§ → cours 1 — chall simple + synthèse → TD
→ cours 2 — chall complexe → 2 TD

Site Web → fonct. groupes.

https://pageperso.lis-lab.fr/alexandra.bac/enseignement/styled-2/styled-13/index.html

§3. Fonctions

ex: afficher 1 tableau → code → erreurs pour le valider : tester / débugger

évite de multiplier les erreurs

code : débuggé 1 fois — tester → pas dupliqué.

objectifs

Fonctions :

→ éviter la répétition de code

règle : un code n'est jamais répété — efficacité de maintenance — lisibilité (*)

ex : afficher le contenu d'un tableau ...

→ ce qui doit être exécuté plusieurs fois → fonction

fonction → définie 1 fois

fonction → appelée autant de fois que nécessaire

→ principe :

→ entrée(s) de la fonction : données (arguments) — variables, tableaux ....

→ sortie : 1 résultat retourné

I. Syntaxe des fonctions

Version simple

⇝ fonctions sont déclarées avant d'être | utilisées / appelées ← avant le main

```
#include ...
# define
fonctions
int main ()
{ :
}
```

⇝ syntaxe :

```
type_retour   mon_fct ( type1  var1 , type2  var2  .... )
{
        :  code
        return (...) ;
}
```

type du résultat renvoyé

liste des arguments (entrées)

prototype de la fonction

résultat

ex :

```
int   main ( )
{
        :
        return 0 ;
}
```

int : renvoie un entier

main : nom de la fct

( ) : pas d'arguments

return 0 : renvoie 0

ex : fonction calculant x^n — pas implémenté en C ...

v.1 ↓

$x \times \dots \times x$

entrée : 1 entier (n)
1 float (x)

rés / sortie : float (x^n)

⚠ en C ds math.h

pow : double x double e → double

int 3 : x^3

pow(x, y) — $x^y = e^{y \ln x}$

```
float puissance (float x, int n)
{
    ⋮
    return ... ;
}
```

→ <u>variables locales</u> : on peut déclarer des variables en début de la fct (idem ce qu'on faisait ds le main)

⚠ ces variables sont <u>locales</u> à la fonction
↓
détruites à la fin de la fonction

⚠ les arguments sont aussi des variables locales <u>initialisées</u> avec les valeurs passées à l'appel.

<u>ex</u> :

```
float puissance (float x, int n)
{
    float res = 1 ;
    int i ;
    for(i = 0; i < n ; i = i+1)
    {
        res = res * x ;
    }
    return res ;
}
```

variables locales

$\underbrace{x \times \cdots \times x}_{n \text{ fois}}$ x

nouveau mode de <u>retour</u> du résultat ⚠

≠ printf
↓
affiche et ne renvoie rien

<u>printf</u> → pas de printf ds les fonctions → retournent le résultat
→ dans <u>le</u> main

```c
#include <stdio.h>

// Fonctions

double puissance (double x, int n)
{
  double res = 1 ;
  int i ;

  for (i=0; i<n; i=i+1)
  {
    res = res * x ;
  }
  return res ;
}

// main

int main ()
{
  double a = 2.5, b ;
  int n = 4 ;

  // puissance(a, 5); // Fait un calcul - le renvoie - mais pas récupéré ...
  b = puissance(a,5) ; // résultat récupéré -> affecté à b
  printf("b puissance n : %g\n", puissance(b,n)) ;

  b = puissance(b,n) ;

  return 0 ;
}
```

float → réels sur 32 bits (32 0/1)
| 4 octets

double → 64 bits
| 8 octets
(précision x2
taille mém x2)

appel de la fonction
↓
on lui passe des arguments.

x local à puissance
initialisé

```c
float puissance (float x, int n)
{
  float res = 1;
  int i;
  ⋮

  return res;
}
```

(local
var x du main)

```c
int main ()
{ float a = 2.5, x ;

  x = puissance (a, 2); ①
  a = puissance (x, 5); ①
  ⋮

}
```

2 appels

puissance lancée
sur a / 2
arg 1 arg 2

???

que passe-t-on de a ?
→ valeur de a ? → C
→ variable a !

Quand on passe une variable en argument à une fct :

→ en C, C++, java :
on passe la valeur de la variable
passage par valeur.

→ en python ... :
on passe la variable
passage par référence

var. locale initialisée

```
float puissance (float x, int n)
{
    float res = 1;
    int i;
    ⋮
    return res;
}
```

2.5 ; 2 ; 1 ; 2.5² ; 5 ; 2.5²

```
int main ()
{   float a = 2.5, x;
    x = puissance (a, 2);   ①
    a = puissance (x, 5);   ②
    ⋮
}
```

var. locales détruites du main

puissance lancée sur a | 2
arg 1 | arg 2
on passe les valeurs
2.5 | 2
init. des var. arguments

arg 1 | arg 2
x | 5
on passe : valeurs
2.5² | 5
init. des var. arguments

⇝ type vide :

certaines fonctions ne renvoient rien

fonctions à effet de bord → entrées / sorties
→ modifications mémoires
→ affichages

ex : printf

$x = 2$ ⇝ modifie x ....

type vide en C : void

ex : fonction d'affichage d'un float :

```
void affiche_float (float x)
{
    printf("%f\n", x);
}
```

Conseil :

```
int main (void)
{
    ⋮
}
```

⇝ type tableaux :

pour passer un tableau en argument ⇝ type tableau..

type nom [ ]

nom est un tableau de cases type

```
#define N 10
```

ex :

```
void affiche_tab ( int T[], int n)
{
    int i;
    for (i=0 ; i < n ; i = i+1)
    { printf("%d ", T[i]);
    }
}
printf("\n");
```

⚠ var. muette de tab.

syntaxe pour le type tableau comme argument

```
int main ()
{
    int tab[N];
    int tab2[2*N];
    affiche_tab(tab, N);
    affiche_tab(tab2, 2*N);
```

création tableaux

ex:

```
void ma_fct (int x, int y) ①
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

2 3

var. locales init. avec les valeurs reçues

inverse x / y

③ x ← 3
y ← 2

④ Destruction des variables locales ~~x, y, tmp~~

```
int main ()
{
    int a = 2, b = 3;
    ma_fct (a, b);
```

on passe la valeur des var. comme args ①

2 3

⚠ passage par valeur en C

a, b sont inchangées

Règle temporaire

→ une fonction C ne peut pas modifier une variable passée en argument

sauf si c'est un tableau

→ passer des pointeurs ...

```
int ma_fct ( float x, int n ....)
{
    :
}
```

```
ma_fct (a, i)
```

modif. impossible

ex:

```
void saisin (int T[], int n)
{
    int i;
    for(i=0; i<n; i=i+1)
    {
        printf("Saisir case suivante :");
        scanf("%d", &T[i]);
    }
}
```

OK

T tableau ...

```
void saisin2 (int i)
{
    printf("Entrez i:");
    scanf("%d", &i);
}
```

i locale ...

Structure du code :

<u>Version standard</u> (≠ version simple du début) → déclaration + implémentation simultanées.

Différence :

① → déclaration du prototype de la fct → prototype : 1ère ligne de la fct spécifiant son type

prototype → types entrées / sorties

ex :

```
float puissance (float [x] opt
                 int [n]) ;
↳ float puissance (float, int) ;
```

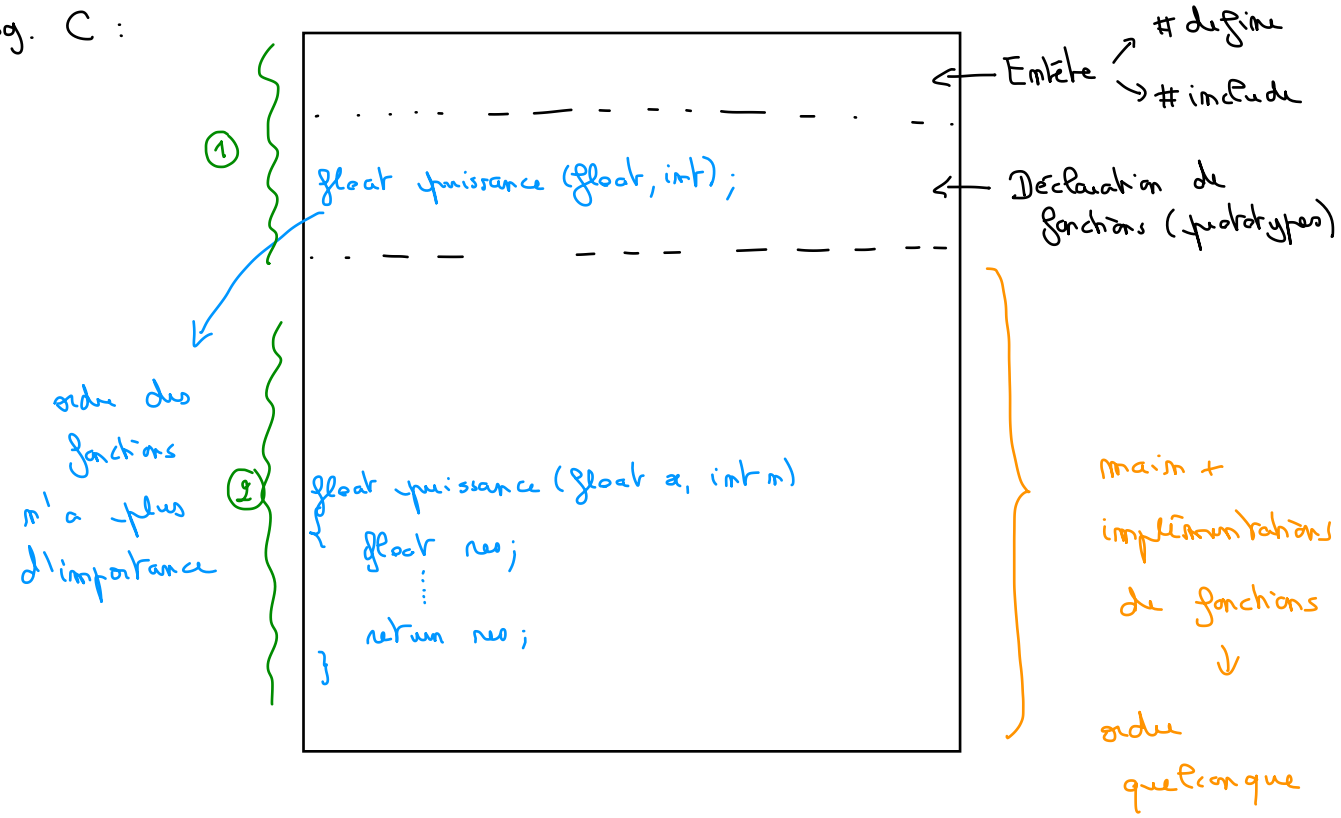
② → implémentation des fonctions
↳ code.

idem version "simple"

ex :

```
float puissance (float x, int n)
{
    ⋮ code
}
```

Prog. C :



Et après .....

⇝ évolution vers bibliothèques → code séparé en 2 fichiers

① → fichier .h contient entêtes / déclarations

② → fichier .c) implémentations

Règle : quand une fct reçoit un tableau en arg
↓
elle reçoit aussi sa longueur.

avant la compil. ≡ recherche / remplacer N → 10 ...

ex :

```
# include ...
# define N 10
# define M 30

void affiche (float tab[], int m)
{
    :
}

int main ()
{   float tab1[N], tab2[2*N], tab3[M];
    affiche (tab1, N);
    affiche (tab2, 2*N);
    affiche (tab3, M);
    :
}
```

← taille des tableaux manipulés / créés.

⚠ noms de var ≠ macros variables # define

m → N → 10

nom de variable ≠ ~~xe~~

10, 10, 30) OK

⚠ Parenthésages ...

(5 * 3) + 2 → précédence

à mettre ⚠

if ((T[0] == 5) && (i < n) || (...))

?

⚠ for / while ⟷ ~~return~~ → si on interrompt un while par un ~~return~~

↓

s'arrête à cause d'une des conditions

↓

code facile à comprendre / debugger

↓

oblige à lire tout le while en détails

ex: recherche x ds T ⇝ renvoie le n° case ou -1 si absent

V1 ⇝ cracra...

```
int recherche (float x, float T[], int n)
{
    int i=0;
    while (T[i]!=x)
    {  i=i+1;
       if (T[i]==x)
       { return i; }
       if (i==n)
       { return -1; }
    }
}
```

sortie du while ≡ car la cond. devient fausse ↓ $T[i]==x$

2 cas où sort de la fonction ....

↓

version propre :

```
int recherche (float x, float T[], int n)
{
    int i=0;
    while ((i<n) && (T[i]!=x))
    { i=i+1; }

    // Conditions d'arrêt
    if (i==n)   // x ∉ T
    { return -1; }
    else   // T[i]==x
    { return i; }
}
```

NON

Cond d'arrêt
$i==n$
ou
$T[i]==x$

⋮

```
int main()
{ int a;
  float T[N];
  float x;
  ⋮
  a = recherche(x,T,N);
  printf("%d\n",
         recherche(x,T,N))
```

II. Récursivité

Fonction récursive : fonction qui s'appelle elle-même.

↳ autorisé en C.

ex : calcul de x^n ⚠ algo à connaître

→ version naïve : $\underbrace{x \times \dots \times x}_{x^n}$ x → $n-1$ $\times$

Version optimale : récursive

x^n → cas 1 → $x^{2m} = (x^m)^2$ si n pair, $n = 2 \cdot m$ → $m = n/2$

cas 2 → $x^{m+1} = x \times x^m$ sinon $n = m+1$ (impair → $m-1$)

cas terminaux → x^n sans appel réc.

cas terminal

$n = 0 \rightsquigarrow x^0 = 1$

on finit toujours par tomber sur ce cas

$x^{21} \longrightarrow x \times x^{20}$ calc. récursif

appel réc. (*)

$x^{20} \longrightarrow (x^{10})^2$

$x^{10} \longrightarrow (x^5)^2$

$x^5 \longrightarrow x \times x^4$

renvoie $((x \times 1)^2)^2$

n décroît strictement

$x^4 \longrightarrow (x^2)^2$ $\quad (x \times 1)^2$ $\quad ((x \times 1)^2)^2$

$x^2 \longrightarrow (x^1)^2$ $\quad x \times 1$

renvoie $(x \times 1)^2$

$x^1 \longrightarrow x \times x^0 \longrightarrow x$

renvoie $x \times 1$

$x^0 \longrightarrow 1$

résultat renvoyé : 1

(*)

$$x \times \left(\left(x \times \left((x \times 1)^2\right)^2\right)^2\right)^2$$

① ② ③ ④ ⑤ ⑥ ⑦

x^{21}

$$x^{21} = \underbrace{x \times \cdots \times x}_{20 \times}$$

$$x^m \longrightarrow O(\log_2 m)$$

multiplications.

$$x^{128} = (x^{64})^2 = ((x^{32})^2)^2 = (x^{16})^{2^2} = (x^8)^{2^{2^2}}$$

$$= (x^4)^{2^{2^{2^2}}} = (x^2)^{2^{2^{2^{2^2}}}} = x^{2^{2^{2^{2^{2^{2^2}}}}}}$$

128 × $\qquad$ 7 ×

Méthode de conception de fonctions récursives

① Conception (brouillon)

a) ⇝ Identifier l'appel récursif

fct ⟶ fct sur des arguments plus petits

ex : fct(n) ⟶ fct(n-1)
fct(n/2) ....

b) ⇝ Déterminer les cas terminaux (cas où on peut conclure sans récursivité).

② Implémentation

⇝ commence par les cas terminaux ⚠

⇝ puis cas général

ex de x^n

① Conception

a) ⟶ appel réc : $x^n = (x^{n/2})^2$ si n pair ✓

$= x \times (x^{n-1})$ sinon

b) ⟶ cas terminaux :
un seul pour $n = 0$ $(x^0 = 1)$

②

```
float puissance (float x, int n)
{ float tmp;
  // Cas terminaux
  if (n == 0)
  { return 1; }
  else // cas général
```

```
{   if ((n % 2) == 0)
    {tmp = puissance (x, n/2) ;
     return (tmp * tmp) ; }
    else
    { tmp = puissance (x, n-1) ;
      return (x * tmp) ; }
  }
}
```

4

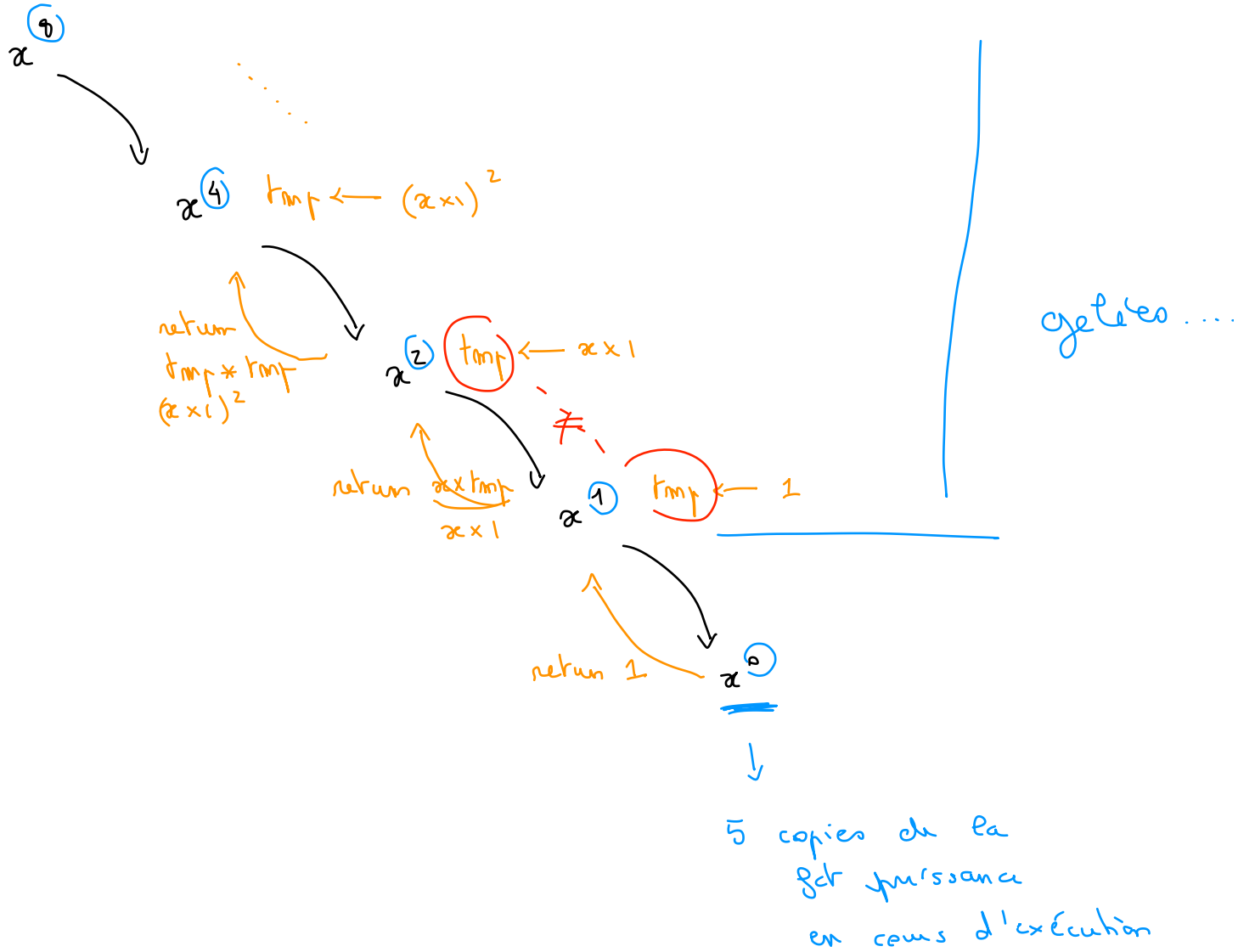
gdf

fin → renvoie 1 rés.

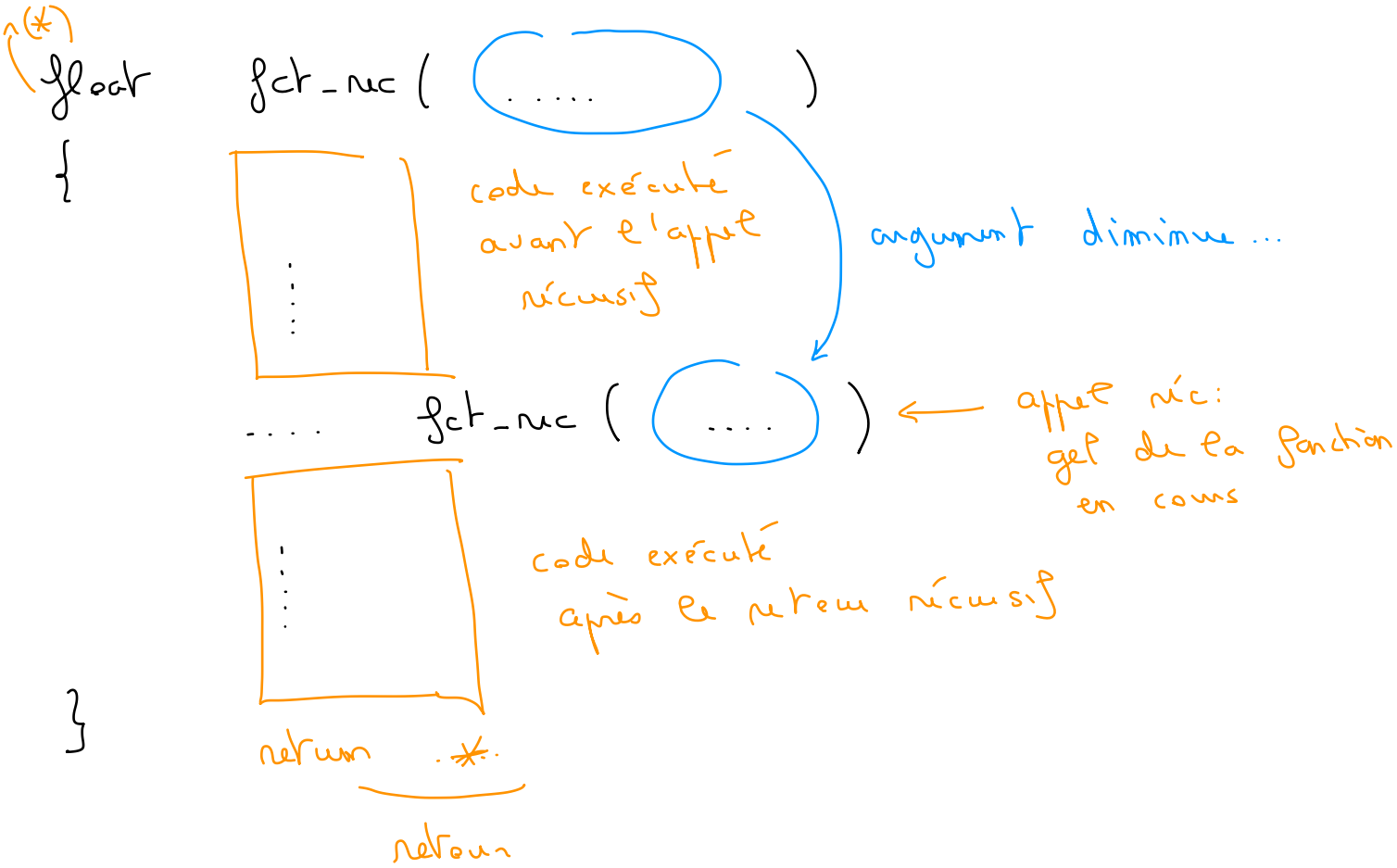
remonter le résultat

appel réc.

suite de la fct après le retour réc.



5 copies de la fct puissance en cours d'exécution


(*)
float fct_rec (.....)
{
code exécuté avant l'appel récursif
argument diminue ...
.... fct_rec (....)
appel réc: gel de la fonction en cours
code exécuté après le retour récursif
}
return .*.
return