

# Introduction à la Programmation des Algorithmes

## 2.2. Langage C – Instructions de boucles

François Fleuret

<https://fleuret.org/11x001/>



**UNIVERSITÉ  
DE GENÈVE**

Avec ce que nous avons vu jusqu'ici, il nous est impossible d'exécuter plusieurs fois le même bout de programme.

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

```
int main(void) {  
    printf("0\n");  
    printf("1\n");  
    printf("2\n");  
    printf("3\n");  
    printf("4\n");  
    printf("5\n");  
    return 0;  
}
```

Pourtant une telle fonctionnalité est indispensable dans virtuellement toute situation concrète:

- calcul numérique,
- traitement de données,
- gestion d'évènements,
- etc.

## Instructions `while`, `do/while` et `for`

Une première instruction de contrôle du flux qui permet de répéter l'évaluation d'une clause est le `while`, qui a la forme suivante:

```
while(condition)
    clause_a_repeter
```

où `condition` définit la condition qui doit être vraie, et calcule un résultat qui prend la valeur "vrai" ou "faux".

Si résultat est "vrai" alors `clause_a_repeter` est évaluée, et le programme retourne avant le `while`.

Donc `clause_a_repeter` sera évaluée encore et encore, **tant que** `condition` est vraie.

Comme pour `if` la condition doit impérativement être entre parenthèses.

```
1  int n = 0;
2
3  while(n < 6) {
4      printf("%d\n", n);
5      n++;
6  }
```

affiche

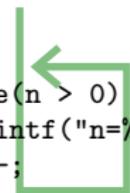
```
0
1
2
3
4
5
```

Il est important de remarquer que si la condition n'est pas vraie initialement, la clause n'est même pas exécutée une fois:

```
while(n > 0) {  
    printf("n=%d\n", n);  
    n--;  
}  
  
printf("Fini n=%d!", n);
```

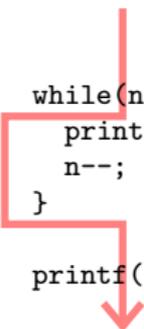
Il est important de remarquer que si la condition n'est pas vraie initialement, la clause n'est même pas exécutée une fois:

```
while(n > 0) { True  
    printf("n=%d\n", n);  
    n--;  
}  
  
printf("Fini n=%d!", n);
```



Il est important de remarquer que si la condition n'est pas vraie initialement, la clause n'est même pas exécutée une fois:

```
False while(n > 0) {  
    printf("n=%d\n", n);  
    n--;  
}  
printf("Fini n=%d!", n);
```



```
1  int n = 0;
2
3  while(n > 0) {
4      printf("%d est strictement positif\n", n);
5      n--;
6  }
7
8  while(n < 3) {
9      printf("%d est strictement plus petit que 3\n", n);
10     n++;
11 }
```

affiche

```
0 est strictement plus petit que 3
1 est strictement plus petit que 3
2 est strictement plus petit que 3
```

Une deuxième instruction de contrôle du flux qui permet de répéter l'évaluation d'une clause et le `do-while`, qui a la forme suivante:

```
do {  
    clause_a_repeter  
} while(condition);
```

`clause_a_repeter` est évaluée, puis, si la condition est "vraie", le programme retourne avant le `do`.

Donc `clause_a_repeter` sera évaluée encore et encore, **tant que** condition est vraie, mais **la clause sera toujours évaluée au moins une fois**.

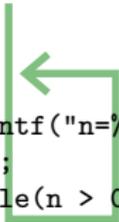
- La condition doit impérativement être entre parenthèses,
- `clause_a_repeter` peut être une seule ou plusieurs expressions terminées par un `;`, mais **est forcément entre `{}`**.

Donc, contrairement à ce qui se passe avec `while`, avec `do ... while` la clause est toujours exécutée une fois:

```
do {  
    printf("n=%d\n", n);  
    n--;  
} while(n > 0);  
  
printf("Fini n=%d!", n);
```

Donc, contrairement à ce qui se passe avec `while`, avec `do ... while` la clause est toujours exécutée une fois:

```
do {  
    printf("n=%d\n", n);  
    n--;  
} while(n > 0); True  
  
printf("Fini n=%d!", n);
```



Donc, contrairement à ce qui se passe avec `while`, avec `do ... while` la clause est toujours exécutée une fois:

```
do {  
    printf("n=%d\n", n);  
    n--;  
} while(n > 0);  
printf("Fini n=%d!", n);
```

False



```
1  int n = 0;
2
3  do {
4      printf("%d est strictement positif\n", n);
5      n--;
6  } while(n > 0);
7
8  do {
9      printf("%d est strictement plus petit que 5\n", n);
10     n++;
11 } while(n < 5);
```

affiche

```
0 est strictement positif
-1 est strictement plus petit que 5
0 est strictement plus petit que 5
1 est strictement plus petit que 5
2 est strictement plus petit que 5
3 est strictement plus petit que 5
4 est strictement plus petit que 5
```

La majorité des exécutions répétées sont des boucles du type

```
1 k = 0;
2
3 while(k < 10) {
4     printf("k=%d\n", k);
5     k++;
6 }
```

avec:

- une initialisation (ici `k = 0`),
- une condition (ici `k < 10`),
- une mise à jour (ici `k++`), et
- une clause à répéter (ici `printf("k=%d\n", k)`).

Une troisième instruction de contrôle du flux spécifique pour ce cas est le `for`, qui a la forme suivante:

```
for(initialisation; condition; mise_a_jour)
    clause_a_repeter
```

qui est exactement équivalent à

```
initialisation
while(condition) {
    clause_a_repeter
    mise_a_jour
}
```

Comme avec `while`, la clause peut ne jamais être exécutée si la condition n'est jamais vraie.

```
1  int n;  
2  
3  for(n = 0; n < 5; n++)  
4    printf("n=%d\n", n);
```

affiche

```
n=0  
n=1  
n=2  
n=3  
n=4
```

La condition peut être une expression complexe.

```
1  int i = 0, j = 0;
2
3  for(i = 0; i + 25 >= i * i; i++) {
4      printf("i=%d j=%d\n", i, j);
5      j += i;
6  }
```

affiche

```
i=0 j=0
i=1 j=0
i=2 j=1
i=3 j=3
i=4 j=6
i=5 j=10
```

## Des vrais algorithmes

Nous avons vu qu'un algorithme pour calculer  $\sqrt{q}$  consiste à résoudre  $x^2 - q = 0$  par dichotomie:

1. définir  $a_0 = 0$ ,  $b_0 = q + 1$
2. itérer tant que  $b_n - a_n \geq \epsilon$ :
  - $c_n = \frac{a_n + b_n}{2}$
  - Si  $c_n^2 - q \geq 0$  alors  $a_{n+1} = a_n$  et  $b_{n+1} = c_n$ ,
  - sinon  $a_{n+1} = c_n$  et  $b_{n+1} = b_n$ .

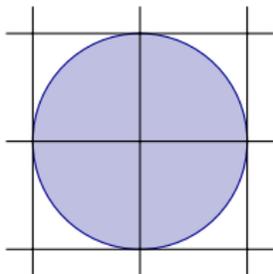
```
1 float epsilon = 1e-6, q = 2;
2 float a = 0, b = 1 + q, c;
3
4 while(b - a >= epsilon) {
5     c = (a + b) / 2;
6     if (c * c - q >= 0) {
7         b = c;
8     } else {
9         a = c;
10    }
11 }
12
13 printf("sqrt(%f) ~ %f\n", q, c);
```

affiche

sqrt(2) ~ 1.414213

Nous pouvons calculer une valeur approchée de  $\pi$  en comptant la proportion de point d'une grille régulière inscrits dans  $[0, 1]^2$  qui sont dans le disque

$$\mathcal{D} = \{(x, y) \in \mathbb{R}^2, x^2 + y^2 \leq 1\}.$$



$$N = \left\| \left\{ (i, j) \in \mathbb{Z}^2, -1 \leq \delta i \leq 1, -1 \leq \delta j \leq 1 \right\} \right\|$$

$$M = \left\| \left\{ (i, j) \in \mathbb{Z}^2, \delta^2 i^2 + \delta^2 j^2 \leq 1 \right\} \right\|$$

$$\pi \simeq 4 \frac{M}{N}$$

```
1 float x, y, N = 0, M = 0, delta = 5e-4;
2
3 for(x = -1; x <= 1; x += delta) {
4     for(y = -1; y <= 1; y += delta) {
5         N++;
6         if(x * x + y * y <= 1) M++;
7     }
8 }
9
10 printf("PI ~ %f\n", 4 * M / N);
```

affiche

PI ~ 3.141493

Lister des nombres premiers:

```
1  int n, m;
2  for(n = 2; n < 25; n++) {
3      m = 2;
4      while(m < n && n % m > 0) m++;
5      if(m == n) printf("%d\n", n);
6  }
```

affiche

```
2
3
5
7
11
13
17
19
23
```

Décomposer un entier en facteurs premiers:

```
1  int m, n = 43758;
2
3  while(n > 1) {
4      m = 2;
5      while(n % m > 0) m++;
6      printf("%d\n", m);
7      n /= m;
8  }
```

affiche

```
2
3
3
11
13
17
```

Instructions `break`, `continue`, `switch`, et `goto`

Des instructions que nous utiliserons très (très!) peu sont `break`, `continue`, `case` et `goto`, qui ne respectent pas la structure modulaire des clauses et ne sont utiles que dans des cas très particuliers.

Il arrive parfois que l'on veuille interrompre une boucle sans même finir la clause. L'instruction `break` permet de sortir immédiatement de la boucle dans laquelle elle se trouve.

```
1  n = 0;
2
3  while(n < 10000) {
4      n++;
5      if(n%17 == 0 && n%3 == 0) break;
6      n++;
7  }
8
9  printf("n=%d\n", n);
```

affiche

n=51

L'instruction `continue` ignore la suite dans l'itération courante de la boucle et va directement au début de la prochaine itération.

```
1  for(int n = 0; n < 6; n++) {  
2      printf("n=%d\n", n);  
3      if (n % 2 == 1) continue;  
4      printf("pair!\n");  
5  }
```

affiche

```
n=0  
pair!  
n=1  
n=2  
pair!  
n=3  
n=4  
pair!  
n=5
```

Il arrive assez fréquemment que l'on veuille exécuter des clauses associées à des valeurs spécifiques d'une expression.

L'instruction `switch` permet d'évaluer une expression et de continuer le programme à un endroit correspondant à la valeur obtenue spécifiée à l'aide du mot clé `case` ou `default`..

```
switch(expression) {  
  case valeur_1:  
    expression;  
    ...  
  
  case valeur_k:  
    expression;  
  
  default:  
    expression;  
};
```

L'utilisation de `break` permet de continuer l'exécution du programme après le bloc du `switch`.

```
1  n = 2;
2
3  switch(n) {
4  case 0:
5      printf("zéro\n");
6      break;
7  case 1:
8      printf("un\n");
9      break;
10 case 2:
11     printf("deux\n");
12     break;
13 default:
14     printf("beaucoup\n");
15     break;
16 }
```

affiche

deux

Une instruction de contrôle du flux rarement utilisée est `goto`. Elle permet de faire revenir/continuer le programme à un endroit arbitraire spécifié par un **label**. Ce dernier est un identifiant suivi du symbole `:`.

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



Il est difficile d'écrire un programme modulaire et sans erreur avec des `goto`. Cette instruction est réservée à des situations exotiques.

La principale utilisation de `goto` consiste à sortir directement de plusieurs boucles imbriquées.

```
for(...) {  
    for(...) {  
        for(...) {  
            ...  
            if (...) goto mon_label_place_apres;  
            ...  
        }  
    }  
}  
mon_label_place_apres:
```

```
1  int a, b, c;
2
3  for(a = 1; a < 1000; a++) {
4      for(b = 1; b < 1000; b++) {
5          for(c = 1; c < 1000; c++) {
6              if((a + b) * (b + c) == 4 * (a * b + b * c)) goto mon_label_place_apres;
7          }
8      }
9  }
10 mon_label_place_apres:
11
12  printf("a=%d b=%d c=%d\n", a, b, c);
```

affiche

a=4 b=1 c=11

Fin