Programmation impérative

Cours 5 : Modularisation de programmes C, makefile et débogage

Catalin Dima

Modules

- Les programmes peuvent être décomposés en modules : fichiers séparés contenant du code C.
- Un module peut être inclus dans un programme C si on a besoin des fonctionnalités décrites dans celui-ci.
- Un module comprend :
  - Les déclarations de types et fonctions (interface) : fichier `.h` (header).
  - Les définitions des fonctions (code) : fichier `.c`
- Un module est inclus dans un programme à l'aide de la directive `#include` (comme les bibliothèques !).
- Intérêt de la modularité :
  - Grandes applications (dizaines de milliers de lignes).
  - Modules correspondant à des bibliothèques, à inclure en fonction des besoins.
- Chaque module peut faire appel à un autre module.

Exemple de module

Fichier `complex.h`

```
#ifndef _MON_MODULE_H
#define _MON_MODULE_H

#include "liste.h"   // là-dedans on a la définition
                     // des listes de complexes !
struct complex{
   float re, im;
};

struct complex conjugue(struct complex x);
float valeur_poly(liste_complex l, struct complex x);

#endif
```

Exemple de module

- Le header contient que les déclarations : variables globales, types de données, fonctions.
- Utilisation de `#ifndef/#define/#endif` : directives de pré-compilation, le fichier est à prendre en compte seulement s'il n'a pas déjà été inclus.
- Le code des deux fonctions se trouve dans le `complex.c`
- Notez l'utilisation des guillemets dans la directive `include` : chemin d'accès relatif ! On peut y mettre aussi des chemins d'accès absolus.
  - Dans le cas des `#include<stdio.h>` & co., les `</>` indiquent des bibliothèques à chercher dans le répertoire des bibliothèques de compilation
  - Exemple pour `stdio.h` : répertoire `/usr/include/`.

Exemple de module

Fichier `complexe.c`

```
#include "complex.h"  // sinon pas de définition de struct complex !
#include "liste.h"

struct complex conjugue(struct complex x){
   struct complex res = {x.re, -x.im};
   return res;
}

struct complex valeur_poly(liste_complex l, struct complex c){
   // corps de cette fonction
}
```

Compilation des modules avec `gcc`

- La compilation des programmes C est un processus en plusieurs phases :
  - Pré-compilation : contrôle de la syntaxe, inclusion des headers, suppression des commentaires, traitement des directives `#define`.
  - Traduction en code assembleur.
  - Traduction du code assembleur en code machine et production de fichiers `.o`, un fichier `.o` par fichier `.c` donné.
  - Édition des liens entre les différents codes machine obtenus (et donc inclusion des fonctions !).
- `gcc` peut s'arrêter dans les différentes phases de la compilation, suivant les options :
  - Option `-E` : pré-compil seulement.
  - Option `-S` : compilation sans assemblage, produit des fichiers `.s`.
  - Option `-c` : compilation et assemblage, sans édition de liens, peut prendre en compte des fichiers déjà compilés, produit des fichiers `.o`
  - Option `-o` : édition de liens, peut prendre en compte des fichiers `.o` déjà assemblés.
- Pour compiler donc un programme `main.c` qui inclut un module `complex.c` il faut :
  - `gcc -c complex.c -o complex.o` : fichier assemblé dans `complex.o`, sinon il sera dans `a.out` et cela va générer des problèmes !
  - `gcc -c main.c -o main.o` : pareil, attention à bien nommer le résultat de l'assemblage !
  - `gcc -o main.exe complex.o main.o`.
- Bien-sûr, il faut que `main.c` contienne un `#include "complex.h"`, sinon le compilo râle car ne trouvant pas les définitions des fonctions `conjugue` et `valeur_poly`.

Fichiers `makefile` et commande `make`

- Tout ça demande de lancer en exécution des commandes très compliquées !
- L'utilitaire `make` permet d'automatiser cela.
- Les commandes à exécuter sont mises dans un fichier `Makefile`, dans des cibles.
- Il suffira d'appeler `make` avec le nom de la cible pour que la commande correspondante soit exécutée.
- Même plus, `make` est plus intelligent : il n'exécute une commande que si ses prémises ont changé depuis

Structure d'un `Makefile` très simple

```
cible : dépendances
[tabulation] commande
```

- `cible` : le nom qui sera appelé pour exécuter la commande.
- `dépendances` : les règles nécessaires à l'exécution de la commande.
- `commande` : dans notre cas, le lancement des différentes phases de la compilation.
- Si une cible est un nom de fichier, la commande correspondante sera exécutée que si sa dépendance a changé.

Exemple :

```
all: complex.o main.o
    gcc -o main.exe complex.o main.o
complex.o: complex.c
    gcc -c complex.c -o complex.o
main.o : main.c complex.h
    gcc -c main.c -o main.o
clean:
    rm -rf *.o  // commande pour supprimer tous les fichiers .o !
```

Makefiles plus complexes

- Possibilité de définir et utiliser des variables :

```
MONCOMPIL = gcc
FLAGSCOMPIL = -W -Wall
main.o : main.c complex.h
    $(MONCOMPIL) -c main.c -o main.o $(FLAGSCOMPIL)
```

- Variables internes :
  - $@ : nom de la cible.
  - $< : nom de la première dépendance.
  - $^ : liste de dépendances.
  - $* : nom du fichier sans suffixe.
- Règles génériques : utilisation du % en tant que caractère joker.

```
%.o: %.c
    $(MONCOMPIL) -o $@ -c $< $(FLAGSCOMPIL)
```

- ... et d'autres...

Déboguage

- Notre programme déconne...
  - Il fait des `segmentation fault`, des `stack overflow`,...
  - ... il n'entre pas dans un `if` ou dans un `for`...
  - ... ou tout simplement il donne des faux résultats !
- Il faut le déboguer :
  - le lancer en exécution jusqu'aux points sensibles,
  - l'interrompre dans son exécution pour le faire avancer par petits pas,
  - afficher les valeurs des variables qui posent problème,
  - voir même modifier les valeurs de certaines variables pour identifier les solutions possibles.

Déboguage à l'aide de gdb

- gdb exécute des fichiers exécutables, pas les `.c` !
- Pour cela, il faut compiler les `.c` avec l'option `-g` de `gcc`, qui permet à gdb d'exécuter pas-à-pas le programme :
  - Relier les lignes de code qui correspondent aux instructions en code-machine (y compris tests, boucles, appels de fonctions)
  - Connaître les zones de stockage de chaque variable.
  - Connaître le type et la structure de chaque variable (e.g. `FILE`).
- Une fois compilé (donc avec `gcc -g nomfichier.c`), on lance gdb en lui donnant en paramètre le nom de l'exécutable.
- ... et gdb prend la main sur l'exécution de notre programme !

Commandes de gdb

- `run` : (re)lance en exécution le programme – qui pourra s'exécuter sans arrêt, ou jusqu'au premier breakpoint rencontré.
- `break` : place un breakpoint à la ligne indiquée (dans le fichier source `.c` !), ou à l'entrée d'une fonction donnée en paramètre. `gdb` interrompt l'exécution du programme lorsqu'il rencontre un tel breakpoint.
- `step` : programme interrompu (mais pas fini !), `gdb` exécute une seule instruction dans le code source. Si cette instruction est un appel de fonction, `step` avance jusqu'à la première instruction du corps de la fonction.
- `print` : affiche la valeur d'une variable ; toutes les règles C de référencement des variables/champs/adresses/indices de table doivent être respectées. Si la variable est de type `struct`, l'ensemble de la structure sera affichée, avec les noms de chaque champ.
  - Attention aux différentes variables locales qui peuvent avoir le même nom dans des fonctions différentes !
  - `gdb` ne sait pas trouver pour vous mêmes les valeurs des variables qui n'appartiennent pas à la fonction de l'instruction courante !
- `list` : affiche le code source entourant l'endroit où le programme est interrompu.
- `next` : avance d'un pas, mais, à la différence de `step`, exécute d'un seul coup un appel de fonction.
- `continue` : avance d'un seul coup jusqu'au prochain breakpoint.
- `clear` ou `delete` : supprime un breakpoint (donné par un no. de ligne ou no. de breakpoint).
- `quit` : sort du programme.

Commandes (un peu plus) avancées de gdb

- `break <ligne> if <condition>` : breakpoint conditionnel, l'exécution est interrompue à la ligne donnée si la condition est satisfaite.

  La condition doit utiliser les variables de la fonction en cours d'exécution !
- `next <nbrelignes>` : avancer plusieurs lignes.
- `finish` : termine la fonction courante.
- `until <no_ligne>` : avance jusqu'à la ligne indiquée.
- `info breakpoints` : affiche l'information sur les breakpoints posés.
- `watch <expression>` : interrompt l'exécution seulement lorsque l'expression change de valeur.
- `backtrace` : affiche la pile d'appels de fonctions.
- `print <expression>` : affiche la valeur de l'expression, évaluée à partir des valeurs courantes de variables.
- `print <var>=<valeur` : modifie la variable, en lui affectant la valeur (ou le résultat de l'expression) indiquée.
  - En fait il s'agit d'une expression de type affectation !
  - Attention au type des variables !
- Raccourcis : `r` pour `run`, `q` pour `quit`, `b` pour `break`, `w` pour `watch`, `p` pour `print`, `c` pour `continue`, `l` pour `list`, etc.
- `help <commande>` : aide en ligne.