Méthodologie de la programmation

Chapitre 7

Aperçu du langage C : mémoire, modules, compilation

Pablo Rauzy <pr@up8.edu>
pablo.rauzy.name/teaching/mp

Aperçu du langage C : mémoire, modules, compilation

Allocation dynamique de mémoire

- On a vu comment allouer de la mémoire en C en déclarant des variables.
- La question qui se pose maintenant est comment gérer dynamiquement la mémoire ?
- Par exemple, si on ne sait pas avant l'exécution du programme de combien d'éléments on aura besoin dans un tableau.

- La fonction **malloc** est déclarée dans l'*entête* **stdlib.h**.
- Son *prototype* est :
  - **void** *malloc (**size_t** size);
- Elle tente d'allouer **size** octets de mémoire et retourne un pointeur vers la zone mémoire allouée.
- En cas d'erreur, elle retourne **NULL**.
- Exemple :

```
1  grades = malloc(count * sizeof(*grades));
2  if (grades == NULL) {
3    printf("Erreur allocation mémoire\n");
4    exit(1);
5  }
```

- Attention, **malloc** n'initialise pas la mémoire !

- Quand on alloue de la mémoire soi-même, il faut penser à la *libérer* pour éviter les *fuites* de mémoire.
- On libère la mémoire avec la fonction **`free`**, aussi déclarée dans l'entête **`stdlib.h`**.
- Son prototype est :
  - `void free (void *ptr);`
- Le pointeur que l'on passe en argument doit avoir été retourné par **`malloc`** précédemment.
- Exemple :

```
1  free(grades);
```

- Attention à ne plus utiliser un pointeur passé à **`free`** !

- Il existe aussi les fonctions `calloc` et `realloc` :
  - `void *calloc (size_t nmemb, size_t size);`
  - `void *realloc (void *ptr, size_t size);`
- La première alloue `nmemb * size` octets et initialise la mémoire à 0.
- La seconde sert à changer la taille allouée à un pointeur (possiblement en déplaçant la mémoire).
- Dans les deux cas, il faut aussi ne pas oublier d'appeler `free` quand on a plus besoin de la mémoire.

Programmation modulaire

- Un *module* en C est un couple de fichier d'implémentation (`.c`) et d'entête (`.h`).
- Dans l'entête, on déclare les *prototypes* des fonctions implémentées, et souvent les structures définies par le module.

Programmation modulaire

- Un *module* en C est un couple de fichier d'implémentation (`.c`) et d'entête (`.h`).
- Dans l'entête, on déclare les *prototypes* des fonctions implémentées, et souvent les structures définies par le module.
- Remarque : on peut voir l'entête comme l'*interface* du module, et dans ce cas on peut pratiquer l'*encapsulation*, comme on l'a vu en programmation orientée objet avec Python.
  - Ne déclarer que les prototypes des fonctions *publiques*.
  - Ne faire que le `typedef` et garder la structure *opaque*.

Programmation modulaire

- Un *module* en C est un couple de fichier d'implémentation (`.c`) et d'entête (`.h`).
- Dans l'entête, on déclare les *prototypes* des fonctions implémentées, et souvent les structures définies par le module.
- Remarque : on peut voir l'entête comme l'*interface* du module, et dans ce cas on peut pratiquer l'*encapsulation*, comme on l'a vu en programmation orientée objet avec Python.
  - Ne déclarer que les prototypes des fonctions *publiques*.
  - Ne faire que le `typedef` et garder la structure *opaque*.

→ Voyons l'exemple du module `point`.

Compilation

- La compilation d'un programme C passe par quatre grandes étapes :
  - le "prétraitement",
  - la compilation,
  - l'assemblage,
  - l'édition de liens.
- Par défaut, GCC fait toutes les étapes.

- C'est l'étape qui traite les `#define`, `#include`, etc.
- On peut demander à GCC de s'arrêter après cette étape avec l'argument `-E` :
  - `gcc -E point.c -o point.i`

- Cette étape transforme le code C en code assembleur.
- On peut demander à GCC de s'arrêter après cette étape avec l'argument `-S` :
  - `gcc -S point.c -o point.s`
  - `gcc -S point.i -o point.s`
- À partir de cette étape le code n'est plus portable (l'assembleur dépend de l'architecture cible).

- Cette étape transforme le code assembleur en fichier *objet*.
- On peut demander à GCC de s'arrêter après cette étape avec l'argument `-c` :
  - `gcc -c point.c -o point.o`
  - `gcc -c point.i -o point.o`
  - `gcc -c point.s -o point.o`

- Cette étape fait le lien entre les noms définies dans différents fichiers objets (les différents modules du projet, mais aussi les bibliothèques, etc.).
- C'est la dernière étape pour produire du code binaire :
  - `gcc point.c -o point`
  - `gcc point.i -o point`
  - `gcc point.s -o point`
  - `gcc point.o -o point`

Compilation de projet

- Quand on a plusieurs modules dans un projet on peut les compiler tous en même temps :
  - `gcc point.c square.c main.c -o geometry`
- Mais quand le projet devient un peu conséquent, cela peut poser problème de devoir tout recompiler à chaque fois, même quand on a modifié seulement un fichier.

Compilation séparée

- En demandant à GCC de générer des fichiers objets pour chaque module, on peut ne recompiler que ceux qu'on a modifiés, puis se contenter de faire l'édition des liens sur les fichiers objets :
  - `gcc -c point.c -o point.o`
  - `gcc -c square.c -o square.o`
  - `gcc -c main.c -o main.o`
  - `gcc point.o square.o main.o -o geometry`

Les moteurs de production

- On peut automatiser ce processus grâce à Make, qui est un *moteur de production*.
- Dans un `Makefile`, on écrit des *règles* qui décrivent quelles commandes permettent de produire une *cible*.
- On exprime aussi les dépendances entres les cibles.
- Ensuite la commande `make` sera capable à partir du `Makefile` et de l'état des fichiers de reproduire la cible finale en ne lançant que les commandes nécessaires.
- Dans notre exemple :
  - La cible `geometry` dépend de `point.o`, `square.o`, et `main.o`, et s'obtient avec `gcc point.o square.o main.o -o geometry`.
  - La cible `point.o` dépend de `point.c` et `point.h`, et s'obtient avec `gcc -c point.c -o point.o`.
  - La cible `square.o` dépend de `square.c`, `square.h`, et `point.h`, et s'obtient avec `gcc -c square.c -o square.o`.
  - La cible `main.o` dépend de `main.c`, `point.h`, et `square.h`, et s'obtient avec `gcc -c main.c -o main.o`.
  - Les cibles qui sont des fichiers existants n'ont pas besoin de règle pour être produites.

- En langage Make, ce que l'on vient de voir s'écrit :

```
1  geometry: main.o point.o square.o
2          gcc main.o point.o square.o -o geometry
3
4  main.o: main.c point.h square.h
5          gcc -std=c99 -c main.c -o main.o
6
7  point.o: point.c point.h
8          gcc -std=c99 -c point.c -o point.o
9
10 square.o: square.c point.h square.h
11         gcc -std=c99 -c square.c -o square.o
```

- Quand on appelle la commande `make` on peut lui préciser une cible à construire, sinon il prend la première.
- Make permet d'utiliser des variables et des raccourcis pour rendre plus génériques et donc plus évolutifs les `Makefile`s.

- Voici à quoi ressemblerait un vrai `Makefile` pour ce petit projet :

```
1  CC = gcc
2  CFLAGS = -std=c99 -c
3  LDFLAGS =
4
5  OBJ = main.o point.o square.o
6  BIN = geometry
7
8  all:: $(BIN) # "fausse" cible
9
10 $(BIN): $(OBJ)
11         $(CC) $^ -o $@
12
13 $(OBJ):
14         $(CC) $(CFLAGS) $< -o $@
15
16 clean::
17         rm -f $(BIN) $(OBJ) $(DEP) *~
18
19 main.o: main.c point.h square.h
20 point.o: point.c point.h
21 square.o: square.c point.h square.h
```

- Les dernières lignes peuvent être générées automatiquement par GCC avec l'argument `-MM`.
- Remarquez la règle "clean".

- Voici à quoi ressemblerait un `Makefile` un peu plus avancé :

```
1  CC = gcc
2  CFLAGS = -std=c99 -g -c
3  LDFLAGS = -g
4
5  SRC = main.c point.c square.c
6  OBJ = $(subst .c,.o,$(SRC))
7  BIN = geometry
8  DEP = _dependencies.mk
9
10 all:: $(DEP) $(BIN)
11
12 $(BIN): $(OBJ)
13         $(CC) $^ -o $@
14
15 $(OBJ):
16         $(CC) $(CFLAGS) $< -o $@
17
18 $(DEP): $(SRC)
19         $(CC) -MM $^ > $@
20
21 clean::
22         rm -f $(BIN) $(OBJ) $(DEP) *~
23
24 -include $(DEP)
```