# Le C — Fonctions

https://tinyurl.com/34wca5kb


Nathanael Demacon

# Fonctions

Une fonction est un morceau de code réutilisable qui prend des paramètres en entrée et renvoie un résultat.

Elles permettent de découper le code pour améliorer la lisibilité et la maintenabilité.

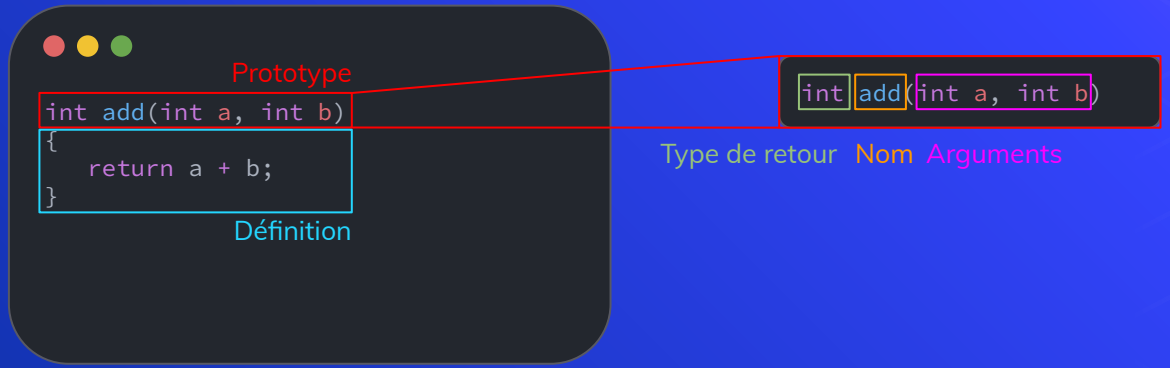
# Fonctions

Vous connaissez et utilisez déjà les fonctions.

```c
#include <stdio.h>

int main()
{
   printf("Hello, world!\n");

   int age;
   scanf("%d", &age);

   return 0;
}
```

# Déclaration

Une fonction se déclare en indiquant son type de retour, son nom et la liste de ses arguments s' il y en a. On ajoute ensuite un bloc d'instructions, c'est sa définition.



# Déclaration

Une fonction peut ne rien renvoyer. Pour ça nous indiquons le type de retour `void`.

```c
#include <stdio.h>

void sayHi()
{
  printf("Hi\n");
}
```

Il est aussi possible d'utiliser `return` sans passer de valeur, ce qui terminera la fonction.

```c
#include <stdio.h>

void sayIsThree(int value)
{
    if (value == 3)
    {
        printf("It's three\n");
        return;
    }

    printf("It's not three\n");
}
```

# Appeler une fonction

On appelle une fonction en indiquant son nom suivi de parenthèses. Entre les parenthèses, nous passons des expressions pour chaque argument demandé.

```
int add(int a, int b)
{
   return a + b;
}

int main()
{
   int result = add(1, 2); // 3

   return 0;
}
```

# Appeler une fonction

Pour appeler une fonction, il faut que celle-ci soit déclarée avant l'appel. Dans le cas où c'est impossible, on doit déclarer son prototype avant sa définition.

```
void a()
{
   b(); // error
}

void b()
{
   // ...
}
```

```
void b();

void a()
{
   b(); // ok
}

void b()
{
   // ...
}
```

# Spécificité des arguments

Les arguments d'une fonction ne peuvent pas être modifiés. Lors de leur passage, les arguments sont copiés avant d'arriver à la fonction.

```c
#include <stdio.h>

void increment(int a)
{
    a++;
}

int main()
{
    int a = 0;
    increment(a);
    printf("%d", a); // 0
}
```

```c
#include <stdio.h>

int push_grade(int *grades, int count, int grade)
{
    grades[count] = grade;
    return count + 1;
}

int main()
{
    int grades[100];
    int count = 0;

    count = push_grade(grades, count, 15);
    count = push_grade(grades, count, 20);
    count = push_grade(grades, count, 11);

    for (int i = 0; i < count; i++)
    {
        printf("%d\n", grades[i]);
    }

    return 0;
}
```

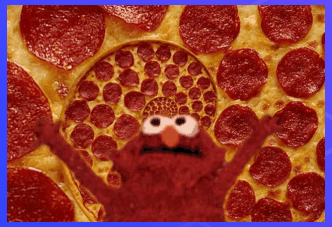
# Récursion

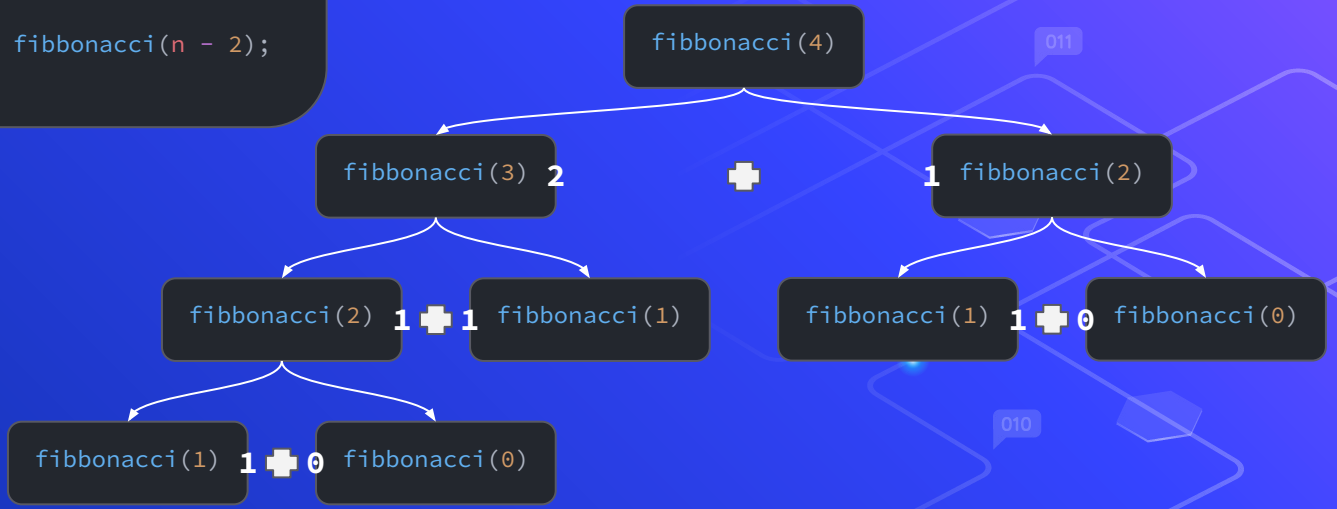
Une fonction qui s'appelle elle-même est une fonction récursive.

```c
int fibbonacci(int n)
{
    if (n <= 1)
    {
        return n;
    }

    return fibbonacci(n - 1) + fibbonacci(n - 2);
}
```

# Récursion

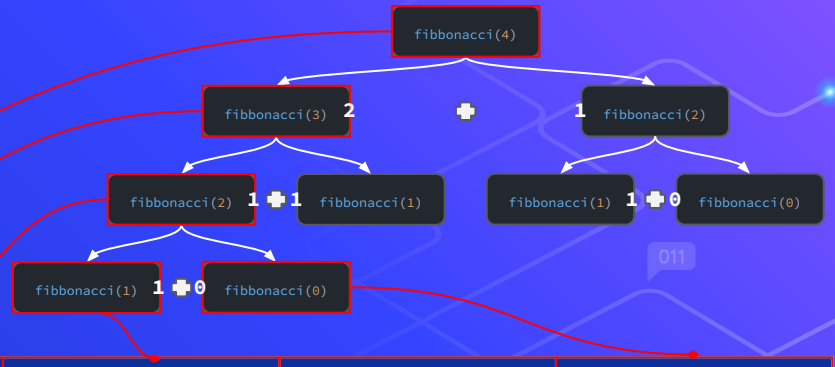
```c
int fibbonacci(int n)
{
   if (n <= 1)
   {
       return n;
   }

   return fibbonacci(n - 1) + fibbonacci(n - 2);
}
```





# Les limites de la récursion

Bien que les fonctions récursives soient pratiques, il existe une limite notable à celles-ci : la “call stack”. Cette stack augmente à chaque appel de fonction (récursive et non-récursive), remplissant la mémoire petit à petit.




| Step 1 | Step 2 | Step 3 | Step 4 | Step 5 | Step 6 |
|---|---|---|---|---|---|
| | | | fibbonacci(1) | | fibbonacci(0) |
| | | fibbonacci(2) | fibbonacci(2) | fibbonacci(2) | fibbonacci(2) |
| | fibbonacci(3) | fibbonacci(3) | fibbonacci(3) | fibbonacci(3) | fibbonacci(3) |
| fibbonacci(4) | fibbonacci(4) | fibbonacci(4) | fibbonacci(4) | fibbonacci(4) | fibbonacci(4) |