,
                        char *zip_code, City *city, char *place_known_as) {
    assert(this);
    if (!gender) return NULL;
    this->name = name;
    this->first_name = first_name;
    this->gender = gender;
    if (! (this->address=Address_new(street_nbr,street,zip_code,city,place_known_as))) {
        return NULL;
    }
    return this;
}

int Person_destruct(Person *this) {
    assert(this);
    return Address_del(this->address);
}

```

La troisième et dernière relation entre deux types est la relation IS A, qui spécialise un type. Cette relation est réalisable pas le même mécanisme que la relation HAS A, à une différence près : on sera souvent amené, pour des raisons exposées dans la suite, à calculer l'adresse de l'objet du type parent inclus dans l'objet du type dérivé. Ce calcul s'effectue grâce à la fonction (ou macro) `offsetof`, définie dans `stddef.h`, de la façon suivante :

```
Derive *this = (Derive *) ((char *)parent - offsetof(Derive, membre_parent));
```

Pour cette raison, on préférera mettre `membre_parent` au début de la structure `Deriv`, de telle sorte qu'à la compilation toute cette ligne ne génère aucun calcul (l'offset valant alors 0). Bien sur, comme rien n'interdit d'avoir plusieurs parents, tous ces offsets ne peuvent pas être nuls.

Ce mécanisme, lorsqu'il réalise une relation de spécialisation, est désigné *héritage*. On dit que le sous-type hérite de son type parent.

Voyons pour illustrer l'héritage le type abstrait `Worker`, spécialisant le type abstrait `Person` :

```

--- worker.h ---

#include "gender.h"

typedef struct Worker Worker;

/* This is a documentation for Worker
 * blablabla
 * You cannot use "=" to copy it.
 */

```

```

Worker *Worker_construct(Worker *this, unsigned salary,
                        char *name, char *first_name, Gender *gender
                        unsigned street_nbr, char *street,
                        char *zip_code, City *city, char *place_known_as);

--- worker_i.h ---

#include "worker.h" // mon header public
#include "person_i.h" // header privé de Person

struct Worker {
    Person person; // relation IS_A
    unsigned salary;
};

--- worker.c ---

#include "worker_i.h"

Worker *Worker_construct(Worker *this, unsigned salary,
                        char *name, char *first_name, Gender *gender
                        unsigned street_nbr, char *street,
                        char *zip_code, City *city, char *place_known_as) {
    assert(this);
    if (! Person_construct(&this->person, name, first_name, gender,
                          street_nbr, street, zip_code, city, place_known_as)) {
        return NULL;
    }
    this->salary = salary;
    return this;
}

int Worker_destruct(Worker *this) {
    assert(this);
    return Person_destruct(&this->person);
}

```

Rien de vraiment neuf, donc, et il semble pour le moment que cette relation n'apporte rien de fondamentalement différent par rapport à la relation HAS A.

7. Spécialisation

Spécialiser un type, c'est créer un type qui dérive d'un autre type et qui en modifie certaines méthodes (on dit qu'il les *surcharge*). Par exemple, nous allons voir le type `Worker` qui est une spécialisation du type `Person` en modifiant la méthode `Person_print` pour qu'elle affiche également le `salary`. Parce que le type `Person` possède cette propriété que son comportement peut être modifié par des types dérivés, on dit qu'il est *polymorphe*.

Le seul mécanisme sur lequel nous pouvons nous appuyer est le pointeur de fonction. Au lieu d'appeler directement une méthode *spécialisable* de `Person`, on doit donc systématiquement employer un pointeur de fonction du type `Person` et pointant normalement sur la méthode de `Person`, mais modifié par le constructeur de `Worker` pour pointer sur la méthode correspondante, spécialisé, de `Worker`.

Certaines méthodes sont donc susceptibles d'être *surchargées* ; on ne peut donc les appeler directement que si on est certain du type réel de l'objet que l'on manipule - c'est à dire uniquement si on a créé l'objet soit même. Ainsi, une méthode de `Worker` peut-elle appeler directement une méthode de `Person` sur son objet `person`.

Nous avons néanmoins un problème : que se passe t-il si, un `Worker` se faisant passer pour un simple objet de type `Person`, le programme qui le manipule en vient à vouloir le détruire ? Il appellera alors la méthode `Person_del` en lui donnant l'adresse de la partie `Person` de l'objet de type `Worker`, ce qui ne fonctionnera pas (cela débouchera probablement sur un appel à `free` avec une adresse invalide, induisant un comportement indéfini). Pour faire face à cette éventualité, il faut que le libérateur d'un type polymorphe soit systématiquement surchargeable.

Lorsqu'on veut offrir cette possibilité de surcharge, on a donc au moins deux pointeurs sur fonction.

Souvent, un type surchargeable permet de surcharger la plupart de ces méthodes, ce qui fait beaucoup de pointeurs de fonction. Il serait idiot d'inclure directement tous ces pointeurs de fonctions dans chaque objet de ce type et dans chaque objet de tout type dérivé, pour deux raisons :

- la place mémoire consommé à contenir tous ces pointeurs identiques serait considérable ;
- cela offrirait la possibilité de modifier ces pointeurs au cas par cas d'un objet à l'autre d'un même type, ce qui n'est pas le comportement souhaité ; en effet, si notre catégorisation en types à un sens, alors tous les objets d'un même type doivent absolument se comporter de façon identique ;

C'est pourquoi l'objet ne contient pas directement un pointeur vers chaque fonction surchargée, mais un pointeur vers une structure unique, privée et constante qui contient, elle, les pointeurs vers les méthodes qui implémentent le comportement de ce type. On dénommera cette structure une *interface*. La structure de cette interface devra naturellement toujours être publique (faire des accesseurs pour chacun des membre de cette structure n'apporte rien et obscurcit considérablement le code). Par contre, l'interface elle même pour un type donné n'a pas besoin d'être publique : une simple constante, déclarée localement en static dans le constructeur suffit.

Le constructeur du type parent devra donc initialiser son pointeur vers son interface vers cette interface définie localement, et offrir un accesseur pour que les constructeurs des types dérivés puissent modifier cette valeur. Un second accesseur en lecture devra être disponible (retournant un `struct type_itf *`) publiquement pour l'appel des méthodes surchargées.

Illustrons tout ceci avec les constructeurs de `Person` et `Worker` :

```
--- person.h ---
```

```
/* interface */
struct person_itf {
    void (*print)(Person *this, FILE *stream);
    void (*del)(Person *this);
};

/* accesseurs */
const struct person_itf *Person_itf(Person *this);
void Person_set_itf(Person *this, const struct person_itf *new_itf);

/* protected */
void Person_print(Person *this, FILE *stream);
void Person_del(Person *this);

--- person_i.h ---

struct Person {
    const struct person_itf *itf;
    char *name, *first_name;
    Gender *gender;
    Address address;
};

const struct person_itf *Person_itf(const Person *this) {
    assert(this);
    return &this->itf;
}

void Person_set_itf(Person *this, const struct person_itf *new_itf) {
    assert(this);
    this->itf = new_itf;
}

void Person_print(Person *this, FILE *stream) {
    /* ... */
}

void Person_del(Person *this) {
    /* ... */
}

Person *Person_construct(Person *this, char *name, char *first_name, Gender *gender,
                        unsigned street_nbr, char *street,
                        char *zip_code, City *city, char *place_known_as) {
    static const struct person_itf itf = { Person_print, Person_del };
    assert(this);
    if (!gender) return NULL;
    this->name = name;
    this->first_name = first_name;
    this->gender = gender;
    if (!Address_construct(&this->address,
                          street_nbr, street, zip_code, city, place_known_as)) {
        return NULL;
    }
}
```

```

    }
    this->itf = &itf; /* default interface */
    return this;
}

--- worker.h ---

Person *Worker_parent_Person(const Worker *this);

--- worker.c ---

void Worker_print(Person *person, FILE *stream) {
    assert(person);
    Worker *this = (Worker *) ((char *)person - offsetof(Worker, person));
    Person_print(person, stream); // appel direct !
    fprintf(stream, "Salary = %u\n", this->salary);
}

void Worker_del(Person *person) {
    assert(person);
    Worker *this = (Worker *) ((char *)person - offsetof(Worker, person));
    /* ... */
}

Worker *Worker_construct(Worker *this, unsigned salary,
                        char *name, char *first_name, Gender *gender,
                        unsigned street_nbr, char *street,
                        char *zip_code, City *city, char *place_known_as) {
    const struct person_itf p_itf = { Worker_print, Worker_del };
    assert(this);
    if (! Person_construct(&this->person, name, first_name, gender,
                          street_nbr, street, zip_code, city, place_known_as)) {
        return NULL;
    }
    Person_set_itf(&this->person, &p_itf); // surcharge
    this->salary = salary;
    return this;
}

Person *Worker_parent_Person(const Worker *this) {
    assert(this);
    return &this->person;
}

```

Vous remarquerez que les fonctions surchargées prennent en premier argument un pointeur vers le type surchargé. Les fonctions qui surchargent les fonctions initiales reçoivent donc elles aussi en premier paramètre non pas le pointeur `this` vers l'objet du type dérivé, mais un pointeur vers l'objet du type initial contenu dans l'objet `this`. Il faut donc commencer par calculer `this` grâce à la macro `offsetof`.

On dénommera toujours le type de l'interface : `struct type_itf` et les pointeurs sur fonction qu'il contient du name des méthodes sans préfixe (`del` et `print` plutôt que `Type_del` et `Type_print`). On dénommera :

```
const struct type_itf *Type_itf(const Type *this);
```

l'accesseur à cette donnée. Un type polymorphe oblige également à définir un accesseur au parent d'un objet car les utilisateurs du type hérité souhaitant profiter du polymorphisme devront appeler les méthodes du type parent avec l'adresse de l'objet parent inclus dans l'objet hérité. Notez que cet accesseur n'a jamais besoin d'être surchargeable (les relations d'héritage étant constantes, comme les autres relations d'utilisation et de composition).

```
Ancestor *Type_parent_ancestor(const Type *this);
```

On fera donc, `person` étant un pointeur sur `Person` :

```
Person_itf(person)->print(person);
```

pour appeler la méthode `print` de `Person`, quelque soit la provenance de `person` (qu'il pointe vers un objet créée individuellement, ou bien qu'il fasse partie d'un `Worker`).

Si nous avons eu `worker` de type `Worker` ou d'un type dérivé (notez bien que tout type dérivé d'un type polymorphe est, a priori, polymorphe) :

```
Person *person = Worker_parent_person(worker);  
Person_itf(person)->print(person);
```

Pour finir, si vous utilisez la notation

```
{ .membre = valeur }
```

pour initialiser une interface, ne perdez pas de vue que les fonctions non surchargées seront initialisées à `NULL` et ne garderons donc pas la valeur de l'interface du parent, contrairement à ce qu'une lecture rapide du source laisse supposer. On préférera donc initialiser explicitement tous les membres de ces structures.

8. Optimisations

La méthode de développement exposée induit deux sources de lourdeur :

- la moindre utilisation d'un objet nécessite de nombreux appels de fonctions ;

- l'initialisation des objets nécessite d'appeler des fonctions avec une très longue liste de paramètres ;

Nous allons donc maintenant chercher à améliorer ces deux points.

La solution aux trop nombreux appels de fonctions est évidente : il faut *inliner* les méthodes les plus simples et les plus fréquemment appelées, en implémentant directement ces méthodes dans le fichier de déclarations publiques, précédées des mots-clés `inline static`. Les allocateurs / libérateurs sont de bons candidats à cette pratique, ainsi que les accesseurs.

Malheureusement, il faudra alors que le type abstrait auquel elles se rapportent soit complètement défini. Il faudra donc définir ce type directement dans ce fichier, ou bien inclure un fichier de déclarations privées (ce qui à l'avantage de signifier à l'utilisateur de la bibliothèque qu'il ne doit pas utiliser directement ces définitions). Idéalement, il faudrait qu'il existe en C un mots-clés pour supprimer la définition d'un type (l'équivalent du `undef` du préprocesseur) de telle sorte qu'on puisse *dé-définir* le type en sortant des déclarations publiques.

Autre inconvénient : il restera toujours une dépendance de fichier à fichier entre le source de l'utilisateur et le fichier de déclarations privées, ce qui n'est pas optimal si l'utilisateur génère ces dépendances automatiquement pour une `Makefile`.

L'avantage d'inliner des fonctions possède en revanche une qualité importante : la manœuvre est indolore pour l'utilisateur du type abstrait. On peut donc laisser ce travail pour la fin.

Réduire le nombre de paramètres à passer aux allocateurs / constructeurs est un problème plus complexe. Si l'allocateur est une fonction inline, le problème se pose uniquement pour le constructeur, qui lui exécute souvent beaucoup de code qu'il ne convient pas d'inliner. Le problème est surtout critique pour ces quelques types abstraits qui sont souvent construit et détruit au cours du programme ; ceux qui ne sont créés que rarement ne posant pas de problème de performance particuliers.

L'idée qui vient naturellement, dans ce cas là, est d'utiliser une structure pour passer tous les paramètres avec l'adresse de cette structure. L'appelant devra quand même remplir la structure en question, ce qui n'est pas toujours mieux que de tout recopier sur la pile. Cette solution convient donc surtout au cas où ce sont les mêmes paramètres qui sont employés pour créer de nombreux objets.

Il faut ici remarquer que si la liste de paramètre d'un constructeur est très longue, c'est souvent parce que le type abstrait hérite d'un autre type, qui lui même est dérivé d'un autre, etc... Il faut alors parfois envoyer tous les paramètres pour les différents constructeurs successifs qui se renvoient les paramètres sans y toucher eux même.

C'est dans un tel cas de figure que la *structure initialisante* peut être utile, si elle regroupe les paramètres constructeur par constructeur. Pour reprendre notre exemple (qui ne le requiert pas vraiment), il faudrait :

```
--- address.h ---
```

```
struct address_construct_list {
    unsigned street_nbr;
    char *street, *zip_code;
    City *city;
    char *place_known_as;
};

Address *Address_construct(Address *this, struct address_construct_list *parameters);

--- person.h ---

struct person_construct_list {
    char *name, *first_name;
    Gender *gender;
    struct address_construct_list address_parameters;
};

Person *Person_construct(Person *this, struct person_construct_list *parameters);

--- worker.h ---

struct worker_construct_list {
    unsigned salary;
    struct person_construct_list *person_parameters;
};

Worker *Worker_construct(Worker *this, struct worker_construct_list *parameters);
```

Le constructeur de Worker peut alors facilement appeler le constructeur de Person :

```
Person_construct(&this->person, &parameters->person_parameters);
```

qui lui même pourra facilement appeler Address_construct.

Cette méthode est cependant peut satisfaisante :

- elle oblige l'utilisateur à remplir "à la main" de fastidieuses structures ;
- elle ne fonctionne vraiment que si les constructeurs ne font que propager les paramètres à d'autres constructeurs, ce qui n'est pas toujours le cas (un type dérivé peut ne pas laisser tant de choix à son utilisateur quant à la construction des types dont il dérive) ;

Il n'y a cependant pas d'autre solution en C⁶.

Notes

1. Attention : certaines de ces définitions sont très éloignées des usages C++

2. Pour eux par exemple le type position est le même sous toute ses formes, et les coordonnées cartésiennes ou polaires ne sont que deux représentations possibles de ce type.
3. Remarquez que ce type entre en relation USES_A avec le type City
4. plus exactement, dans la portée
5. on voit se dessiner le type général ensemble, dont hash, liste, etc, sont des spécialisations...
6. En langage machine, on pourrait, sur beaucoup d'architectures, préserver la fenêtre des paramètres d'un appel à l'autre pour appeler les constructeurs des classes parentes sans les réempiler.