TP 5

Programmation en C (LC4)

Semaine du 26 février 2007

1 Jouons avec les pointeurs

Dans cette section, on convertit des pointeurs en **unsigned int**, pour voir un peu comment cela se passe dans les entrailles de la machine.

Exercice 1 Soit le code suivant :

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    int *t = malloc(4 * sizeof(int));
    printf("%u %u %u %u %u\n", sizeof(int), (unsigned int) t,
            (unsigned int) (t + 1), (unsigned int) &t[1], (unsigned int) (&t[1] - t));
    free(t);
    exit(0);
}
```

Sachant que les entiers sont codés sur 4 octets, que pensez-vous qu'il va afficher ? Faites l'expérience pour vérifier.

Exercice 2 De même avec ce code :

```
#include <stdio.h>
struct ploum { char x; char y; int z; };
struct plam { int x; char y; };
int main(int argc, char *argv[]) {
    struct ploum a;
    printf("%u %u %u %u %u\n", sizeof(char), (unsigned int) &a,
            (unsigned int) &a.x, (unsigned int) &a.y, (unsigned int) &a.z);
    printf("%u\n", sizeof(struct plam));
    exit(0);
}
```

Ici, les résultats sont un peu plus surprenants.

2 Plusieurs fichiers !

Dans cette partie on reprend la partie "Chaînes de caractères" du TD 5. On va séparer les fonctions en plusieurs fichiers pour voir comment compiler le tout. La compilation en plusieurs fichiers séparés présente plusieurs avantages :

- la programmation est modulaire, donc plus compréhensible,
- la séparation en plusieurs fichiers produit des listings plus lisibles,
- la compilation est plus rapide car seule une partie du code est recompilée.

Commencez par récupérer les fichiers `dico.h` et `dico.c` contenant la déclaration et l'initialisation de la variable dico de type **struct** traduction * (il s'agit de la traduction française d'un millier de mots anglais) et de la variable taille_dico de type **int** :

`http://www.di.ens.fr/~pmaurel/LC4/TP5/dico.h`
`http://www.di.ens.fr/~pmaurel/LC4/TP5/dico.c`

Les fonctions du TD seront dans un fichier `traduction.c` et leur déclaration dans `traduction.h`. Enfin un fichier `main.c` contiendra la fonction **int** main(**void**).

Exercice 3 Pour que chaque fichier `.c` compile séparément (avec `gcc -o`), trouvez les inclusions qu'il faut mettre au début de chacun de ces fichiers `.c` (avec `#include`).

Exercice 4 Compilez le tout en utilisant un fichier `Makefile` et la commande `make`.

Rappel : Les fichiers `Makefile` sont structurés grâce aux *règles*. Ce sont elles qui définissent ce qui doit être exécuté ou non. Une *règle* est une suite d'instructions qui seront exécutées pour construire une *cible*, mais uniquement si des *dépendances* sont plus récentes.

La syntaxe d'une règle est la suivante :

cible: dépendances
    commandes

Attention : n'oubliez pas qu'avant chaque ligne de *commandes* il faut absolument une *tabulation*.

3 Pile (Rappel du TD 4)

Une première méthode pour implémenter une pile d'entiers consiste à stocker tous les éléments de la pile dans un simple tableau (le sommet de la pile se trouve dans la dernière case de ce tableau). Lorsqu'on veut ajouter (empiler) un élément, on recopie tous les éléments (et le nouvel élément) dans un tableau plus grand d'une case (de même, lorsqu'on veut enlever (dépiler) le sommet, on recopie tous les éléments sauf le sommet dans un tableau plus petit d'une case). On utilise la structure suivante :

struct pile_simple { **int** taille; **int** *elements; };

Exercice 5 Écrire des fonctions :

void empile_pile_simple(**struct** pile_simple *pile, **int** n)
int depile_pile_simple(**struct** pile_simple *pile)

(On supposera que depile_pile_simple() n'est jamais appelée alors que la pile est vide).

Cette implémentation est simple mais inefficace puisque les éléments de la pile sont recopiés à chaque opération. Il est plus astucieux de garder un tableau partiellement rempli : lorsque la pile déborde, plutôt que d'allouer un tableau plus grand d'une case, on alloue un tableau deux fois plus grand et lorsque l'on dépile un élément, on ne recopie dans un tableau plus petit que si le tableau est aux trois quarts vide, auquel cas on divise sa taille par deux.

On utilise donc la structure suivante :

struct pile_amortie { **int** taille, capacite; **int** *elements; };

où capacite représente le nombre de cases du tableau elements, tandis que taille représente le nombre de cases effectivement remplies. À partir de la taille$^{\text{ème}}$ case, il y a des cases non utilisées.

Exercice 6 Écrire des fonctions :

void empile_pile_amortie(**struct** pile_amortie *pile, **int** n)
int depile_pile_amortie(**struct** pile_amortie *pile)