

TP2: Fonctions et boucles

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

Seuls certains exercices sont à rendre, ils sont explicitement indiqués au fil du TP. Les autres exercices servent à introduire des notions, ou bien à les approfondir pour les élèves souhaitant aller plus loin.

A Le type bool

On étudie de plus près les conditions des *if-else* et des boucles.

Exercice 1

Recopiez le programme suivant, et exécutez-le pour observer son comportement:

```
1 #include <stdio.h>
2
3 int main(){
4     if (0){
5         printf("Oui\n");
6     } else {
7         printf("Non\n");
8     }
9     return 0;
10 }
```

Modifiez ensuite le programme en remplaçant le 0 de la condition par un 1, un 5, un -85.2, etc... Que semble être la règle du comportement du if-else ?

En réalité, les conditions des if-then et des for sont des valeurs au même titre que les entiers et les flottants:

```
1 int main(){
2     int x = 0;
3     int y = 2;
4     int b1 = (x==y) || (x > y*5) ; // vaut 0
5     int b2 = (x < y); // vaut 1
6     printf("%d %d\n", b1, b2);
7 }
```

Plus précisément, en C, la valeur 0 représente le faux, et les autres valeurs représentent le vrai. Cependant, on privilégie l'utilisation de la valeur 1 pour représenter le vrai.

Donner le type `int` aux valeurs de vérité est un peu contre-intuitif. On introduit donc un nouveau type: le type `bool`, qui représente les booléens. Ce terme vient du nom de Georges Boole, un mathématicien et logicien anglais. Les **booléens** sont au nombre de 2: `true` et `false`. Les opérateurs logiques `&&` (ET), `||` (OU INCLUSIF) et `!` (NON) sont trois opérations

sur les booléens. On parle **d'algèbre de Boole** (on en reparlera au cours de l'année quand vous aurez fait un peu plus de maths).

En C, `true` vaut 1 et `false` vaut 0. En fait, le type `bool` est simplement un autre nom pour désigner les entiers 8 bits, mais on l'utilise afin de signaler que l'on manipule des valeurs de vérités: c'est un type qui sert à rendre le code **lisible**.

Pour utiliser le type `bool`, et les valeurs `true` et `false`, il faut utiliser une nouvelle librairie: ***stdbool.h***.

```
1 #include <stdbool.h>
2 #include <stdio.h>
3 int main(){
4     bool b1 = true;
5     bool b2 = b1 && (2<1); // true ET false: b2 vaut false
6     printf("%d %d\n", b1, b2); // affiche "1 0"
7 }
```

Exercice 2: (à rendre)

Q1. Écrivez un programme créant deux variables booléennes `a` et `b` valant `true` et `false` respectivement, puis calculant et affichant $a, b, a \& b, a || b$ et $!a$). Il n'y a pas de format spécifique aux booléens, vous pouvez utiliser `%d`.

Q2. Complétez les tables des valeurs de $\&\&$ et $||$:

x	y	$x \& y$
0	0	?
0	1	?
1	0	?
1	1	?

x	y	$x y$
0	0	?
0	1	?
1	0	?
1	1	?

B Fonctions

Lorsque l'on définit une fonction autre que `main`, on doit systématiquement écrire en commentaire ce qu'elle fait. Ce commentaire, dit de **documentation**, sert à expliquer aux personnes qui lisent le code à quoi sert la fonction, comment l'utiliser, et quelles restrictions éventuelles s'appliquent aux arguments. Par exemple:

```
1  #include <stdio.h>
2  /* Ici on met une explication globale de ce que fait le programme */
3
4  /* Affiche x ne fois, sur n lignes distinctes.
5     n doit être positif ou nul */
6  void multi_affiche(float x, int n){
7      for (int i = 0; i < n; i++){
8          printf("%d\n", x);
9      }
10 }
11
12 /* Calcule et renvoie (a puissance n) modulo b. n doit être positif
13    ou nul, b doit être supérieur ou égal à 2. */
14 int puissance_mod(int a, int n, int b){
15     int result = 1;
16     for (int i = 0; i < n; i++){
17         result = (result * a) % b;
18     }
19     return result;
20 }
21
22 int main(){
23     int x = puissance_mod(17, 6, 13);
24     multi_affiche(0.15, x);
25     return 0;
26 }
```

Bon commentaire Le but d'un commentaire de documentation n'est pas d'expliquer les mécanismes internes de la fonction, mais plutôt d'expliquer comment l'utiliser. Un bon commentaire de documentation permet à une personne qui le lit d'utiliser la fonction sans aller en lire le code source, de la même manière que vous utilisez `printf` sans savoir comment elle est implémentée. Ainsi, sans même aller lire le corps des fonctions `multi_affiche` et `puissance_mod`, vous savez précisément à quoi elles servent et **comment les utiliser**.

Voici des règles **absolues** concernant les commentaires de fonction:

- Décrire la valeur renvoyée et/ou les actions effectuées, en fonction des entrées;
- Ne pas confondre “afficher” et “renvoyer”;
- Systématiquement faire apparaître **le nom de chaque paramètre**;
- Ne pas écrire “Ceci est une fonction qui prend en paramètre X et fait Y.”, écrire directement “Fait Y.”

À partir de maintenant, lorsque vous écrivez une fonction, vous devez la commenter en suivant ces règles.

Exercice 3: (à rendre)

Créez un fichier `mes_fonctions.c`, définissez les fonctions suivantes, et écrivez des commentaires appropriés. **Attention**, vous ne pouvez pas recopier l'énoncé des questions car celui-ci ne répond pas aux critères précédents !

1. Une fonction qui prend en entrée deux entiers et détermine si le premier divise le deuxième, en renvoyant un booléen.
2. Une fonction qui, étant donné un flottant non nul, l'affiche puis affiche son inverse.
3. Une fonction qui étant donné x, y, z, t , calcule $3x + 5y - 6.25z + t$, l'affiche, et renvoie son carré.
4. La fonction `main`, qui devra tester les fonctions précédentes sur plusieurs valeurs. Pas besoin de la commenter.

C Assertions

Il arrive d'écrire des fonctions qui ne peuvent s'exécuter que sur certaines entrées. Par exemple, une fonction calculant la racine d'un nombre, ou bien une fonction calculant la factorielle d'un entier, ne sont pas sensées être appelées sur des entiers négatifs.

Définition 1. Une **précondition** est une hypothèse devant être vérifiée pour qu'une fonction s'exécute sans erreur.

Voyons une des manières dont on peut tester les préconditions d'une fonction.

Exercice 4

Recopiez et compilez le code suivant:

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  /* Calcule et renvoie la factorielle de n un entier positif ou nul. */
5  int factorielle(int n){
6      assert(n>=0);
7      if (n == 0){
8          return 1;
9      } else {
10         return n * factorielle(n-1);
11     }
12 }
13
14 int main(){
15     int n;
16     scanf("%d", &n);
17     printf("%d\n", factorielle(n));
18     return 0;
19 }
```

Exécutez le programme avec comme entrée $n = 5$, puis $n = -3$. Que se passe-t'il ?

La fonction `assert`, de la librairie `<assert.h>`, permet donc de déterminer si une condition est remplie ou non, et à arrêter l'exécution et produire un message d'erreur si elle ne l'est pas. La vérification de préconditions est une utilisation classique d'`assert`, mais on peut aussi l'utiliser aussi pour valider des tests.

Dans la suite de ce TP, vous devez diviser vos programmes en fonctions, de façon à bien exhiber l'organisation logique derrière le programme. Il faudra systématiquement commenter les fonctions et leur ajouter des assertions testant les préconditions comme expliqué ci-dessus.

Exercice 5: (à rendre)

Reprenez votre code de l'exercice 3 (pas la peine de créer un nouveau fichier, modifiez directement le code dans le dossier de l'exercice 3). Rajoutez à chaque fonction des assertions, si besoin est.

Programmes sans fin

Nous allons maintenant nous intéresser aux boucles et aux fonctions récursives, deux outils puissants mais qui peuvent nous amener à écrire des programmes qui ne terminent pas si l'on ne fait pas attention. Lorsque vous lancez un programme dans le terminal, vous pouvez l'arrêter de force avec Ctrl+C. Il faudra parfois le faire plusieurs fois pour que le programme réponde.

D Boucles for

Dans cette partie, vous devez utiliser des boucles for.

Exercice 6: (à rendre)

- Q1.** Écrivez un programme demandant de rentrer un entier n et affichant successivement tous les entiers entre 1 et n .
- Q2.** Écrivez un programme qui affiche un escalier: Il lit dans le terminal un entier n puis affiche n lignes. La i -ème ligne (à partir de $i = 0$) sera composée de $2i + 1$ fois le caractère "-", suivi d'un unique caractère "|". Le résultat doit ressembler à:

```
-|
---|
-----|
-----|
-----|
```

Vous devrez définir une fonction `void affiche_ligne(int k)` qui gère l'affichage de la k -ème ligne et une fonction `void affiche_escalier(int n)` qui gère l'affichage d'un escalier de n lignes.

E Boucles while

Dans cette partie, vous devez utiliser des boucles while.

Exercice 7: (à rendre)

Écrivez un programme qui demande à l'utilisateur de rentrer des nombres, jusqu'à ce que l'utilisateur rentre un nombre strictement négatif, et qui affiche alors la somme de tous les nombres positifs rentrés. *Indication: utilisez une variable pour stocker le dernier nombre rentré, et une variable pour stocker la somme.*

Exercice 8: (à rendre)

Le but de cet exercice est d'écrire un petit jeu de devinette, un programme qui génère un nombre aléatoire entre 0 et 4999, et qui tente de le faire deviner à l'utilisateur, de la manière suivante:

- L'utilisateur rentre un nombre;
- Le programme lui dit "Plus haut" ou "Plus bas" si le nombre n'est pas bon;
- Lorsque l'utilisateur a trouvé le bon nombre, le programme affiche "Gagné" et s'arrête.

On rappelle que l'utilisation d'aléatoire se fait via les fonctions `srand` et `rand` de la librairie `stdlib.h`, et que l'initialisation de la graine avec `srand` se fait généralement avec la fonction `time` de la librairie `time.h`.

- Q1.** Écrire une fonction `bool verifier(int cible, int essai)` qui prend en entrée deux entiers `cible` et `essai`, qui représentent respectivement le nombre cible et un essai de l'utilisateur. Cette fonction doit afficher le résultat de l'essai (plus haut, plus bas, égal) et renvoyer un booléen indiquant si l'essai est correct.
- Q2.** En utilisant la fonction précédente, implémenter le jeu de devinette.
- Q3.** Connaissez-vous une stratégie efficace pour ce jeu ?

F Fonctions récursives

Dans cette partie, vous n'avez pas le droit d'utiliser les boucles !

Une fonction est dite **récursive** si elle s'appelle elle-même dans son propre code. C'est le cas de la fonction `factorielle` donnée dans l'exercice 4: elle utilise le fait que pour $n \in \mathbb{N}^*$, $n! = n \times (n-1)!$, et que $0! = 1$. Comme les boucles, la récursivité permet de répéter des opérations (potentiellement à l'infini). Pour écrire une fonction récursive, il faut se demander:

- dans quel cas l'opération à effectuer est triviale;
- dans les autres cas, comment se ramener à un cas plus petit.

Par exemple, si l'on veut créer une fonction qui prend en entrée $n \in \mathbb{N}^*$ et affiche un décompte depuis n jusqu'à 1:

- si $n = 1$ c'est trivial, on affiche 1
- sinon, on commence par afficher n , puis il reste à afficher un décompte depuis $n-1$

D'où le code:

```
1 void affiche_decompte(int n){
2     if (n == 1){
3         printf("1\n");
4     } else {
5         printf("%d\n", n);
6         affiche_decompte(n-1);
7     }
8 }
```

Exercice 9

Pour cette exercice, créez un unique fichier C, implémentez les fonctions suivantes et écrivez dans le main du code permettant de tester ces fonctions **sur plusieurs entrées**.

- Q1.** Écrivez une fonction prenant en entrée un entier n et affichant les n premiers entiers dans l'ordre croissant. (*Indication: quelle est la dernière valeur à afficher ?*)
- Q2.** Écrivez une fonction prenant en entrée un entier et affichant chaque chiffre de cet entier sur une ligne différente. Si l'entier est négatif, le signe '-' doit apparaître au début sur une ligne à part. Par exemple, sur l'entrée -894 , votre programme doit afficher:

```
1 -
2 8
3 9
4 4
```

(*Indication: utilisez des divisions modulo 10.*)

- Q3.** Écrivez une fonction qui, étant donné un entier n , affiche n fois "O" sur la même ligne, puis revient à la ligne.
- Q4.** Écrivez une fonction qui, étant donné un entier n , affiche pour chacun de ses chiffres le nombre correspondant de "O" sur une ligne. Si l'entier est négatif, le signe '-' doit apparaître au début sur une ligne à part. Par exemple pour -894 , le

programme doit afficher:

```
1 -
2 00000000
3 00000000
4 0000
```

Tout comme les variables, les fonctions en C peuvent être déclarées et définies à deux endroits différents du code. Par exemple:

```
1 #include <stdio.h>
2 #include <assert.h>
3
4 // renvoie le double de n
5 int doubler(int n);
6
7 // renvoie le quadruple de n
8 int quadrupler(int n){
9     return doubler(doubler(n));
10 }
11
12 int doubler(int n){
13     return 2*n
14 }
15
16 int main(){
17     int a = quadrupler(5);
18     assert(a == 20); // on peut aussi utiliser assert pour tester les fonctions !
19     printf("%d\n", a);
20     return 0;
21 }
```

Dans ce code, on a **déclaré** la fonction `doubler`, puis on l'a utilisé pour **définir** la fonction `quadrupler`, et enfin on a **défini** la fonction `doubler`. Notez que l'on doit préciser le type de retour à la déclaration ET à la définition.

Déclarer une fonction sans la définir sert à signaler au compilateur que la fonction existe et que l'on va la définir à un moment. Si vous regardez le code source de gros logiciels écrits en C (VLC, OBS Studio par exemple), vous pourrez voir que les fichiers sources sont séparés en deux familles: les fichiers `.h`, où l'on déclare les fonctions, et les fichiers `.c`, où l'on définit les fonctions. Plus tard dans l'année, vous apprendrez à faire des projets avec plusieurs fichiers.

Exercice 10: (à rendre)

Écrivez un programme contenant deux fonctions, `ping` et `pong`, toutes deux de signature `void` → `void`, telles que:

- `ping()` affiche "Ping", attend une seconde, et appelle `pong()`
- `pong()` affiche "Pong", attend une seconde et appelle `ping()`

Vous aurez besoin de la fonction `sleep`, dans la librairie `<unistd.h>`. Cette fonction prend en argument un entier n , et met en pause le programme pendant n secondes.

G Suite de syracuse

Soit $x \in \mathbb{N}$. La suite de Syracuse de x est une suite $(u_n)_{n \in \mathbb{N}}$ à valeurs dans $\mathbb{N}$ définie par:

$$\begin{aligned} u_0 &= x \\ u_{n+1} &= \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{si } u_n \text{ est impair.} \end{cases} \end{aligned}$$

Voici la suite de Syracuse pour quelques valeurs de x :

x	u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7
1	1	4	2	1	4	2	1	4
5	5	16	8	4	2	1	4	2
6	6	3	10	5	16	8	4	2
2	2	1	4	2	1	4	2	1

On remarque que pour les 4 valeurs données, la suite de Syracuse devient cyclique à 1-4-2-1... éventuellement. Cette suite fait l'objet d'une conjecture très célèbre: la **Conjecture de Syracuse**:

Conjecture 1. Pour tout $x \in \mathbb{N}^*$, en notant $(u_n)_{n \in \mathbb{N}}$ la suite de Syracuse de x , il existe $k \in \mathbb{N}^*$ tel que $u_k = 1$.

Autrement dit, cette conjecture affirme que la suite de Syracuse de tout entier $x \in \mathbb{N}^*$ devient cyclique à 4-2-1 éventuellement. Ce problème est toujours ouvert aujourd'hui¹.

Définition 2. Étant donné $x \in \mathbb{N}^*$, et $(u_n)_{n \in \mathbb{N}}$ sa suite de Syracuse, le **temps de vol** de x est le plus petit $k \in \mathbb{N}$ tel que $u_k = 1$. Si la suite n'atteint jamais 1, le temps de vol de x est $+\infty$.

Exercice 11: (à rendre)

Pour cet exercice, créez un unique fichier C, dans lequel vous écrirez plusieurs fonctions. Attention, certains calculs vont sortir des entiers 32 bits, vous devez donc utiliser le type `long int`, qui tient sur 64 bits. Le symbole pour afficher des `long int` avec printf est `%ld`:

```
1 long int x = 4295000000;
2 printf("%ld\n", x);
```

- Q1.** Créez une fonction `long int suivant(long int x)` qui, étant donné $x \in \mathbb{N}^*$, calcule le terme suivant dans la suite de Syracuse.
- Q2.** Créez une fonction `long int syracuse(long int x, int n)` qui, étant donnés $x \in \mathbb{N}^*$ et $n \in \mathbb{N}$, calcule le n -ème de la suite de Syracuse. Utilisez la fonction précédente pour simplifier votre code. Combien vaut le n -ième terme de la suite de Syracuse de x dans les cas suivants:

1. $x = 9, n = 6$

3. $x = 1023, n = 729$

2. $x = 77, n = 128$

4. $x = 1234567, n = 52397$

¹Si vous trouvez une preuve ou un contre-exemple, n'hésitez pas à m'en faire part.

Q3. Créez une fonction `long int temps_de_vol(long int x)` qui, étant donné $x \in \mathbb{N}^*$, calcule son temps de vol. Quel est le temps de vol de x dans les cas suivants:

- | | |
|-------------|----------------|
| 1. $x = 1$ | 4. $x = 28$ |
| 2. $x = 26$ | 5. $x = 77030$ |
| 3. $x = 27$ | 6. $x = 77031$ |

Q4. Écrivez une fonction `long int plus_long_vol(long int N)` qui, étant donné un entier N , renvoie l'entier $x \in \llbracket 1, N \rrbracket$ ayant le plus long temps de vol. Donnez le résultat de cette fonction, ainsi que le temps de vol correspondant, pour:

- | | |
|----------------|-------------------|
| 1. $N = 10$ | 5. $N = 100000$ |
| 2. $N = 100$ | 6. $N = 1000000$ |
| 3. $N = 1000$ | 7. $N = 10000000$ |
| 4. $N = 10000$ | |

Q5. Combien de temps prend votre programme environ pour la dernière valeur ? Quelles améliorations pouvez-vous proposer pour le rendre plus efficace ?

Q6. (Bonus) Implémentez une amélioration proposée.

Exercice 12: (à rendre)

- Q1.** Écrire une fonction `rand_int(int a, int b)` prenant en entrée deux entiers $a \leq b$ et renvoyant un entier aléatoire uniforme de l'intervalle $\llbracket a, b \rrbracket$.
- Q2.** Dans le main, à l'aide d'une boucle, générer 10 fois de suite un nombre entre 0 et 99. Ajouter l'instruction `srand(time(NULL));` d'initialisation au début du main, et vérifiez que votre programme est correct.
- Q3.** Déplacer le `srand` pour le mettre au début de la fonction `rand_int`, et tester à nouveau. Expliquer le résultat observé.

H Exercice Libre

Exercice 13: (à rendre)

Imaginez un programme qui utilise des fonctions, des boucles `for` et `while`. Pensez bien à planifier votre programme, à réfléchir en amont aux fonctions dont vous aurez besoin. Commentez ce programme (et **toutes** les fonctions !) pour expliquer succinctement ce qu'il fait. Si vous n'avez pas d'idées:

- Un programme qui affiche des lignes de “*” de longueurs qui varient avec le temps (sinusoidal, linéaire, autre), créant une sorte de vague dans le terminal
- Un programme qui teste si un nombre est premier
- Un programme qui calcule les n premiers termes de la suite de Fibonacci
- Un programme qui calcule la moyenne, le min et le max des nombres rentrés par l'utilisateur, au fur et à mesure qu'il les rentre.

Si vous souhaitez calculer des racines, des cosinus, ou d'autres opérations mathématiques classiques, vous aurez besoin de la librairie `<math.h>`. Pour l'utiliser, il faut non seulement écrire `#include <math.h>` au début de votre programme C, mais aussi rajouter l'option `-lm` (comme **L**ink **M**ath) à la fin de la commande de compilation. Vous pouvez trouver en ligne la liste des fonctions existantes dans cette librairie.

Pensez à bien tester vos programmes avec plusieurs valeurs, dont des valeurs limites / très grandes, afin de vérifier le bon fonctionnement de votre code.