Introduction à la Programmation des Algorithmes

3.1. Langage C – Fonctions et librairies

François Fleuret
https://fleuret.org/11x001/

Il arrive fréquemment qu'une sous-partie d'un programme constitue un tout cohérent avec un sens clair, et dont les interactions avec le reste du programme se font via un petit nombre de grandeurs:

- trier des nombres,
- multiplier des matrices,
- afficher une image,
- lire un fichier,
- etc.

Tous les langages de programmation permettent de définir des **fonctions**, qui sont des fragments de programmes auxquels sont associés des identifiants.

Cela permet d'organiser le code source et de faciliter sa compréhension, en particulier en évitant de répliquer les mêmes bouts de programmes.

Fonctions

Dans cet exemple, nous voulons garder celui de deux entiers qui a le plus petit nombre de diviseurs:

```
1   int n1 = 10503, n2 = 22703;
2
3   int nb_div_n1 = 0;
4   for(int k = 1; k <= n1; k++)
5     if(n1 % k == 0) nb_div_n1++;
6
7   int nb_div_n2 = 0;
8   for(int k = 1; k <= n2; k++)
9     if(n2 % k == 0) nb_div_n2++;
10
11  int m;
12  if(nb_div_n1 <= nb_div_n2) { m = n1; }
13  else { m = n2; }
```

Les lignes 3–5 calculent dans `nb_div_n1` le nombre de diviseurs de `n1`, et les lignes quasi identiques 7–9 calculent dans `nb_div_n2` le nombre de diviseurs de `n2`. Les lignes 11–13 copie dans `m` l'entier avec le plus petit nombre de diviseurs.

Ce programme peut être réécrit avec une **fonction** qui effectue le calcul que l'on veut faire plusieurs fois:

```
1  int nb_div(int n) {
2    int nb_div_n = 0;
3    for(int k = 1; k <= n; k++)
4      if(n % k == 0) nb_div_n++;
5    return nb_div_n;
6  }
7
8  int main(void) {
9    int n1 = 10503, n2 = 22703;
10
11   int m;
12   if(nb_div(n1) <= nb_div(n2)) { m = n1; }
13   else { m = n2; }
14
15   return 0;
16 }
```

Les lignes 1–6 définissent une fonction nommée `nb_div` qui reçoit un argument de type `int` et retourne une valeur de type `int`. Les lignes 11–13 peuvent être écrites sans savoir comment est programmée `nb_div`, et sont faciles à comprendre.

La définition d'une fonction en C se fait de la façon suivante avec l'opérateur `()`:

```
1  type_du_resultat nom_de_la_fonction(type1 arg1, type2 arg2, ...) {
2    instructions
3  }
```

Lorsqu'elle est utilisée:

- chaque argument a la forme d'une variable locale ayant le type correspondant, dont la portée est entre les `{}` qui encadrent le **corps** de la fonction, et dans laquelle est **copiée** la valeur passée au moment de l'appel,
- son exécution s'arrête dès qu'un `return` est rencontré, et la valeur de l'expression de ce `return` est la valeur finalement calculée par la fonction.

L'utilisation de cette fonction se fait avec l'opérateur `()`:

```
nom_de_la_fonction(expression1, expression2, ...)
```

qui peut être combiné avec les autres opérateurs arithmétiques et relationnels que nous avons vus.


```c
float moyenne(float a, float b) {
  return (a + b) / 2;
}

int main(void) {
  int a = 3;
  float x = 5.0, y = 11.0;
  float m;
  m = moyenne(a * x, y);
  return 0;
}
```

Ce qui nous donne par substitution (avec a=3, x=5 et y=11):

```
m = moyenne(a * x, y)
m = moyenne(3 * x, y)
m = moyenne(3 * 5, y)
m = moyenne(15, y)
m = moyenne(15, 11)
m = (15 + 11) / 2
m = 26 / 2
m = 13
```

13 (et m a changé)

Il peut arriver qu'une fonction ne doive pas retourner de résultat et simplement avoir des effets tels qu'un affichage, un son, ou l'écriture dans un fichier. On parle alors d'**effets de bords**, le type de retour doit être `void` et il n'y a pas d'instruction `return`.

```c
#include <stdio.h>

void affiche_carre(int n) {
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
      if(i == 0 || i == n-1 || j == 0 || j == n-1) {
        printf("**");
      } else {
        printf("..");
      }
    }
    printf("\n");
  }
}
```

```c
int main(void) {
  affiche_carre(8);
  affiche_carre(5);
  return 0;
}
```

affiche

```
****************
**............**
**............**
**............**
**............**
**............**
**............**
****************
**********
**......**
**......**
**......**
**********
```


Le compilateur détecte et indique une erreur si une fonction doit retourner une valeur selon sa définition mais ne possède pas de return.

```c
#include <stdio.h>

int plus_petit_diviseur(int n) {
  for(int k = 2; k < n; k++) {
    if(n % k == 0) return k;
  }
}

int main(void) {
  printf("%d\n", plus_petit_diviseur(2344343));
  return 0;
}
```

```
clang test.c && ./a.out
test.c:7:1: warning: control may reach end of non-void function [-Wreturn-type]
}
^
1 warning generated.
```

Le compilateur produit quand même un fichier exécutable, bien que le comportement de ce dernier soit mal défini.

Bien que cela rende parfois le programme plus compliqué à comprendre, une fonction peut contenir plusieurs `return`. Elle s'interrompt et retourne une valeur dès qu'elle en rencontre un.

```
1  float valeur_absolue(float x) {
2    if(x >= 0) return x;
3    else return -x;
4  }
5
6  int plus_petit_diviseur(int n) {
7    for(int k = 2; k < n; k++) {
8      if(n % k == 0) return k;
9    }
10   return n;
11 }
```


Une fonction peut évidement faire appel à d'autres fonctions, ce qui permet de réaliser des programmes très complexes globalement avec une complexité toujours limitée localement.

```
1  float surface_disque(float rayon) {
2    float pi = 3.14159265359;
3    return pi * rayon * rayon;
4  }
5
6  float volume_cylindre(float rayon, float hauteur) {
7    return surface_disque(rayon) * hauteur;
8  }
9
10 int main(void) {
11   float v;
12   v = volume_cylindre(1.2, 25.0);
13   return 0;
14 }
```

Dans certaines situations, il arrive que l'on veuille utiliser une fonction avant de l'avoir programmée. Cela peut être le cas pour rendre le programme plus clair à lire, ou si des fonctions s'appellent les unes les autres.

Contrairement à d'autres langages le C ne regarde pas tout le source pour voir les fonctions qui existent. Par conséquent il doit avoir été informé de l'existence d'une fonction avant son utilisation.

On peut déclarer une fonction en indiquant son **prototype** qui spécifie uniquement son nom, le type de la valeur qu'elle retourne et les types de ses arguments. La définition peut venir après.


```
1  #include <stdio.h>
2
3  int plus_petit_diviseur(int n);
4
5  int main(void) {
6    printf("%d\n", plus_petit_diviseur(2344343));
7    return 0;
8  }
9
10 int plus_petit_diviseur(int n) {
11   for(int k = 2; k < n; k++) {
12     if(n % k == 0) return k;
13   }
14   return n;
15 }
```

La ligne 3 déclare la fonction `plus_petit_diviseur` sans la définir, et elle peut donc être utilisée ligne 6. Sa définition vient lignes 9–14, après celle de `main` lignes 5–7.

Un détail technique important en C est que les variables locales qui existent là où la fonction est appelée n'existent pas dans la fonction. On parle de **portée lexicale**: les variables accessibles sont spécifiées dans le code source, et ne dépendent pas de ce qui se passe lors de l'exécution du programme.


```c
float surface_disque(float rayon) {
  float pi = 3.14159265359;
  return pi * rayon * rayon;
}

int main(void) {
  for(int k = 0; k < 5; k++) {
    float a = k + 1;
    printf("rayon %f surface %f\n", a, surface_disque(a));
  }

  for(int l = 0; l < 3; l++) {
    float b = 1 / (float) (l + 1);
    printf("rayon %f surface %f\n", b, surface_disque(b));
  }

  return 0;
}
```

Ici par exemple les variables `k`, `a`, `l`, et `b` ne sont pas accessibles dans la fonction `surface_disque`.

En particulier, **modifier les arguments dans le corps de la fonction ne modifie rien aux variables qui existent là où la fonction est appelée.**

Il est parfois nécessaire de le faire, auquel cas on peut communiquer à la fonction l'adresse en mémoire d'une variable. Nous reviendrons là dessus dans la partie du cours sur les **pointeurs**.

De plus, le C ne permet pas de définir des fonctions locales, avec une portée limitée.

Plus généralement, le langage C n'est pas un **langage fonctionnel** dans lequel une fonction est un type comme un autre et où l'on peut:

- avoir des variables à valeur fonctionnelle, et
- faire des opérations sur les fonctions comme on peut le faire avec les autres types de données.

OCAML est un exemple de langage fonctionnel.

```
# let f = fun x -> x + 1;;
val f : int -> int = <fun>
# let g = fun u -> u * 2;;
val g : int -> int = <fun>
# let compose = fun (f, g) -> (fun x -> f(g(x)));;
val compose : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
# let h = compose(f, g);;
val h : int -> int = <fun>
# h(3);;
- : int = 7
```


Les fonctions peuvent recevoir des arguments de types structures et renvoyer des résultats de type structure.

Si nous définissons

```
1  typedef struct {
2    int a, b;
3  } paire;
```

Alors nous pouvons par exemple avoir

```
1  void affiche(paire p) {
2    printf("(%d, %d)\n", p.a, p.b);
3  }
4
5  int le_max(paire p) {
6    if (p.a >= p.b) return p.a;
7    else return p.b;
8  }
9
10 int main(void) {
11   paire q = { 123, 456 };
12   affiche(q);
13   printf("%d\n", le_max(q));
14   return 0;
15 }
```

qui affiche

```
(123, 456)
456
```


Ainsi que

```
1  paire doublon(int k) {
2    paire p;
3    p.a = k;
4    p.b = k;
5    return p;
6  }
7
8  int main(void) {
9    paire u;
10   u = doublon(17);
11   affiche(u);
12   return 0;
13 }
```

qui affiche

```
(17, 17)
```

En revanche on ne peut pas faire de même avec des tableaux.

Librairies

Il existe un grand nombre de fonctions déjà programmées et regroupées dans ce que l'on appelle des **librairies**.

Plusieurs librairies importantes sont fournies en standard avec les compilateurs C. Le compilateur permet de compiler à l'avance ces fonctions pour ne pas avoir à le faire à chaque fois. Il faut néanmoins lui indiquer de lire des fichiers **header**, où sont spécifiés les prototypes des fonctions:

- `stdio.h` pour les entrées/sorties, dont `printf`,
- `stdlib.h` pour des fonctions variées dont les conversions de chaînes de caractères en valeurs numériques, la gestion de la mémoire, et la génération de nombres aléatoires,
- `math.h` pour les fonctions mathématiques.

Pour `math.h` il faut en plus donner l'option `-lm` au compilateur pour lui dire de "lier" le fichier déjà compilé. Sinon:

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void) {
5    float a = 0.2;
6    printf("%f %f\n", sin(a), cos(a));
7    printf("%f\n", pow(sin(a), 2) + pow(cos(a), 2));
8    return 0;
9  }
```

```
clang test.c
/tmp/test-c941a9.o: In function `main':
test.c:(.text+0x1f): undefined reference to `sin'
test.c:(.text+0x35): undefined reference to `cos'
test.c:(.text+0x76): undefined reference to `sin'
test.c:(.text+0x80): undefined reference to `pow'
test.c:(.text+0xa3): undefined reference to `cos'
test.c:(.text+0xad): undefined reference to `pow'
clang-7: error: linker command failed with exit code 1 (use -v to see invocation)
```

```c
#include <stdio.h>
#include <math.h>

int main(void) {
  float a = 0.2;
  printf("%f %f\n", sin(a), cos(a));
  printf("%f\n", pow(sin(a), 2) + pow(cos(a), 2));
  return 0;
}
```

affiche

```
0.198669 0.980067
1.000000
```


Quelques-unes des fonctions de `math.h`:

Trigonometric functions

`cos`	Compute cosine
`sin`	Compute sine
`tan`	Compute tangent
`acos`	Compute arc cosine
`asin`	Compute arc sine
`atan`	Compute arc tangent

Exponential and logarithmic functions

`exp`	Compute exponential function
`log`	Compute natural logarithm
`log10`	Compute common logarithm
`log2`	Compute binary logarithm

Power functions

`pow`	Raise to power
`sqrt`	Compute square root

Rounding and remainder functions

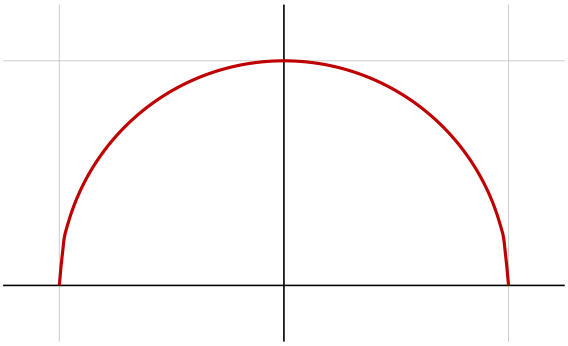
`ceil`	Round up value
`floor`	Round down value
`nearbyint`	Round to nearby integral value

Other

`fabs`	Compute absolute value
`abs`	Compute absolute value
`isinf`	Is infinity
`isnan`	Is Not-A-Number

Ce header définit également les deux constantes `M_E` et `M_PI` respectivement égales à e et π.

$$\pi = 2 \int_{-1}^{1} \sqrt{1 - x^2} dx$$




```c
#include <stdio.h>
#include <math.h>

int main(void) {
  float s = 0, dx = 1e-4;
  for(float x = -1; x <= 1; x += dx) {
    s += sqrt(1 - x * x) * dx;
  }
  printf("%f\n", 2 * s);
  return 0;
}
```

affiche

```
3.141501
```

La fonction

```
int rand(void)
```

de `stdlib.h` retourne une valeur aléatoire prise uniformément entre 0 et `RAND_MAX` inclues.

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

int main(void) {
  for(int k = 0; k < 5; k++) {
    printf("%f\n", (float) rand() / (float) RAND_MAX);
  }
  return 0;
}
```

affiche

```
0.840188
0.394383
0.783099
0.798440
0.911647
```



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

int binomial(float p, int n) {
  int m = 0;
  for(int k = 0; k < n; k++) {
    if((float) rand() / (float) RAND_MAX <= p) m++;
  }
  return m;
}

int main(void) {
  int n = 10;
  int nb[n + 1];

  for(int k = 0; k <= n; k++) nb[k] = 0;

  for(int i = 0; i < 100000; i++) {
    nb[binomial(0.5, n)]++;
  }

  for(int k = 0; k <= n; k++) {
    printf("%d: %d\n", k, nb[k]);
  }

  return 0;
}
```

```
0: 82
1: 1041
2: 4394
3: 11660
4: 20560
5: 24517
6: 20532
7: 11706
8: 4442
9: 974
10: 92
```