Introduction à la programmation
TD 5

Polytech Marseille - 1ère année

Filière Génie Biomédical

Dans toute la suite, on considèrera que le type des listes de caractères est implémenté par :

```
typedef char tdata ;

struct zmaillon {
   tdata elt ;
   struct zmaillon * suiv ;
} ;

typedef struct zmaillon maillon ;
typedef maillon * liste ;
```

Vous mettrez ces lignes au début de votre programme, juste après les include et define.

Nous avons défini un type `tdata` pour faciliter l'adaptation du programme si l'on veut changer le type contenu dans les listes ... il suffit alors de modifier la première ligne ...

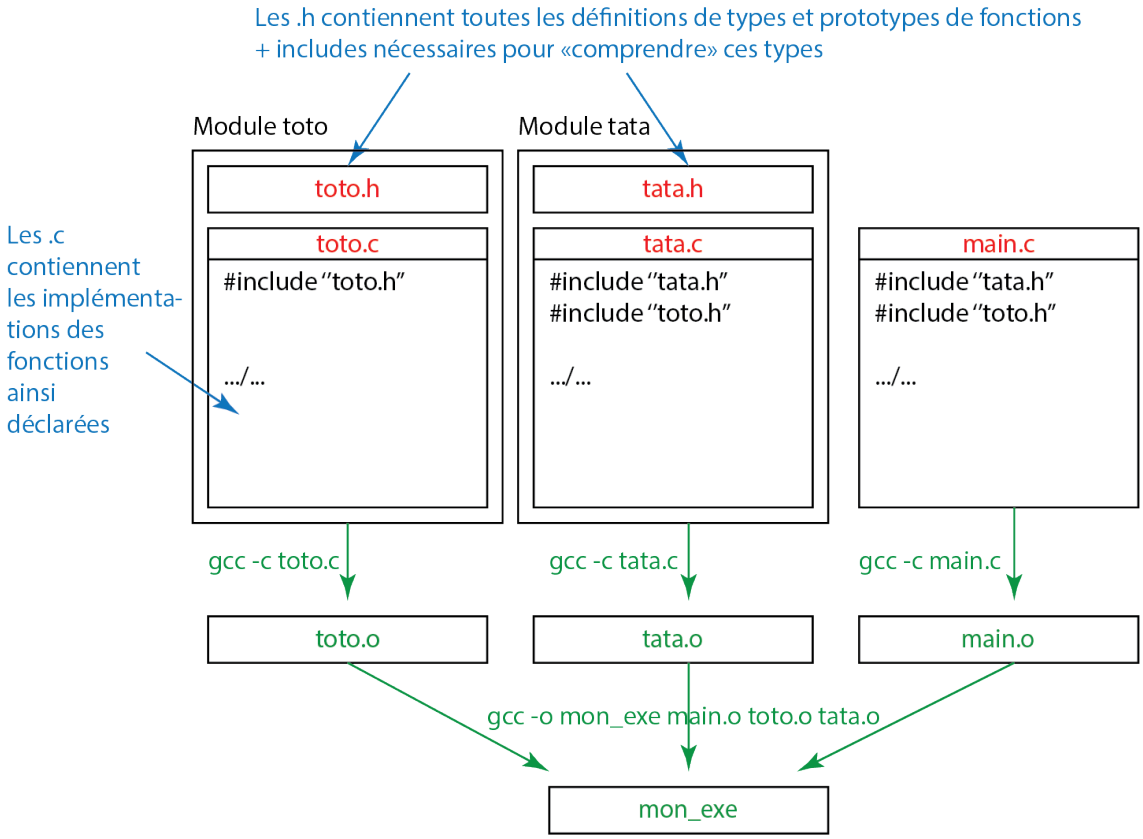
1 Programmation modulaire

Un module regroupe un ensemble de fonction cohérent offrant une "boîte à outils" clé en mains pour les autres parties du programme. La programmation modulaire consiste donc à découper un programme en tels modules. Par exemple, un programme utilisant les listes pour stocker des chaînes de caractères pourra naturellement être découpé en :

- un module `lists` contenant la déclaration des listes de chaînes de caractères et toutes les fonctions de base sur les listes
- un module `main` utilisant le module `lists` pour implémenter un dictionnaire stockant des chaînes de caractères et offrant à l'utilisateur différentes fonctions d'utilisation.

On imagine maintenant qu'un programme est constitué de deux modules :

- toto
- tata (utilisant toto)
- et un main utilisant l'un et l'autre



Tout l'intérêt est que chaque morceau est compilé indépendamment pour fabriquer un code intermédiaire (compilation avec l'option `-c` pour fabriquer un `.o`). Pour l'exécutable est fabriqué en groupant tous ces fichiers intermédiaire. Ainsi lorsque l'on fait des modifications, on ne recompile que les modules ayant été modifiés ainsi que l'exécutable final.

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
```

2 Implémentation du TAD

Exercice 1. Ecrire les fonctions du type abstrait de données des listes. On rappelle leur prototype :

```
liste creer_liste_vide () ;
int est_liste_vide (liste l) ;
liste ajouter (tdata e, liste l);
```

```
tdata tete (liste l) ;
liste queue (liste l) ;
```

Exercice 2. Vous testerez alors vos fonctions à partir du main :

(i) Déclarer deux variables listes `l1` et `l2`.

(ii) Les initialiser à la liste vide.

(iii) Dans `l1` mettre la liste $['a';' b']$ et dans `l2` la liste $['c';' d']$ (pour cela, vous utiliserez les fonctions du TAD).

3 Fonctions utilisant les listes

Dans toutes les fonctions qui suivent, vous utiliserez, si possible, les fonctions du TAD. Vous ne "vous attaquerez aux pointeurs" que si c'est vraiment nécessaire (efficacité, facilité d'implémentation ...).

Exercice 3. Ecrire une fonction permettant d'afficher une liste et la tester sur `l1` et `l2` :

```
void afficher (liste l) ;
```

Exercice 4. Ecrire une fonction calculant la longueur d'une liste :

```
int longueur (liste l) ;
```

Exercice 5. On veut écrire une fonction copiant une liste (donc renvoyant une copie de la liste passée en entrée).

```
liste copier (liste l) ;
```

(i) Pourquoi n'est-il pas facile de réaliser la fonction de manière itérative (avec un for ou un while) ? Si vous essayez de le faire dans quel ordre est la liste résultat ?

(ii) Ecrire la fonction récursive réalisant la copie (et n'utilisant que le TAD) - vous constaterez vous-mêmes sa simplicité ...

Exercice 6. On veut écrire une fonction de concaténation (mettre bout à bout deux listes). Donc par exemple, la concaténation de `l1` et `l2` doit renvoyer $['a';' b';' c';' d']$.

```
liste concatener (liste l1, liste l2) ;
```

(i) Ecrire tout d'abord une fonction récursive n'utilisant que les fonctions du TAD (et ne touchant pas aux pointeurs).

(ii) Que pensez-vous de son efficacité ? Quel est son inconvénient ?

(iii) Ecrire ensuite un fonction itérative réalisant la concaténation en manipulant les pointeurs (en fait un pointeur).

(iv) Quel est son inconvénient ? (modification des arguments) ... Conclusion, on ne peut pas tout avoir ...