TP10 : Pot pourri

Programmation en C (LC4)

Semaine du 1 avril 2008

1 Quelques pièges avec les fichiers

Exercice 1 La fonction `gets` sert à lire une ligne depuis `stdin`, et l'écrit dans un tableau qu'on lui passe en argument. Entrer le code suivant, et l'exécuter en lui entrant une looooooongue ligne :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int x=0;
  char buf[10];
  gets(buf);
  printf("%d\n",x);
  exit(0);
}
```

Que se passe-t-il?

Exercice 2 Exécuter le code suivant :

```
#include <stdio.h>
#include <unistd.h>
int main(int argc,char **argv) {
  fputs("ploum",stdout);
  sleep(20); // La fonction sleep sert a mettre le programme en pause
             // pendant le nombre de secondes passees en argument

}
```

Que se passe-t-il?

2 Le préprocesseur

Avant la compilation proprement dite, le code source d'un `.c` subit d'abord une transformation au niveau du texte lui-même. C'est le texte transformé qui est vraiment compilé.

Exercice 3 Par exemple, entrer le code suivant :

```
#include <stdio.h>

int main(int argc,char **argv) {
  int c;
  do c=fgetc(stdin); while (c!=EOF);
  exit(0);
}
```

mettons dans un fichier `truc.c`, et tapez la commande `gcc -E -o truc truc.c`. Au lieu de compiler, cela met dans le fichier `truc` le code source transformé. Observer le contenu de ce fichier.

3 Compilation séparée

Lorsque l'on écrit un gros programme, il devient vite pénible d'avoir tout dans un même `.c` : il est plus dur de s'y retrouver, et, comme l'on recompile tout à chaque fois, les temps de compilation s'allongent. Heureusement, on peut très bien découper son programme en plusieurs fichiers. En plus de résoudre les problèmes déjà cités, cela permet de réutiliser plus facilement certains bouts de codes dans d'autres programmes : il suffit de récupérer un petit `.c` entier, plutôt que d'aller copier-coller des bouts d'un plus gros fichier.

Cela mène à une discipline de programmation que l'on peut qualifier de modulaire : on regroupe dans un `.c` des fonctions liées (par exemple, si l'on a besoin de listes et d'arbres binaires de recherche, on va faire un `.c` définissant les fonctions sur les listes, et un `.c` définissant les fonctions sur les arbres), et, pour faire un programme complet, on assemble plusieurs `.c` dont certains font référence à d'autres.

L'assemblage consiste à compiler séparément chaque `.c`, avec la commande `gcc -c`. Par exemple, si l'on a un `truc.c`, la commande `gcc -c truc.c` va créer un fichier `truc.o`, qui contient le code des fonctions définies dans `truc.c`, traduit en langage machine, mais sous une forme qui n'est pas directement exécutable. Les fichiers `.o` sont un peu l'équivalent en C des `.class` produits par `javac`. Pour créer un programme exécutable, il faut ensuite rassembler tous les `.o` que l'on a créés à l'étape précédente. Cela se fait en les donnant en argument à `gcc`. Par exemple, `gcc truc.o bidule.o chose.o`. Un **et un seul** des `.o` que l'on assemble doit définir une fonction `main` (le point où l'exécution du programme commence).

Exercice 4 Télécharger le fichier `http://www.tabareau.fr/exemple_decoupe.tar`, puis extraire son contenu avec la commande `tar xf exemple_decoupe.tar`. Compiler les `.c` en des `.o`, puis les assembler.

Pour pouvoir, dans un `.c`, appeller des fonctions ou utiliser des variables globales définies dans un autre `.c`, il faut que le compilateur soit au courant que ces fonctions ou variables sont définies quelque part (sinon, il ne les connait pas puisqu'elles ne sont pas définies dans le fichier qu'il est en train de compiler). On met le compilateur au courant en déclarant les identificateurs définis dans d'autres `.c`, avec leur type. Par exemple, si un autre fichier définie une variable `int x` et une fonction `void f(void)`, il faut les déclarer ainsi avant de s'en servir :

```
extern int x;
void f(void);
```

Si le fichier `.c` externe définit un type que l'on veut pouvoir manipuler à l'extérieur, il faut aussi déclarer ce type. Soit l'on recopie sa définition, soit l'on se contente d'un `struct machin;` (mais pour exporter un `typedef`, il faut exporter sa définition complète, même si la `struct` qui intervient dedans n'est pas déclarée complètement, c'est-à-dire `struct machin; typedef struct machin machin;`), auquel cas on ne pourra manipuler des valeurs de ce type que via des pointeurs, puisque le compilateur ne connait pas leur définition.

Exercice 5 Récupérer le fichier `http://www.tabareau.fr/a_decouper.c`, et le découper en trois `.c` : l'un contenant les fonctions travaillant sur les listes, le deuxième les fonctions sur les arbres de recherche, et le troisième la fonction `main`. Pour tester, ajouter `-Wall` aux options de `gcc`, pour être sur que les fonctions sont bien déclarées.

Recopier l'interface d'un `.c` dans tous ceux qui l'utilisent est fastidieux, et peu robuste (si on modifie l'interface, il ne faut pas oublier d'aller la modifier dans tous les autres fichiers). Heureusement, le `#include` vient à notre aide. Pour chaque `truc.c`, on écrit son interface dans un fichier nommé par convention `truc.h`. Ensuite, dans tout `bidule.c` qui utilise des choses définies dans `truc.c`, il suffit de mettre un `#include "truc.h"`. De la sorte, la version actuelle de l'interface de `truc.c` sera incluse à chaque recompilation de `bidule.c`.

Exercice 6 Modifier le découpage de l'exercice précédent pour utiliser des `.h`.

Recompiler les `.c` qui en ont besoin (i.e. qui ont été modifiés) est aussi une tache fastidieuse. Pour se simplifier la vie, il existe un utilitaire nommé `make` qui automatise une partie du travail. On rentre dans un fichier nommé `Makefile` une description du travail à faire pour batir le programme, et, lorsque l'on appelle `make`, ce dernier va lancer les commandes de compilation qu'il faut (mais il ne recompile que les `.c` qui ont été modifiés depuis leur dernière compilation). Par exemple, si l'on a les fichiers `truc.c` et `chose.c` servant à batir le programme `bidule`, on entre dans le fichier `Makefile` les deux lignes suivantes :

```
bidule: truc.o chose.o
        gcc -o bidule truc.o chose.o
```

Attention, c'est un caractère «tabulation», et non une série d'espaces qui se situe avant le `gcc`. Ce caractère se tape avec la touche juste au-dessus de CapsLock. Cela se lit : pour créer `bidule`, il faut que `truc.o` et `chose.o` aient été créés au préalable, puis il faut exécuter la commande `gcc -o bidule truc.o chose.o`. `make` possède des règles automatiques qui lui permettent de deviner tout seul comment créer `truc.o` et `chose.o` à partir de `truc.c` et `chose.c`, donc les deux lignes ci-dessus suffisent, mais si l'on utilisait un langage que make ne connait pas, il faudrait ajouter des règles spécifiques. Pour créer `bidule`, il faut entrer la commande `make bidule`, et `make` se débrouille alors tout seul.

Exercice 7 Écrire un `Makefile` pour le projet de l'exercice précédent.

En fait, si l'on modifie un `.h`, il faut probablement recompiler tous les `.c` qui l'incluent. `make` n'est pas assez intelligent pour le détecter lui-même, il faut lui expliquer explicitement qu'un `.c` dépend d'un certain nombre de `.h`. Cela se fait en incluant une règle pour chaque `.c`. Par exemple, si `truc.c` inclue `chose.h` et `machin.h`, il faut ajouter la ligne suivant au `Makefile` :

```
truc.o: chose.h machin.h
```

Exercice 8 Adapter de la sorte le `Makefile` de l'exercice précédent.

Cerises sur le gateau On peut définir des variables dans le `Makefile`. Par exemple, au lieu d'écrire

```
bidule: truc.o chose.o
        gcc -o bidule truc.o chose.o
```

on peut écrire :

```
OBJS=truc.o chose.o

bidule: $(OBJS)
        gcc -o bidule $(OBJS)
```

Les `$(OBJS)` sont partout remplacés par `truc.o chose.o`. Cela évite des erreurs, par exemple si l'on ajoute ou enlève un fichier, on ne risque pas d'oublier de le faire à certains endroits.

On peut définir des règles génériques. Par exemple, on pourrait définir ainsi une règle servant à compiler les `.c` (si elle n'existait pas déjà) :

```
.SUFFIXES: .c .o
.c.o:
        gcc -c $<
```

qui explique à `make` que pour fabriquer un `.o` à partir d'un `.c`, il faut passer le nom de ce `.c` en argument à la commande `gcc -c` (c'est le `$<` qui indique l'endroit où le nom du `.c` doit être inséré dans la ligne de commande).