Plusieurs fichiers en C

Le but de ce document est d'expliquer *comment* fonctionne C avec plusieurs fichiers. Ce document contient des figures et des exemples afin d'illustrer ce qui est écrit. Si quelque chose semble peu clair, prévenez-moi et j'ajusterai.

I. Mise en contexte, Motivation.

Pour des projets simples, écrire l'entièreté d'un code dans un seul fichier est possible. On distingue généralement cinq parties à un code C en un seul fichier :

(1) libraires

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

(2) structures

```c
typedef struct {
  float x;
  float y;
} vec2;
```

(3) signatures

```c
vec2 add(vec2, vec2);
vec2 scale(float, vec2);
```

(4) implémentations

```c
vec2 add(vec2 u, vec2 v) {
  return { u.x + v.x, u.y + v.y };
}

vec2 scale(float a, vec2 v) {
  return { a * v.x, a * v.y };
}
```

(5) fonction *main*

```c
int main() {
  vec2 u = { 0, 2 };
  vec2 v = { 1, 3 };
  vec2 w = add(u, scale(2, v));
  printf("x: %d, y: %d\n", w.x, w.y);
  return 0;
}
```

Un avantage majeur, c'est que ça marche sans aucun problèmes. Mais, il existe un inconvénient important :

> *L'informaticien est fondamentalement fainéant.*
>
> — Tout le monde

On aime transformer notre code en blocs que l'on peut assembler et désassembler comme on le souhaite. Par exemple, au bout de trois implémentation d'une file, on peut en avoir un peu ras le bol (et ça se comprend). On aimerai réutiliser le code, pour se concentrer sur la partie *réellement* intéressante : le programme principal.

II. Quels fichiers ?

De la décomposition du code en « blocs », on peut se dire qu'un bloc doit correspondre à un fichier. Presque . . . En réalité, on utilise *deux* fichiers.

En C, on utilise *deux* types de fichiers, que l'on différentie par leur extension : les fichiers en `.c` et les fichiers en `.h`. Que contiennent ces deux fichiers ?

- Les fichiers `.h` contiennent les `#include<...>` pour les librairies, les structures, et les signatures de fonctions.
- Les fichiers `.c` contiennent les implémentations des fonctions, et notamment de la fonction *main*.

C'est pour ça que, en général, un bloc se décompose en deux fichiers que l'on nomme `[bloc].h` et `[bloc].c`.

module « bloc » ***en-tête header*** ***implémentation***

Bon, jusque là, on n'a définit que les blocs, pas comment relier les blocs entre-eux.

III. Relier les modules.

Imaginons que l'on doit implémenter une file. Pour cela, on veut utiliser une implémentation déjà réalisée d'une liste chaînée. On peut analyser ce que contient `linked_list.h` (le fichier *header* du module de liste chaînée).

`linked_list.h`

```
typedef struct node_t { ... } node;
typedef struct { ... } linked_list;

node node_create(int);
linked_list ll_create();
bool ll_is_empty(linked_list);
void ll_add_to_start(node, linked_list);
void ll_add_to_end(node, linked_list);
node ll_remove_first(linked_list);
node ll_remove_last(linked_list);
void node_free(node);
void ll_free(linked_list);
```

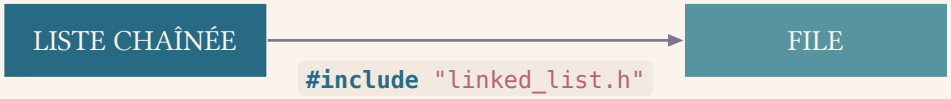
On aura plus ou moins de signatures dans le fichier en-tête. Dans ce cas-ci, on aura toutes les fonctions nécessaires pour implémenter une file. Il ne reste plus qu'à le faire . . .

Commençons par le fichier en-tête pour la file. Il contiendra un type `queue` correspondant à une file. On ajoutera également les signatures des fonctions que l'on définira après.

`queue.h`

```
#include "linked_list.h"

typedef linked_list queue;

queue queue_create();
void queue_enqueue(node, queue); // enfile
node queue_dequeue(queue);       // défile
void queue_free(queue);
```

Là est un moment très important. Par la puissance du `#include "..."`, les modules FILE et LISTE CHAÎNÉE sont maintenant reliés. ***Attention !*** La syntaxe est *légèrement* différente à celle pour les librairies : pour les librairies, on utilise `#include <...>` et pour les modules définis soi-même, on utilise `#include "..."`.



Ce lien entre deux modules est *orienté* : le module FILE utilise le module LISTE CHAÎNÉE mais pas inversement.

À quoi sert ce lien ? On peut, dans le fichier en-tête, utiliser les types définis dans le module importé. Là, on utilise les types `linked_list` et `node` dans le fichier `queue.h` alors qu'il est défini dans `linked_list.h`.

Qu'en est-il de l'implémentation du module FILE ? On aimerai pouvoir utiliser *plus* que les types : on aimerai utiliser les fonctions que l'on définit dans LISTE CHAÎNÉE.

queue.c

```c
#include "queue.h"
#include <assert.h>

queue queue_create() {
  return ll_create();
}

void queue_enqueue(node n, queue q) {
  ll_add_to_start(n, q);
}

node queue_dequeue(queue q) {
  assert(!queue_is_empty(q));
  return ll_remove_last(q);
}

[...]
```

Dans ce fichier, on commence généralement par inclure le fichier en-tête associé (là, `queue.h`). On peut aussi utiliser des librairies (`assert.h` par exemple), ou d'autres fichiers en-tête (qui ne sont pas déjà inclus dans `queue.h`).

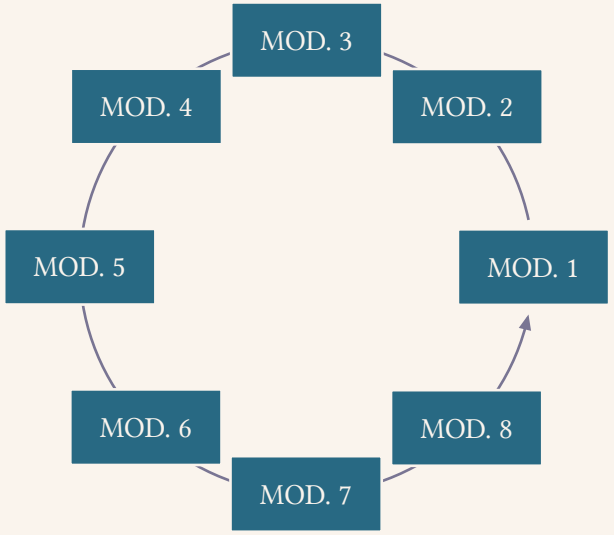
Ensuite, on définit les implémentations des fonctions, en ayant accès aux modules inclus (là, on utilise les fonction du module LISTE CHAÎNÉE).

C'est fait ! Vous savez comment définir plusieurs modules, et les agencer pour former un programme. Il reste cependant quelques pièges à éviter.

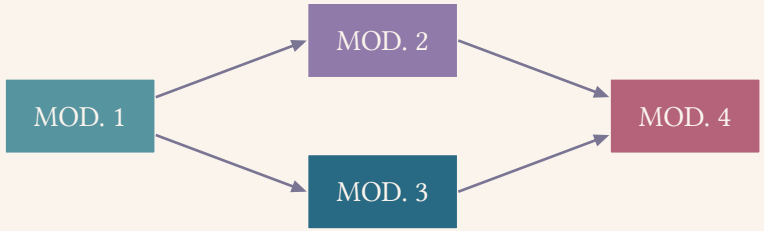
IV. Les pièges à éviter.

IV.1. Les cycles.

Lorsque l'on inclus des fichiers, il est important d'éviter de créer des cycles. En effet, si l'on essaie d'inclure un des fichiers, on doit donc inclure un autre fichier, ..., et on se retrouve à inclure le 1er fichier. On essaie d'inclure le 1er fichier dans le 1er fichier ! Et ça, `gcc`, il n'aime pas du tout.



Le graphe des `#include "..."` n'a pas besoin d'être un arbre, juste d'être acyclique. On peut avoir un module qui importe deux autres modules, et ces deux autres modules importent un même module. Un dessin sera plus simple pour comprendre.



Ça ressemble à un cycle, mais ce n'en n'est pas un, comme le graphe est *orienté*.

IV.2. Importer plusieurs fois un même module.

Reprenons l'exemple avec le diagramme en losange. Que se passe-t-il si l'on inclus le module 4 ?

(1) On commence à inclure le module 4.
(2) Le module 4 a besoin du module 2.
  (a) On commence à inclure le module 2.
  (b) Le module 2 a besoin du module 1.
    (i) On commence à inclure le module 1.
    (ii) Le module 1 est inclus.
  (c) Le module 2 est inclus.
(3) Le module 4 a besoin du module 3.
  (a) On commence à inclure le module 3.
  (b) Le module 3 a besoin du module 1.
    (i) On commence à inclure le module 1.
    (ii) Le module 1 est inclus.
  (c) Le module 3 est inclus.
(4) Le module 4 est inclus.

Le module 1 est inclus *deux fois*. Et ça, `gcc`, il n'aime pas du tout.

Pour cela, on utilise une astuce : les commandes préprocesseur. (Les commandes préprocesseur *en général* ne sont pas au programme, mais elles le sont dans le cadre des inclusions de fichiers.)

Le préprocesseur est un programme très simple qui est lancé avant le compilateur C. C'est notamment lui qui s'occupe des inclusions `#include<...>` et `#include "..."`. Il nous permet également de définir des variables, et de tester si une variable est définie.

L'astuce est que, lorsqu'on inclus un module pour la première fois, on définit une variable qui dit que ce module a déjà été inclus. On peut donc *skip* l'inclusion les fois suivantes.

Reprenons l'exemple de `linked_list.h`. La commande `#ifndef` permet de tester si une variable n'est pas définie (*if not defined*). La commande `#define` définit une variable (sans aucune valeur, mais on s'en fiche, on veut juste savoir que cette variable a été définie). Et enfin, la commande `#endif` ferme le *if* commencé avec `#ifndef`.

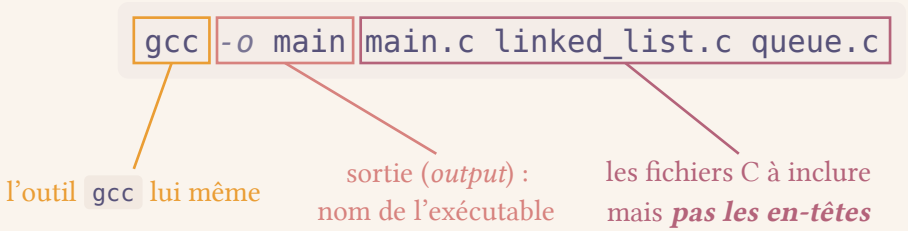
linked_list.h
```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

typedef struct node_t { ... } node;
[...]
void ll_free(linked_list);

#endif
```

Voilà ! Cette astuce permet d'éviter d'inclure deux fois le même module.

V. Compiler avec `gcc`.

L'outil `gcc` est un compilateur C. C'est notamment celui qui est utilisé pendant les oraux en MPI (pour les TPs d'informatique). Il est également très utilisé dans l'industrie, car il est *open-source* (*i.e.* libre d'accès, et d'être modifié sans problèmes de *copyright*).

Pour compiler plusieurs fichiers (par exemple, `linked_list.c`, `queue.c` et `main.c`), on utilise la commande sur la page suivante.



Et c'est tout pour la compilation ! Il suffira d'exécuter `./main` (ou le nom que vous avez après le `-o`) et le programme s'exécutera.

VI. Exercice d'application.

On continue sur l'exemple de ce document . . .

Q1. Coder un module LISTE CHAÎNÉE. Il contiendra les mêmes fonctions que celles du fichier `linked_list.h` (page 2).

Q2. Coder un module FILE. Il contiendra les mêmes fonctions que celles du fichier `queue.h` (page 2).

Q3. Coder un module ARBRE BINAIRE (*binary tree* en anglais) où les valeurs seront des flottants. Il contiendra plusieurs fonctions :
- création d'un arbre binaire, création d'un nœud ;
- calcul de hauteur et de taille d'un arbre ;
- parcours en profondeur en ordre préfixe, infixe et suffixe d'un arbre ;
- parcours en largeur (on utilisera une file).

Q4. Coder un module ABR (arbre binaire de recherche, ou *binary search tree* en anglais). On utilisera le module ARBRE BINAIRE, et on ajoutera les fonctions :
- insertion/suppression d'une valeur d'un ABR ;
- calcul du min/max d'un ABR.