Chapitre 3
Types abstraits de données
Listes

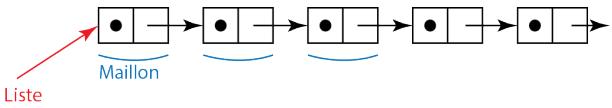
Polytech Marseille - IRM 3ème année
Alexandra Bac

Algorithmique et structures de données

Carte mentale du cours : Chapitre3 - https://mm.tt/1145546945?t=I61qLvmSOn

Table des matières

1 Implémentation en C de listes chaînées




```
struct zMaillon {
   elt Elt ;
   struct zMaillon *suiv ; } ;
typedef struct zMaillon Maillon ;
typedef Maillon * Liste ;
```

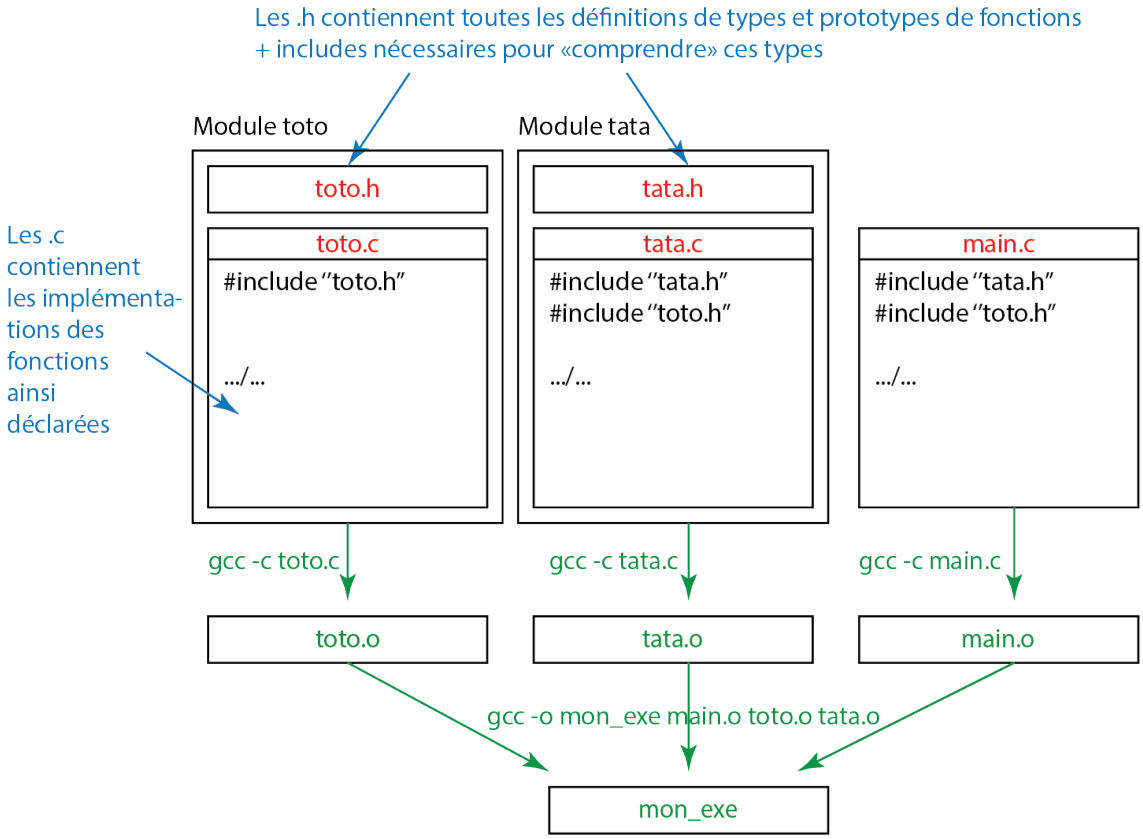
ou ce qui revient au même (avec de l'habitude) :

```
typedef struct zMaillon {
   elt Elt ;
   struct zMaillon *suiv ; } Maillon, * Liste ;
```

2 Programmation modulaire

On imagine qu'un programme est constitué de deux modules :
- toto
- tata (utilisant toto)

et d'un main utilisant l'un et l'autre.



Tout l'intérêt est que lorsque l'on fait des modifications, on ne recompile que les modules ayant été modifiés.

Mais la compilation serait longue sans automatisation (4 commandes gcc ici ...) et il serait dommage que la recompilation des parties modifiées ne soit pas automatique.

On utilise donc la commande `make` qui assure la compilation de manière automatique. Elle lit ses instructions dans un fichier `Makefile` qui doit se trouver dans le même répertoire que les fichiers à compiler (au moins dans un mode "pour débutants").

Un Makefile est constitué :
- de lignes initiales définissant des variables (nom du compilateur, options de compilation - si l'on veut les changer ... il n'y aura qu'une ligne à changer ...)
- puis des instructions de compilation. Elles sont obligatoirement écrites sur deux lignes :

```
label: dependance1, dependance2 ...
    directive de compilation
```

  - Le label est un nom permettant de désigner la directive (généralement le nom du fichier que l'on est en train de compiler, par exemple `toto.o`, `main.o` ou `mon_exe.o`).
  - Les dépendances sont les fichiers à surveiller : dès que l'un d'eux est modifié, la présente directive sera relancée

— La ligne de la directive de compilation commence **obligatoirement par une tabulation** (attention !!! pas d'espaces, ça ne passe pas), puis on écrit la commande qui doit être exécutée (par exemple `gcc -c toto.c`).
— On ajoute souvent à la fin une directive `clean` permettant d'effacer les .o intermédiaires et l'exécutable pour relancer une compilation propre.
— Enfin, `make` exécuter la première directive qu'il rencontre dans le fichier Makefile ... donc on ajouter également souvent une directive :
`all: mon_exe`

ne faisant rien ... c'est une forme de `goto` permettant de renvoyer `make` vers le bon point d'entrée dans le fichier (qu'on pourra alors mettre dans l'ordre que l'on veut ...)

Le Makefile correspondant au programme précédent est :

```
CC = gcc
CCOPTS = -Wall

all: mon_exe


toto.o: toto.c toto.h
    ${CC} ${CCOPTS} -c toto.c

tata.o: tata.c tata.h
   ${CC} ${CCOPTS} -c tata.c

main.o: main.c
    ${CC} ${CCOPTS} -c main.c

mon_exe: main.o toto.o tata.o
    ${CC} ${CCOPTS} -o mon_exe main.o toto.o tata.o

clean:
    rm *.o ; rm mon_exe
```

Dernier point important, comme les modules peuvent être éventuellement inclus plusieurs fois (ici le `main` utilise `tata` et `toto`, mais `tata` inclut lui-même `toto`), tous les fichiers `.h` respectent la convention suivante : **on y définit une macro du style `__NOM__` qui lui sera propre et permettra de savoir si on est déjà passé dans le .h ou non**.

Ainsi `tata.h` sera de la forme :

```
#ifndef __TATA_H__
#define __TATA_H__
...
...
... // le code
...
...
#endif
```