TP 11 : pointeurs de fonctions et préprocesseur

Programmation en C (LC4)

Semaine du 23 avril 2007

1 Les pointeurs de fonction

Un pointeur est une variable contenant une adresse mémoire. Les pointeurs peuvent également contenir l'adresse d'une fonction, c'est ce que l'on appelle un pointeur de fonction. Ces pointeurs peuvent ainsi servir à « passer en paramètre une fonction » à une autre fonction pour ensuite l'appeler dans cette autre fonction.

Exercice 1 Utilisation simple des pointeurs de fonction.

- Écrire une fonction fois_deux() qui multiplie par deux un entier.
- Écrire une fonction appliquer_tableau(**int** (*f)(**int**), **int** *t, **int** n) qui applique une fonction f(), au tableau t de taille n.
- Écrire une fonction main() qui teste appliquer_tableau().

Exercice 2 Utilisation des pointeurs de fonction dans un algorithme de tri de tableau.

Voici un algorithme de tri des éléménts d'un tableau dans l'ordre croissant :

```
void tri_croissant(int *t, int n) {
    int i, i_min, j, tmp;
    for (i = 0; i < n - 1; i++) {
        i_min = i;
        for (j = i; j < n; j++)
            if (t[j] < t[i_min])
                i_min = j;
        tmp = t[i_min];
        t[i_min] = t[i];
        t[i] = tmp;
    }
}
```

Voici un algorithme de tri des éléments d'un tableau dans l'ordre décroissant :

```
void tri_decroissant(int *t, int n) {
    int i, i_max, j, tmp;
    for (i = 0; i < n - 1; i++) {
        i_max = i;
        for (j = i; j < n; j++)
            if (t[j] > t[i_max])
                i_max = j;
        tmp = t[i_max];
        t[i_max] = t[i];
        t[i] = tmp;
    }
}
```

Ces deux algorithme sont identiques, à l'exception de l'opérateur de comparaison. Afin d'éviter de dupliquer du code, nous allons utiliser les pointeurs de fonctions.

- Écrire une fonction superieur(**int** a, **int** b) qui renvoie 1 si a est supérieur à b, 0 s'ils sont égaux et -1 sinon.
- Écrire une fonction inferieur (**int** a, **int** b) qui renvoie 1 si a est inférieur à b, 0 s'ils sont égaux et -1 sinon .
- Écrire une fonction tri (), qui tri un tableau t de taille n selon une fonction compare().
- Écrire une fonction main() qui teste tri ().

2 Préprocesseur

Un « préprocesseur » est un programme qui procède à des transformations sur un code source, avant l'étape de compilation ou d'interprétation proprement dite. Dans le langage C, le préprocesseur se contente de procéder à des substitutions de chaînes de caractères en fonction de règles définies par l'utilisateur. On les utilise généralement pour rajouter des macros, inclure d'autres fichiers et pour autoriser la compilation conditionnelle et la compilation séparée.

Jusqu'à présent, vous utilisiez le préprocesseur C le plus fréquemment via les directives :

```
#include "entete.h"
#include <entete.h>
```

dont le rôle est de copier le contenu d'un fichier dans le fichier courant. On l'emploie généralement pour inclure les en-têtes des bibliothèques systèmes, telles que les fonctions mathématiques (<math.h>) ou les fonctions d'entrée/sortie standard (<stdio.h>).

3 Macros

Les macros sont utilisées en C pour définir de petits morceaux de code qui seront réutilisés plusieurs fois à divers endroits du programme. Durant l'exécution du préprocesseur, chaque appel à une macro est remplacé par la définition de cette macro (en copiant textuellement les arguments donnés dans cette définition).

Exercice 3 Écrire des macros :

- qui définit une constante π,
- qui calcule le cube d'un nombre,
- qui calcule la somme de deux nombres,
- qui calculant le volume d'une sphère ($V = \frac{4\pi R^3}{3}$).

Utiliser ces macros dans une fonction main() qui affiche la somme des volumes de n sphères : on passera en argument au programme les $R_i - 1$ où R_i est le rayon de la $i^{\text{ème}}$ sphère.

Vous pourrez utiliser la fonction **double** strtod(**const char** *s, NULL) qui convertit une chaîne de caractère s en **double** : combien de fois cette fonction est-elle appelée à l'exécution de votre programme ?

4 Compilation conditionnelle

Le préprocesseur C permet de faire la compilation conditionnelle, c'est-à-dire de disposer de plusieurs versions d'un même programme ou d'un extrait dans un même fichier source. Typiquement, les programmeurs utilisent cette technique pour compiler différemment le programme en fonction de son état d'avancement, de la plateforme de destination, des vérifications désirées, ou pour s'assurer que les fichiers d'en-tête ne sont inclus qu'une seule fois.

Exercice 4 La compilation conditionnelle permet donc d'orienter la compilation suivant certains critères, ici, nous allons voir comment remédier aux dépendances cycliques entre modules grâce à la compilation conditionnelle.

- Créer cinq fichiers `toto.h`, `toto.c`, `tata.h`, `tata.c`, et `main.c`.
- Dans `toto.h` est déclarée la fonction somme() qui calcule la somme de deux entiers, cette fonction étant définie dans `toto.c`.

- Dans `tata.h` est déclarée la fonction produit() qui calcule le produit de deux entiers, cette fonction étant définie dans `tata.c`.
- On impose que le fichier `tata.h` doit dépendre (c'est-à-dire inclure) du fichier `toto.h` et réciproquement.
- Le fichier `main.c` appelle ces deux fonctions.

Écrire un `Makefile` qui permet de compiler l'ensemble de ces fichiers. Que se passe t-il? Résoudre le problème en utilisant des directives de compilation conditionnelle.

Exercice 5 Le fichier d'en-tête <assert.h> défini la macro assert () qui prend en argument un booléen (un entier) qui indique une condition qui doit être vérifiée à l'exécution du programme, et qui, si cette condition n'est pas vérifiée, met fin à l'exécution du programme et affiche un message d'erreur.

Écrire un programme qui vérifie avec assert () si le nombre d'arguments qui lui ont été passés est supérieur à 3. Compiler ce programme tel quel puis en rajoutant un #**define** NDEBUG en haut du fichier source. Décrire ce que donne l'exécution du programme compilé dans les deux cas et proposer une définition de la macro assert ().

5 Programmation modulaire

La programmation modulaire repose sur l'utilisation de modules, qui sont des briques dans lequelles sont définies des éléments par le programmeur (structures de donnée), ainsi qu'un ensemble de fonctions associées à ces éléments. Cette méthode de regroupement permet de réaliser une encapsulation comparable par certains aspects à celle de la programmation objet, et permet l'organisation du code source en unités de travail logiques. Ce style de programmation facilite grandement la réutilisabilité et le partage du code, et est particulièrement utile pour la réalisation de bibliothèques.

Exercice 6 Le but de cette exercice est de masquer l'implémentation d'un vecteur 3D et d'empêcher la manipulation de vecteur 3D autrement que via les fonctions définies dans l'interface.

- Définir dans un fichier `vecteur.c` une structure vecteur composé de trois **double**, ainsi que les fonctions (qui travailleront sur des pointeurs sur la structure vecteur) :
  - nouveau_vecteur() qui alloue la mémoire nécessaire à un vecteur et initialise ses coefficients,
  - detruit_vecteur() qui libère la mémoire utilisée par un vecteur,
  - affiche_vecteur () qui affiche les coefficients d'un vecteur,
  - multiplie_vecteur() qui renvoie le produit d'un vecteur par un **double**,
  - ajoute_vecteur() qui renvoie la somme de deux vecteurs,
  - soustrait_vecteur() qui renvoie la différence de deux vecteurs,
  - normalise_vecteur() qui renvoie un vecteur unitaire à partir du vecteur donné en argument,
  - vecteur_oppose() qui renvoie l'opposé d'un vecteur,
  - produit_scalaire() qui renvoie le produit scalaire de deux vecteurs
- Définir dans un fichier `vecteur.h` le type pointeur vers une structure vecteur (le type **struct** vecteur est inconnu dans ce fichier, mais il est possible de définir un type pointeur vers un type inconnu). Rajouter les déclarations des fonctions implémentées dans `vecteur.c`.
- Remarquer qu'il est impossible, si l'on inclus uniquement l'en-tête `vecteur.h`, d'accéder aux coefficients du vecteur : l'implémentation de ce type est complémentement opaque pour l'utilisateur.
- Écrire dans un fichier `main.c` une suite d'instructions calculant (et affichant) la réflexion $\vec{r}$ d'un vecteur $\vec{v} = (4, 5, 6)$ par rapport à $\vec{u} = (1, 2, 3)$. *Rappel* : $\vec{r} = 2\left(\vec{v}.\frac{\vec{u}}{\|\vec{u}\|}\right)\frac{\vec{u}}{\|\vec{u}\|} - \vec{v}$
- Écrire un `Makefile` qui permet de compiler l'ensemble de ces fichiers.

Exercice 7 — *(difficulté : ● ● ●)* Le but de cette exercice est de définir deux types vecteurs : l'un utilisant des **float**, l'autre utilisant des **double** sans avoir à réécrire en deux exemplaires toutes les déclarations et définitions de types et de fonctions.

Indications :

- Renommer le fichier `vecteur.c` en `vecteur-impl.c` et remplacer toutes les occurrences de **double** par REEL : le fichier `vecteur-impl.c` sera inclus par vecteur.c qui définira la macro REEL comme **double** ou **float**.
- Le nom du type vecteur sera aussi une macro.
- Renommer de même le fichier `vecteur.h` en `vecteur-decl.h` (et supprimer la directive de compilation conditionnelle pour éviter l'inclusion multiple). Le fichier `vecteur.h` sera le fichier inclus par l'utilisateur tandis que `vecteur-decl.h` contiendra les déclarations des types et fonctions en fonction de macros définies dans `vecteur.h`.
- En langage C, il est interdit de "surcharger" les fonctions : on ne peut définir qu'une seule fonction avec le même nom. Il faut donc, par exemple, rajouter un suffixe aux noms de fonctions d'après le type de vecteur sur lequel elles travaillent (utiliser l'opérateur ## de concaténation dans une macro).