

TP 2 — rappels de C - bis

Master première année, option EEA

<http://magphy.ujf-grenoble.fr/ratchov/eea/>

14 septembre 2004

1 Déclarations, définitions, gestion de projet

1.1 Utilisation de déclarations

Rappeler à quoi sert une déclaration. Concernant une même fonction, est-ce qu'on peut avoir dans le même fichier `.c` :

- une déclaration et une définition
- une définition sans déclaration, quand ?
- plusieurs fois la même déclaration
- plusieurs fois la même définition

Exercice 1 *Construire un programme (qui fait ce que vous voulez) et qui contient une fonction `func()` sans arguments en plus de la fonction `main()`, n'oubliez pas sa déclaration (i.e. son prototype).*

Compilez et essayez le programme. Le compilateur `gcc` a une option signalant les définitions oubliés. Compilez le programme (appelé `main.c`) avec la commande :

```
gcc -Wall -Wmissing-prototypes -o main main.c
```

Débrouillez-vous pour que votre programme ne génère aucun warning. Quelle est la différence entre les deux déclarations :

- `void func(void);`
- `void func();`

1.2 Compilation séparée

Lorsque un projet dépasse une certaine taille on préfère fragmenter le source en plusieurs modules (fichiers `.c`). Avec cette approche :

- on gagne en lisibilité
- plusieurs personnes peuvent travailler sur le même projet, chacun travaille sur un module sans interférer avec ses collègues
- après une modification, on n'a pas besoin de recompiler le programme entier, il suffit de recompiler le module modifié.
- on peut utiliser le même module dans deux projets indépendants.

Exercice 2 *Séparez le programme en deux fichiers de sorte que l'un contienne la définition de la fonction `main()` et l'autre celle de `func()`.*

La compilation se fait comme suit :

```
gcc -c main.c
gcc -c func.c
gcc -o prog main.o func.o
```

La première étape compile `main.c` et donne un fichier objet `main.o`, la seconde étape fait la même chose pour `proc.c`, la troisième étape (édition de liens, ou linkage) crée un fichier exécutable où sont inclus (et liés) les deux fichiers objet et les bibliothèques système nécessaires.

Exercice 3 *Recompilez tous les modules avec les options de compilation qui signalent les prototypes (et autres déclarations) manquants (cf. sec. 1.1). Faites de sorte qu'il y ait toutes les déclarations nécessaires, pour que le compilateur n'émette pas de warnings.*

Si tout va bien, vous avez deux fois la même déclaration (une fois dans chaque fichier `.c`). Dans un grand projet il peut y avoir des dizaines de milliers de déclarations, il n'est pas question d'en mettre une copie dans chaque fichier source `.c`. On met toutes les déclarations dans des fichiers `.h` et on utilise la directive `#include` pour l'inclure dans les sources `.c`.

Exercice 4 *Mettez les déclarations de votre programme dans un fichier `decl.h`, et utilisez `#include` dans vos fichiers sources `.c`. Assurez vous que tout marche bien.*

1.3 Gestion de projet avec make

Il est très fastidieux de compiler un par un tous les fichiers sources. L'utilitaire `make` permet d'automatiser la compilation ; son fonctionnement est simple : on décrit dans un fichier nommé `Makefile` toutes les dépendances entre fichiers sources et toutes les commandes de compilation. Ensuite, lorsqu'on démarre `make`, il fait tout le nécessaire pour que le programme se retrouve compilé. Voici à quoi ressemble un `Makefile` :

```
CC = gcc
CFLAGS =

prog.exe:      main.o func.o
               ${CC} main.o func.o -o prog.exe

main.o:        main.c decl.h
               ${CC} ${CFLAGS} -c main.c

func.o:        func.c decl.h
               ${CC} ${CFLAGS} -c func.c
```

Les deux premières lignes définissent les variables `CC` et `CFLAGS`. Dans la suite du `Makefile` au lieu d'écrire `gcc` on écrira `#{CC}`. L'utilité en est que lorsqu'on veut remplacer, par exemple, `gcc` par autre chose il suffira de modifier la définition de la variable `CC` au lieu de modifier chaque occurrence de `gcc`. La suite du `Makefile` est composée de blocs de la forme :

```
cible: source1 source2 source3 ...
      commande
```

Ceci indique à `make` que pour construire le fichier "cible" il faut exécuter "command" et que `cible` dépend des fichiers `source1`, `source2` etc. Ainsi, dans l'exemple ci-dessus, le premier bloc indique que pour fabriquer `prog.exe` il y a besoin de `main.o` et de `func.o` et qu'il faut exécuter "`gcc -o prog.exe main.o func.o`", et ainsi de suite. Pour finir, noter qu'avant la commande il y a une tabulation et pas des espaces. Pour un grand projet `make` est indispensable, parce que :

- il ne va compiler que les modules qui ont besoin d'être recompilés, c'est à dire les modules dont un des sources est plus récent que le fichier cible.
- pour compiler on a juste à taper `make`, sans se soucier de quels modules doivent être recompilés.

Exercice 5 *Faites un `Makefile` pour votre programme. Faites de sorte que la compilation se fasse avec les options qui cherchent les prototypes manquants.*

2 Variables statiques et automatiques

Quelle est la différence entre une variable statique et une variable automatique, ou sont-elles stockées ? Peut on avoir :

- une variable locale automatique ?
- une variable locale statique ?
- une variable globale automatique ?
- une variable globale statique ?

Exercice 6 *Faire une fonction `fact` qui retourne la factorielle de son premier argument (rappel : $n! = 1 \times 2 \times \dots \times n$). Faites la de façon récursive, en remarquant que $n! = n \times (n - 1)!$. À l'entrée de la fonction affichez sur la console la valeur et l'adresse de son argument.*

Exercice 7 *Modifiez la fonction pour qu'à chaque appel elle affiche aussi le nombre de fois qu'elle a été appelée. Faites afficher aussi l'adresse de la variable qui mémorise ce nombre.*

3 Transmission des arguments

3.1 Arguments simples

Exercice 8 *Faites un programme contenant une fonction qui prend un entier comme argument :*

```
void func1(unsigned);
```

Vérifiez que les arguments reçus par la fonction sont des copies des variables passées en argument. Est-ce que l'adresse de l'argument reçu peut être la même que celle de la variable passée en argument ?

Exercice 9 Faites un programme contenant une fonction qui prend un pointeur vers un entier.

```
void func2(unsigned *);
```

Vérifiez que lors d'un appel, le pointeur reçu par la fonction contient bien l'adresse de la variable passée par référence.

Exercice 10 Refaites l'exercice précédent, cette fois-ci en utilisant le passage par référence à la façon C++.

```
void func3(unsigned &);
```

Comparer les "trois" façons de transmettre un entier à une fonction. Lesquelles sont équivalentes.

Exercice 11 Parmi les trois façons de transmettre un entier à la fonction, laquelle (lesquelles) choisirez-vous pour :

- incrémenter l'argument
- afficher l'argument

Faites les programmes correspondants

Exercice 12 si `a` et `b` sont du type `unsigned`, quels sont les appels qui marchent, les appels qui compilent mais qui ne marchent pas, les appels qui ne compilent même pas :

- `func1(a);`
- `func2(&a);`
- `func3(a);`
- `func1(5);`
- `func3(5);`
- `func1(a+b);`
- `func3(a+b);`
- `func2(&a + &b);`
- `func2(&a + 5);`

Expliquez. À votre avis, pourquoi certains programmeurs déconseillent le passage par référence à la C++.

3.2 Transmission de tableaux

Quand on passe un tableau en argument à une fonction, c'est uniquement l'adresse du premier élément du tableau qui est transmise. En effet, en langage C/C++ les tableaux et les pointeurs ont un rôle très similaire, on peut presque toujours utiliser un pointeur à la place d'un tableau. Autrement dit le tableau entier n'est pas copié dans la pile, c'est uniquement son adresse qui y est copiée.

Exercice 13 *Il y a-t-il une différence entre les trois fonctions suivantes :*

- void func1(unsigned *tab);
- void func2(unsigned tab[]);
- void func3(unsigned tab[10]);

Faites une fonction qui prend en argument un tableau d'entiers et qui le remplit avec les nombres 0,1,2,...,N, la valeur de N est à spécifier par l'utilisateur. Comment transmet-on la dimension du tableau à la fonction.