TD11 – Pointeurs sur fonctions (suite)

Langage C (LC4)

semaine du 12 avril 2010

1 Généralisation du TD10

On rappelle la fonction suivante :

```
int iterations(int (*operateur)(int, int), int z, int n, int * tableau) {
    int i;
    int resultat = z;
    for (i = 0; i < n; ++i) {
        resultat = (*operateur)(resultat, tableau[i]);
    }
    return resultat;
}
```

Question 1. Pourquoi n'est-il pas possible de réécrire la fonction `extremum`, qui calcule un extremum d'un tableau, en utilisant directement `iterations` ?

Question 2. Pour pallier cet inconvénient, nous allons supposer que l'opérateur $\top$ donné en argument de `iterations` prend un argument supplémentaire de type `void *`, et que cet argument supplémentaire est aussi fourni en argument de `iterations` qui se contentera de le passer à l'opérateur $\top$.[1]
Modifier la fonction `iterations` dans ce sens.
Puis, grâce à ces modifications, réécrire la fonction `extremum` en utilisant directement la nouvelle version de `iterations`, avec l'argument supplémentaire bien choisi. Puis appliquer en réécrivant les fonctions `minimum` et `maximum`.
On rappelle que `void *` est un type permettant de manipuler un pointeur vers des données quelconques : pour tout type `T`, une donnée de type `T*` est de type `void *`.

1.1 Comptages

Question 3. Écrire, en utilisant `iterations`, une fonction `pairs` qui permet de compter dans un tableau le nombre d'éléments pairs.

Question 4. Généraliser en écrivant une fonction `compte` qui, en utilisant `iterations`, permet de compter le nombre d'éléments vérifiant un certain prédicat P. On supposera que P est donné sous la forme d'une fonction prenant un argument entier a et renvoyant une valeur non nulle si et seulement si $P(a)$ est vraie.

Question 5. Que faudrait-il faire si on voulait passer par les étapes intermédiaires :

1. Cet argument est appelé *fermeture* (ou *closure* en anglais).

– comptage des éléments divisibles par un entier d donné en paramètre
– comptage des éléments plus petits qu'un entier d au sens d'un ordre $\triangleleft$ donné en paramètre (la divisibilité pouvant être vue comme un cas particulier)

2 Utilisation de la fonction `qsort`

Question 6. Écrire une fonction `int my_int_comparator(const void* a, const void* b)` qui permet de comparer deux entier. Il faut "caster" proprement les entiers !

Question 7. Utiliser cette fonction pour trier un tableau d'entier à l'aide de `qsort` :

```
NAME
  qsort - sorts an array

SYNOPSIS
  #include <stdlib.h>

  void qsort(void *base, size_t nmemb, size_t size,
             int(*compar)(const void *, const void *));

DESCRIPTION
  The qsort() function sorts an array with nmemb elements of size size. The
  base argument points to the start of the array.

  The contents of the array are sorted in ascending order according to a com-
  parison function  pointed to by compar, which is called with two arguments
  that point to the objects being compared.

  The comparison function must return an integer less than, equal to, or
  greater than zero if the first argument is considered to be respectively
  less than, equal to, or greater than the second. If two members compare as
  equal, their order in the sorted array is undefined.

RETURN VALUE
  The qsort() function returns no value.
```

Question 8. Écrire une fonction `int my_char_comparator(const void* a, const void* b)` qui permet de comparer deux `char` et l'utiliser pour trier un tableau de `char`.

Question 9. Nous avons maintenant un tableau `array` de type `unsigned int*`, constitué de très grands entiers qui occupent `n` `unsigned int` à trier. Écrire la fonction `compar` à utiliser, ainsi que l'appel à `qsort`.