“factory” en C

Michel Billaud (michel.billaud@laposte.net)

20 novembre 2021

Table des matières

- Les notes sont publiées dans https://www.mbillaud.fr/notes/
- Sources dans https://github.com/MichelBillaud/notes-diverses

Dernières corrections :

- 21 novembre 2021 : typos, liens vers sources. Ajout variante.

1 À quoi ça sert

On suppose qu'on est dans un contexte où on a une approche "programmation objet en C", avec du polymorphisme

1.1 Polymorphisme

Un exemple : des formes qui peuvent être des rectangles, des cercles, etc. Des fonctions "new_quelquechose" sont chargées de créer, par allocation dynamique, des instances :

```
Rectangle *r = new_rectangle(100, 200, 10, 20);  // position et dimensions
Circle    *c = new_circle(300, 300, 50);         // position et rayon
```

Il existe une fonction `display_shape()` qui leur est applicable à tous :

```
display_shape(r);
display_shape(c);
```

Elle produit des textes qui dépendent du type

```
rectangle 10x20 at (100,200)
circle with radius 5 at (300,300)
```

Comme elle accepte des pointeurs de différents types, et que C ne permet pas de surcharger les fonctions, son argument est de type "pointeur générique" :

```
void display_shape(void * this);
```

Dans le même ordre d'idée, on pourra constituer des tableaux (ou des listes, ou …) de pointeurs génériques pour représenter des collections d'objets

```
void * array[] = {
    new_rectangle(100, 200, 10, 20),
    new_circle(300, 300, 50),
    ....
};

for (size_t i = 0; i < sizeof(array)/sizeof(array[0]); i++) {
    display_shape(array[i]);
}
```

Voir un peu plus loin la manière de faire ceci.

1.2 Ce qu'on veut faire

On voudrait disposer d'une fonction "générique" capable de fabriquer des instances *de divers types*, en fonction d'un paramètre qui lui est donné, et de listes de paramètres variables en nombre et en type.

```
// exemple provisoire
void * array[] = {
    new_shape("rectangle", 100, 200, 10, 20),
    new_shape("circle",    300, 300, 50),
    new_shape("text",      100, 100, "hello, world");
```

```
    ....
};
```

De toute évidence, il va falloir utiliser la bibliothèque STDARG.

On pourrait envisager de définir la fonction `new_shape` par un aiguillage du genre

```
void * new_shape(char *type, ...)
{
    void *ptr;
    va_list ap;
    va_start(ap, type);
    if (strcmp(type, "rectangle") == 0) {
        int x = va_arg(ap, int);
        int y = va_arg(ap, int);
        int r = va_arg(ap, int);
        ptr = new_circle(x, y, r);
    } else if (strcmp(type, "circle") == 0) {
        ...
    }
    va_end(ap);
    return ptr;
}
```

mais ça conduit à une fonction dont le code s'allonge en fonction du nombre de types, ce qui peut poser des problèmes de maintenance.

L'idée, que nous allons développer, c'est d'avoir, au lieu d'une fonction, un objet "factory" (fabrique) dont le rôle est construire des "shapes" de différentes sortes

Il comportera deux "méthodes"

- un qui sert à enregistrer une manière supplémentaire de créer des objets,
- une pour fabriquer effectivement un objet à partir de paramètres.

```
// Exemple
ShapeFactory factory = { .... };  // on verra plus tard

// enregistrement
factory_register(&factory, "circle",    "iii",  &build_circle);
factory_register(&factory, "rectangle", "iiii", &build_rectangle);
factory_register(&factory, "text",      "iis",  &build_text);

// fabrication d'un objet
void * p = factory_build(&factory, "text", 100, 100, "hello, world");
```

Le troisième paramètre donne le type des arguments attendus (pour un `Text` : deux entiers et une chaîne). Le 4ième est une fonction auxiliaire qui crée un objet à partir des paramètres extraits.

1.3 Annexe : Objets et polymorphisme

Dans cette partie, on montre rapidement comment avoir des objets et des fonctions polymorphes en C.

1. On définit un type de structure qui indique les pointeurs vers les fonctions qu'on peut appliquer aux objets de façon générique. Ici il n'y en a qu'une, une méthode d'affichage.

```
typedef struct {
    void (*display)(void *);
    // ...
} ShapeMethodsTable;
```

Ce n'est pas un tableau, mais on appelle ça une "table de méthodes".

2. Pour chacune des "classes" (`Circle`, `Rectangle`, …), il y a une instance qui renvoie vers des fonctions spécifiques au type

```
ShapeMethods circleMethodsTable    = { .display = & display_circle };
ShapeMethods rectangleMethodsTable = { .display = & display_rectangle };
```

3. Chaque objet est représenté par une structure dont le **premier** champ pointe vers la table de sa classe

```
typedef struct {
    ShapeMethods *table;  // ici
    int x, y;
    int radius;
} Circle;
```

ce champ est initialisé lors de la construction

```
Circle * new_circle(int x, int y, int radius)
{
    Circle *c = malloc(sizeof(Circle));
    *c = (Circle) {
        .table = &circleMethodsTable,   // ici
        .x = x,
        .y = y,
        .radius = radius
    };
    return c;
}
```

(en cas de difficulté, se renseigner sur l'utilisation de désignateurs pour initialiser les structures).

4. La méthode "générique" `display_shape` utilise cette table comme relais pour appeler la méthode qui correspond au type de l'objet

```
void display_shape(void * this)
{
    ShapeMethodsTable **table = this;   // transtypage
    (*table)->display(this);
}
```

Ici on met à profit le fait que le standard C garantit que le premier champ d'une structure commence physiquement à la même adresse que la structure, cf 6.7.2.15 dans le draft standard C17

> A pointer to a structure object, suitably converted, points to its initial member

Le paramètre `this` pointe donc sur le pointeur vers la table des méthodes, d'où la double indirection.

4. La fonction `display_circle` a comme premier paramètre un pointeur générique[1], qu'il faut transtyper avant usage

```
void display_circle(void * this) {
    Circle *thisCircle = this;
    printf("circle with radius %d at (%d,%d)\n",
        thisCircle->radius,
        thisCircle->x,
        thisCircle->y);
}
```

2 La fabrique

Un objet `Factory` va mémoriser pour chaque identifiant (une chaîne), le type des paramètres (dans une chaîne) et la fonction qui sert à fabriquer l'objet.

2.1 Fonctions de fabrication

La fabrication des objets (de types divers) se fait à partir de paramètres de types différents.

Pour cela nous définissons un type "union" assez grand pour contenir un paramètre quelconque

```
typedef union {
           int i;
           float f;
           char *s;
} BuilderParameter;
```

les fonctions de fabrication sont des **adaptateurs** pour avoir un prototype commun pour les fonctions qui construisent des objets : elles reçoivent un tableau de paramètres et retournent un pointeur générique.

```
void * build_circle(BuilderParameter params[])
{
    return new_circle(params[0].i, params[1].i, params[2].i);
}
```

[1] Un pointeur de type `Circle*` semblerait de prime abord plus naturel, mais causerait un conflit de type lors de la construction de la table `ShapeMethodsTable`. Le champ `display` est de type "pointeur sur une fonction dont l'argument est un pointeur générique", théoriquement incompatible avec "pointeur sur une fonction dont l'argument est un pointeur de `Circle`".

```
void * build_text(BuilderParameter params[])
{
    return new_text(params[0].i, params[1].i, params[2].s);
}
```

2.2 Le type `Factory`

Une structure `Factory` contient une collection de descriptions (nom, types des arguments, “builders”) :

```
typedef struct {
    char * name;
    char * types;
    void * (*builder)(BuilderParameter params[]);
} BuilderDescription;

#define MAX_NB_DESCRIPTIONS_IN_FACTORY 10

typedef struct {
    int nb_descriptions;
    BuilderDescription descriptions[MAX_NB_DESCRIPTIONS_IN_FACTORY];
} ShapeFactory;
```

Une factory doit être initialisée avant usage :

```
ShapeFactory factory = { .nb_descriptions = 0 };
```

2.3 Enregistrement d’un “builder”

L’enregistrement dans une factory est un simple ajout à la table

```
void factory_register(ShapeFactory *factory, char *name, char *types,
                      void * (*builder)(BuilderParameter params[]))
{
    factory->descriptions[factory->nb_descriptions ++] =
        (BuilderDescription) {
            .name = name,
            .types = types,
            .builder = builder
            };
}
```

2.4 Construction d’un objet

Pour construire un objet, il faut

- en fonction du nom indiqué, retrouver la description
- récupérer les paramètres avec les types voulus et les mettre dans un tableau
- appeler le builder.

```
#define MAX_NB_PARAMETERS_IN_BUILDER 10

void * factory_build(ShapeFactory *factory, char name[], ...)
```

```
{
    int index = 0;
    while ((index < factory->nb_descriptions)
    && (strcmp(name, factory->descriptions[index].name) != 0)) {
        index++;
    }

    if (index == factory->nb_descriptions) {
        return NULL;
    }
    BuilderDescription *d = & factory->descriptions[index];

    BuilderParameter parameters[MAX_NB_PARAMETERS_IN_BUILDER];
    va_list ap;
    va_start(ap, name);
    for(int i = 0; d->types[i] != '\0'; i++) {
        switch (d->types[i]) {
        case 'i' :
            parameters[i].i = va_arg(ap, int);
            break;
        case 'f' :
            parameters[i].f = va_arg(ap, double);
            break;
        case 's' :
            parameters[i].s = va_arg(ap, char *);
            break;
        }
        va_end(ap);
    }
    return d->builder(parameters);
}
```

3 Une variante

On peut imaginer une variante : les différents builders d'une factory sont identifiées par une valeur retournée par la factory lors de leur enregistrement.

3.1 Exemple d'utilisation

```
// Exemple
ShapeFactory factory = { .nb_descriptions = 0 };

// enregistrement
const int CIRCLE    = factory_register(&factory, "iii",  &build_circle);
const int TEXT      = factory_register(&factory, "iis",  &build_text);

// fabrication d'un objet
void * p = factory_build(&factory, TEXT, 100, 100, "hello, world");
```

3.2 Les identifiants

En fait, l'identifiant retourné par l'enregistrement sera l'indice de la description enregistrée.

Ceci permettra un accès direct lors des constructions, plus rapide que la recherche de la chaine.

3.3 Le nouveau type `ShapeFactory`

Il n'y a plus de noms à stocker dans les descriptions

```
typedef struct {
    // char * name;           // SUPPRIMÉ
    char * types;
    void * (*builder)(BuilderParameter params[]);
} BuilderDescription;
```

3.4 L'enregistrement

```
int factory_register(ShapeFactory *factory,
                     // char *name,      // SUPPRIMÉ
                     char *types,
                     void * (*builder)(BuilderParameter params[]))
{
    factory->descriptions[factory->nb_descriptions] =
        (BuilderDescription) {
            // .name = name,             // SUPPRIMÉ
            .types = types,
            .builder = builder
            };
    return factory->nb_descriptions++;    // RETOUR DE L'INDICE
}
```

3.5 La construction

La construction utilise directement l'index reçu, ce qui élimine la recherche par nom :

```
void * factory_build(ShapeFactory *factory, int index, ...)
{
    // recherche par nom supprimée

    BuilderDescription *d = & factory->descriptions[index];
    BuilderParameter parameters[MAX_NB_PARAMETERS_IN_BUILDER];
    va_list ap;
    va_start(ap, index);                          // CHANGEMENT
    for (int i = 0; d->types[i] != '\0'; i++) {
        switch (d->types[i]) {
        case 'i' :
            parameters[i].i = va_arg(ap, int);
            break;
```

```
            case 'f' :
                parameters[i].f = va_arg(ap, double);
                break;
            case 's' :
                parameters[i].s = va_arg(ap, char *);
                break;
            }
            va_end(ap);
        }
        return d->builder(parameters);
```

4 Remarques

- Le langage C n'est absolument pas fait pour supporter la programmation orientée objets telle qu'on la connait.
- On arrive quand même à faire des choses qui y ressemblent, au prix de quelques contorsions.
- Au passage, on perd tout le contrôle de types : le compilateur ne peut pas vérifier que l'on passe à une "fonction générique" des adresses d'un type raisonnable.

5 Annexe : code source

Le code source complet est disponible dans DemoFactory.zip