TP 2 : Introduction à C (Partie 2)

13 septembre 2023

1 Rappels

1.1 Variables et Types

— La **déclaration** de variable se fait avec `type variable;` (en remplaçant `type` par le type et `variable` par le nom de la variable)

— L'**assignation** de valeur à une variable après déclaration se fait avec `variable = valeur;` (en remplaçant `variable` par le nom de la variable et `valeur` par la valeur à affecter)

— Les deux peuvent être faits en même temps avec `type variable = valeur;`

— Les types de base que l'on a vu sont `int`, `float` et `char` (et les chaînes de caractères à afficher)

— En C, les variables booléennes sont représentées par 0 (faux) et 1 (vrai). On peut importer `<stdbool.h>` pour faire en sorte que les expressions `true` et `false` (de type `bool`) désignent ces booléens.

1.2 Affichage, Saisie

La fonction pour afficher une chaîne de caractère est la fonction `printf`.

Pour afficher les valeurs de variables entières dans une séquence de caractère à l'intérieur de la fonction `printf`, on utilise la syntaxe suivante :

```
printf("[texte] %d [texte] %d ... ",x1,x2,...);
```

où les valeurs des variables x_i s'afficheront aux positions consécutives des `%d`, dans l'ordre où sont données les variables après la chaîne de caractère : x_1 sur le premier, x_2 sur le deuxième, et ainsi de suite.

Si l'une des variables est un flottant, on utilise `%f` au lieu de `%d` dans la fonction `printf`.

Pour demander à l'utilisateur de saisir la valeur à stocker dans une variable, on utilise la fonction `scanf`. La syntaxe pour demander à l'utilisateur de rentrer la valeur d'une variable entière `x` est la suivante :

```
scanf("%d",&x);
```

De la même manière, on utilise `%f` si l'on souhaite faire saisir la valeur d'un flottant.

1.3 Conditions

La syntaxe des conditions est la suivante :

```
if (condition_1) {
[instructions]
} else if (condition_2) {
[autres instructions]
} else {
[autres instructions]
}
```

Où les conditions s'expriment sont la forme de booléens. Il peut y avoir plusieurs `else if` ou aucun, et il peut ne pas y avoir de `else`.

1.4 Boucles While

La syntaxe des boucles `while` est la suivante :

```
while (condition)
{
    [bloc d'instructions]
}
```

1.5 Boucles For

La syntaxe des boucles `for` est la suivante :

```
for (initialisation ; condition ; incrementation)
{
    [instructions]
}
```

1.6 Fonctions

La syntaxe des fonctions est la suivante :

```
type_R nom_fonction(type_1 arg_1 ,... ,type_N arg_N)
    {
    // [bloc d'instruction]
    return valeur_R;
    }
```

Si l'on déclare une variable à l'intérieur d'une fonction, celle-ci sera locale et ne pourra pas être appelée en-dehors de l'exécution de la fonction. On peut également déclarer des variables globales, à l'extérieur du corps des fonctions.

2 Assertions

On a vu dans le TP précédent que l'on pouvait écrire des programmes pour répondre à un problème donné, défini par une spécification qui est la donnée des entrées admissibles du problème (déterminées par des préconditions) et des sorties qui doivent être obtenues en fonction des entrées.

Les programmes que l'on a vu avaient un défaut : ils pouvaient s'exécuter sur des entrées ne correspondant pas aux préconditions du problème. On a par exemple pu faire des programmes pour calculer le successeur d'un entier dans la suite de Syracuse, où l'entrée pouvait être un entier relatif, alors qu'il doit s'agir normalement d'un entier naturel non-nul.

Pour éviter de pouvoir exécuter le programme sur des entrées qui ne correspondent pas aux préconditions, on peut utiliser les **assertions**. On importe la librairie `assert.h` et on utilise la fonction `assert` pour demander de vérifier, à un moment de l'exécution d'une fonction, qu'une précondition est bien vérifiée. Cette précondition s'exprime sous la forme d'un booléen, et la syntaxe pour tester la précondition est la suivante :

```
assert (precondition);
```

Si une assertion n'est pas vérifiée, le programme termine avec un message d'échec d'assertion.

Exercice 1 :

Reprendre les programmes des exercices 9 et 11 du TP précédent pour y ajouter des assertions correspondant aux préconditions du programme. Tester sur des exemples.

Exercice 2 :

Ecrire un programme qui contient une fonction `syracuse` qui renvoie le successeur d'un entier dans la suite de Syracuse, demande à un utilisateur de rentrer une valeur entière, puis utilise la fonction `syracuse` pour afficher les termes de la suite à partir du nombre donné jusqu'à la première occurrence de 1, ainsi que le nombre de termes affichés (*temps de vol*). On s'assurera que le nombre donné est bien un entier naturel non-nul.

3 Exercices

Exercice 3 :

Ecrire un programme qui demande à l'utilisateur de rentrer la valeur d'un flottant f positif, d'un entier naturel non-nul n et qui calcule la troncature à n décimales de $\sqrt{f}$ (sans utiliser la fonction `sqrt` !)

Exercice 4 :

Ecrire un programme qui demande à l'utilisateur de donner les solutions de 3 calculs successifs affichés dans le terminal (exemple : "3 x 7 = ?") et ne passe au calcul suivant (ou s'arrête à la fin) qu'une fois que la bonne valeur est rentrée.

Exercice 5 :

Ecrire un programme qui demande à l'utilisateur de rentrer la valeur d'un entier naturel n, et affiche la valeur de $n!$.

Exercice 6 :

Ecrire un programme qui demande à l'utilisateur de rentrer la valeur d'un entier naturel n, et affiche la somme des entiers de 1 à n.

Exercice 7 :

Ecrire un programme qui demande à l'utilisateur de rentrer la valeur d'un entier naturel n, et affiche la somme des carrés des entiers de 1 à n.

Exercice 8 :

Ecrire un programme qui demande à l'utilisateur de rentrer la valeur d'un entier naturel n, et affiche la somme des 2^k pour k compris entre 1 à n.

Exercice 9 :

Ecrire un programme qui demande à l'utilisateur de rentrer la valeur d'un entier naturel non-nul n, d'un entier naturel non-nul k, et affiche les k premiers termes de la suite de Syracuse à partir de n.

Exercice 10 :

Ecrire un programme qui demande à l'utilisateur de rentrer la valeur d'un entier naturel non-nul n, d'un seuil s, et affiche les termes de la suite de Syracuse à partir de n jusqu'à ce que l'un d'entre eux soit inférieur à s, ainsi que le nombre de termes affichés.

Exercice 11 :

Ecrire un programme qui demande à l'utilisateur de rentrer la valeur d'un entier naturel non-nul n, d'un seuil s, et affiche les termes de la suite de Syracuse jusqu'à ce que l'un d'entre eux soit inférieur à s.

4 Bonus : Utilisation de l'aléa

Pour utiliser des nombres générés aléatoirement, on peut utiliser la librairie `stdlib.h` ; la fonction `rand()` (sans arguments) renvoie un entier compris entre 0 et une borne supérieure appelée RAND_MAX.

Pour pouvoir bien utiliser l'aléa en ayant des générations de nombres qui soient différentes à chaque exécution, il faut mettre une "graine" (seed), c'est-à-dire un entier qui déterminera la génération de l'aléa dans la suite du programme : si un programme est exécuté deux fois avec une même graine, les nombres générés aléatoirement seront identique. La fonction permettant de rentrer le seed est `srand`.

Pour s'assurer d'avoir des graines différentes (et donc des exécutions avec des résultats différents), on choisit généralement de prendre comme graine la valeur `time(NULL)`, qui donne le nombre de secondes écoulées depuis le 1er janvier 1970. Pour cela, on utilise la librairie `time.h`.

Exercice 12 :

Lancer le programme suivant plusieurs fois :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

```
int main()
{
    srand(time(NULL));
    int x;
    x = rand();
    printf("%d \n",x);
    x = rand();
    printf("%d \n",x);
    x = rand();
    printf("%d \n",x);
    x = rand();
    printf("%d \n",x);
    x = rand();
    printf("%d \n",x);
}
```

Remplacer `time(NULL)` par un entier constant et lancer le programme plusieurs fois. Que se passe-t-il ?

Souvent, lorsque l'on utilise l'aléa, on peut être amené à vouloir générer des valeurs dans un ensemble donné et selon une loi qui ne correspondent pas nécessairement à ce qui est généré par `rand`. Pour cela, on peut avoir recours à des écritures particulières. Par exemple :

- Pour avoir un entier naturel choisi uniformément entre 0 et $k-1$, on peut prendre le reste de la division euclidienne de `rand()` par k.
- Pour avoir un flottant choisi uniformément dans l'intervalle $[0;1]$, on peut utiliser l'expression suivante : `float x = ((float)rand()/(float)(RAND_MAX))`

Exercice 13 :

Ecrire un programme qui demande deux bornes entières a et b et génère et affiche un entier sélectionné uniformément entre a et b.

Exercice 14 :

Ecrire un programme qui demande deux bornes a et b et génère et affiche un flottant sélectionné uniformément dans l'intervalle $[a;b]$.

Exercice 15 (Algorithme de Monte-Carlo) :

Soit une surface S_1 d'aire A_1 finie inconnue à l'intérieur d'une surface S_2 d'aire A_2 finie connue. Pour avoir une estimation en temps borné de A_1/A_2, on peut générer n points uniformément dans la surface S_2 et compter la proportion de points générés à l'intérieur de S_1. Un tel algorithme pour une telle tâche, où le temps de calcul est déterministe mais le résultat est aléatoire, s'appelle un **algorithme de Monte-Carlo**.

Exemple : on considère que S_1 est le disque de centre (0,0) et de rayon 1, et S_2 est le carré dont les abscisses et ordonnées des sommets sont ± 1. Utiliser un algorithme de Monte-Carlo qui demande le nombre de points à générer dans S_2 pour estimer A_1/A_2. *Indice : pour générer un point dans S_2 uniformément, on peut générer son abscisse uniformément dans $[0;1]$ et faire de même pour son ordonnée.*