TP n°3

BOUCLES

I. Prolégomènes

1. À l'aide d'un terminal de commandes, créer un nouveau répertoire nommé **TP3** dans votre répertoire **TP**, et se déplacer à l'intérieur. Ouvrir un nouveau fichier **tp3.c** avec l'IDE VSCodium.

2. Dans **tp3.c**, écrire le squelette de base d'un code C, en tentant de le faire de mémoire sans regarder.

Une deuxième habitude à prendre

Dans ce TP et dans tous les suivants, il vous est demandé de bien compartimenter votre code en différentes fonctions. Cela permet de factoriser votre code et ainsi gagner en lisibilité et en efficacité. Dans la mesure du possible, il faut chercher à éviter une fonction trop longue. Cette remarque inclut également la fonction `main` : essayez de mettre le moins de lignes possible dans cette fonction.

Une troisième habitude à prendre

Dans ce TP et dans tous les suivants, il vous est demandé de régulièrement tester vos fonctions et parsemer ces dernières d'asserts. Cela ne sera plus toujours explicitement exigé, mais vous protégera face aux bugs.

II. Incrémentation d'une variable

Dans la suite de ce TP, nous aurons régulièrement besoin d'incrémenter une variable entière, c'est-à-dire lui ajouter la valeur 1. Bien entendu, il est possible de le faire de la manière suivante `i = i + 1;`. Mais cette opération est tellement courante qu'il existe un raccourci plus idiomatique en C : l'instruction `i++;`. Cette instruction est à l'origine du nom du langage C++, un successeur du C.

3. Vérifier rapidement que cette instruction permet effectivement d'incrémenter une variable entière en C.

III. La boucle for

Schéma général

```c
for (init; condition; update) {
    instr1;
    ...
}
// Suite du programme
```

Exemple

```c
for (int i=0; i<10; i++) {
    int mult = i * 9;
    printf("%d x 9 = %d\n", i, mult);
}
// Suite du programme
```

Une **boucle for** permet de répéter un bloc d'instructions délimité par des accolades. L'en-tête d'une boucle for est constituée du mot-clef `for` puis de de trois éléments entre parenthèses :

- **init**, qui sera exécuté une seule fois avant même de rentrer dans le corps d'instructions de la boucle,
- **condition**, une condition qui, intialement ou après en être sorti, décide si l'on rentre dans le corps de la boucle,
- **update**, qui sera exécuté à chaque fois qu'on atteint la fin du corps de la boucle.

💙 Pour répéter n fois un bloc d'instructions, on peut utiliser l'en-tête `for (int i=0; i<n; i++)`.

4. Prédire et vérifier expérimentalement le nombre exact de tours de boucles que réalisent la fonction d'exemple et les deux fonctions suivantes. Et si l'on retire la condition `1 + 1 == 1` ?

```c
void tourner_dans_le_vide() {
  for (int i=19; i<108; i=i+30) {
      printf("Un tour de boucle\n");
  }
}
```

```c
void tourner_ou_non() {
  for ( ; 1 + 1 == 1 ; ) {
      printf("Un tour de boucle\n");
  }
}
```

5. Implémenter une fonction de prototype `int somme_entiers(int n)` renvoyant la somme des entiers de 1 à n inclus. Tester votre fonction en vérifiant que la somme des entiers de 1 à 100 vaut 5050.

6. Implémenter six fonctions `ligne`, `carré`, `triangle`, `carré_vide`, `triangle_centré` et `zigzag`, toutes de type `void` et prenant chacune un argument `n` de type `int`, qui affichent les formes correspondantes sur n lignes. Les exemples donnés ci-dessous sont pour $n = 7$.

```
#######

#######
#######
#######
#######
#######
#######
#######

#
##
###
####
#####
######
#######

#######
#     #
#     #
#     #
#     #
#     #
#######

   #
  ###
 #####
#######
 #####
  ###
   #

#######
      #
#######
#
#######
      #
#######
```

IV. La boucle while

Schéma général

```c
while (condition) {
    instr1;
    ...
}
// Suite du programme
```

Exemple

```c
int i = 0;
while (i < 100) {
    i = i * 2;
}
// Suite du programme
```

Une boucle while permet de répéter un bloc d'instructions tant que la condition placée en en-tête est vrai. Plus précisément, si la condition est initialement fausse, on ne rentre jamais dans la boucle et on passe directement à la suite du programme. Sinon, on exécute le bloc d'instructions en entier, puis à chaque fois que l'on atteint la fin de ce bloc, on évalue la condition de boucle. Si elle est vraie, on réitère ce processus, mais si elle est fausse on passe à la suite du programme.

En particulier, ce n'est pas parce que la condition est fausse au milieu du bloc d'instructions que l'on sort immédiatement : la vérification ne se fait qu'en amont et à chaque fin de bloc.

7 Réimplémenter la fonction `somme_entiers` à l'aide d'une boucle while au lieu d'une boucle for.

8 Nous souhaitons déterminer expérimentalement la valeur du plus grand entier que l'on peut encoder avec le type `int` en C. Implémenter une fonction qui définit une variable de type `int` à 0, et l'incrémente tant que la nouvelle valeur est plus grande que l'ancienne. Afficher alors la dernière valeur obtenue et comparer à la valeur théorique attendue.

9 Nous souhaitons déterminer la valeur du plus petit flottant strictement positif en C. Implémenter une fonction qui définit une variable de type `float` à 1.0, puis la divise par deux tant que la nouvelle valeur est strictement positive. Afficher alors la dernière valeur non nulle obtenue.

Suite de Syracuse

La suite de Syracuse est d'un entier $N > 0$ est définie par récurrence de la façon suivante :

- $U_0 = N$,
- si U_n est pair, alors $U_{n+1} = \frac{U_n}{2}$,
- si U_n est impair, alors $U_{n+1} = 3 \times U_n + 1$.

Il est conjecturé que cette suite finit toujours par atteindre la valeur 1, peu importe l'entier N choisi initialement.

10 Implémenter une fonction de prototype `void affiche_syracuse(int N)` qui part d'un entier $U_0 = N$ et affiche les termes U_n suivants dans la suite de Syracuse jusqu'à atteindre la valeur 1.

11 Implémenter une fonction de prototype `int syracuse(int N)` qui renvoie le nombre d'étapes nécessaires pour que la suite atteigne la valeur 1 en partant de $U_0 = N$. Quel entier N entre 1 et 100 nécessite le plus d'étapes ?

V. Exercices bonus

12 Implémenter une fonction de prototype `int puissance(int a, int b)` qui calcule et renvoie a^b.

13 Implémenter une fonction de prototype `bool est_premier(int n)` qui décide si un entier strictement positif donné est premier. On pourra utiliser les booléens `true` et `false` dans le `return`.

14 Implémenter une fonction de prototype `int plus_grande_puissance(int n)` qui renvoie la plus grande puissance de 2 qui est plus petite ou égale à `n`.

15 Implémenter une fonction de prototype `int plus_grand_ind_puiss(int n)` qui renvoie le plus grand entier i tel que 2^i soit plus petit ou égale à `n`.

16 Implémenter une fonction de prototype `void affiche_decomposition_bin(int n)` qui affiche la décomposition en binaire d'un entier `n`. Pensez à utiliser les fonctions précédentes.

On s'intéresse à la famille de fonctions $(f_n)_{n\in\mathbb{N}^*}$ suivantes :

```c
int f1(int n) {
  int k = 0;
  for (int i1=0; i1<n; i1++) {
    k++;
  }

  return k;
}
```

```c
int f2(int n) {
  int k = 0;
  for (int i1=0; i1<n; i1++) {
    for (int i2=0; i2<n; i2++) {
      k++;
    }
  }

  return k;
}
```

...

17 Conjecturer ce que renvoient ces fonctions et le vérifier expérimentalement.

La méthode de Héron (ou méthode babylonienne) est une stratégie simple pour estimer la valeur d'une racine carrée d'un réel x. On commence par faire une estimation grossière e de la racine carrée, par exemple $e = \frac{x}{2}$. Puis on obtient une autre borne b en calculant $\frac{x}{e}$ (division flottante). On sait alors que la vraie valeur de la racine carrée est située entre les bornes e et b. Si la précision nous convient, on s'arrête, sinon on réitère le processus avec une nouvelle estimation valant $\frac{e+b}{2}$.

18 Implémenter une fonction de prototype `float heron(float x, float eps)` qui utilise cette stratégie pour calculer la racine carrée de `x` jusqu'à ce que les deux bornes soient plus proches que la tolérance `eps`. En pratique, combien d'étapes sont nécessaires pour `x=108.0` et `eps=0.01` ?