Langage C

Wieslaw Zielonka
zielonka@irif.fr

programme C divisé en plusieurs fichiers

Modularisation de programmes en C

Pourquoi un programme avec plusieurs fichiers source ?

Comment faire en sorte que les fonctions dans un fichiers puissent appeler les fonctions définies dans un autre fichier ?

Comment compiler chaque fichier source ?

Comment rassembler en un seul programme exécutable plusieurs fichiers compilés séparément ?

Comment fabriquer un Makefile pour un programme composé de plusieurs fichiers source ?

Modularisation de programmes en C

En C chaque "module" correspond à un fichier source, par exemple :

- un fichier source avec les fonctions pour le traitement des arbres,
- un autre fichier source pour les fonctions de traitement de listes etc. etc.
- et finalement un fichier avec la fonction main() et peut-être d'autres fonctions qui utilisent aussi bien les arbres que les listes.

Pourquoi un programme avec plusieurs fichiers source ?

Permet diviser le projet en plusieurs modules qui peuvent être développés et testés séparément.

Gain du temps.
Si un seul fichier de plusieurs dizaines milliers de lignes et à chaque modification il faut recompiler 50000 lignes de code?
Le temps de compilation de 200 lignes n'est pas le même que pour 50000 lignes.

Réutilisation du même code dans plusieurs projets.

Création de bibliothèques.

Compilation

`editeur.c --->  éditeur.o`

compilation : le compilateur génère à partir de fichier source *.c un fichier objet *.o

Editeur de lien (linker) rassemble les fichiers *.o en un programme exécutable. Un seul fichier source contient la fonction main()

Appeler les fonctions définies dans un autre fichier

```
/* liste_lignes.c */


typedef struct ligne ligne;
typedef ligne *liste;

struct ligne{
  char * text;
  liste suivant;
};

static liste
supprimer_ligne_suivante(liste l){
}

liste creer_liste(void){  }

liste supprimer_ligne(liste l, int
i){   }

liste ajouter_ligne(liste l, int i,
char *contenu){   }

void remplacer_ligne(liste l, char
*contenu){    }
```

```
/* editeur.c */


liste fichier2liste(char *file){
   FILE *flot = fopen(file,"r+");
   liste l = creer_liste();
   char  *txt;

  while( ... ){
       ........
      txt = lire_ligne(flot);
       l = ajouter_ligne(l, i,
                         twt);

}
```

les fonctions dans editeur.c font appel aux fonctions de liste_ligne.c

Appeler les fonction définies dans un autre fichier

Pour compiler le fichier

`editeur.c`

le compilateur :

1. doit connaître les prototypes de fonctions définies dans

`listes_lignes.c`

Pourquoi?

2. doit savoir ce que c'est le type

`liste`

Préparer un fichier en-tête liste_lignes.h

Ecrire un fichier en-tête

`liste_lignes.h`

qui contient les prototypes de toutes les fonctions définies dans le fichier list_lignes.c **sauf celles qui ont l'attribut static**.

Une fonction **static** est visible uniquement à l'intérieur de fichier *.c qui contient sa définition. Une fonction static est locale, "privée", elle peut être utilisée uniquement dans le fichier où elle est définie. Les prototypes de fonctions static ne sont jamais écrit dans le fichier *.h

REGLE

Le fichier en-tête *.h porte le même nom que le fichier *.c correspondant, juste l'extension est .c est remplacée par .h

Préparer un fichier liste_lignes.h

```
/* liste_lignes.h    */

typedef struct ligne *liste; /* définir le type liste */

extern liste creer_liste(void) ;
extern liste supprimer_ligne(liste l, int i) ;
extern liste ajouter_ligne(liste l, int i, char *contenu) ;
extern void remplacer_ligne(liste l, char *contenu) ;
```

On suppose que l'utilisateur n'accède jamais directement aux champs de struct ligne.

Dans ce cas pas la peine de définir dans le fichier en-tête *.h la structure **`struct ligne.`**
Il suffit de définir `liste` avec `typedef.`

Ajouter un include

```
/* editeur.c */

#include "liste_lignes.h"

liste fichier2liste(char *file){
   FILE *flot =
fopen(file,"r+");
   liste l = creer_liste();
   char  *txt;

  while( ... ){
      ........
      txt = lire_ligne(flot);
       l = ajouter_ligne(l, i,
                         txt);

}
```

Ajouter

```
#include "liste_lignes.h"
```

dans chaque fichier source qui utilise les fonctions de définies dans liste_lignes.c

Notez le nom de fichier *.h entre " ".

Et si l'inclusion s'enchaînent ?

```c
/* editeur.c */

#include "liste_lignes.h"

liste fichier2liste(char *file){
   FILE *flot = fopen(file,"r+");
   liste l = creer_liste();
   char  *txt;

  while( ... ){
      ........
      txt = lire_ligne(flot);
       l = ajouter_ligne(l, i, txt);
}
```

```c
/* editeur.h */


extern liste
fichier2liste(char *file) ;
```

Fabriquer le fichier editeur.h pour editeur.c.

La règle : pour chaque fichier *.c un fichier *.h correspondant **sauf pour le fichier qui contient la fonction** main().

Problème : le type liste n'est pas défini dans editeur.h.

Et si l'inclusion s'enchaînent ?

```c
/* editeur.c */

#include "liste_lignes.h"

liste fichier2liste(char *file){
   FILE *flot = fopen(file,"r+");
   liste l = creer_liste();
   char  *txt;

  while( ... ){
      ........
      txt = lire_ligne(flot);
       l = ajouter_ligne(l, i, txt);
}
```

```c
/* editeur.h */

#include "liste_lignes.h"

extern liste
fichier2liste(char *file) ;
```

Fabriquer le fichier editeur.h pour editeur.c.

La règle : pour chaque fichier *.c un fichier *.h correspondant sauf pour le fichier qui contient la fonction main().
Problème : le type liste n'est pas défini dans editeur.h.

Ajouter #include dans editeur.h.
Conclusion : les fichiers *.h peuvent faire include de d'autres fichiers *.h. Cela peut inclure aux inclusions multiples du même fichier *.h.

Et si l'inclusion s'enchaînent ?

```c
/* main.c */

#include "liste_lignes.h"
#include "editeur.h"

int main(void){



}
```

Le fichier liste_ligne.h" sera inclus deux fois. Ce qui peut provoquer des problèmes.

Quelles solution ?

Ajouter les lignes "magiques" dans le fichier *.h

```
/* liste_lignes.h    */

#ifndef LISTES_LIGNES_H
#define LISTES_LIGNES_H
typedef struct ligne *liste;
extern liste créer_liste(void) ;
extern liste supprimer_ligne(liste l, int i) ;
extern liste ajouter_ligne(liste l, int i, char *contenu) ;
extern void remplacer_ligne(liste l, char *contenu) ;

#endif
```

En rouge les trois lignes magiques pour éviter les inclusions multiples. Notez comment on fabrique les noms de la constante LISTE_LIGNES_H à partir du nom de fichier **liste_lignes.h :**

- **mettre les lettres en majuscule et**
- **remplacer . par _**

Et si les inclusions s'enchaînent ?

```
/* editeur.h */
#ifndef EDITEUR_H
#define EDITEUR_H

#include "liste_lignes.h"

liste file_2_liste(char *file) ;

#endif
```

pour la définition de `liste`

Les directives #ifndef #define #endif empêchent l'inclusion multiples du même fichier *.h

Dernière consigne

```
/* liste_lignes.h    */

#ifndef LISTE_LIGNES_H
#define LISTE_LIGNES_H
typedef struct ligne *liste;

extern liste créer_liste(void) ;

extern liste supprimer_ligne(liste l,
int i) ;

extern liste ajouter_ligne(liste l,
int i, char *contenu) ;

extern void remplacer_ligne(liste l,
char *contenu) ;
#endif
```

```
/* liste_lignes.c */

#include "liste_lignes.h"



struct ligne{
  char * text;
  liste suivant;
};

liste creer_liste(void){  ..... }
```

Chaque fichier *.c doit inclure son propre fichier en tête *.h

Mais pourquoi ? liste_lignes.c connaît bien ces propres fonctions?

On s'assure que les prototypes dans *.h correspondent bien aux définition dans *.c.

Conflits de noms de fonctions

Le linker "voit" toutes les fonctions non-static :

```
/* fichier_a.c  */
int f( int a){        }

int g(int x){

}
```

conflit

```
/* fichier_b.c  */



double g(struct node x){

}
```

Le compilateur peut compiler les deux fichiers *.c pour obtenir deux fichier *.o.

Un programme C ne peut pas contenir deux fonctions qui partent le même nom.

Dans cet exemple il est impossible d'utiliser les deux fichiers .o pour fabriquer un programme exécutable : le linker trouve que le nom g est ambigu même si les deux fonctions g() possèdent les paramètres différents.

Conflits de noms de fonctions

```
/* fichier_a.c  */
int f( int a){        }

static int g(int x){

}
```

pas de conflit

```
/* fichier_b.c  */



static double g(struct node x){

}
```

Les fonctions `static` sont locales à un fichier. Le linker ne s'occupe pas de fonctions static et dans chaque fichier on peut définir une fonction static avec le même nom sans risque de confusion.

Compilation

On suppose que nous avons 5 fichiers source :

liste_lignes.c liste_lignes.h
éditeur.c editeur.h

main.c

main.c contient la fonction main(), il n'y a pas pas de fichiers main.h

compiler en ligne de commande :

```
gcc -g -Wall -c -o  liste_ligne.o liste_ligne.c
gcc -g -Wall -c -o  éditeur.o     éditeur.c
gcc -g -Wall -c -o  main.o        main.c
```

l'option `-c :` compiler mais pas linker, le fichier objet obtenu *.o n'est pas exécutable

Linker les fichiers *.o (et les bibliothèques si besoin) pour obtenir un exécutable main :

```
gcc main.o éditeur.o liste_ligne.o -lm -o mon_programme
```

Maintenant on pourra exécuter le programme :

```
./mon_programme
```

si les fonctions math sont utilisées

variables globales dans un programme C divisé en plusieurs fichiers

Modularisation et variables globales (variables et tableaux de niveau 0)

Les variables globales `static, tableaux static` sont toujours locales à un fichier.

```
/* fich_a.c */

static int a;
static double b;
static double c = 9.9;
static int tab[100];
```

```
/* fich_b.c */

static int a;
static char b ;
static double c = 98.98;
static double tab[200];
```

Les variables a,b,c et tableau tab dans le fichier `fich_a.c` sont visibles uniquement dans ce fichier et n'ont rien à voir avec les variables a,b,c et le tableau tab dans le fichier `fich_b.c.`

Modularisation et variables globales (variables de niveau 0)

Les variables **`non static`** globales sont "visibles" dans tous les fichiers sources qui composent le programme.

Si vous avez un programmes composé de deux fichiers
`fich_a.c` et `fich_b.c`
et dans chaque fichier vous avez une déclaration globale

```
int a;
```

cela désigne la même variable a dans tout le programme. **La variable globale a ne peut être initialisée que dans un seul fichier.**

Vous ne pouvez déclarer une variable globale non static a de type int dans un fichier et une variable globale a non static de type double dans un autre fichier : vous aurez deux déclarations contradictoires.
Vous ne pouvez pas initialiser la même variable de niveau 0 dans deux fichiers sources différents.
Même remarque pour les tableaux déclarés au niveau 0 (en dehors de fonctions).

variables globales (variables de niveau 0) dans plusieurs fichiers

Pour que cela soit vraiment correct et dans l'état d'art :
chaque variable globale doit être déclarée dans un seul fichier :

```
/*  file_a.c */
int a;  /*initialisé automatiquement à 0 */
int b = 6;
```

Dans le fichier file_a.h correspondant :

```
/* file_a.h */
#ifndef FILE_A_H
#define FILE_A_H
extern int a;
extern int b;
#endif
```

Remarque : l'attribut extern n'est pas obligatoire.
Les variables globales doivent être initialisés dans les fichier *.c, jamais dans le fichier en-tête *.h.

variables globales (variables de niveau 0) dans plusieurs fichiers

```
/* file_a.c*/

int y;
```

```
/* file_b.c*/

int y;
```

Il y aura une seule variable globale y dans le programme initialisée par défaut à 0.

Transformer les fichiers sources :

```
/* file_a.c*/

int y;
```

```
/* file_b.c*/

#include "file_a.h"
```

```
/* file_a.h*/

extern int y;
```

variables globales (variables de niveau 0) dans plusieurs fichiers

```
/* file_a.c*/

int z = 3;
```

```
/* file_b.c*/

int z;
```

La mémoire pour la variable z est allouée dans le fichier file_a.o avec la valeur initiale 3.
Dans les deux fichiers z désigne la même variable.

Transformer les fichiers source :

```
/* file_a.c*/

int z = 3;
```

```
/* file_a.h*/

extern int z;
```

```
/* file_b.c*/

#include "file_a.h"
```

variables globales (variables de niveau 0) dans plusieurs fichiers

```
/* file_a.c*/

int z = 3;
```

conflit

```
/* file_b.c*/

int z = 4;
```

Le linker ne pourra pas faire un exécutable en utilisant ces deux fichier.
Le conflit de noms de variables.

et définition de structures dans *.h ou dans *.c ?

```c
/* a.c */

#include <stdlib.h>
#include "a.h"

struct point{
  int x;
  int y;
};



struct point  *f(int a, int b){
  struct point
*p=malloc( sizeof(struct
point));
  p->x=a;
  p->y=b;
  return p;
}
.../* d'autres fonctions*/
```

```c
/* a.h */
#ifndef A_H
#define A_H

struct point;
extern struct point *f(int a,
int b);

/* d'autres prototypes de
fonctions */
#endif
```

```c
#include "a.h"

int main(){
  struct point *p=f(5,6);
}
```

Si les champs de la structure struct point sont utilisés uniquement dans le fichier a.c alors la structure est à définir uniquement dans a.c.

Dans a.h on ajoute juste une sorte de déclaration struct point;

Résumé de fichiers *.h

A mettre dans le fichier *.h :

1. les prototypes de toutes les fonctions qui ne sont pas `static`
2. les déclarations de toutes les variables globales qui ne sont pas `static` (mais si une telle variable est initialisée alors l'initialisation reste dans le fichier *.c)
3. les définitions de types de données utilisés dans 1) et/ou 2)
4. ajouter les lignes "magiques" :

```
#ifndef  NOM_DE_FICHIER_H
#define  NOM_DE_FICHIER_H

   le contenu de fichier *.h
#endif
```

Résumé de fichiers *.h

Où mettre les définitions de struct ? enum ?

Dans le fichier *.c si les champs de struct utilisés juste dans ce fichier, sinon dans *.h.

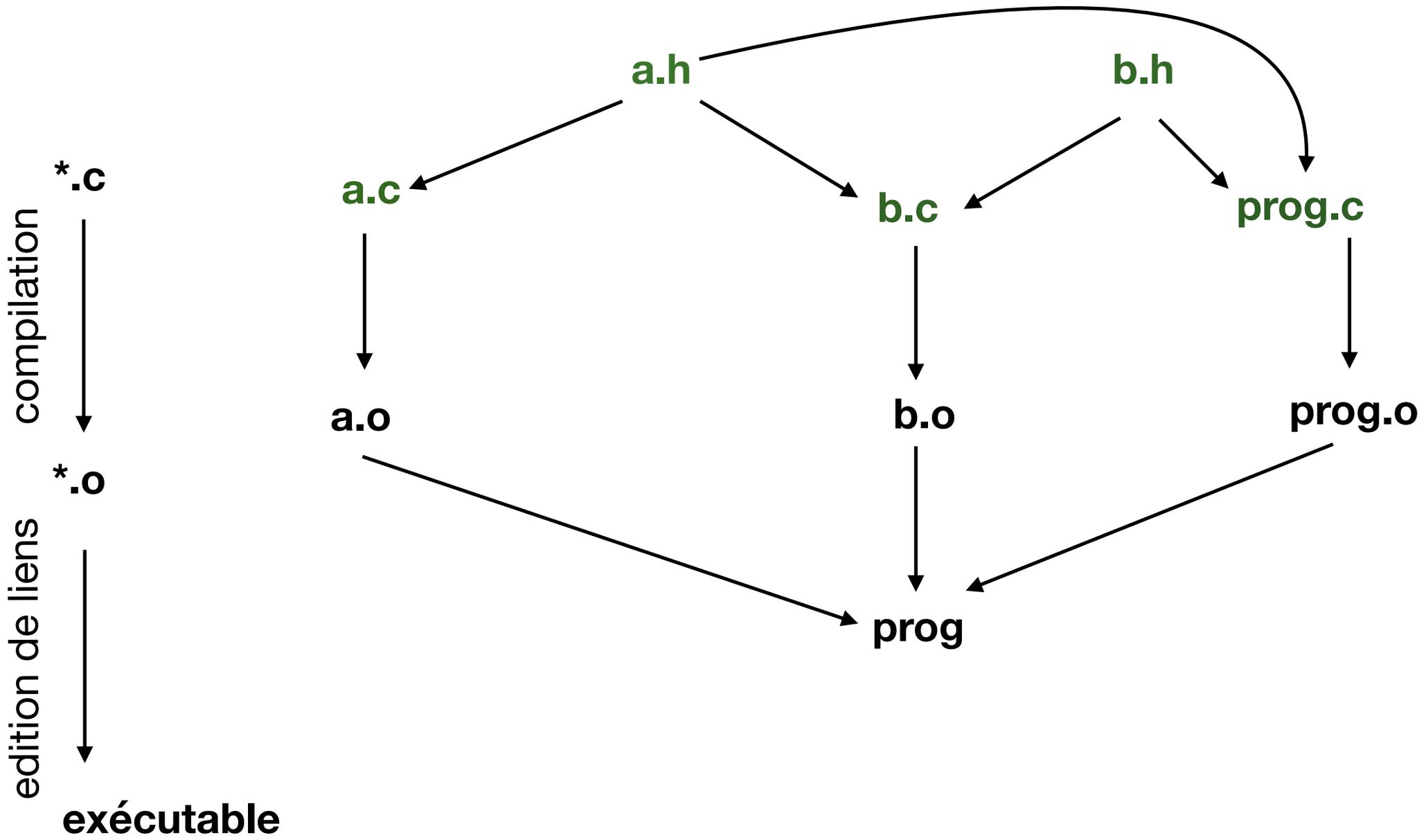
Résumé de fichiers *.h

Un fichier *.h peut faire include d'un autre fichier *h.

Ne jamais mettre les définitions de fonctions dans *.h, uniquement des prototypes.

Ne jamais faire d'inclusion de fichier *.c dans un autre fichier (ni dans un autre *.c ni dans un *h).

graphe de dépendances



prog.c contient la fonction main(),
prog.c est le seul fichier sans prog.h correspondant

Makefile minimaliste (tous les fichiers dans le même répertoire)

```
CC=gcc
CFLAGS=   -Wall -g -pedantic -std=c11
LDLIBS=   -lm

prog: prog.o a.o b.o

a.o : a.c a.h
b.o : b.c b.h a.h
prog.o : prog.c b.h a.h

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

pour compiler juste le nécessaire la commande :

make

pour recompiler tout :

make cleanall
make