IN 101 - Cours 03

28 septembre 2011

présenté par

Matthieu Finiasz

Les fonctions en C

Qu'est-ce qu'une fonction ?

- Une fonction est un segment de code que l'on peut appeler (par exemple depuis le `main`) pour effectuer une tâche spécifique :
  - permet d'effectuer plusieur fois la même opération sans avoir à réecrire le code.

- Nous avons déjà vu des fonctions :
  - `printf` pour afficher des choses,
  - `atoi` pour parser un entier,
  - `malloc` pour allouer de la mémoire,
  - le `main` est aussi une fonction, mais un peu spéciale.

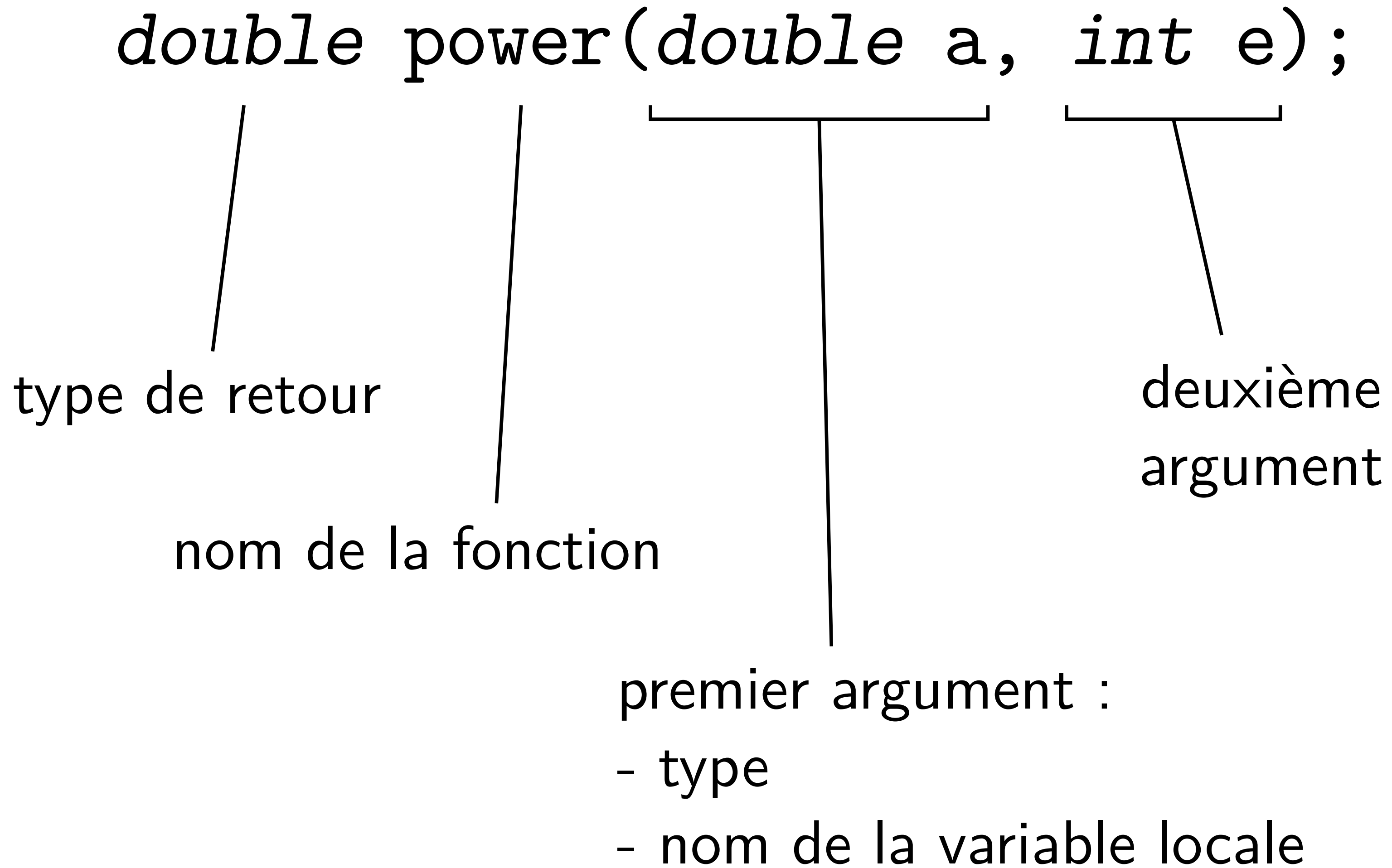
- Comme l'indiquent les exemples ci-dessus :
  - une fonction peut avoir des arguments,
  - elle peut aussi retourner une valeur.

Qu'est-ce qu'une fonction ?

- Une fonction possède deux parties :
  - un prototype (ou signature/en-tête) : c'est ce qui est vu de l'extérieur
    - → par exemple : `int atoi(char a[]);`
  - un corps : c'est le code de la fonction
    - → l'ensemble des instructions à effectuer quand on l'appelle.

- Il suffit de connaître le prototype de la fonction pour l'utiliser :
  - indique le nom de la fonction,
  - indique quels sont les arguments à donner,
  - indique le type de ce qu'elle renvoie quand on l'évalue.
- Si vous utilisez une librairie en C, le manuel indiquera le prototype des fonctions et ce qu'elles font, pas le code.

Exemple de prototype de fonction



Déclaration d'une fonction

- Comme pour les variables, on peut juste déclarer une fonction en donnant son prototype :

```
1 double power(double a, int e);
```

- Où on peut la définir complétement en donnant son code :

```
1 double power(double a, int e) {
2   int i;
3   double res = 1;
4   for (i=0; i<e; i++) {
5     res *= a;
6   }
7   return res;
8 }
```

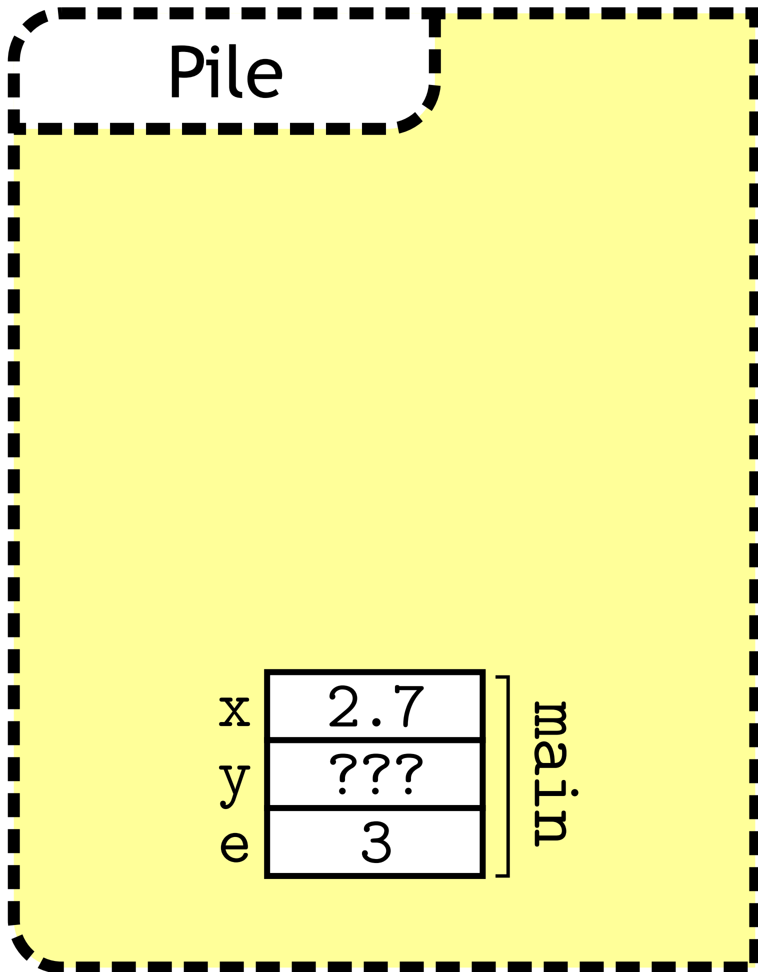
Déroulement d'un appel de fonction

- Une fois qu'une fonction est déclarée on peut l'appeler
  ⚠ il faudra que la définition complète soit aussi donnée.

- Lors de l'appel, il se passe les choses suivantes :
  - de la mémoire est reservée dans la pile pour l'exécution
    → variables locales (et arguments), adresse de retour
  - les arguments de la fonction sont évalués,
  - les valeurs des arguments sont recopiées dans les variables locales,
  - le code de la fonction est exécuté,
  - l'appel à la fonction est "remplacé" par sa valeur de retour.

Déroulement d'un appel de fonction

Étape par étape

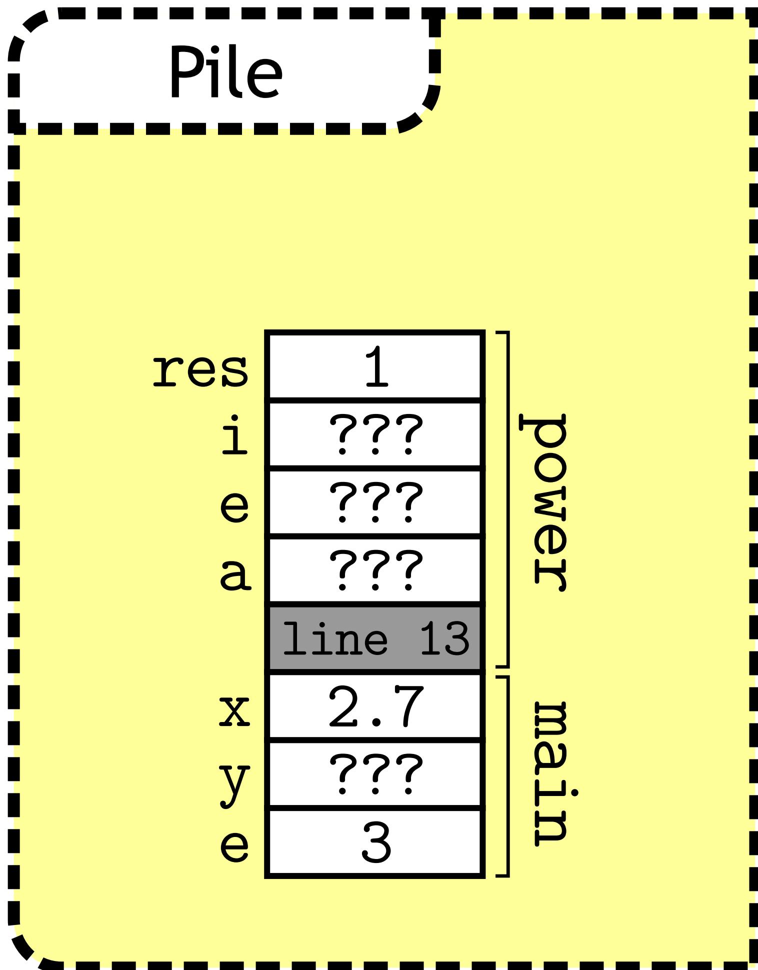



```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4    for (i=0; i<e; i++) {
5      res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- Début du main, les variables locales sont initialisées.

Déroulement d'un appel de fonction

Étape par étape

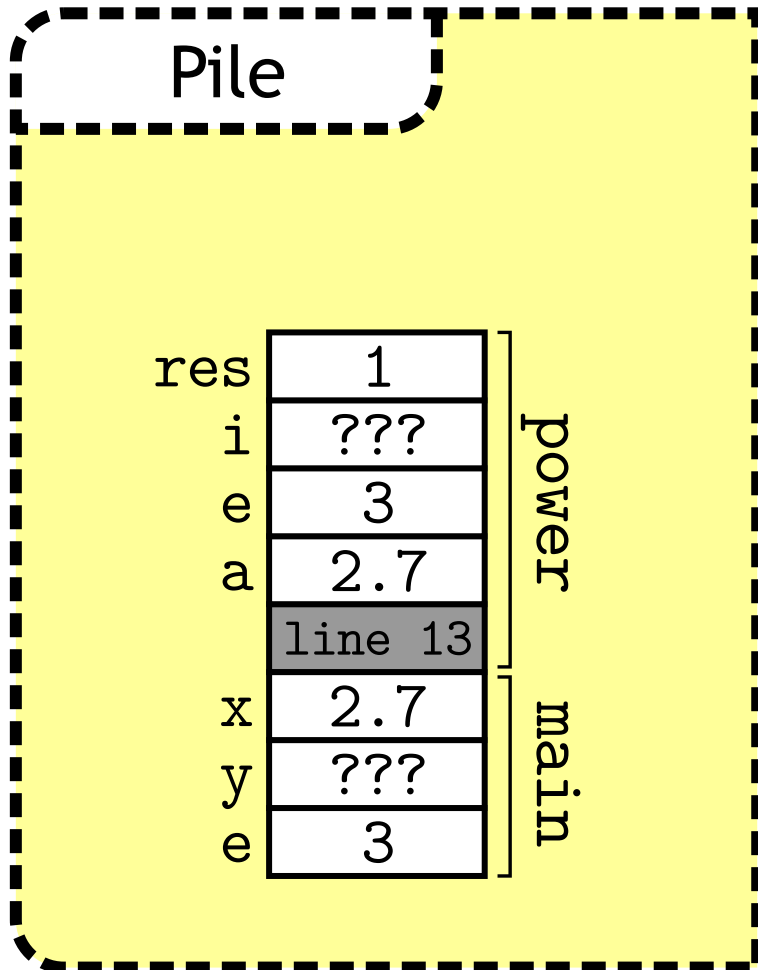



```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4    for (i=0; i<e; i++) {
5      res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13 ▶ y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- Appel à la fonction `power` :
  - on note l'adresse de retour,
  - on reserve la mémoire des variables locales.

Déroulement d'un appel de fonction

Étape par étape

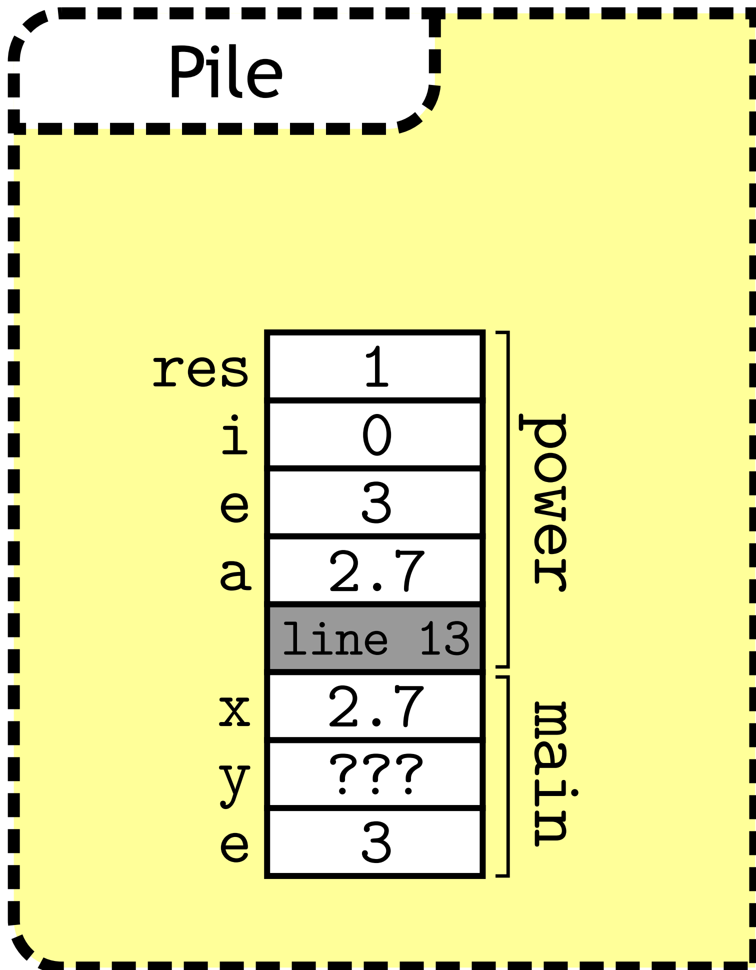



```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4    for (i=0; i<e; i++) {
5      res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13 ▶ y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- On évalue les arguments et on les recopie.

Déroulement d'un appel de fonction

Étape par étape

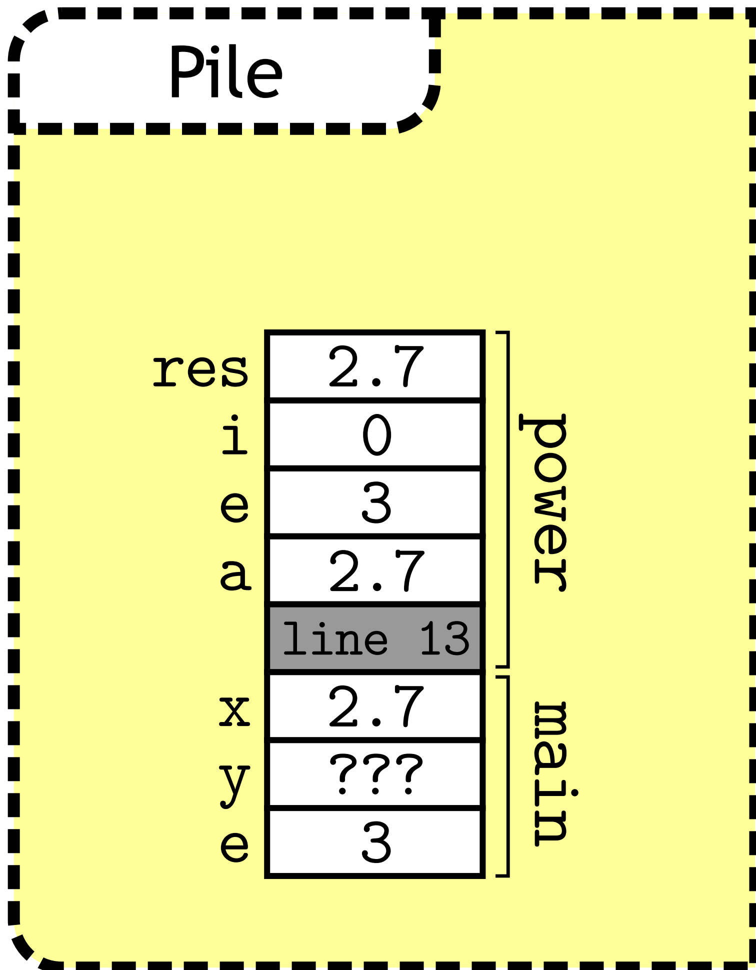



```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4  ▶ for (i=0; i<e; i++) {
5      res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- Début de la boucle `for`, `i=0`.

Déroulement d'un appel de fonction

Étape par étape

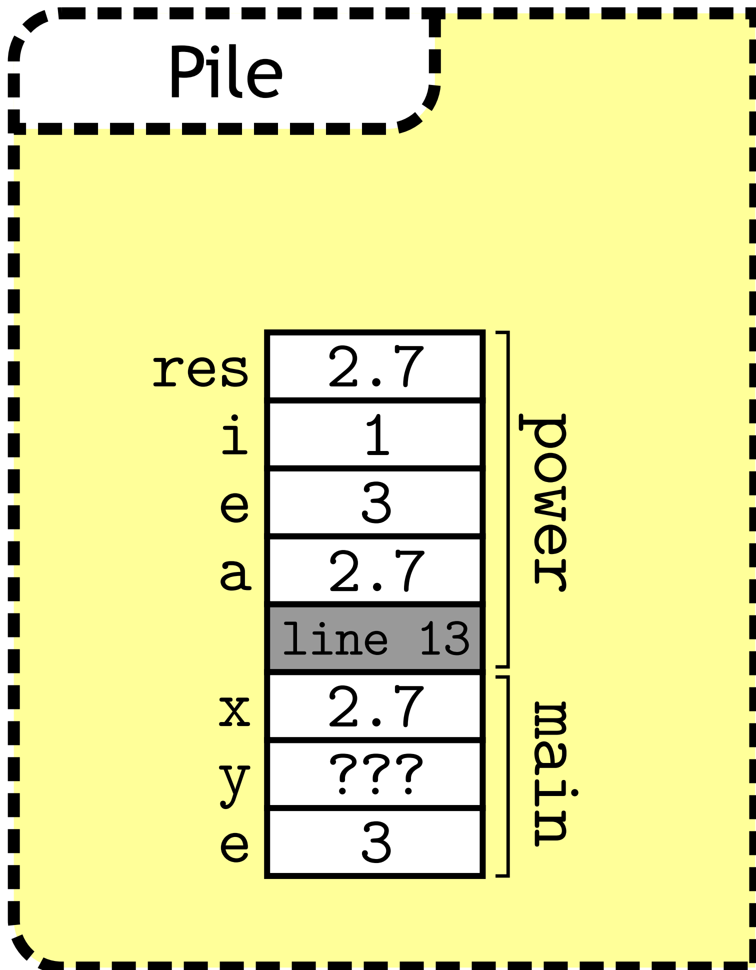



```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4    for (i=0; i<e; i++) {
5  ▶   res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- On met à jour `res`.

Déroulement d'un appel de fonction

Étape par étape

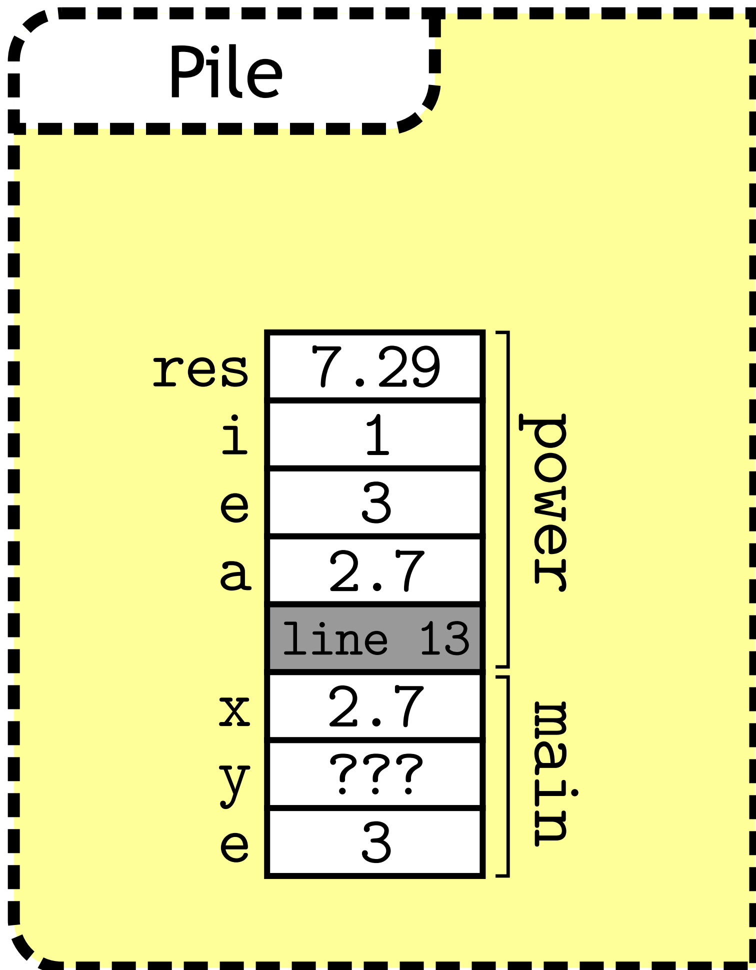



```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4 ▶  for (i=0; i<e; i++) {
5      res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- On incrément `i`.

Déroulement d'un appel de fonction

Étape par étape

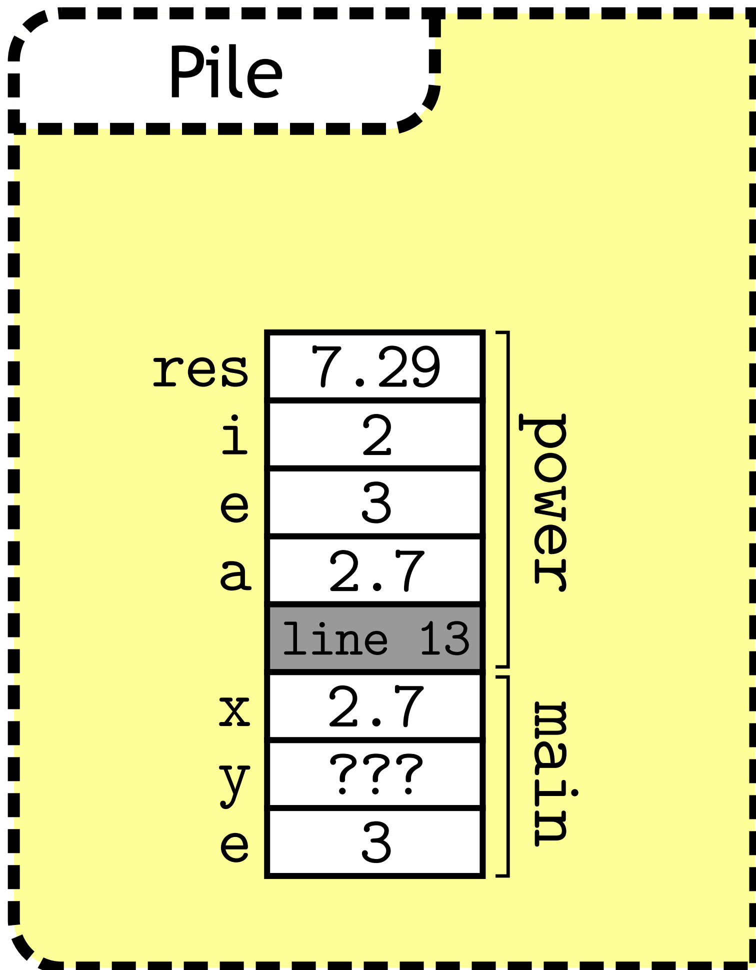



```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4    for (i=0; i<e; i++) {
5 ▶    res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- Et on continue la boucle `for`...

Déroulement d'un appel de fonction

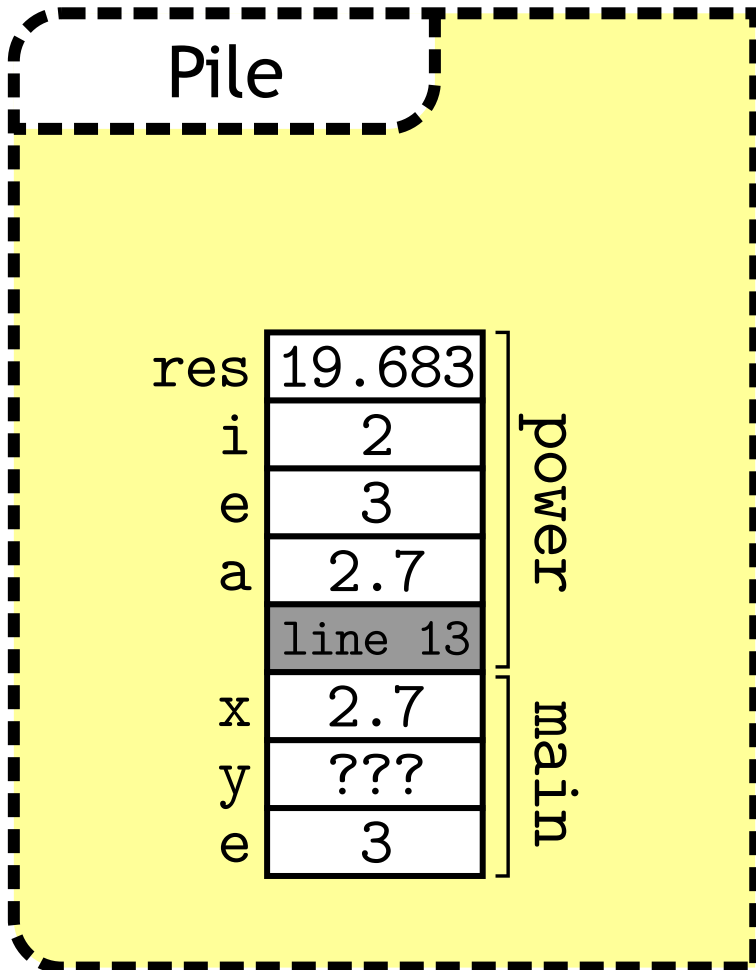
Étape par étape




```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4  ▶ for (i=0; i<e; i++) {
5      res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

Déroulement d'un appel de fonction

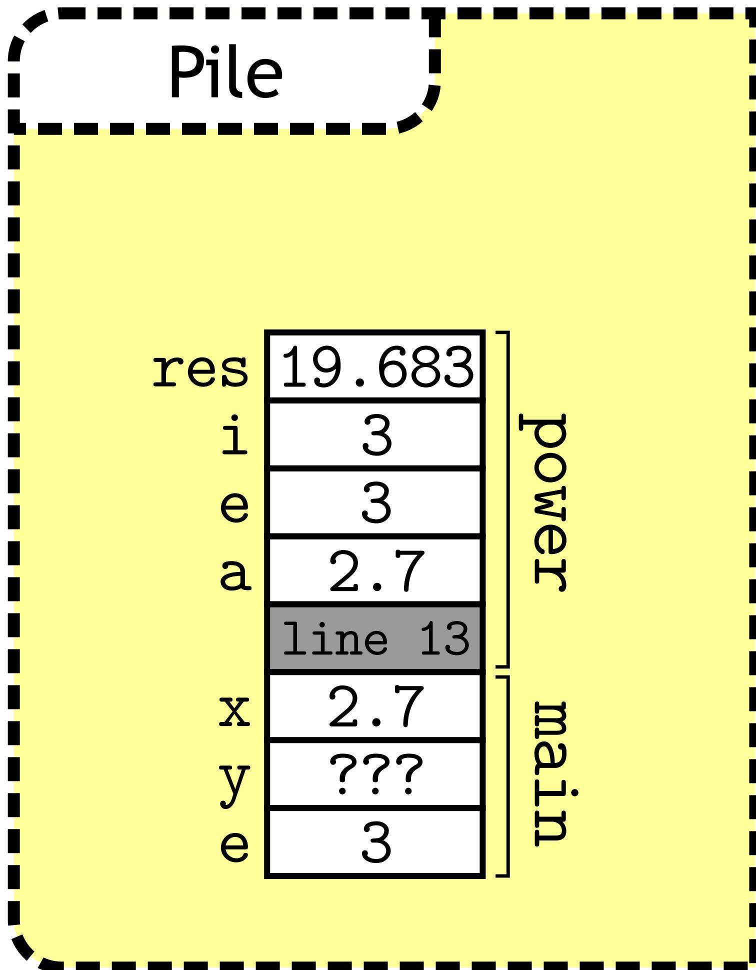
Étape par étape




```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4    for (i=0; i<e; i++) {
5 ▶    res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

Déroulement d'un appel de fonction

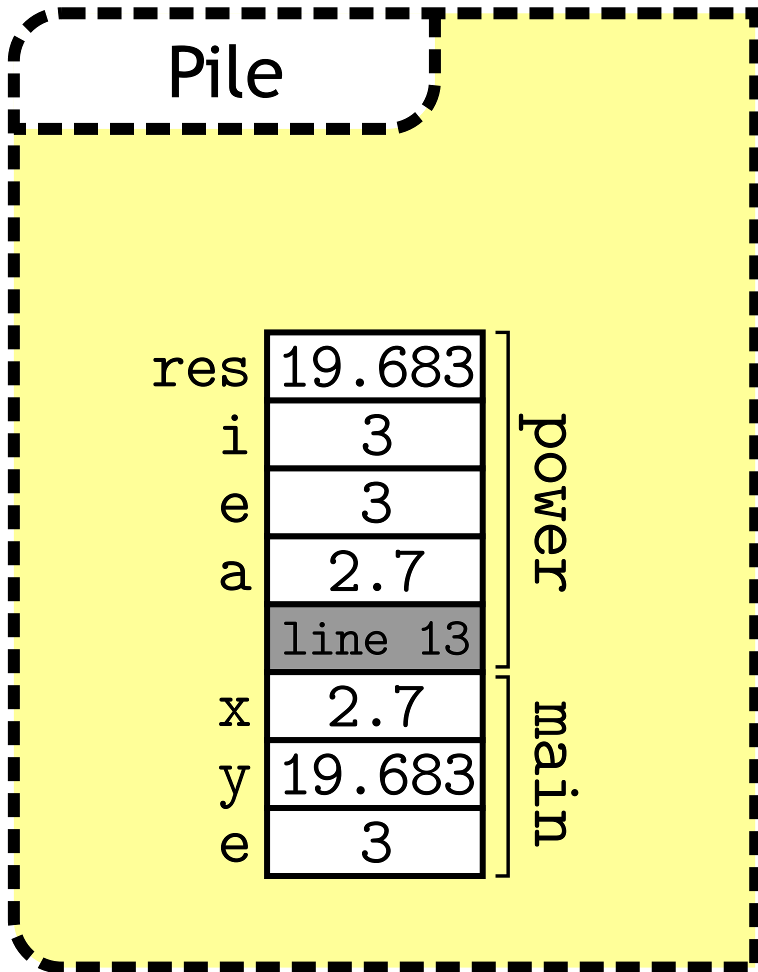
Étape par étape




```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4  ▶ for (i=0; i<e; i++) {
5      res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- Le test `i<e` est faux
  → fin de la boucle `for`.

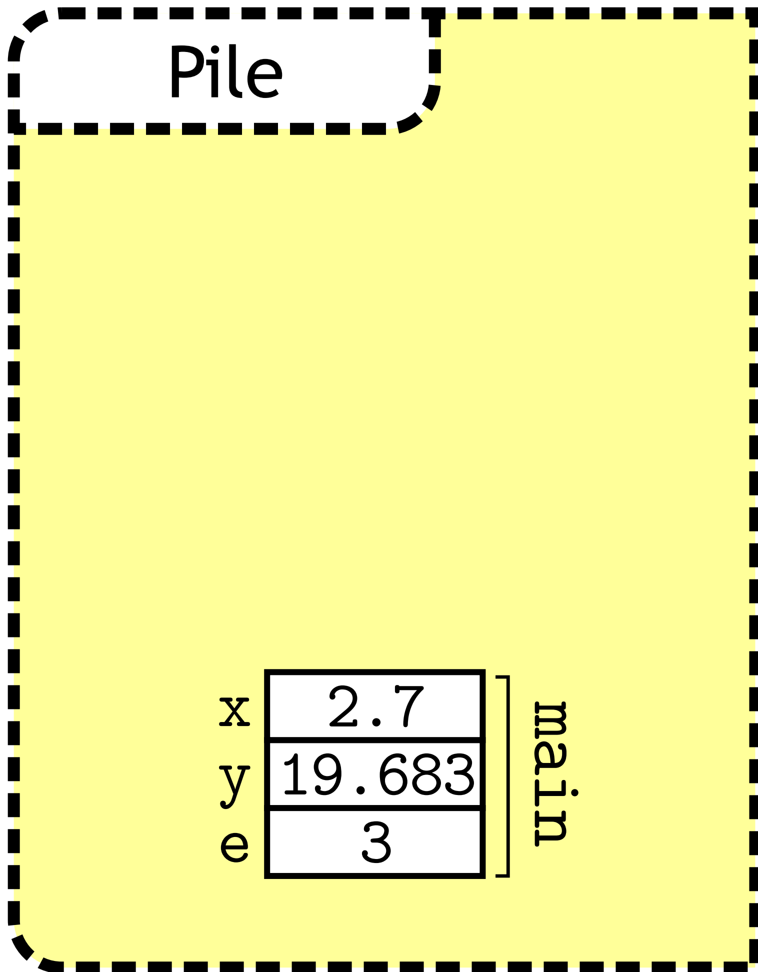
Déroulement d'un appel de fonction
Étape par étape




```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4    for (i=0; i<e; i++) {
5      res *= a;
6    }
7  ▶ return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- On retourne à l'adresse de retour
  - → res est copié dans y.

Déroulement d'un appel de fonction
Étape par étape

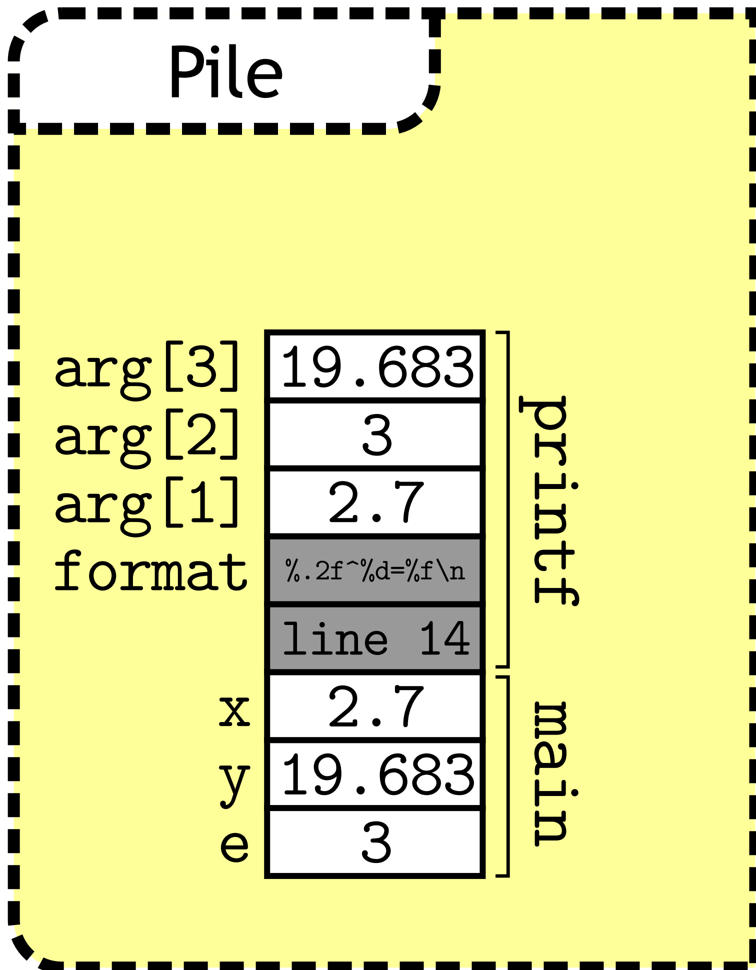



```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4    for (i=0; i<e; i++) {
5      res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13 ► y = power(x,e);
14   printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- Libération de la mémoire de la fonction power

Déroulement d'un appel de fonction

Étape par étape




```
1  double power(double a, int e) {
2    int i;
3    double res = 1;
4    for (i=0; i<e; i++) {
5      res *= a;
6    }
7    return res;
8  }
9
10 int main() {
11   int e = 3;
12   double y, x=2.7;
13   y = power(x,e);
14 ▶ printf("%.2f^%d = %f\n",x,e,y);
15 }
```

- Appel de printf.

Variables globales et locales

- Les variables globales sont déclarées en dehors d'une fonction
  - elles sont accessibles/modifiables depuis n'importe où.
- Les variables locales sont propres à chaque fonction :
  - la fonction `power` ne peut lire/modifier les variables du `main`,
  - plusieurs variables locales peuvent avoir le même nom (dans des fonctions différentes) sans conflit,
  - si une variable locale a le même nom qu'une variable globale, seule la variable locale est visible,
    → à éviter en général...
  - tout ce qui est stocké dans des variables locales est supprimé à la fin de la fonction, quand la mémoire est libérée.
    → Il faut renvoyer le résultat avec `return`, ou écrire ailleurs dans la mémoire (dans le tas, par exemple).

Fonctions sans valeur de retour

Le type *void*

- Une fonction avec un type de retour doit toujours terminer son exécution par un `return`, sinon le compilateur affiche une erreur.
- Certaines fonctions n'ont pas besoin de renvoyer quelque chose :
  - leur type de retour est alors *void*,
  - elles peuvent quand même utiliser `return`, mais sans argument :

```
1  void print_string (char str[]) {
2    int i = 0;
3    while (1) {                    // boucle infinie
4      if (str[i] == '\0') {        // test de fin de chaîne
5        return;                    // on quitte la fonction
6      }
7      printf("%c", str[i]);        // on affiche une caractère
8      i++;
9    }
10 }
```

- Le `return` permet d'interrompre l'execution de la fonction.

Utilisation de fonctions sans valeur de retour

```
1  double power(double a, int e);
2  void print_string(char str[]);
3
4  int main() {
5    double y;
6    y = power(2.7,3);   // le résultat est stocké dans y
7    power(3.5,12);      // le résultat du calcul est perdu
8    print_string("Hello World!");
9    y = print_string("Hello World!");  // erreur !
10 }
```

- On peut ne pas utiliser la valeur retournée par une fonction :
  - par exemple, `printf` retourne le nombre de caractères écrits.
- Une fonction *void* doit en revanche toujours être utilisée seule.

Fonctions récursives

Qu'est-ce qu'un fonction récursive ?

- Certains calculs s'écrivent naturellement de façon récursive,
  - par exemple, le calcul du n-ième terme d'une suite récurrente :

$$\begin{cases} U_n = f(U_{n-1}), \\ U_0 = u_0. \end{cases}$$

- Cela se programme avec des fonctions qui s'appellent elles-même
  - c'est ce que l'on appelle des fonctions récursives.

```
1 #define UZERO 1
2 int U(int n) {
3   if (n == 0) {
4     return UZERO;
5   }
6   return f(U(n-1));
7 }
```

Utilisation de fonctions récursives

- En C, une fonction récursive s'écrit comme une fonction normale
  - l'appel à elle-même se fait comme n'importe quel appel de fonction.

- En revanche, il faut une condition d'arrêt :
  - une valeur d'entrée pour laquelle il n'y a pas d'appel récursif,
  - les appels récursifs doivent toujours l'atteindre à un moment
    - → sinon, le programme ne termine pas.
  - en général, une fonction récursive commence par un test
    - → si la condition d'arrêt est atteinte, `return`.

- Pour une suite récurrente, la condition d'arrêt correspond aux conditions initiales.

Exemple : le calcul de puissance

Version récursive simple

- Un calcul de puissance peut aussi s'écrire de façon récursive :

$$\begin{cases} a^e = a \times a^{e-1}, \\ a^0 = 1. \end{cases}$$

- L'exposant e décroit de 1 à chaque appel
  → il converge donc bien vers 0.

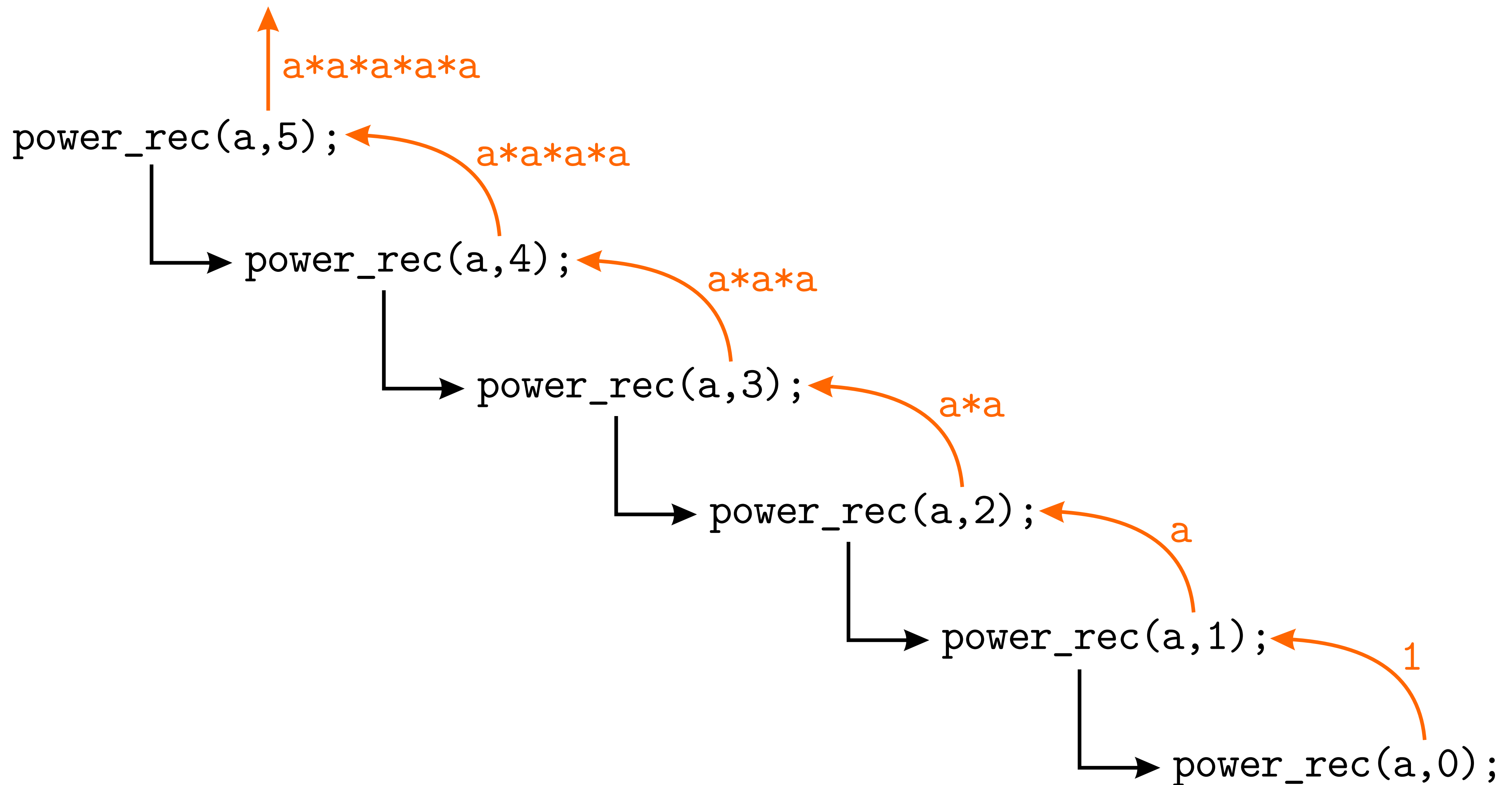
```
1 double power_rec(double a, unsigned int e) {
2   if (e == 0) {
3     return 1;
4   }
5   return a * power_rec(a,e-1);
6 }
```

Exemple : le calcul de puissance
Version récursive simple

```
power_rec(a,5);
  └─> power_rec(a,4);
        └─> power_rec(a,3);
              └─> power_rec(a,2);
                    └─> power_rec(a,1);
                          └─> power_rec(a,0);
```

Exemple : le calcul de puissance

Version récursive simple



Exemple : le calcul de puissance

Square and multiply

- On peut aussi utiliser la récurrence pair/impair suivante :

$$\begin{cases} a^{2e} = a^e \times a^e, \\ a^{2e+1} = a \times a^{2e}, \\ a^1 = a. \end{cases}$$

```
1  double power_rec2(double a, unsigned int e) {
2    double tmp;
3    if (e == 1) {
4      return a;
5    }
6    if (e%2 == 0) {
7      tmp = power_rec2(a,e/2);
8      return tmp*tmp;
9    }
10   return a * power_rec2(a,e-1);
11 }
```

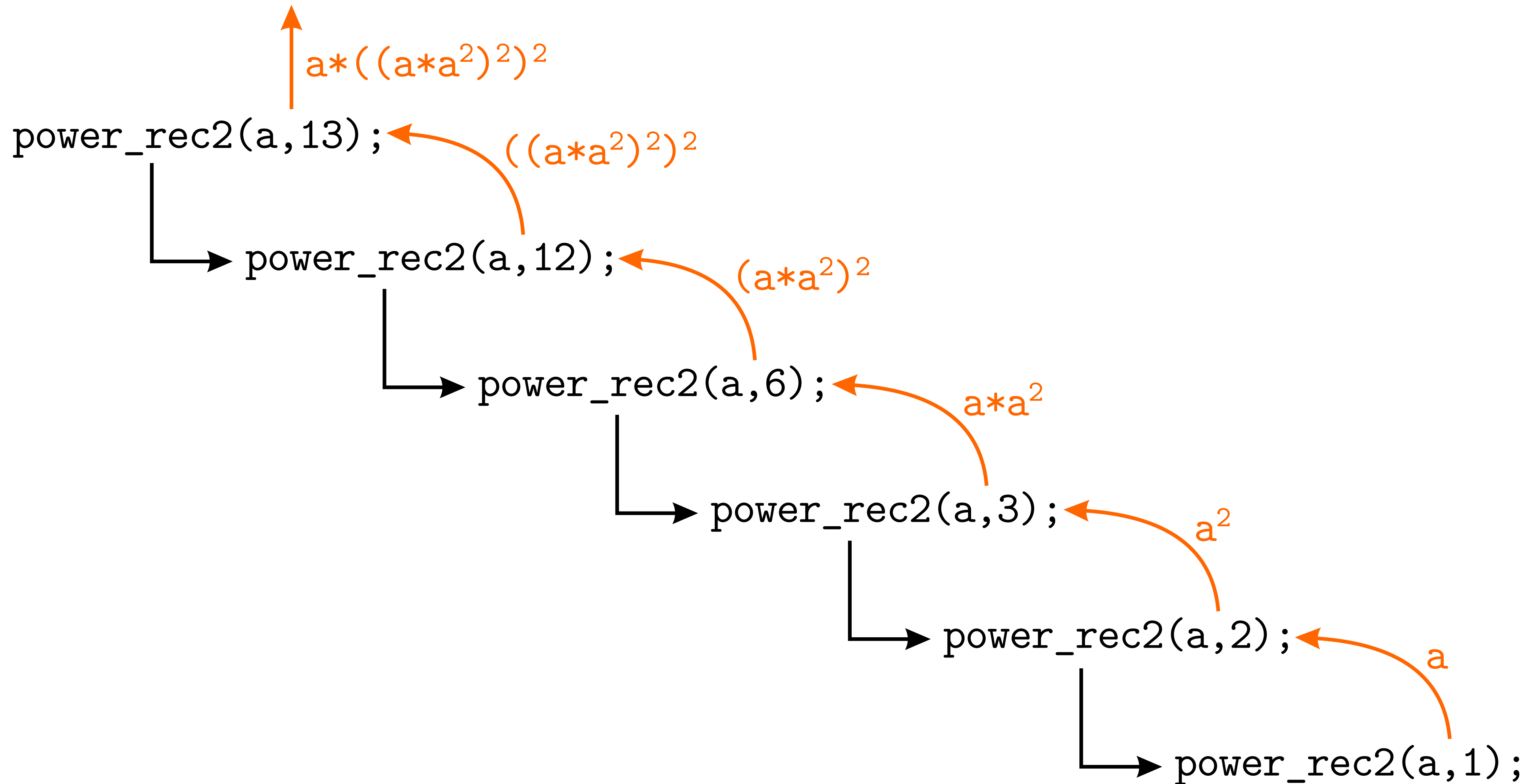
Exemple : le calcul de puissance

Square and multiply

```
power_rec2(a,13);
  └─> power_rec2(a,12);
        └─> power_rec2(a,6);
              └─> power_rec2(a,3);
                    └─> power_rec2(a,2);
                          └─> power_rec2(a,1);
```

Exemple : le calcul de puissance

Square and multiply



Appels récursifs multiples

- Une fonction récursive peut aussi faire plusieurs appels récursifs
  - pour une suite récurrente double,
  - dans les algorithmes de tri (*cf.* cours 05).

- Cela se passe comme des appels de fonctions normaux :
  - on entre dans le premier appel récursif,
  - éventuellement, on a d'autres appels récursifs,
  - quand le premier appel est fini, on passe au deuxième.

    → Les appels récursifs ne sont pas lancés en même temps.

Appels récursifs multiples

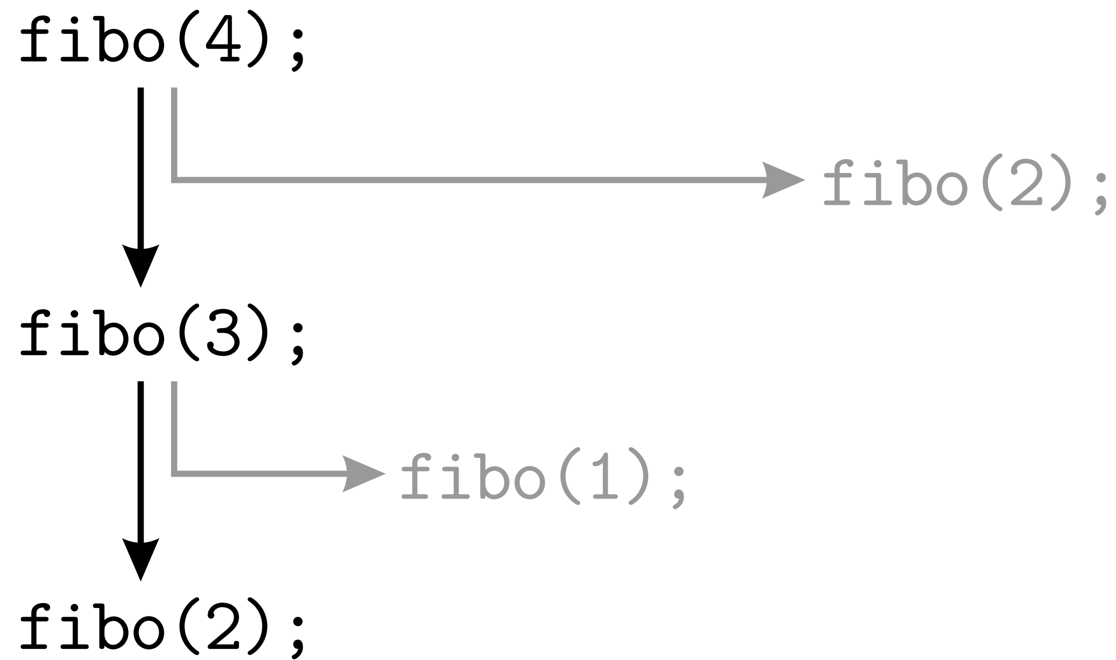
Fibonacci : $F_n = F_{n-1} + F_{n-2}$ et $F_0 = 0, F_1 = 1$

```
fibo(4);
```

```
fibo(3);
```

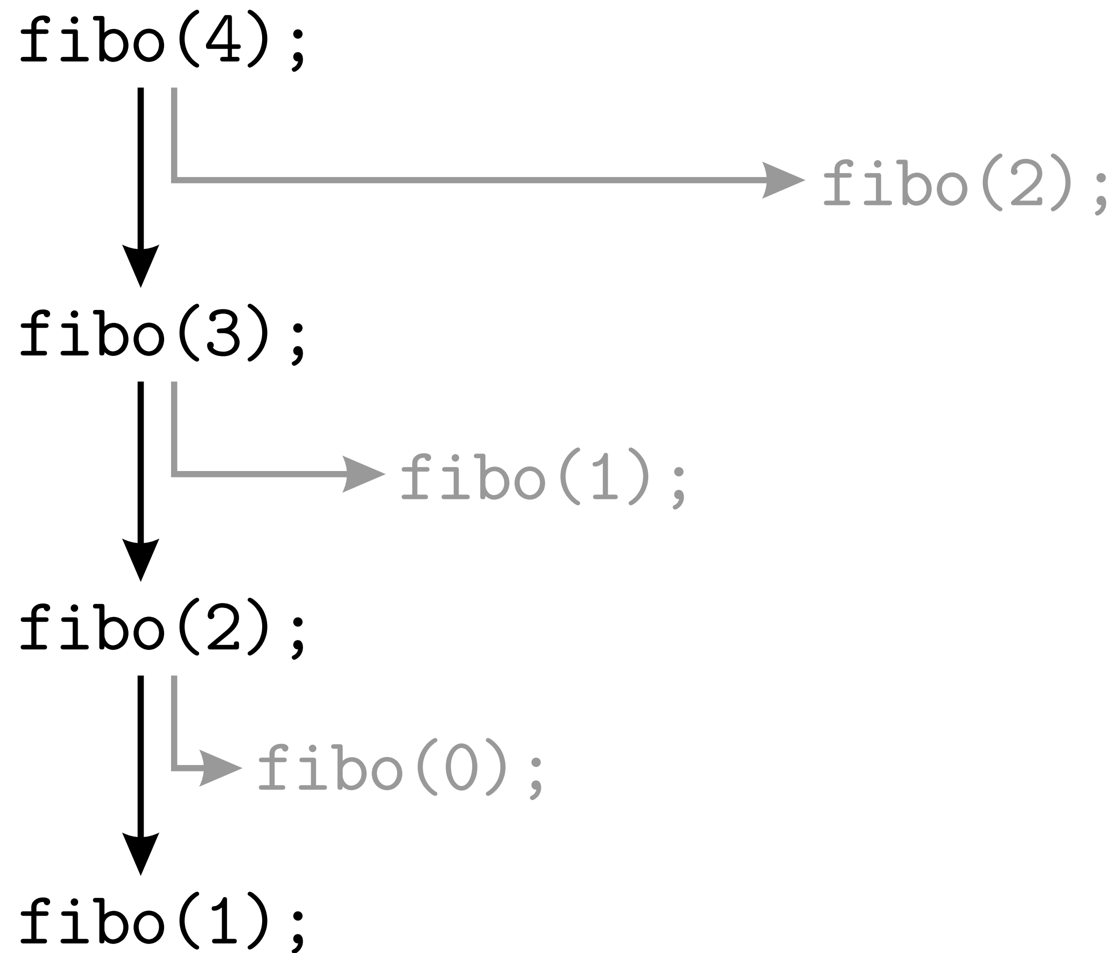
Appels récursifs multiples

Fibonacci : $F_n = F_{n-1} + F_{n-2}$ et $F_0 = 0, F_1 = 1$



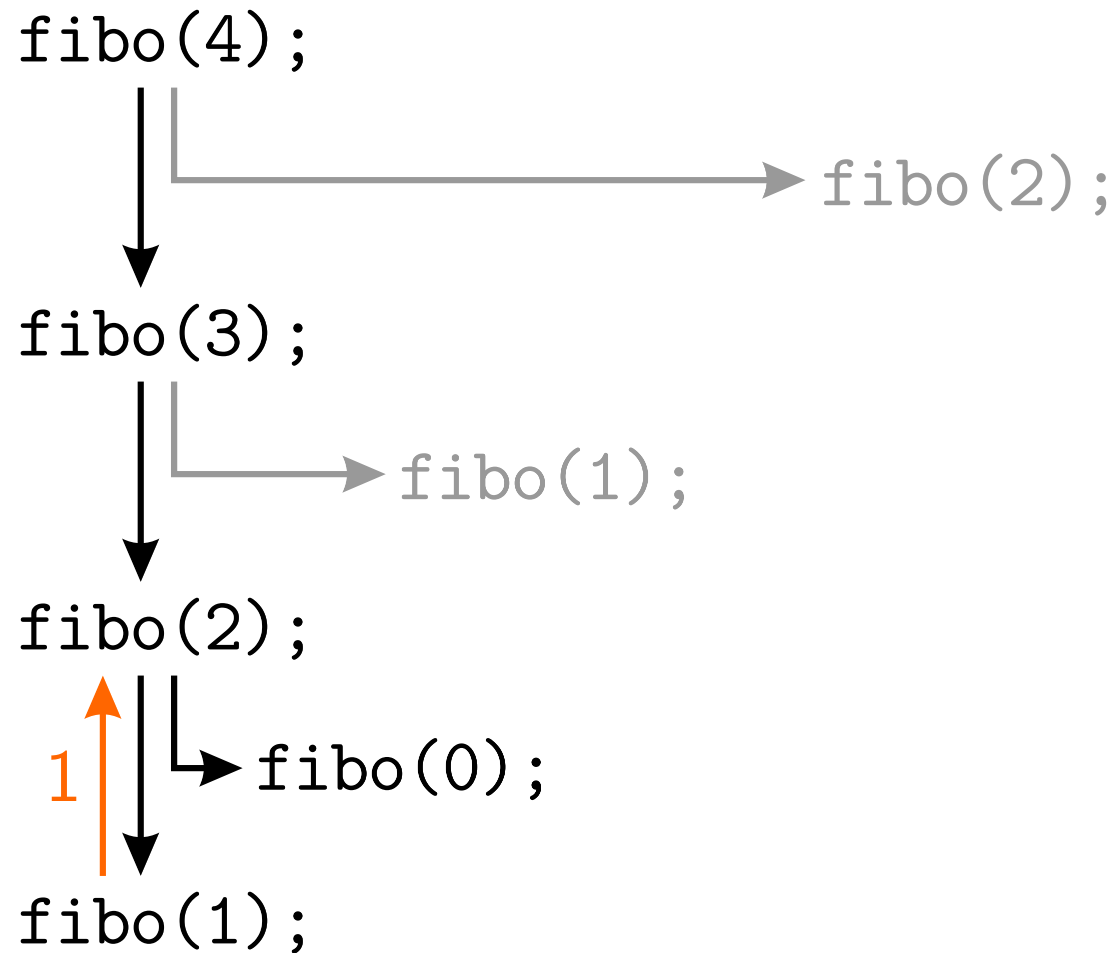
Appels récursifs multiples

Fibonacci : $F_n = F_{n-1} + F_{n-2}$ et $F_0 = 0, F_1 = 1$



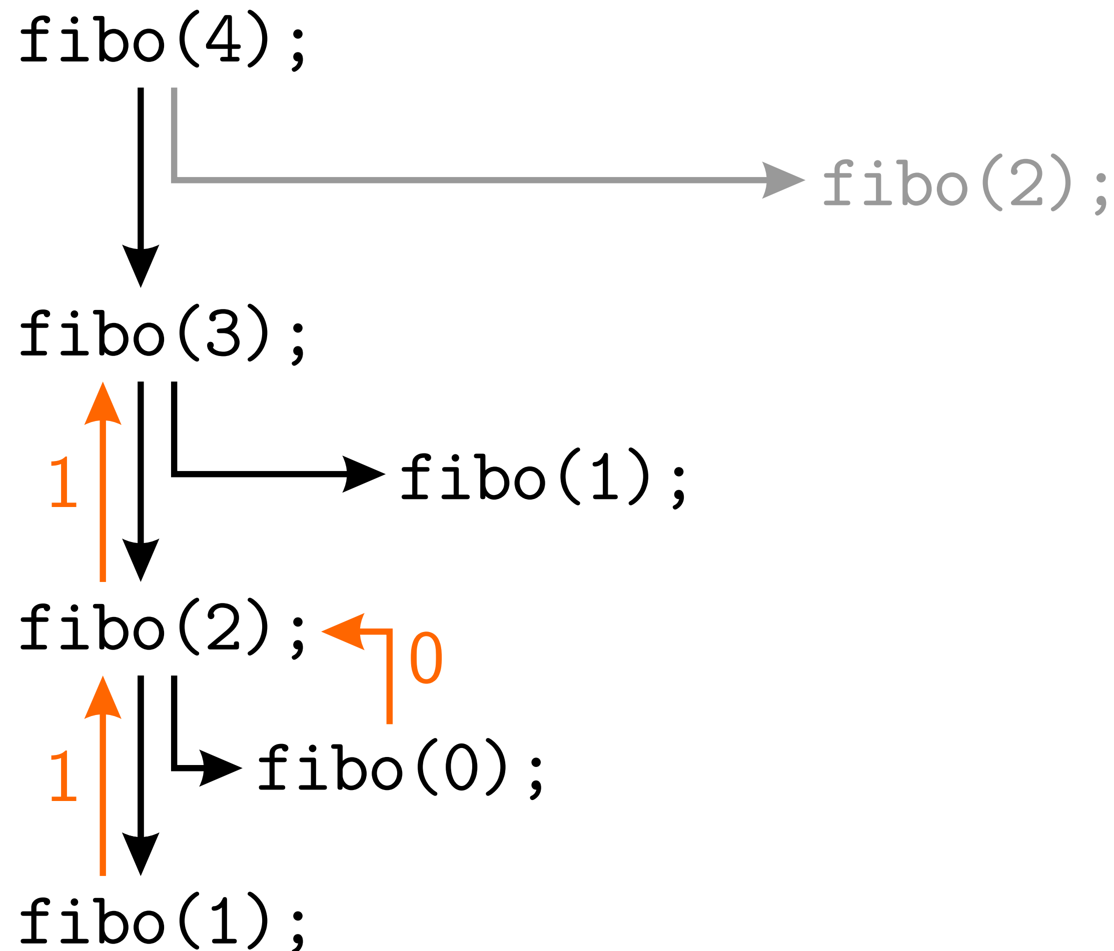
Appels récursifs multiples

Fibonacci : $F_n = F_{n-1} + F_{n-2}$ et $F_0 = 0, F_1 = 1$



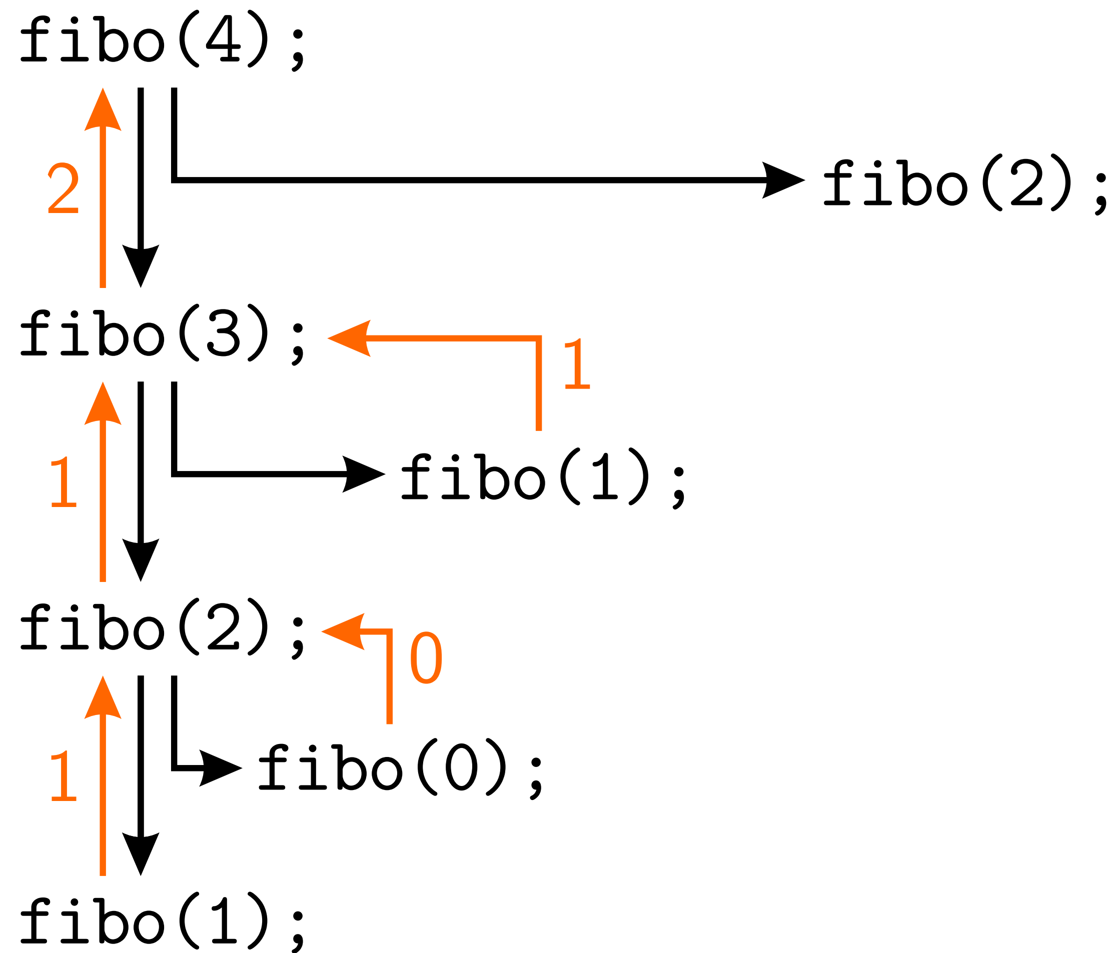
Appels récursifs multiples

Fibonacci : $F_n = F_{n-1} + F_{n-2}$ et $F_0 = 0, F_1 = 1$



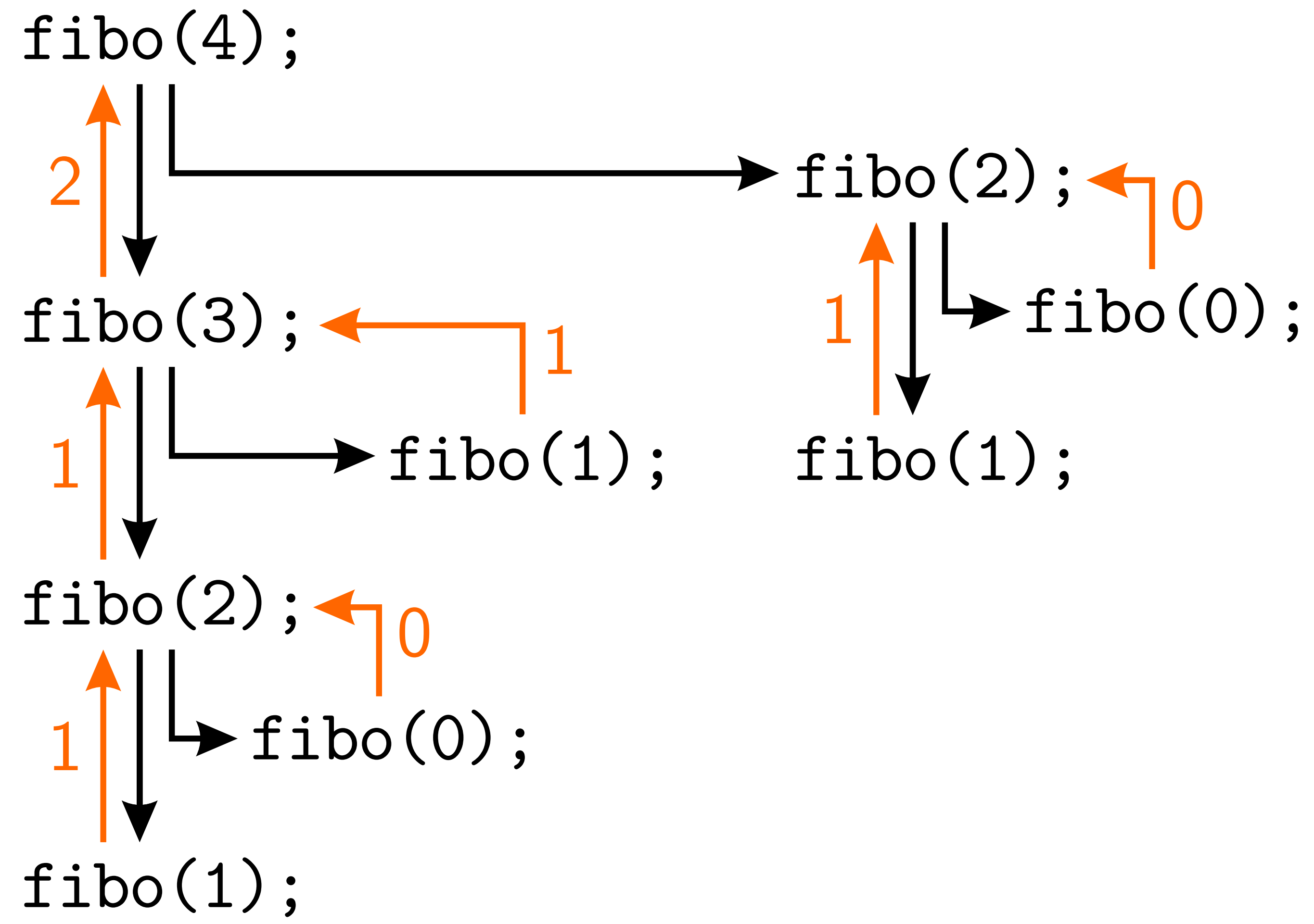
Appels récursifs multiples

Fibonacci : $F_n = F_{n-1} + F_{n-2}$ et $F_0 = 0, F_1 = 1$



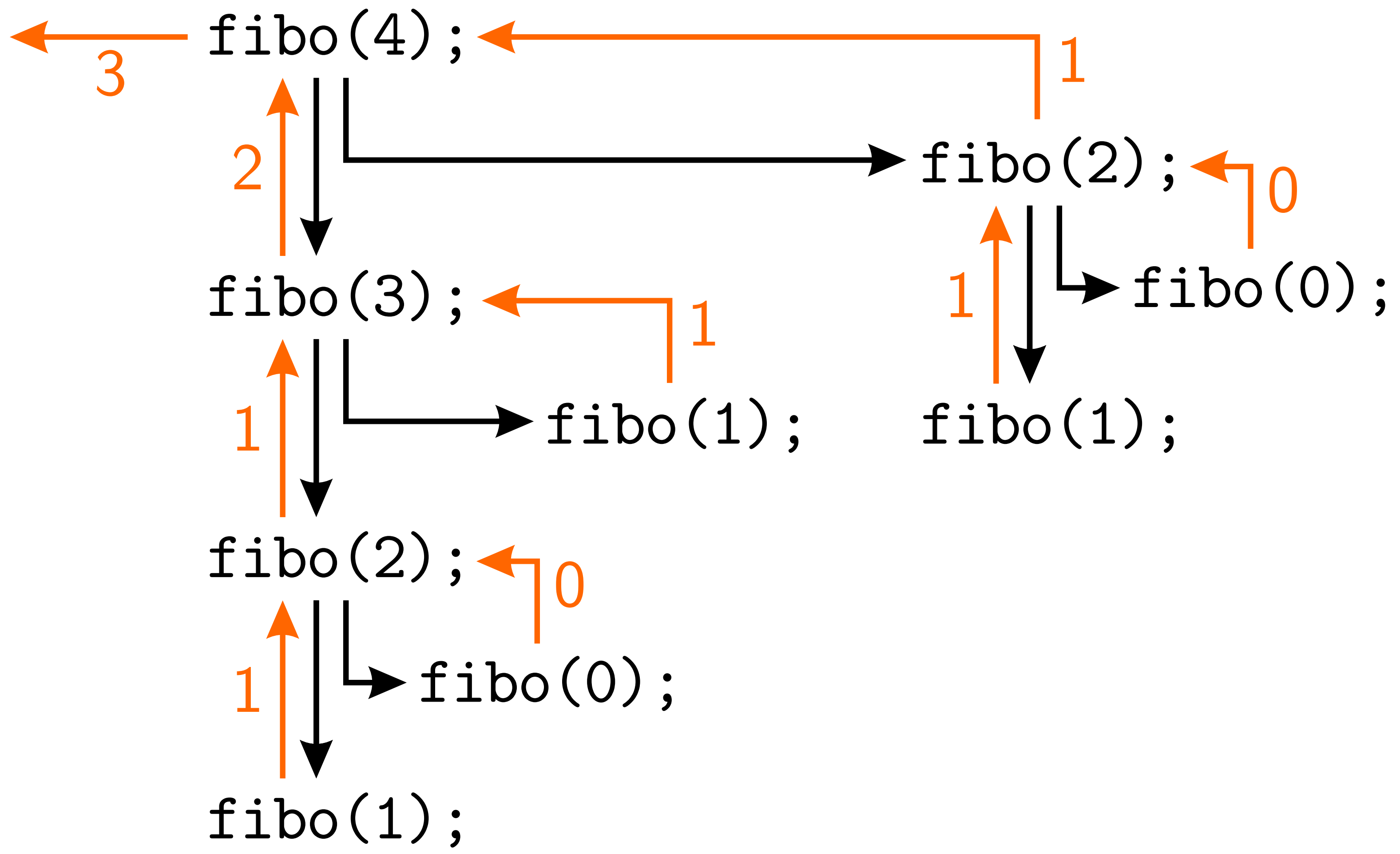
Appels récursifs multiples

Fibonacci : $F_n = F_{n-1} + F_{n-2}$ et $F_0 = 0, F_1 = 1$

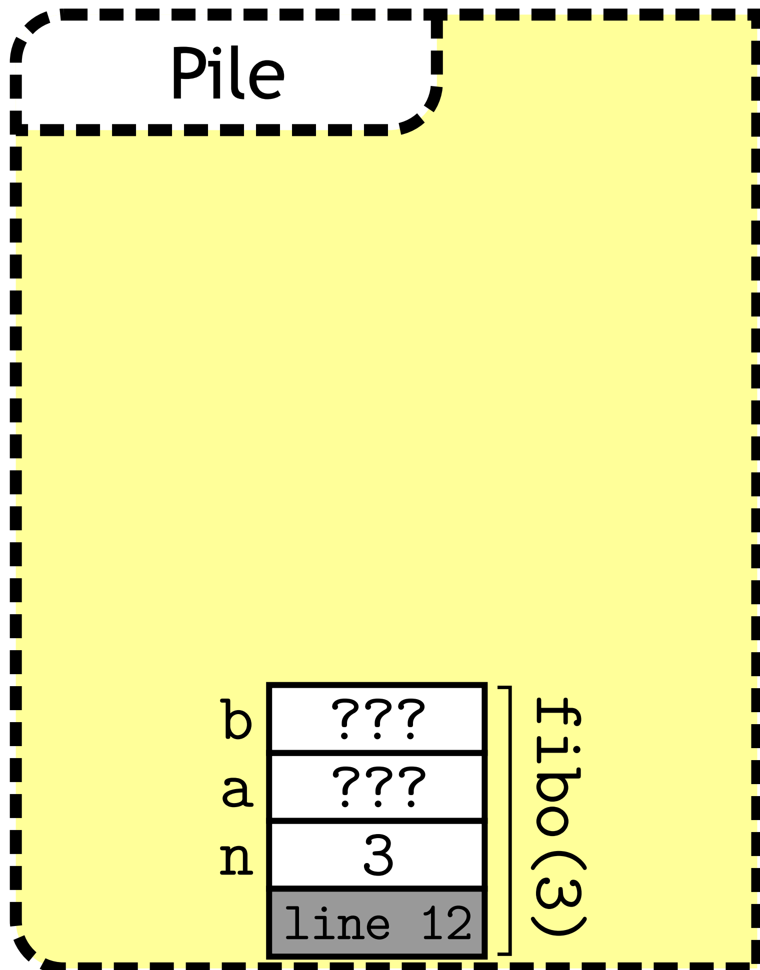


Appels récursifs multiples

Fibonacci : $F_n = F_{n-1} + F_{n-2}$ et $F_0 = 0, F_1 = 1$



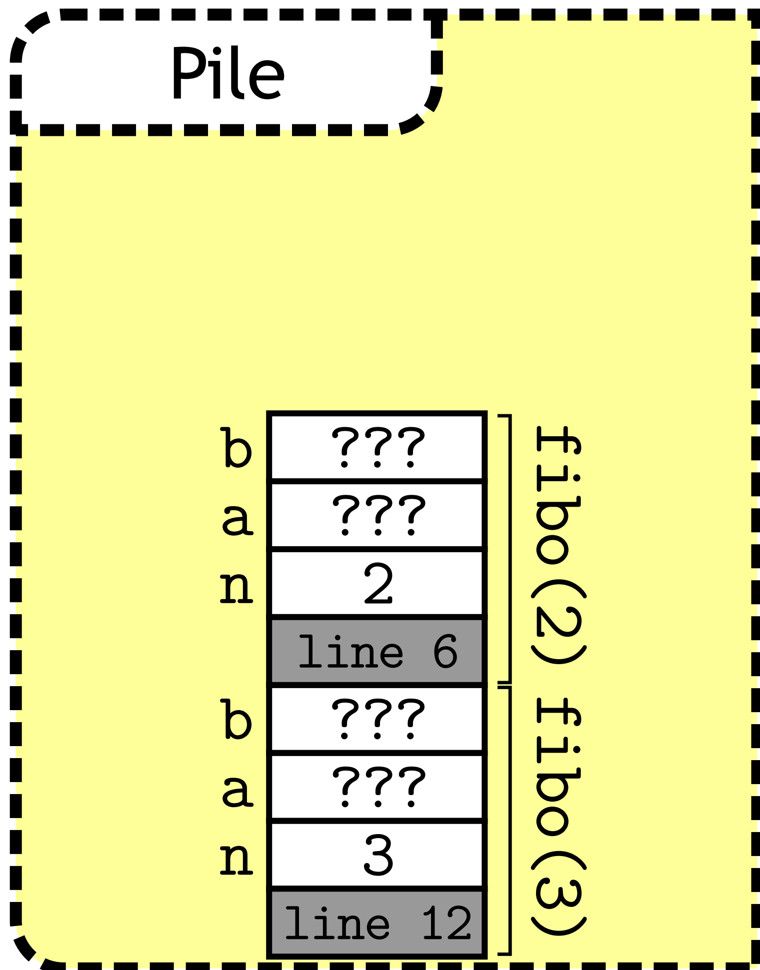
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6    a = fibo(n-1);
7    b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12 ▶ printf("%d\n", fibo(3));
13 }
```

- On commence par appeler `fibo(3)`.

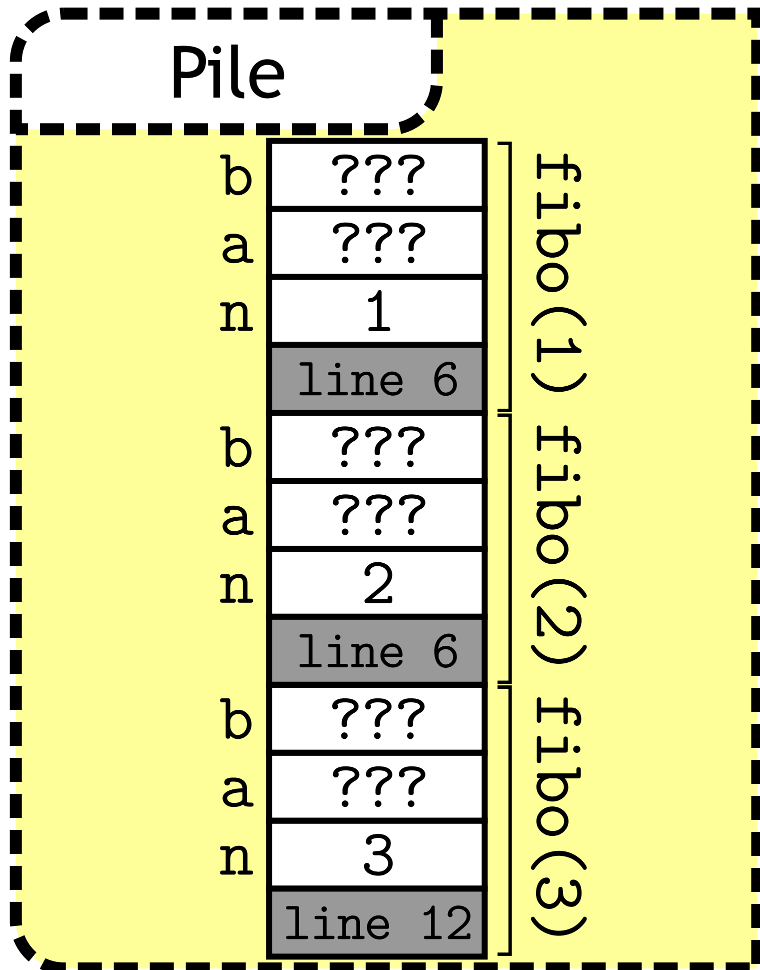
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6  ▶ a = fibo(n-1);
7    b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- `fibo(3)` appelle `fibo(2)`...

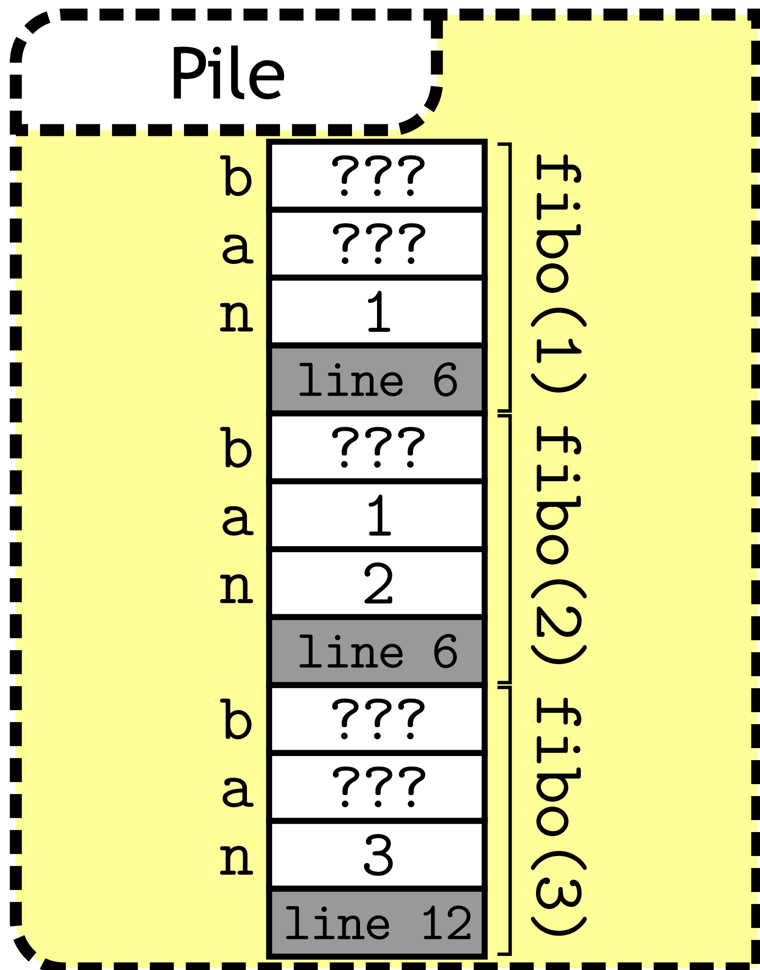
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6  ▶ a = fibo(n-1);
7    b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- `fibo(3)` appelle `fibo(2)` qui appelle `fibo(1)`.

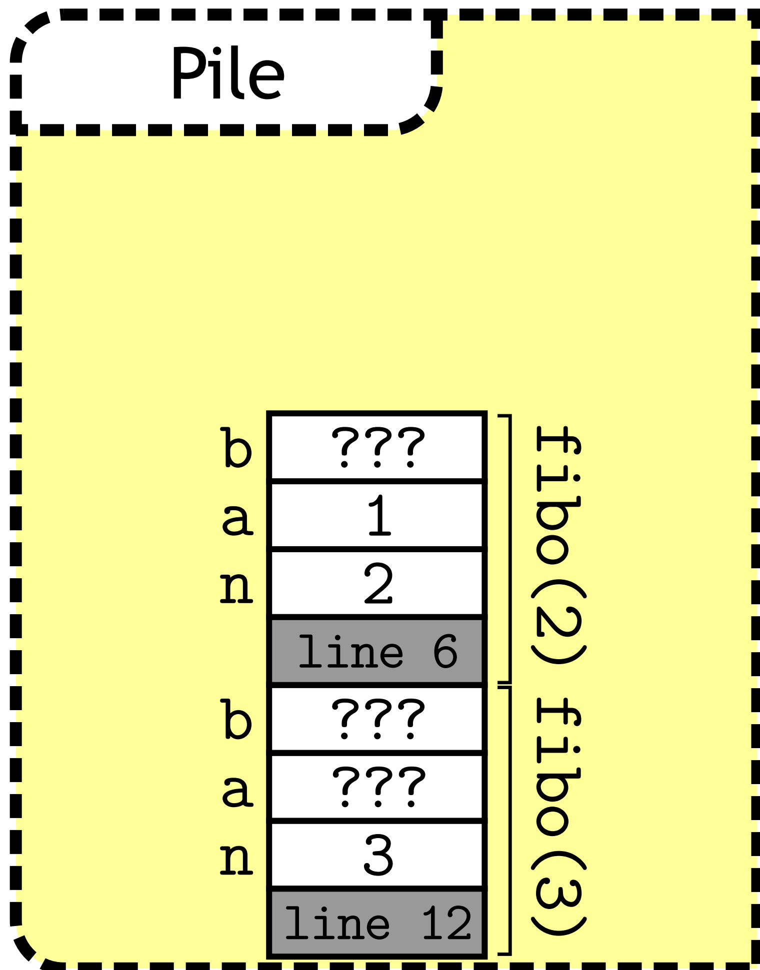
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4  ▶   return n;
5    }
6    a = fibo(n-1);
7    b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- Le test (n < 2) est vrai :
  - fibo(1) retourne 1 à la ligne 6
    → a = 1 dans fibo(2).

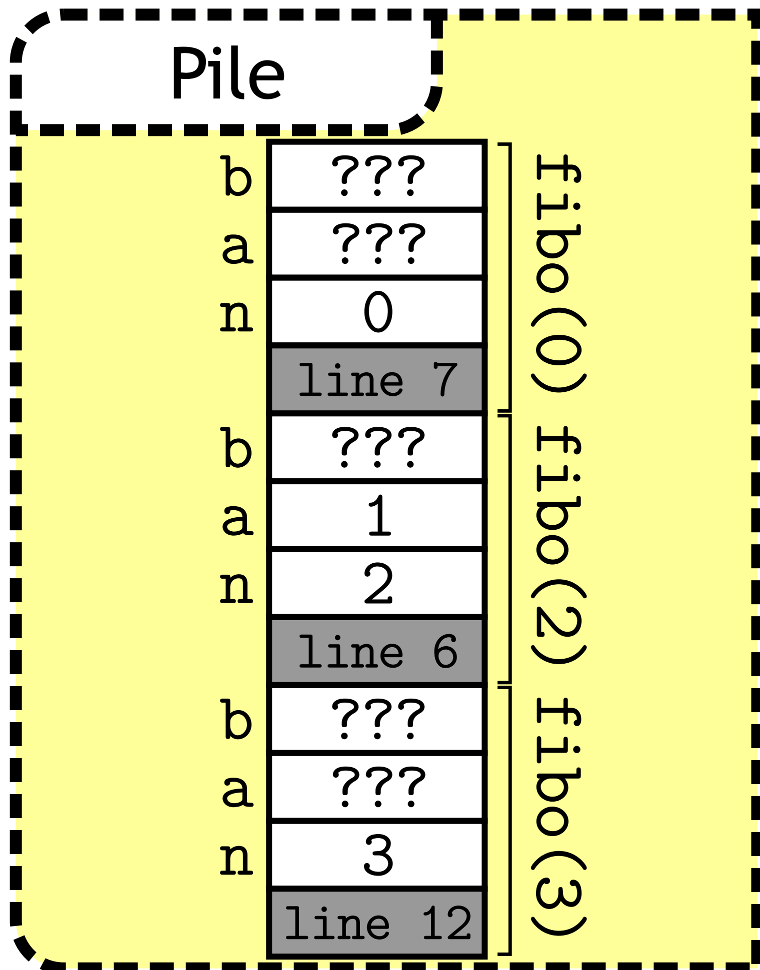
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6    a = fibo(n-1);
7  ▶ b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- On continue l'exécution de `fibo(2)`.

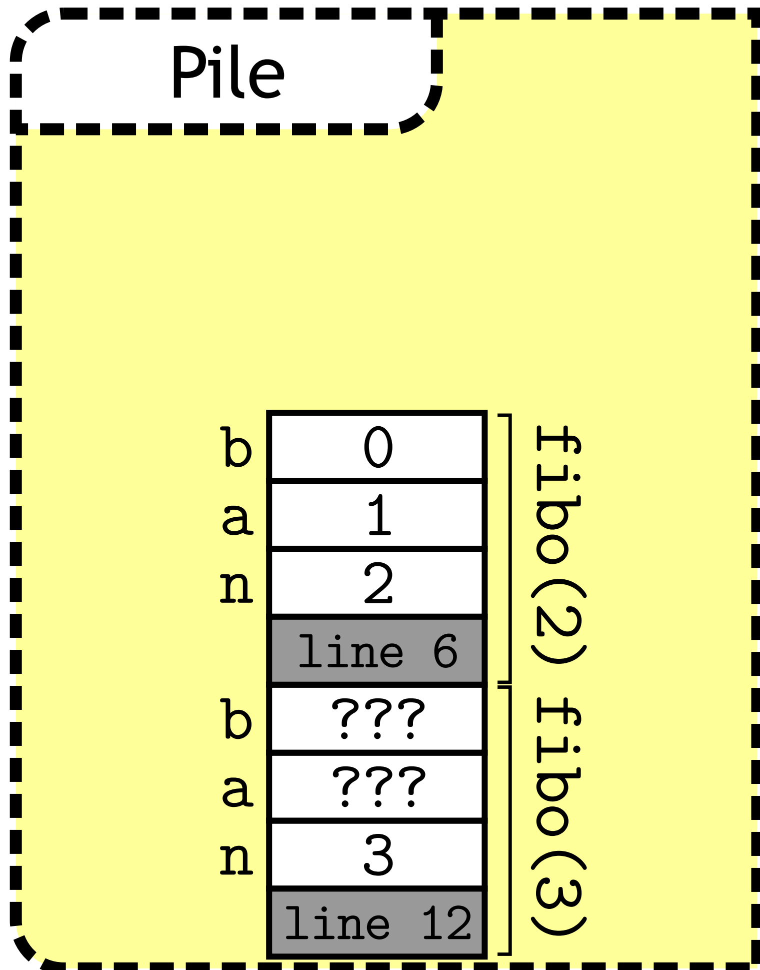
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6    a = fibo(n-1);
7  ▶ b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- fibo(2) appelle fibo(0)...

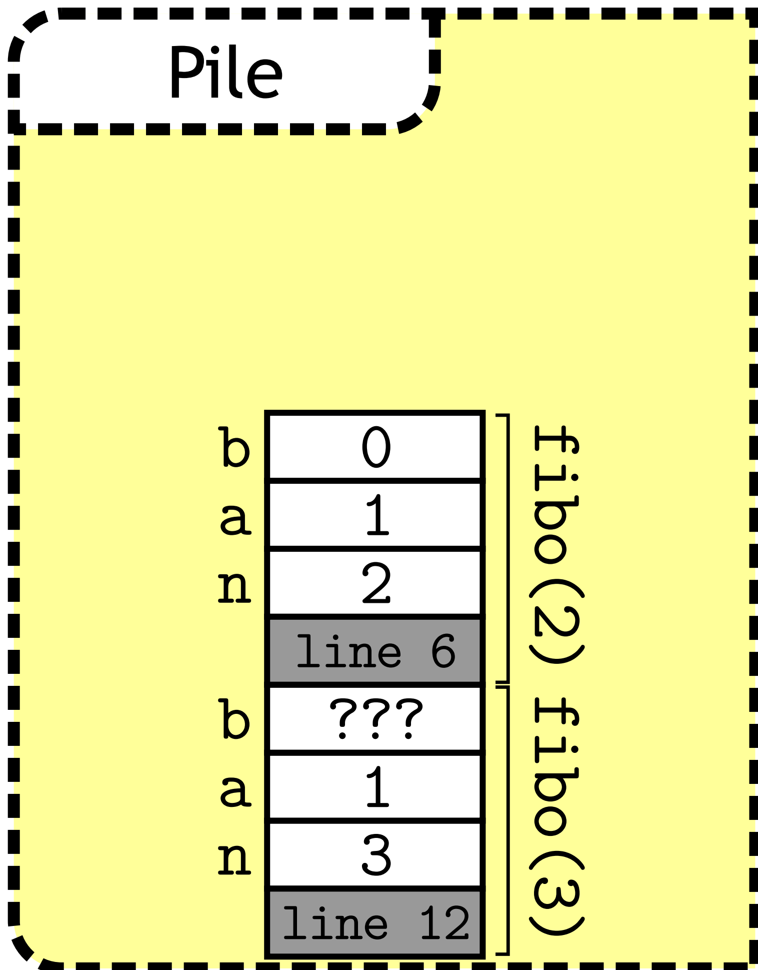
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6    a = fibo(n-1);
7  ▶ b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- `fibo(2)` appelle `fibo(0)` qui retourne 1.

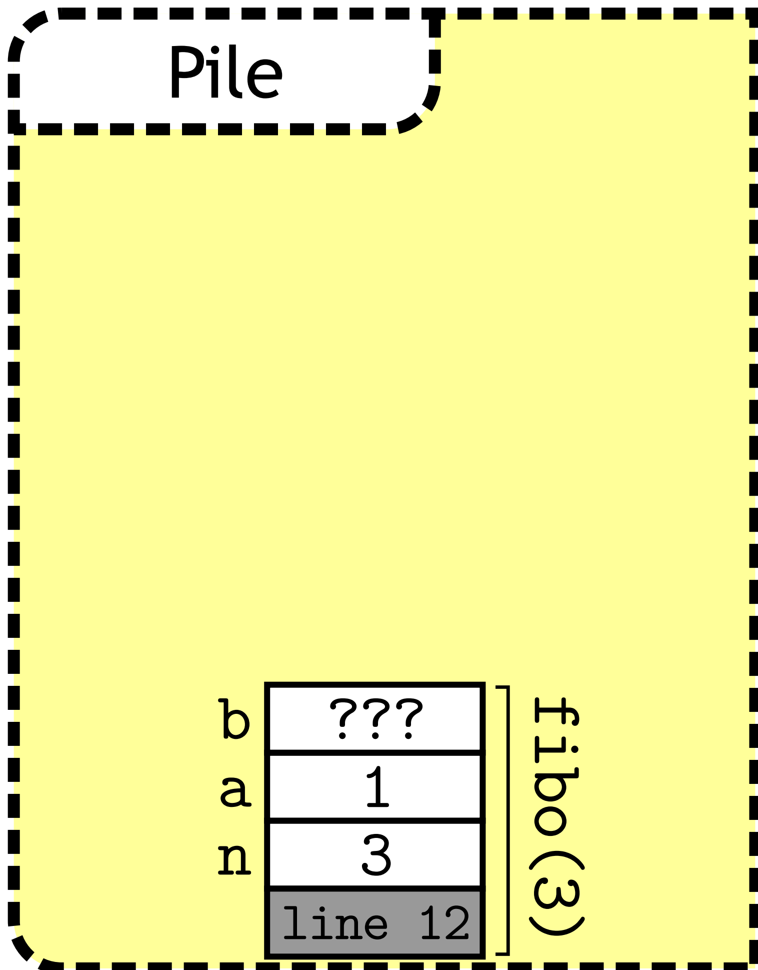
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6    a = fibo(n-1);
7    b = fibo(n-2);
8  ▶ return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- `fibo(2)` retourne `a + b` à la ligne 6 de `fibo(3)`.

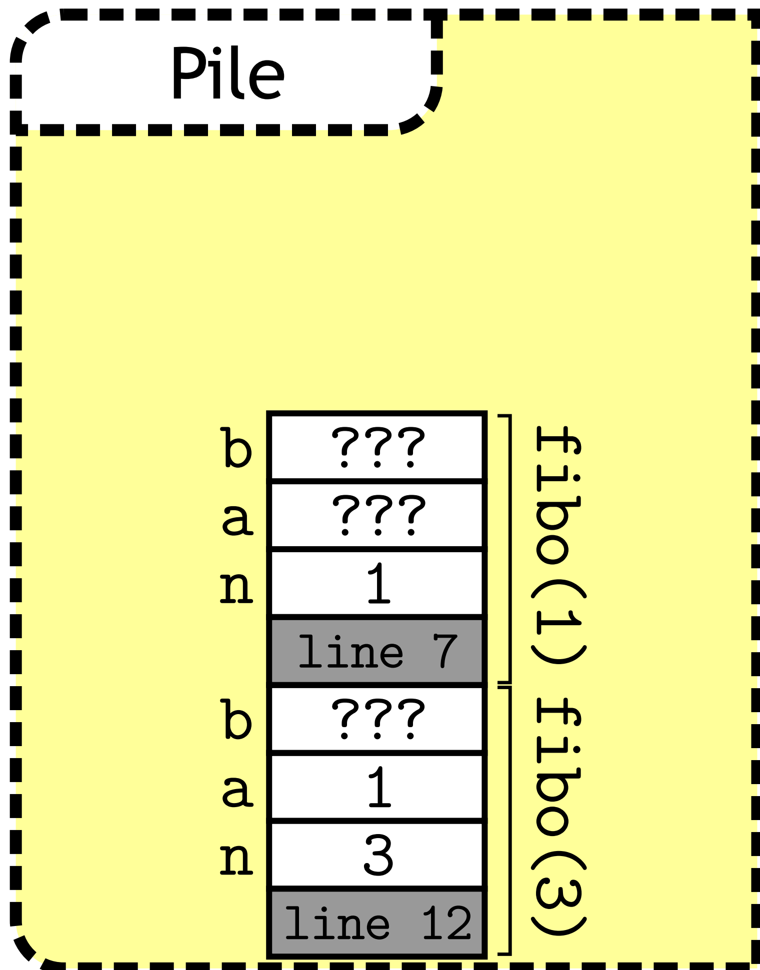
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6    a = fibo(n-1);
7  ▶ b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- L'exécution de `fibo(3)` continue.

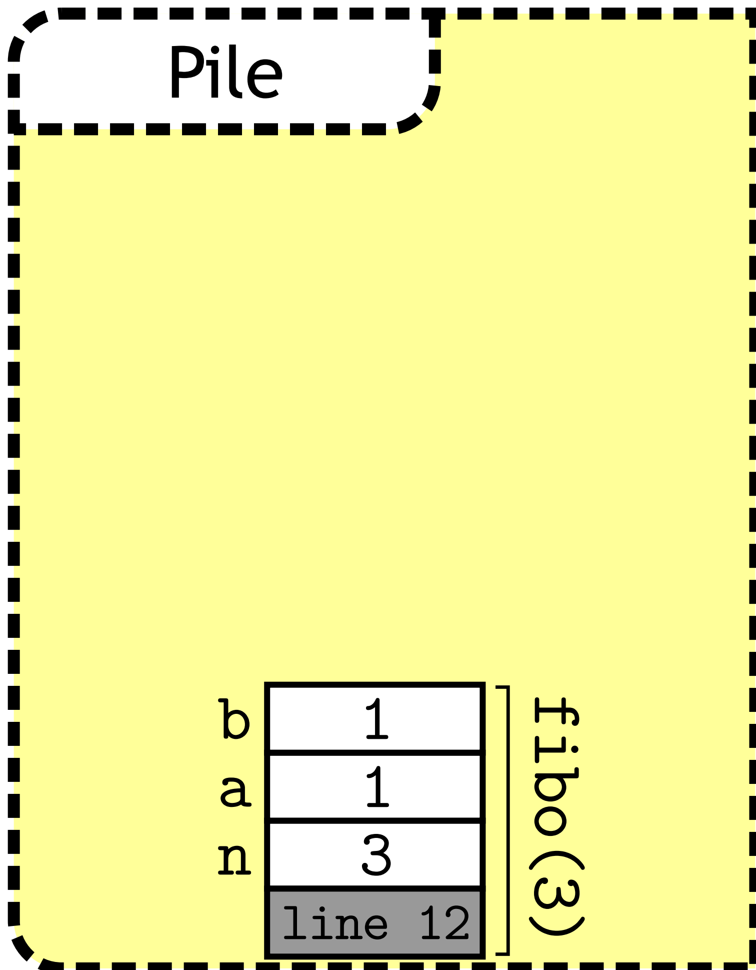
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6    a = fibo(n-1);
7  ▶ b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- Appel de `fibo(1)`.

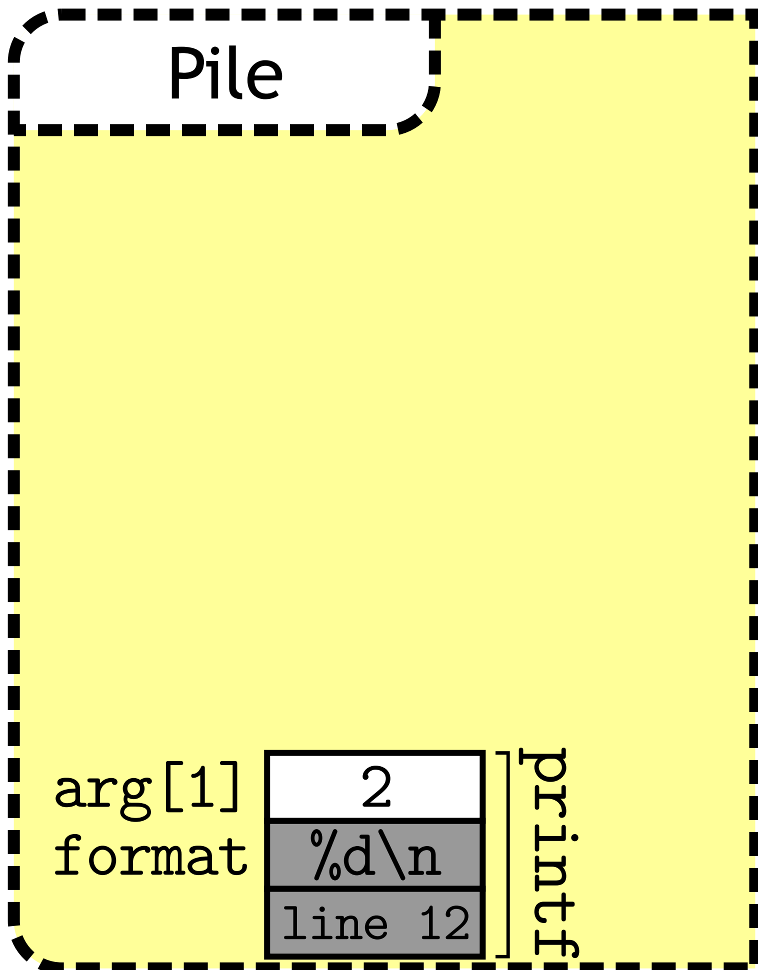
Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6    a = fibo(n-1);
7    b = fibo(n-2);
8  ▶ return a + b;
9  }
10
11 int main() {
12   printf("%d\n", fibo(3));
13 }
```

- L'exécution de `fibo(3)` va se terminer.

Déroulement d'un appel de fonction récursive




```
1  int fibo(unsigned int n) {
2    int a,b;
3    if (n < 2) {
4      return n;
5    }
6    a = fibo(n-1);
7    b = fibo(n-2);
8    return a + b;
9  }
10
11 int main() {
12 ▶ printf("%d\n", fibo(3));
13 }
```

- fibo(3) retourne 3 et on appelle donc printf.

Ce qu'il faut retenir de ce cours

- Les fonctions permettent de rendre le code plus modulaire :
  - décomposer un algorithme complexe en opérations simples,
    → simplifie l'analyse,l'optimisation ou la relecture,
  - écrire des librairies de fonctions,
    → le prototype suffit à utiliser une fonction.

- Les fonctions récursives simplifient l'écriture du code :
  - correspondent souvent à la façon naturelle d'écrire un algorithme,
  - bien utilisées, elles sont aussi rapide qu'une boucle
    → exemple de la récursion terminale.

⚠ Les appels de fonctions sont compliqués pour le compilateur, mais pas pour l'utilisateur,
→ il ne faut pas se priver !