Pré-TP2: Introduction aux boucles et fonction

MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

Consignes

Ce document sert de courte introduction aux fonctions et aux boucles, pour les élèves ayant terminé le TP1 et souhaitant travailler le TP2 en avance.

A Fonctions

Le terme mathématique de "fonction" désigne une boîte noire transformant une entrée en sortie de manière systématique. Par exemple, la fonction $f : x \mapsto x^2$ est telle que $f(2) = 4$, $f(10) = 100$, et ainsi de suite. Cependant, on ne sait pas en lisant la définition de la fonction **comment** la calculer en pratique. Ainsi, la fonction

$$g : x \mapsto \begin{cases} 1 & \text{si l'hypothèse de Riemann est vraie} \\ 0 & \text{sinon} \end{cases}$$

est bien définie, mais personne ne sait la calculer (pour l'instant).

En informatique, une **fonction** est une suite d'opérations à exécuter sur des variables d'entrées afin de fabriquer un résultat. En C, lorsque l'on crée une fonction, on doit préciser le **type** de ses entrées ainsi que celui de la valeur de retour. Par exemple :

```c
// Renvoie |x|
int valeur_absolue(int x){
  if (x < 0){
    return -x;
  else {
    return x;
  }
}
```

Cette fonction prend en entrée un entier x, et calcule sa valeur absolue $|x|$, qui est aussi un entier. On peut s'en servir en utilisant la même notation que les fonctions mathématique classiques :

```c
int main(){
  int x;
  scanf("%d", &x); // lit un entier (potentiellement négatif)
  int y = valeur_absolue(x);  // stocke la valeur absolue dans y

  return 0;
}
```

Dans le code ci-dessus, on dit que l'on a **appelé** la fonction `valeur_absolue` sur la valeur de `x`, et que l'on a stocké le résultat dans `y`.
Autre exemple :

```
1 // Affiche x entre crochets
2 void affiche_crochets(int x){
3   printf("[%d]", x);
4 }
```

Cette fonction prend en entrée un entier x et l'affiche entre crochets dans le terminal. Elle ne **renvoie** rien : on ne pourrait pas écrire `y = affiche_crochets(x);`.
Dernier exemple :

```
1 // Génère un nombre aléatoire entre 10 et 20 inclus
2 int random_10_20(){
3   return (rand () % 11) + 10;
4 }
```

Cette fonction ne prend pas de paramètre, et génère un nombre aléatoire. Le code suivant montre un exemple d'utilisation des deux fonctions précédentes :

```
1 int main(){
2   srand(time(NULL));
3   int a = random_10_20();
4   int b = random_10_20();
5   affiche_crochets(a);
6   affiche_crochets(b);
7   return 0;
8 }
```

On remarque que, contrairement à une fonction mathématique, une fonction informatique peut ne rien prendre en entrée, et/ou ne rien renvoyer/calculer. Pire encore : une fonction peut renvoyer deux résultats différents si on l'appelle sur la même entrée (grâce à l'aléatoire par exemple), ce qui est impensable en mathématiques !

Notons que `printf`, `scanf`, et même `main` sont des fonctions.

B Boucles

Les boucles sont des outils algorithmiques permettant de répéter des instructions un certain nombre de fois. Il y a deux sortes de boucles : pour/for et tant-que/while.

Boucle pour Voici un exemple d'algorithme utilisant une boucle pour :

Algorithme 1 : `est_premier`(n)

Entrée(s) : $n \in \mathbb{N}, n \geq 2$
Sortie(s) : OUI si n est premier, NON sinon

```
1 pour i = 2 à n − 1 faire
2     si i divise n alors
3         retourner NON
  // Après la boucle, tous les diviseurs ont été testés
4 retourner OUI
```

Pour exécuter la boucle, on fait varier i de 2 à $n-1$ (si $n = 2$ alors on ne lance pas du tout la boucle), et pour chaque valeur de i, on exécute le corps de la boucle : si i divise n on renvoie immédiatement la réponse NON. A la fin de la boucle, on a essayé de diviser n par tous les potentiels diviseurs, sans en trouver, on renvoie donc OUI, car n est forcément premier.

En C, la syntaxe des boucles for est un peu lourde. Voici comment on implémenterait l'algorithme précédent en C :

```c
// Renvoie 1 si n est premier, 0 sinon
int est_premier(int n){
  for(int i = 2; i < n; i++){
    if (n % i == 0){
      return 0;
    }
  }
  return 1;
}
```

Dans ce code, la boucle pour est indiquée par le mot-clé `for`, et est constitué de 4 parties :

- Le corps de la boucle : c'est le code qui est exécuté pour chaque valeur de i
- L'initialisation `int i = 2` : c'est là qu'on nomme l'indice de boucle et qu'on donne sa valeur initiale
- La condition de boucle `i < n` : tant que cette condition est vraie, on continue à boucler, et on s'arrête lorsqu'elle devient fausse. Ici, on s'arrête dès que i atteint n : on a donc testé $i = 2, 3, \ldots, n-1$.
- L'incrément `i++` : cette instruction augmente la valeur de i de 1. Elle a le même effet que `i = i + 1`.

Autre exemple de boucle sur le même modèle :

```c
for (int k_1 = 7; k_1 <= 48; k_1++){
  printf("%d\n", k_1);
}
```

Cette boucle va afficher tous les entiers $k_1 = 7, 8, \ldots, 48$. Comme l'inégalité de la condition est large, la valeur 48 est aussi utilisée.

Notons que l'on peut utiliser le mot clé `for` pour faire des boucles assez complexes, mais pour l'instant on se restreint à la forme suivante :

```
for (int i = DEBUT; i < FIN; i++){ // ou bien i <= FIN
  CORPS
}
```

où seuls DEBUT, FIN, CORPS et le nom de la variable i peuvent changer.

Boucle tant-que Une boucle tant-que permet de répéter des instructions **tant qu'**une certaine condition est vérifiée. Par exemple :

```
int x = -1;
while (x < 0){
  scanf("%d", &x);
}
```

Ce code lit des entiers dans le terminal jusqu'à en trouver un positif : il continue de lire **tant que** x est strictement négatif. Notons qu'une boucle tant-que peut ne jamais s'arrêter :

```
while (1 + 1 == 2){
  printf("...");
}
```

Dans le terminal, appuyer sur CTRL + C permet d'interrompre l'exécution d'un programme. Ça peut être utile si jamais vous lancez un programme qui tourne indéfiniment...

Voyons un autre exemple : reprenons la fonction `est_premier`, et écrivons-la avec une boucle tant-que :

```
// Renvoie 1 si n est premier, 0 sinon
int est_premier(int n){
  int i = 2; // diviseur potentiel
  while (i < n && n % i != 0){
    i = i + 1;
  }
  (A completer)
}
```

Cette boucle continue de s'exécuter tant que $i < n$ ET que i ne divise pas n. Donc, elle s'arrête lorsque $i \geq n$ OU i divise n. Si elle s'arrête car $i \geq n$, cela signifie que l'on a essayé tous les diviseurs potentiels sans en trouver un. Sinon, cela signifie que i est un diviseur de n, qui n'est donc pas premier. On peut donc compléter la fonction ainsi :

```
  if (i >= n){
    return 1;
  } else {
    return 0;
  }
```

En fait, en C les conditions sont des valeurs : 1 pour vrai et 0 pour faux. On pourrait donc même écrire directement :

```
  return (i >= n); // renvoie 1 si i >= n, 0 sinon
```