

Compilation multi-fichiers

Aide-mémoire

MP2I Lycée Pierre de Fermat

En C, les gros projets peuvent très facilement atteindre plusieurs centaines, plusieurs milliers de lignes de code. Afin de segmenter le code et de faciliter le développement, on sépare le code en *plusieurs fichiers*. Dans un programme multi-fichiers, seul un fichier .c contient une fonction *main*. L'étape de compilation sert à lier tous les fichiers entre eux pour produire un unique exécutable.

On considère le code suivant comme exemple récurrent de cette fiche :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct animal {
6      char* espece;
7      char* nom;
8  } animal_t;
9
10
11 typedef struct zoo{
12     int n_animaux;
13     animal_t** animaux; // tableaux de pointeurs d'animaux
14 } zoo_t;
15
16 /*Affiche les informations de l'animal a */
17 void infos_animal(animal_t* a){
18     printf("%s (%s)\n",
19           a->nom, a->espece);
20 }
21
22 /* Affiche le nombre d'animaux du zoo z, puis liste chaque animal */
23 void infos_zoo(zoo_t* z){
24     printf("%d animaux:\n", z->n_animaux);
25     for (int i = 0; i < z->n_animaux; ++i){
26         infos_animal(z->animaux[i]);
27     }
28 }
29
30 /* Libère la mémoire réservée pour un zoo */
31 void free_zoo(zoo_t* z){
32     free(z->animaux);
33     free(z);
34 }
35
36 /* Renvoie le nombre d'animaux de z correspondant à l'espèce e */
37 int compter_espece(zoo_t* z, char* e){
38     int compteur = 0;
39     for (int i = 0; i < z->n_animaux; ++i){
40         if (strcmp(z->animaux[i]->espece, e) == 0){
41             compteur++;
42         }
43     }
44     return compteur;
45 }
```

```

46
47 int main(){
48     animal_t* lion = malloc(sizeof(animal_t));
49     animal_t* girafe = malloc(sizeof(animal_t));
50
51     lion->nom = "Grorf";
52     lion->espece = "Lion";
53
54     girafe->nom = "Fleurine";
55     girafe->espece = "Giraffe";
56
57     zoo_t* z = malloc(sizeof(zoo_t));
58     z->n_animaux = 2;
59     z->animaux = malloc(2*sizeof(animal_t*));
60     z->animaux[0] = a1;
61     z->animaux[1] = a2;
62
63     infos_zoo(z);
64
65     free(a1);
66     free(a2);
67     free_zoo(z);
68
69     return 0;
70 }

```

Ce programme modélise un zoo ayant plusieurs animaux. Pour l’instant, tout le code est réuni dans un seul grand fichier C. Cependant, en C, et dans tous les langages de programmation, on **divise le code en plusieurs fichiers**, selon une organisation logique. Une pratique assez courante est d’avoir un fichier C par structure définie.

Par exemple, ici, on peut mettre tout ce qui touche aux animaux ensemble, et tout ce qui touche aux zoos ensemble. Nous allons donc créer trois fichiers :

```

├─ animal.c
├─ zoo.c
└─ main.c

```

Pour compiler le programme, on précise alors tous les fichiers :

```
gcc main.c animal.c zoo.c -o prog
```

L’ordre des fichiers .c n’importe pas. On peut toujours utiliser les différentes options de compilation :

```
gcc main.c animal.c zoo.c -o prog -Wall -Wextra -g -lm
```

En réalité, lorsque l’on divise ainsi le code, on doit créer non seulement des fichiers C contenant les définitions des différentes structures et fonctions, mais également des fichiers C contenant uniquement les **déclarations**. On appelle ces fichiers des **en-têtes**, ou **headers**, et leur extension est ".h".

Le rôle d’un header est de documenter les différentes fonctions et types qui sont défini dans le fichier C correspondant, et permet de les utiliser dans d’autres fichiers.

C’est donc dans le header que l’on écrit le commentaire de documentation d’une fonction. Dans notre exemple, on aura les fichiers sources suivants :

```

├─ animal.h
├─ animal.c
├─ zoo.h
├─ zoo.c
└─ main.c

```

Les fichiers .h servent à annoncer les différentes structures et fonctions qui existent, et s’utilisent avec `#include`, comme les bibliothèques. Les fichiers .c servent à implémenter ces fonctions, et ont pour vocation d’être compilés.

Regardons plus en détail le contenu des fichiers .h et .c pour comprendre comment ils s'agencent, et comment le compilateur les gère.

Le fichier animal.h contiendra :

```
1 /* Données des animaux */
2 typedef struct animal{
3     char* espece;
4     char* nom;
5 } animal_t;
6
7 /*Affiche les informations de l'animal pointé par a*/
8 void infos_animal(animal_t* a);
```

et le fichier animal.c contiendra :

```
1 #include "animal.h"
2 #include <stdio.h>
3
4 void infos_animal(animal_t* a){
5     printf("%s (%s), %d ans\n", a->nom, a->espece);
6 }
```

Aucun de ces deux fichiers ne contient de main. On peut imaginer que l'on a créé notre propre (petite) librairie, comme `<string.h>` ou `<stdio.h>`.

Ensuite, si l'on utilise le type `animal_t` et la fonction `infos_animal` dans un autre fichier C du programme, on devra écrire au début du fichier en question : `#include "animal.h"` pour que le compilateur sache que `animal_t` et `infos_animal` existent dans ce fichier C. Par exemple, dans le fichier zoo.h, on aura :

```
1 /* Données des zoos */
2 #include "animal.h"
3
4 typedef struct zoo{
5     int n_animaux;
6     animal_t** animaux;
7 } zoo_t;
8
9 /* Affiche le nombre d'animaux du zoo z puis liste chaque animal */
10 void infos_zoo(zoo_t* z);
11
12 /* Libère la mémoire réservée pour un zoo */
13 void free_zoo(zoo_t* z);
14
15 /* Renvoie le nombre d'animaux de z correspondant à l'espece e */
16 int compter_espece(zoo_t* z, char* e);
```

et dans zoo.c :

```
1 #include "zoo.h"
2 #include "animal.h"
3
4 #include <stdio.h>
5 #include <string.h>
6
7 void infos_zoo(zoo_t* z){
8     printf("%d animaux:\n", z->n_animaux);
9     for (int i = 0; i < z->n_animaux; ++i){
10         infos_animal(z->animaux[i]);
11     }
12 }
13
14 void free_zoo(zoo_t* z){
15     free(z->animaux);
16     free(z);
17 }
```

```

18
19 /* Renvoie le nombre d'animaux de z correspondant à l'espece e */
20 int compter_espece(zoo_t* z, char* e){
21     int compteur = 0;
22     for (int i = 0; i < z->n_animaux; ++i){
23         if (strcmp(z->animaux[i]->espece, e) == 0){
24             compteur++;
25         }
26     }
27     return compteur;
28 }

```

Enfin, dans main.c :

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #include "animal.h"
5 #include "zoo.h"
6
7
8 int main(){
9     animal_t* a1 = malloc(sizeof(animal_t));
10    animal_t* a2 = malloc(sizeof(animal_t));
11
12    a1->nom = "Grorf";
13    a1->espece = "Lion";
14    a1->age = 17;
15    a1->poids = 180;
16
17    a2->nom = "Fleurine";
18    a2->espece = "Giraffe";
19    a2->age = 21;
20    a2->poids = 963;
21
22    zoo_t* z = malloc(sizeof(zoo_t));
23    z->n_animaux = 2;
24    z->animaux = malloc(2*sizeof(animal_t*));
25    z->animaux[0] = a1;
26    z->animaux[1] = a2;
27
28    infos_zoo(z);
29
30    free(a1);
31    free(a2);
32    free(z->animaux);
33    free(z);
34
35    return 0;
36 }

```

Concrètement, la commande `#include` a pour effet de recopier intégralement le contenu du fichier spécifié.

Si l'on essaie de compiler ces fichiers avec la commande plus haut, on obtient plusieurs erreurs disant que l'on a défini plusieurs fois la structure `struct animal`. La raison est simple : `main.c` fait référence à `animal.h`, et à `zoo.h` qui fait lui même référence à `animal.h`, donc le code présent dans `animal.h` est recopié deux fois dans le `main`.

Pour éviter cela, il faut préciser au compilateur que le contenu de nos fichiers `.h` ne doit être inclus qu'une seule fois, et que si l'on essaie d'inclure un fichier une deuxième fois, il faut l'ignorer. On appelle ce procédé un **include guard**, et pour l'implémenter on utilise deux commandes préprocesseur :

- `#define`, que nous avons déjà vu, permet de définir une macro. Par exemple, `#define BLA 64` fait que chaque instance de `BLA` dans le code est remplacée par `64` : on définit la macro `BLA`
- `#ifndef` (comme **if not defined**) permet de prendre en compte le code qui suit uniquement si la macro spécifiée n'est pas définie. La fin du bloc de code à recopier ou non est marquée par `#endif`. Par exemple :

```
1 #define BLA // on définit BLA, sans lui donner de valeur: c'est une macro vide
2
3 #ifndef BLA
4 /* Ici, du code qui ne sera pas pris en compte par le compilateur
5    car BLA est une macro définie */
6 #endif
7
8 #ifndef BLI
9 /* Ici, du code qui sera pris en compte car BLI n'est pas une macro définie */
10 #endif
```

Le principe est donc de définir une macro pour chaque header, et de mettre tout le header dans un grand `#ifndef`. Par exemple :

```
1 /* Données des animaux */
2 #ifndef ANIMAUX_H
3 #define ANIMAUX_H
4
5 typedef struct animal{
6     char* espece;
7     char* nom;
8 } animal_t;
9
10 /*Affiche les informations de l'animal pointé par a*/
11 void infos_animal(animal_t* a);
12
13 #endif
```

Dans notre code, dans le fichier `main.c`, on a inclus deux fois ce header : une fois directement et une fois via `zoo.h`. La première fois, `ANIMAUX_H` n'est pas définie, le code est donc recopié, et `ANIMAUX_H` est défini (vide, mais défini). La deuxième fois, la macro est définie, on ne recopie donc rien.

Nous allons également rajouter un include guard dans `zoo.h`, même si ce n'est pas strictement nécessaire car il n'est inclus qu'une seule fois, au cas où l'on vienne rajouter des fichiers `.c` / `.h` au projet plus tard.

```
1 #ifndef ZOO_H
2 #define ZOO_H
3 /** Données des zoos: un zoo est un tableau d'animaux. **/
4
5 #include "animal.h"
6
7 typedef struct zoo{
8     int n_animaux;
9     animal_t** animaux;
10 } zoo_t;
11
12
```

```

13 /* Affiche le nombre d'animaux du zoo z puis liste chaque animal */
14 void infos_zoo(zoo_t* z);
15
16 /* Libère la mémoire réservée pour un zoo */
17 void free_zoo(zoo_t* z);
18
19 /* Renvoie le nombre d'animaux de z correspondant à l'espece e */
20 int compter_espece(zoo_t* z, char* e);
21 #endif

```

On peut alors compiler le programme sans erreur :

```
gcc main.c animal.c zoo.c -o petit_zoo -Wall -Wextra -g
```

L'archive compagnon de cette fiche contient les 5 fichiers décrits ci-dessus : `main.c`, `animal.h`, `animal.c`, `zoo.h`, `zoo.c`.

Pragma once On voit qu'il faut donner à chaque include guard un nom différent, mais que l'on rajoute essentiellement toujours les mêmes lignes de code. En pratique, la plupart des compilateurs acceptent l'instruction préprocesseur `#pragma once` qui simule un include guard lorsqu'elle est mise au début d'un fichier. Par exemple, au lieu de l'include guard du fichier `animal.h`, on aurait pu écrire :

```

1 #pragma once
2 /* Données des animaux */
3
4 typedef struct animal{
5     char* espece;
6     char* nom;
7 } animal_t;
8
9 /*Affiche les informations de l'animal pointé par a*/
10 void infos_animal(animal_t* a);

```

A savoir

Pour résumer :

- Le code est divisé en fichiers selon une organisation logique et structurée.
- Chaque fichier `.c` (à part le `main`) doit avoir un fichier `.h` correspondant, où sont déclarées les structures et les fonctions utilisées. Le fichier contenant la fonction `main` n'a généralement pas de header.
- Un fichier `A.c` doit contenir la ligne `#include "A.h"` et, s'il utilise des fonctionnalités d'un fichier `B.c`, il doit également contenir la ligne `#include "B.h"`. **Les fichiers `.h` servent à être inclus.**
- La documentation va dans les fichiers d'en-tête. Une personne qui lit le fichier `A.h` doit être capable d'utiliser le code du fichier `A.c` sans jamais l'ouvrir, juste en lisant les commentaires de documentation.
- La commande de compilation doit faire intervenir tous les fichiers `.c` nécessaires. **Les fichiers `C` servent à être compilés.**

Vous devez savoir **créer des programmes séparés en plusieurs fichiers**. Vous devez donc également connaître l'existence du mécanisme d'**include guard**, et vous devez savoir l'implémenter en utilisant `#ifndef` et `#define`. Vous pouvez utiliser `#pragma once` dans votre code.

Pour aller plus loin : Makefile

En pratique, lorsqu'un projet C a des centaines de fichiers sources, on ne veut pas taper à la main la commande de compilation. On peut utiliser un **makefile** pour automatiser la compilation. Les makefiles peuvent être extrêmement complexe, et utilisent leur propre langage, mais sont assez pratiques lorsque l'on s'habitue à leur syntaxe. Vous pouvez vous renseigner sur internet sur l'utilisation basique des makefiles si ça vous amuse !