Langage C
Modularisation de programmes C

1 Un programme C divisé en plusieurs fichiers

Pourquoi diviser un programme en plusieurs fichiers ?

- *lisibilité du code* – le programme est plus facile à comprendre si on regroupe dans chaque fichier les fonctions qui constituent un ensemble, qui agissent sur les mêmes données (un peu comme une classe en java).
- *facilité des tests* – on peut faire un jeu de tests séparé pour les fonctions de chaque fichier source,
- *gain de temps* – chaque fichier peut être compilé séparément, et si l'on modifie les fonctions d'un fichier, il suffit de recompiler seulement ce fichier. Recompiler un fichier de quelques centaines ou quelques dizaines de lignes prend peu de temps par rapport à la recompilation d'un fichier de dizaines ou centaines de milliers de lignes,
- *réutilisabilité* – un module (l'ensemble des fonctions d'un fichier source) peut facilement être réutilisé par d'autres programmes,
- *maintenabilité* – le découpage en modules bien conçus diminue l'interdépendance. Il devient plus facile de faire des changements dans le code ou dans le choix de structures de données sans avoir à modifier le reste du programme.

Quand un programme C est divisé en plusieurs fichiers sources, la compilation se déroule en deux phases : d'une part la compilation de chacun des fichiers ; d'autre part la phase de *linkage*, la liaison des fichiers produits par la compilation.

1. La compilation de chaque fichier source `*.c` produit un *fichier objet* `*.o`. Les fichiers objets ne sont pas exécutables. On suppose qu'il y a exactement un fichier source contenant une fonction `main`.
2. Dans la phase de linkage, un programme – le *linker* – assemble tous les fichiers objets en un programme exécutable. Le linker résout aussi toutes les références vers les bibliothèques externes.

Bien sûr un Makefile bien conçu facilite grandement ces deux tâches.

2 Contraintes sur le choix des noms

Une variable définie à l'extérieur de toute fonction est appelée une variable *globale* ou de *niveau* 0. Les fonctions en C sont toutes globales (de niveau 0) puisqu'une fonction ne peut être définie à l'intérieur d'une autre.
Comme vous le savez (ou devez le savoir), un programme C ne peut pas définir deux fonctions portant le même nom (même si ces deux fonctions n'ont pas les mêmes paramètres). Cela reste vrai lorsque le programme est divisé en plusieurs fichiers.
Par exemple, si nous avons deux fichiers `liste.c` et `arbre.c` contenant chacun une définition [1] de fonction et si ces deux fonctions portent le même nom, par exemple `afficher`,

1. Rappelons que la définition d'une fonction contient le corps de la fonction entre les accolades.

chaque fichier pourra être compilé séparément, ce qui produira les fichiers objets `liste.o` et `arbre.o`. En revanche, le linker ne pourra pas construire un exécutable utilisant simultanément ces deux fichiers objets. A cause de ce conflit de noms, on ne pourra pas faire un programme utilisant à la fois les listes et les arbres, à moins de modifier l'un des deux modules pour résoudre ce conflit.

3 Fonctions locales à un fichier – `static`

L'ajout de l'attribut `static` à une déclaration de fonction dans un fichier rend cette fonction *locale* à ce fichier. La fonction ne peut plus être en conflit avec d'autres fonctions de même nom définies dans d'autres fichiers sources : elle est visible à l'intérieur du fichier, et uniquement dans celui-ci.
En particulier, toute fonction auxiliaire qui n'a pas vocation à être utilisée à l'extérieur du fichier où elle est définie *doit* être déclarée comme `static` : ceci évite des conflits de nom inutiles et pénibles à gérer.
Dans un projet réel, le créateur d'un module doit rédiger une documentation détaillée du module pour ses utilisateurs potentiels. Mais ces utilisateurs ne pouvant accéder aux fonctions `static`, il est inutile de les documenter (tout comme on ne documente pas les éléments `private` en Java).

4 Les fichiers d'en-tête `*.h`

Soit `liste.c` un module définissant des listes chaînées ainsi que toutes les fonctions de gestion de ces listes. Par exemple, `liste.c` commencera de la manière suivante (pour simplifier, les instructions du corps des fonctions entre les accolades ne sont pas écrites) :

```c
/* liste.c */
struct node{
  int val;
  struct node *suivant;
};
typedef struct node *liste;
liste ll_ajouter(liste l, int i){    /* ... */   }
liste ll_chercher(liste l, int i){   /* ... */   }
void  ll_afficher_liste(liste l){    /* ... */   }
liste ll_supprimer_premier(liste l){ /* ... */   }
/* d'autres fonctions */
```

Dans un autre fichier, par exemple `main.c`, certaines fonctions se servent des listes ainsi que des fonctions de gestion définies dans `liste.c` :

```c
/*   main.c  */
#include <stdio.h>
int main(){
  liste l = ll_ajouter(NULL, 1);
  l = ll_ajouter(l, 5);
  /* etc. */
}
```

Tel quel, le fichier `main.c` n'est pas compilable : le compilateur ne reconnaît ni le type `liste`, ni la fonction `ll_ajouter`. Une solution naïve serait de d'ajouter au début de `main.c` les définitions de structures de données de `liste.c`, ainsi que les prototypes [2] de ses fonctions de gestion :

```c
/*   main.c    */
#include <stdio.h>
struct node{
  int val;
  struct node *suivant;
};
typedef struct node *liste;
liste ll_ajouter(liste l, int i);
liste ll_chercher(liste l, int i);
void  ll_afficher_liste(liste l);
liste ll_supprimer_premier(liste l);   /* etc */
int main(){
  liste l = ll_ajouter(NULL, 1);
  l = ll_ajouter(l, 5);               /* etc. */
}
```

Cette seconde version de `main.c` compile maintenant correctement, et le linker peut assembler `main.o` et `liste.o` en exécutable. Cependant, cette solution est loin d'être satisfaisante : programmer par copier-coller est à la fois peu élégant et propice aux erreurs. C'est plutôt à l'auteur du module `liste.c` de fournir aux utilisateurs du module les éléments nécessaires à son usage : c'est à cela que sert ce qu'on appelle un *fichier d'en-tête*.

Pour chaque fichier `*.c` qui ne contient pas de fonction `main`, l'auteur doit fournir un fichier d'en-tête `*.h` contenant les définitions de types et les prototypes des fonctions non `static` définies dans ce module (ne mettez **jamais** de prototypes de fonctions static dans les fichiers `*.h`). Dans notre exemple, le fichier `liste.h` (portant le même nom que `liste.c`, mais avec l'extension `.h`) s'écrira de la manière suivante :

```c
/*    liste.h   */
#ifndef LISTE_H
#define LISTE_H
struct node{
  int val;
  struct node *suivant;
};
typedef struct node *liste;
extern liste ll_ajouter(liste l, int i);
extern liste ll_chercher(liste l, int i);
extern void  ll_afficher_liste(liste l);
extern liste ll_supprimer_premier(liste l);  /* etc */
#endif
```

2. Rappelons que le prototype d'une fonction ne contient pas le corps de la fonction entre les accolades : il se termine immédiatement par un point-virgule après la parenthèse refermant sa suite d'arguments. Ce prototype permet au compilateur de connaître le *typage* de la fonction, sans pour autant connaître son implémentation si elle est définie à l'extérieur du fichier compilé.

Le fichier `liste.h` contient trois lignes « magiques » qui commencent par `#`, deux lignes au début de fichier et une qui termine le fichier.

Le nom `LISTE_H` qui apparaît dans `#ifndef` à la première ligne est un nom fabriqué à partir de nom de fichier `liste.h` en remplaçant les lettres minuscules par les lettres majuscules et le point par le caractère `_` (souligné).

Les lignes qui commencent par `#` sont ce qu'on appelle des directives de preprocessing. Sans entrer dans des détails trop techniques, ces trois lignes magiques permettent de garantir que le fichier `liste.h` ne sera jamais inclus deux fois dans un autre fichier, ce qui peut provoquer des erreurs. Pour un programme simple c'est une précaution sans doute inutile mais il vaut mieux prendre de bonnes habitudes. Nous imposerons donc la règle suivante :

Tout fichier d'en-tête doit contenir ces trois lignes magiques. De plus le nom qui suit `#ifndef` doit être construit à partir du nom du fichier d'en-tête en remplaçant les lettres minuscules par les lettres majuscules et le point par le caractère `_`.

Noter que les fichiers d'en-tête de la librairie standard [3] suivent (presque) les mêmes règles, par exemple `string.h` commence par `#ifndef _STRING_H` et termine par `#endif`.

Un fois `list.h` écrit, le fichier `main.c` peut prendre la forme suivante :

```c
/* main.c */
#include <stdio.h>  /* inclusion d'un header standard */
#include "liste.h"  /* inclusion de liste.h - noter les guillemets doubles */
int main(){
  liste l = ll_ajouter( NULL, 1);
  l = ll_ajouter( l, 5);
  /* etc */
}
```

Comme pour les `includes` de fichier d'en-tête standards, le pré-processeur remplacera la ligne `#include "liste.h"` par le texte du fichier `liste.h`. Nous pouvons aussi modifier le fichier `liste.c` en écrivant :

```c
/*   liste.c  */
#include "liste.h"  /* inclusion de liste.h */

/*  les définitions de fonctions */
liste ll_ajouter(liste l, int i){    /* ... */ }
liste ll_chercher(liste l, int i){   /* ... */ }
void  ll_afficher_liste(liste l){    /* ... */ }
liste ll_supprimer_premier(liste l){ /* ... */ }
/* d'autres fonctions */
```

Autrement dit, les définitions de types sont déplacées dans `liste.h`, et ce fichier d'en-tête est inclus dans `list.c`.

Remarque. Pourquoi cette dernière inclusion ? La raison est simple : l'auteur du module doit pouvoir garantir que le typage des fonctions dans `list.c` (l'impl'ementation effective

3. Allez voir le contenu du répertoire `/usr/include`.

du module) est conforme aux prototypes écrits dans `list.h` (l'information fournie aux utilisateurs du module). Si l'auteur décide de changer le typage d'une des fonctions de `liste.c` mais oublie de faire de même dans `liste.h`, la compilation de `liste.c` échouera, ce qui l'obligera à mettre les deux fichiers en accord. Sans cette inclusion, il peut être difficile de retrouver ce type d'erreurs (une fonction avec deux typages différents dans un `*.c` et son `*.h` associé).

Résumé.

1. Pour chaque fichiers `*.c` qui ne contient pas de `main`, on écrit un fichier d'en-tête avec le même nom et l'extension `.h`.
2. On met dans le fichier d'en-tête `*.h` les trois lignes qui commencent par `#`, avec le nom qui suit `#ifndef` fabriqué à partir de nom de fichier.
3. On met dans le fichier en-tête `*.h` les prototypes de toutes les fonctions **non** `static` définies dans `*.c`. Les prototypes peuvent être précédés par `extern` mais ce n'est pas obligatoire.
4. On transfère dans `*.h` les définitions de types qui sont utilisés par ces fonctions (définitions globales).
5. On fait l'inclusion de fichier `*.h` dans tous les fichiers sources `*.c` qui utilisent les fonctions de ce `*.h`.

5 Partage de variables globales

Une déclaration *identique* d'une variable globale dans plusieurs fichiers est possible, à condition que cette variable soit initialisée dans *un seul* fichier ou qu'elle ne soit pas initialisée du tout [4]. On peut par exemple écrire la déclaration globale `int compteur = 5;` dans `liste.c` et la déclaration globale `int compteur;` dans `arbre.c`. Le nom `compteur` désignera dans les deux fichiers la *même* variable de type `int` – le même espace mémoire – initialisée à 5. La mémoire pour cette variable est allouée dans le module qui l'initialise, ici dans `liste.c`.

En revanche, si l'on déclare dans deux fichiers distincts deux variables globales de même nom mais *de types différents*, par exemple si `liste.c` contient la déclaration globale `int compteur;` et si `arbre.c` contient la déclaration globale `long compteur;` le linker ne pourra pas lier `liste.o` et `arbre.o` pour produire un exécutable : `compteur` ne peut pas être à la fois `int` et `long`.

Comme pour les fonctions, une variable globale avec l'attribut `static` est une variable visible uniquement dans le fichier où elle est déclarée. Cette variable n'apparaît donc jamais dans un fichier en-tête. A partirce point, nous supposerons que tout variable globale est non `static`.

4. Rappelons que, quand une variable globale n'est pas initialisée explicitement elle sera initialisée par défaut à 0. Cette initialisation par défaut concerne uniquement les variables globales et non pas les variables locales à une fonction.

Recommandations. Pour enlever toute ambiguïté, il est **fortement recommandé** que chaque variable globale ne soit déclarée que dans **un seul** fichier `*.c` et, si elle doit être initialisée, qu'elle le soit uniquement dans ce fichier.
Par exemple, nous pouvons écrire dans `list.c` la déclaration globale :

```
/* globale dans liste.c */
int compteur = 5;
```

et dans le `liste.h` correspondant écrire :

```
/* dans liste.h */
extern int compteur;
```

Comme pour les prototypes de fonctions dans les fichiers en-tête `*.h` l'attribut `extern` n'est pas obligatoire mais il est recommandé.

6 Makefile

Il nous reste à écrire un `Makefile` pour un programme composé de plusieurs fichiers. Celui que nous présentons ici est le plus simple possible. Au début de fichier `Makefile`, on déclare les variables de `make`

```
CC=gcc
CFLAGS= -Wall  -g
LDLIBS= -lm
```

`LDLIBS` seulement si le programme utilise des bibliothèques. Après ces définitions on ajoute les dépendances.

La toute première dépendance, écrite toujours avant toutes les autres, est celle permettant de générer le fichier exécutable.
Supposons par exemple que le programme soit découpé en trois fichiers `main.c`, `liste.c` et `arbre.c` – un seul de ces fichiers contenant une fonction `main`. On souhaite que l'exécutable porte le nom `main`.
Pour produire cet exécutable, le linker doit disposer des trois fichiers objets `main.o`, `liste.o` et `arbre.o`. On écrit donc :

```
main : main.o liste.o arbre.o
```

Cette dépendance sera utilisée par `make` pour générer la commande lançant le linker, qui assemble les fichiers objets, résoudra tous les liens, et produira le fichier exécutable `main`.

Après cette première dépendance, pour chaque fichier objet généré par le compilateur, on indique quels sont les fichiers nécessaires à cette compilation (code-source, fichiers d'en-têtes), par exemple :

```
main.o : main.c liste.h arbre.h

liste.o : liste.c liste.h

arbre.o : arbre.c liste.h arbre.h
```

`make` utilisera ces dépendances pour générer les commandes de compilation qui produiront chaque `*.o`. Notez que chaque `*.c` doit être le premier élément de sa liste, juste après les deux-points. A la fin de `Makefile` vous pouvez ajouter les commandes habituelles de nettoyage :

```
cleanall:
      rm -rf *.o main *~
clean:
      rm -rf  *~
```

où les lignes indentées commencent par le caractère TAB.

6.1 Quand les règles implicites ne suffisent pas

Le Makefile décrit dans la section précédente est suffisant quand les noms de fichiers correspondent, par exemple `main.c` est compilé vers `main.o` et qui donne un exécutable `main`. Mais si nous voulons un exécutable qui s'appelle `toto` il faut écrire une commande explicite dans ligne qui suit la dépendance :

```
toto : main.o liste.o arbre.o
       $(CC) $^ $(LDLIBS) -o $@
```

La commande qui suit la dépendance utilise les variables de `make` :
- `$(CC)` sera bien sûr remplacé par la valeur de la variable `CC`,
- `$(LDLIBS)` sera remplacé par la valeur de la variable `LDLIBS`,
- `$^` sera remplacé par tout ce qui est à droite de deux points, c'est-à-dire `main.o liste.o arbre.o`
- et `$@` sera replacé par ce qui est à gauche de deux point, c'est-à-dire par `toto`.

La ligne de commande commence par le caractère de tabulation (l'indentation de la ligne est provoquée par ce caractère et non pas par une suite d'espaces).

7 Cacher des structures de données

Cette section n'est pas obligatoire, mais elle contient des informations importantes sur la modularisation de programmes en C.
Dans la section 4 nous avons montré sur un exemple de module de listes chaînées `liste.c` comment fabriquer le fichier d'en-tête `liste.h` correspondant. Le fichier `liste.h` de la section 4 contient la définition du type de structure **struct node**.
Chaque module `*.c` qui inclut `liste.h` « connaît » les champs de `struct node` : les fonctions de ces modules peuvent donc, en principe, accéder directement aux champs de `struct node` sans utiliser les fonctions définies dans `liste.c`[5]. Autrement dit, nous dévoilons à l'utilisateur du module les détails de son implémentation.

5. Cette situation est similaire à celle d'une classe Java dont les variables d'instances sont toutes `public` : l'utilisateur de la classe peut accéder directement aux variables d'instances sans utiliser les méthodes de cette classe.

Pourtant, si le module `liste.c` est bien conçu, l'utilisateur n'a aucune raison de connaître le détail de la définition de `struct node` : toutes les opérations autorisées sur les listes sont implémentées dans le module `liste.c`, et il n'est pas souhaitable qu'une autre partie du programme dépende de leur représentation interne.
Le but est donc d'enlever la définition de structure

```
struct node{
   int val;
   struct node *suivant;
 };
```

du fichier `liste.h`, et de la remettre dans `liste.c`, tout en rendant visible le *nom de type* `liste` dans le fichier `liste.h` sans en donner la définition exacte. Il s'avère que c'est possible. Le fichier `liste.h` peut s'écrire :

```
/*    liste.h   */
#ifndef LISTE_H
#define LISTE_H
typedef struct node *liste;
extern liste ll_ajouter(liste l, int i);
extern liste ll_chercher(liste l, int i);
extern void  ll_afficher_liste(liste l);
extern liste ll_supprimer_premier(liste l);
/* etc */
#endif
```

et l'on peut remettre

```
struct node{
   int val;
   struct node *suivant;
 };
```

dans `liste.c`.
Supposons que le fichier `main.c` fait inclusion de `liste.h`. De point de vue de `main.c` le type `liste` est un pointeur vers une structure `struct node` dont les champs sont inconnus. Tant que dans `main.c` on n'essaie pas d'accéder aux champs de la structure `struct node` le compilateur est capable de compiler le fichier `main.c` et de produire `main.o`.
Autrement dit, le compilateur peut compiler un module dont les fonctions utilisent des pointeurs vers un type de structure même si, au moment de cette compilation, les champs de ces structures ne sont pas connues, et lorsque ces fonctions n'accèdent pas aux champs de ces structures.