Polymorphisme en C : une idée

Michel Billaud (michel.billaud@laposte.net)

6 novembre 2021

Table des matières

Dernières corrections : 6 novembre.

1 Objectif : exprimer le polymorphisme

Dans cette note on montre comment réaliser une sorte de polymorphisme en C, à savoir la possibilité d'appliquer la même fonction à des objets de types différents, avec un comportement spécifique pour chaque type d'objet.

On aura une séparation entre **interface** (au sens Java) et les types concrets qui implémenteront cette interface.

2 Idéees pour la réalisation

2.1 Interface / implémentations

L'interface recense les fonctions applicables indifféremment aux types qui l'implémentent ("fonctions virtuelles"). Ici on aura une interface `Animal` avec deux fonctions `Animal_Talk()` et `Animal_Feed()`.

1. Chaque type aura une table des fonctions virtuelles (dite v-table, pour *virtual table*). En fait, la vtable est une structure avec des pointeurs vers les fonctions qui implémentent les fonctions virtuelles.

```
typedef void (*ANIMAL_TALK_METHOD)(void *);
typedef void (*ANIMAL_FEED_METHOD)(void *, char *);

typedef struct {
    ANIMAL_TALK_METHOD talk;
    ANIMAL_FEED_METHOD feed;
} AnimalVtable;
```

2. Tout objet qui implémente un animal aura comme premier champ un pointeur vers la vtable. On introduit un type "Animal abstrait" comme modèle :

```
typedef struct  {
    AnimalVtable *vtable; // table de pointeurs de fonctions
} Animal;
```

3. La vtable est utilisée comme "relais" par des fonctions applicables à tout `Animal` :

```
void Animal_Talk(void *this)
{
    Animal *animal = this;
    animal->vtable->talk(this);
}

void Animal_Feed(Animal *this, char *food)
{
    Animal *animal = this;
    animal->vtable->feed(this, food);
}
```

2.2 Exemple d'implémentation : le type `Dog`

1. Le type `Dog`, possède un attribut spécifique (`name`) :

```
typedef struct {
    AnimalVtable *vtable;
    char * name;                // attribut
} Dog;
```

2. La v-table de Doc contient les adresses des fonctions qui implémentent `talk()` et `feed()`

```c
void Dog_Talk(void *this)                // fonction talk
{
    Dog *dog = this;
    printf("%s> Wah!\n", dog->name);
}

void Dog_Feed(void *this, char * what)   // fonction feed
{
    Dog *dog = this;
    (void) dog;
    printf("thanks for %s !\n", what);
}

AnimalVtable DogVtable = {               // vtable pour Dog
    .talk = & Dog_Talk,
    .feed = & Dog_Feed
};
```

3. Une fonction `new_Dog` est chargée d'allouer une nouvelle instance, et initialise sa vtable et ses champs :

```c
Dog *new_Dog(char *name)
{
    Dog * p = malloc(sizeof(Dog));
    p->vtable = & DogVtable;
    p->name = name;
    return p;
}
```

2.3 La fonction `main()`

- construit un tableau de pointeurs d'animaux de types différents,
- applique à chacun d'eux les fonctions `Animal_Talk` et `Animal_Feed`, pour montrer les comportement spécifiques
- libère les objets alloués.

```c
int main()
{
    Animal * animals[] = {
        (Animal *) new_Dog("Medor"),
        (Animal *) new_Fish("Yellow"),
        (Animal *) new_Dog("Rex")
    };

    for (int i = 0; i < 3; i++) {
        Animal_Talk(animals[i]);
        Animal_Feed(animals[i], "the kibbles"); // croquettes
    }

    for (int i = 0; i < 3; i++) {
        free(animals[i]);
```

```
    }
}
```

3 Annexes

3.1 Code complet

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


// tout objet polymorphe a une table de fonctions
// comme premier membre


typedef void (*ANIMAL_TALK_METHOD)(void *);
typedef void (*ANIMAL_FEED_METHOD)(void *, char *);

typedef struct {
    ANIMAL_TALK_METHOD talk;
    ANIMAL_FEED_METHOD feed;
} AnimalVtable;

typedef struct  {
    AnimalVtable *vtable; // table de pointeurs de fonctions
} Animal;


void Animal_Talk(void *this)
{
    Animal *animal = this;
    animal->vtable->talk(this);
}

void Animal_Feed(Animal *this, char *food)
{
    Animal *animal = this;
    animal->vtable->feed(this, food);
}

// ----------------------------------------
typedef struct {
    AnimalVtable *vtable;
    // les attributs vont ici
    char * name;
} Dog;

void Dog_Talk(void *this)
{
```

```
    Dog *dog = this;
    printf("%s> Wah!\n", dog->name);
}

void Dog_Feed(void *this, char * what)
{
    Dog *dog = this;
    (void) dog;
    printf("thanks for %s !\n", what);
}

AnimalVtable DogVtable = {
    .talk = & Dog_Talk,
    .feed = & Dog_Feed
};

Dog *new_Dog(char *name)
{
    Dog * p = malloc(sizeof(Dog));
    p->vtable = & DogVtable;
    p->name = name;
    return p;
}

// ----------------------------------------
typedef struct {
    AnimalVtable *vtable;
    char *color;
} Fish;


void Fish_Talk(void *this)
{
    Fish *fish = this;
    printf("%s fish -> Bloup!\n", fish->color);
}

void Fish_Feed(void *this, char * what)
{
    Fish *fish = this;
    (void) fish;
    (void) what;
    printf("No thanks\n");
}

AnimalVtable fishVtable = {
    .talk =  & Fish_Talk,
    .feed =  & Fish_Feed
};
```

```c
Fish *new_Fish(char *color)
{
    Fish * p = malloc(sizeof(Fish));
    p->vtable = & fishVtable;
    p->color = color;
    return p;
}

int main()
{
    Animal * animals[] = {
        (Animal *) new_Dog("Medor"),
        (Animal *) new_Fish("Yellow"),
        (Animal *) new_Dog("Rex")
    };

    for (int i = 0; i < 3; i++) {
        Animal_Talk(animals[i]);
        Animal_Feed(animals[i], "the kibbles"); // croquettes
    }

    for (int i = 0; i < 3; i++) {
        free(animals[i]);
    }
}
```

3.2 Compilation et exécution

La compilation ne nécessite pas d'option spécifique. Le code est conforme au standard c17.

```
gcc -Wall -Wextra -pedantic -std=c17 demo.c -o demo
```

Exécution

```
./demo
Medor> Wah!
thanks for the kibbles !
Yellow fish -> Bloup!
No thanks
Rex> Wah!
thanks for the kibbles !
```