

Correction TP1

MP2I Lycée Pierre de Fermat
guillaume.rousseau@ac-toulouse.fr

Exercice 9

Version du code avec retour à la ligne:

```
1 #include <stdio.h>
2
3 int main(){
4     // Afficher une salutation
5     printf("Hello world\n");
6     return 0;
7 }
```

Exercice 10

Code complété:

```
1 #include <stdio.h>
2
3 int main(){
4     // créer une variable entière x, initialement nulle
5     int x = 0;
6
7     // créer une variable entière y, valant 5 initialement
8     int y = 5;
9     // modifier x en lui assignant x+y
10    x = x + y;
11    // modifier y en lui assignant y+x
12    y = x + y;
13    // répéter les deux dernières étapes une fois de plus
14    x = x + y;
15    y = x + y;
16    // afficher x et y
17    printf("x vaut %d, y vaut %d\n", x, y);
18    return 0;
19 }
```

Une remarque de style: la convention en C est de mettre des espaces autour des signes [=] et des opérateurs tels que [+]. On écrit donc `x = x + y` et pas `x=x+y`.

Exercice 11

On remarque que le programme affiche des 0 en plus à la fin du `6.99`. Par défaut, `printf` affiche les flottants avec un certain nombre de chiffres après la virgule. Il est possible de spécifier le nombre de chiffres que l'on veut afficher, ce qui arrondira le nombre. Pour cela, on utilise le format suivant:

```
1 float prix = 6.99
2 printf("%.4f", x); // affiche 6.9900
3 printf("%.2f", x); // affiche 6.99
4 printf("%.1f", x); // affiche 7.0
```

Exercice 12

On remarque qu'en utilisant le type `int`, la division est tronquée. L'ordinateur a en fait effectué une division entière, c'est à dire euclidienne: $5 = 2 \times 2 + 1$.

On peut aussi se demander ce qui arrive si `x`, `y` sont entières et `z` flottante:

```
1 #include <stdio.h>
2
3 int main(){
4     // version float
5     float x = 5;
6     float y = 2;
7     float z = x / y;
8     printf("Résultat avec float: %f\n", z);
9
10    // version int
11    int x2 = 5;
12    int y2 = 2;
13    int z2 = x2 / y2;
14    printf("Résultat avec int: %d\n", z2);
15
16    // et si z reste float ?
17    int x3 = 5;
18    int y3 = 2;
19    float z3 = x3 / y3;
20    printf("Résultat avec x,y int et z float: %f\n", z3);
21
22 }
```

Exercice 13

```
1 #include <stdio.h>
2
3 int main(){
4     int x = 3;
5     float y = 0.25;
6     float somme_f = x + y;
7     int somme_i = x + y;
8
9     return 0;
10 }
```

On remarque que la variable flottante contient bien 3.25, tandis que la variable entière

contient 3. Si l'on utilise 0.75 au lieu de 0.25, la valeur de `somme_i` reste 3. On peut donc en déduire que lorsque l'on passe d'une valeur flottante à une valeur entière, on **tronque** à la valeur inférieure, sans forcément arrondir.

Exercice 14

Lorsque l'on exécute le code, si l'on rentre deux entiers, tout se passe correctement. Si l'on essaie de rentrer autre chose (un nombre à virgule, des lettres, etc...), le programme semble faire n'importe quoi et affiche des valeurs sans aucun sens, qui peuvent même varier d'une exécution à l'autre. Voici la manière dont les deux `scanf` du programme fonctionnent:

1. Le premier `scanf` lit dans le terminal. Il saute les caractères blancs comme les espaces et les retours à la ligne, et essaie de lire un nombre entier (donc constitué uniquement de chiffres). S'il lit autre chose, il s'arrête immédiatement, **sans retirer le caractère qui l'a fait s'arrêter**.
2. Le deuxième `scanf` fait la même chose. Si le premier `scanf` a été arrêté par un caractère non reconnu, le deuxième `scanf` bloque également dessus, et ne fait rien.

C'est ce qui explique que si l'on rentre une seule lettre, comme 'a', et que l'on appuie sur entrée, le programme semble "sauter" le deuxième `scanf`: en réalité, les deux sont bien exécutés mais se font bloquer par la lettre.

Néanmoins, le programme affiche tout de même des valeurs pour les deux variables, et la somme est cohérente. En C, lorsque l'on lance un programme, toutes les variables ont une valeur initiale aléatoire, qui dépend de ce qui se trouvait dans la mémoire au moment du lancement (et donc ce qui a été potentiellement laissé par un autre programme). Lorsqu'un appel à `scanf` réussit, la variable utilisée est modifiée, mais en cas d'échec, la valeur initiale n'est pas remplacée. On peut le vérifier en rajoutant un affichage avant les scans:

```
1  #include <stdio.h>
2
3  int main (){
4      int x = 45; // valeur initiale définie
5      int y;      // valeur initiale non-définie
6      printf("Valeurs initiales: x=%d y=%d\n", x, y);
7
8      printf (" Entrez x: ");
9      scanf ("%d", &x);
10     printf (" Entrez y: ");
11     scanf ("%d", &y);
12
13     printf ("x vaut %d, y vaut %d, la somme vaut %d\n", x, y, x+y);
14     return 0;
15 }
```

Exercice 17

Pour le second programme, on peut remarquer qu'en rentrant -0.33 , le programme affiche A au lieu de B, comme si la condition $-0.33 < -0.33$ était évaluée comme vraie. Ce comportement étrange peut être expliqué par la manière dont les nombres flottants sont représentés sur un ordinateur.

Il existe en réalité deux types distincts pour les nombres flottants: `float` et `double`. On verra au chapitre 2 la manière précise dont ces types fonctionnent, mais l'idée est qu'ils utilisent un système assez proche de l'écriture scientifique, et le type `double` est deux fois plus précis, donc a deux fois plus de "chiffres après la décimale".

Le nombre -0.33 n'est pas représentable de manière exacte par les flottants (à nouveau, on verra pourquoi au chapitre 2). Donc, si l'on écrit:

```
1 float a = -0.33;
```

ou bien:

```
1 double b = -0.33;
```

on se retrouve en fait avec des approximations de -0.33 . De plus, en C, lorsque l'on écrit un nombre directement dans le code source, celui-ci est traité comme un `double` par défaut. Donc, si l'on écrit:

```
1 float x = -0.33;
2 if (x < -0.33){
3     ...
4 }
```

on compare un `float` approximant -0.33 avec un `double` approximant -0.33 , ce qui explique le résultat observé dans le programme ! Pour résoudre le problème, on peut forcer le -0.33 du code à être un `float` en écrivant `-0.33f`: le "f" final indique que c'est un `float`:

```
1 #include <stdio.h>
2
3 int main(){
4     float x;
5     printf("Rentrez un nombre: ");
6     scanf("%f", &x);
7
8     if (x < -0.33f){
9         printf("Cas A\n");
10    } else if (x <= 7.89f) {
11        printf("Cas B\n");
12    } else {
13        printf("Cas C\n");
14    }
15
16    return 0;
17 }
```

Ceci dit, il est toujours possible de faire bugger le programme: si l'on rentre -0.3300000001 , l'arrondi fait que le nombre est traité comme -0.33 , et l'on passe dans le cas B !

Exercice 18

Programme médiane:

```
1 #include <stdio.h>
2
3 int main(){
4     int x, y, z;
5     printf("Rentrez les trois nombres: ");
6     scanf("%d %d %d", &x, &y, &z);
7
8     printf("La médiane de %d, %d et %d est ", x, y, z);
9     if ((y <= x && x <= z) || (z <= x && x <= y)){
10         printf("%d\n", x);
11     } else if ((x <= y && y <= z) || (z <= y && y <= x)){
12         printf("%d\n", y);
13     } else {
14         printf("%d\n", z);
15     }
16
17     return 0;
18 }
```

Il faut bien penser à tester les programmes ! Si vous avez utilisé des inégalités stricts dans le programme de la médiane, il y a de fortes chances pour que votre programme ne fonctionne pas correctement lorsque l'on rentre trois nombres dont deux sont égaux.

Pour le programme “gabu”, il faut éviter de simplement faire un if-else avec 4 cas afin de factoriser un peu le code. L'idée est d'afficher “ga” si le nombre est multiple de 3, puis, indépendamment, d'afficher bu si le nombre est multiple de 5. Enfin, on rajoute une condition pour traiter le cas particulier: les nombres multiples ni de 3 ni de 5:

```
1 #include <stdio.h>
2
3 int main(){
4     int x;
5     printf("Rentrez un nombre: ");
6     scanf("%d", &x);
7
8     if (x % 3 == 0){
9         printf("ga");
10    }
11    if (x % 5 == 0){
12        printf("bu");
13    }
14    if (x % 3 != 0 && x % 5 != 0){
15        printf("%d", x);
16    }
17    // retour à la ligne dans tous les cas
18    printf("\n");
19
20
21    return 0;
22 }
```

Afin de gérer les retours à la ligne correctement dans les 4 cas de figure, on n'affiche le retour à la ligne qu'à la toute fin, une fois que l'on a fini le reste.

Exercice 19

Une fois que l'on a identifié la logique de l'exercice précédent, il suffit de rajouter les conditions nécessaires pour les deux syllabes en plus:

```
1  #include <stdio.h>
2
3  int main(){
4      int x;
5      printf("Rentrez un nombre: ");
6      scanf("%d", &x);
7
8      if (x % 3 == 0){
9          printf("ga");
10     }
11     if (x % 5 == 0){
12         printf("bu");
13     }
14     if (x % 7 == 0){
15         printf("zo");
16     }
17     if (x % 11 == 0){
18         printf("meu");
19     }
20     if (x % 3 != 0 && x % 5 != 0 && x % 7 != 0 && x % 11 != 0){
21         printf("%d", x);
22     }
23     // retour à la ligne dans tous les cas
24     printf("\n");
25
26
27     return 0;
28 }
```

Exercice 20

- Q1.** Le type `int` va jusqu'à $2^{32}-1$, soit environ 2.15×10^9 . Un calcul rapide montre que 2.15×10^9 secondes équivaut à environ 68 ans. Autrement dit, ce type permettrait de stocker la date jusqu'en $1970 + 68 = 2038$.
- Q2.** En faisant le même calcul, on trouve que sur le type `long int`, le dépassement se passera dans plus de 292 milliards d'années, ce qui devrait suffire.
- Q3.** Programme pour calculer le mois et l'année:

```
1  #include <stdio.h>
2  #include <time.h>
3
4  int main(){
5      long int t = time(NULL);
6
7      // nombre d'années écoulées depuis 01/01/1970
8      int nb_annees = t / (3600 * 24 * 365.24);
9
10     int annee = nb_annees + 1970;
11 }
```

```

12     int secondes_par_mois = 3600 * 24 * 30.44;
13     // nombre de mois écoulés depuis le 01/01/1970
14     int nb_mois = t / (secondes_par_mois);
15
16     // mois actuel: reste modulo 12
17     int mois = (nb_mois % 12) + 1; // janvier = 01 et pas 00
18
19     printf("Mois: %d, année: %d \n", mois, annee);
20 }

```

Exercice 21

On remarque qu'en lançant le programme avec quelques secondes d'intervalles, les nombres aléatoires générés sont différents d'une exécution à l'autre. En revanche, quand on lance le programme plusieurs fois en une seconde, la graine reste la même, et les nombres générés aussi. Notons qu'au sein d'une exécution, les valeurs de $\bar{x}$ et $\bar{y}$ sont générées totalement aléatoirement et indépendamment.

En enlevant l'instruction `srand(time(NULL));`, les nombres générés sont toujours les mêmes d'une exécution à l'autre: ceci est dû au fait que la seed est toujours la même (probablement 0).

Pour tirer un nombre entre 0 et 99 inclus:

```

1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4
5  int main(){
6      srand(time(NULL));
7
8      int a = rand() % 100;
9
10     printf("%d\n", a);
11 }

```

Pour tirer un nombre entre 10 et 20, on peut remarquer qu'il y a 11 valeurs possibles. On commence donc par tirer un nombre entre 0 et 10 (inclus), il ne reste plus qu'à décaler le résultat de 10:

```

1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4
5  int main(){
6      srand(time(NULL));
7
8      int a = 10 + (rand() % 11);
9
10     printf("%d\n", a);
11 }

```

A retenir: pour tirer un nombre aléatoire parmi k valeurs, on peut commencer par tirer entre 0 et $k - 1$, ce qui se fait en prenant `rand() % k`. Pour tirer un nombre entre a et b , la formule générale est donc `a + rand() % (b-a+1)`.