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 :

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


typedef struct livre{
  char* espece;
  char* nom;
} animal_t;


typedef struct zoo{
  int n_animaux;
  animal_t** animaux;
} zoo_t;


/*Affiche les informations de l'animal pointé par a*/
void infos_animal(animal_t* a){
  printf("%s (%s)\n",
         a->nom, a->espece);
}

/* Affiche le nombre d'animaux du zoo z puis liste chaque animal  */
void infos_zoo(zoo_t* z){
  printf("%d animaux:\n", z->n_animaux);
  for (int i = 0; i < z->n_animaux; ++i){
    infos_animal(z->animaux[i]);
  }
}

/* Libère la mémoire réservée pour un zoo */
void free_zoo(zoo_t* z){
  free(z->animaux);
  free(z);
}

/* Renvoie le nombre d'animaux de z correspondant à l'espece e */
int compter_espece(zoo_t* z, char* e){
  int compteur = 0;
  for (int i = 0; i < z->n_animaux; ++i){
    if (strcmp(z->animaux[i]->espece, e) == 0){
      compteur++;
    }
  }
```

```c
  return compteur;
}


int main(){
  animal_t* lion = malloc(sizeof(animal_t));
  animal_t* girafe = malloc(sizeof(animal_t));

  lion->nom = "Grorf";
  lion->espece = "Lion";

  girafe->nom = "Fleurine";
  girafe->espece = "Giraffe";

  zoo_t* z = malloc(sizeof(zoo_t));
  z->n_animaux = 2;
  z->animaux = malloc(2*sizeof(animal_t*));
  z->animaux[0] = a1;
  z->animaux[1] = a2;

  infos_zoo(z);

  free(a1);
  free(a2);
  free_zoo(z);

  return 0;
}
```

Ce programme modélise un zoo ayant plusieurs animaux. Pour l'instant, tout le code est réuni dans un seul grand fichier C.

Lorsque l'on fait un programme multi-fichiers, on rassemble le code par thématique. Par exemple, ici, on peut mettre tout ce qui touche aux animaux ensemble, tout ce qui touche aux zoos ensemble, et le main seul. Nous allons donc créer trois fichiers :

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

En réalité, lorsque l'on divise ainsi le code, on crée 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 fonctions qui existent, et s'utilisent avec #include, comme les librairies. Les fichiers .c servent à implémenter ces fonctions, et ont pour vocation d'être compilés. La commande de compilation à taper sera :

```
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
```

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 :

```c
/* Données des animaux  */
typedef struct animal{
  char* espece;
  char* nom;
} animal_t;

/*Affiche les informations de l'animal pointé par a*/
void infos_animal(animal_t* a);
```

et le fichier animal.c contiendra :

```c
#include "animal.h"
#include <stdio.h>

void infos_animal(animal_t* a){
  printf("%s (%s), %d ans\n", a->nom, a->espece);
}
```

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 code sache que animal_t et infos_animal existent. Par exemple, dans le fichier zoo.h, on aura :

```c
/* Données des zoos  */
#include "animal.h"

typedef struct zoo{
  int n_animaux;
  animal_t** animaux;
} zoo_t;

/* Affiche le nombre d'animaux du zoo z puis liste chaque animal  */
void infos_zoo(zoo_t* z);

/* Libère la mémoire réservée pour un zoo */
void free_zoo(zoo_t* z);

/* Renvoie le nombre d'animaux de z correspondant à l'espece e */
int compter_espece(zoo_t* z, char* e);
```

et dans zoo.c :

```c
#include "zoo.h"
#include "animal.h"

#include <stdio.h>
#include <string.h>

void infos_zoo(zoo_t* z){
  printf("%d animaux:\n", z->n_animaux);
  for (int i = 0; i < z->n_animaux; ++i){
    infos_animal(z->animaux[i]);
  }
}

void free_zoo(zoo_t* z){
  free(z->animaux);
  free(z);
```

```c
}

/* Renvoie le nombre d'animaux de z correspondant à l'espece e */
int compter_espece(zoo_t* z, char* e){
  int compteur = 0;
  for (int i = 0; i < z->n_animaux; ++i){
    if (strcmp(z->animaux[i]->espece, e) == 0){
      compteur++;
    }
  }
  return compteur;
}
```

Enfin, dans main.c :

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

#include "animal.h"
#include "zoo.h"


int main(){
  animal_t* a1 = malloc(sizeof(animal_t));
  animal_t* a2 = malloc(sizeof(animal_t));

  a1->nom = "Grorf";
  a1->espece = "Lion";
  a1->age = 17;
  a1->poids = 180;

  a2->nom = "Fleurine";
  a2->espece = "Giraffe";
  a2->age = 21;
  a2->poids = 963;

  zoo_t* z = malloc(sizeof(zoo_t));
  z->n_animaux = 2;
  z->animaux = malloc(2*sizeof(animal_t*));
  z->animaux[0] = a1;
  z->animaux[1] = a2;

  infos_zoo(z);

  free(a1);
  free(a2);
  free(z->animaux);
  free(z);

  return 0;
}
```

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 certaines fonctions. 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 utilise pour cela 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.

— #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 :

```c
#define BLA 64

#ifndef BLA
/*
  du code qui ne sera pas pris en compte par le compilateur
*/
#endif

#ifndef BLI
/*
  de code qui sera pris en compte
*/
#endif
```

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

```c
/* Données des animaux  */
#ifndef ANIMAUX_H
#define ANIMAUX_H

typedef struct animal{
  char* espece;
  char* nom;
} animal_t;

/*Affiche les informations de l'animal pointé par a*/
void infos_animal(animal_t* a);

#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.

Ce mécanisme s'appelle un **#include guard**.

Même si `zoo.h` n'est inclus qu'une seule fois, nous allons également rajouter un include guard, en prévision d'éventuels ajouts de fichiers / structures au code :

```c
#ifndef ZOO_H
#define ZOO_H
/*
  Données des zoos
*/

#include "animal.h"

typedef struct zoo{
```

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

On peut alors compiler le programme sans erreur :

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

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 la commande

```
1  #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  /* Données des animaux  */
2  #pragma once
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

Vous devez savoir créer des programmes séparés en plusieurs fichiers. Vous devez également connaître l'existence du mécanisme d'include guard, mais vous pouvez utiliser #pragma once dans votre code.

Pour résumer :

- Le code est divisé en fichiers selon une organisation logique et structurelle
- Chaque fichier .c 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 utilisant des fonctionnalités d'un fichier B.c doit contenir la ligne #include "A.h". Les **headers** servent à être **inclus**
- La commande de compilation doit faire intervenir tous les fichiers .c nécessaires. Les **fichiers C** servent à être **compilés**.

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 !