

CI 4

4- LANGUAGE C

LES FONCTIONS 1 ET LA PORTÉE DES VARIABLES

Loïc Cuvillon

l.cuvillon@unistra.fr

Sommaire chapitre 4

Les fonctions 1

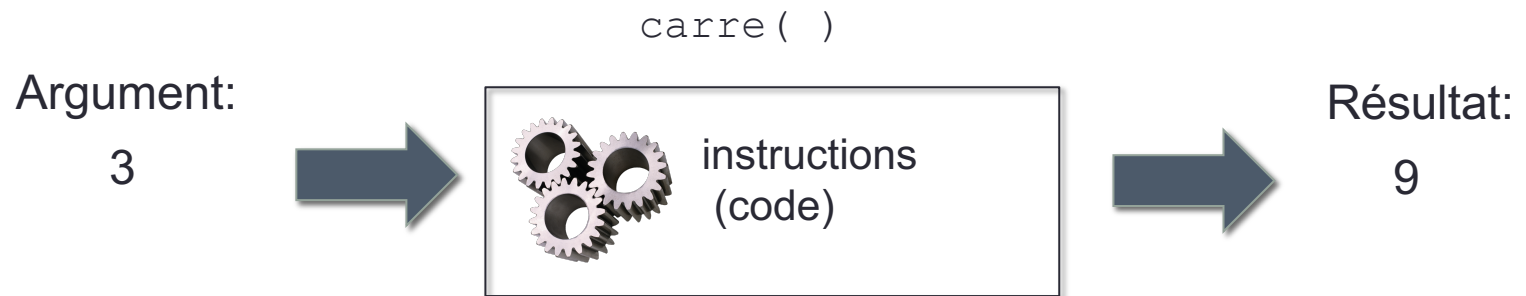
définition	178
passage de paramètres par valeur	187
mécanisme du passage par valeur	191
récurtivité	206
prototype d'une fonction	210

Rappel: Structure d'un programme C

- Un programme C est constitué d'une ensemble de fonctions (bloc d'instructions) qui peuvent s'invoquer l'une l'autre
- La première fonction à exécuter porte un nom spécifique :
`int main()`

Fonction dans un programme C

- Une fonction : élément du programme qui réalise un traitement des arguments/paramètres en entrée et retourne un résultat



- Usage des fonctions :
 - décomposition d'un programme en sous-programmes plus simples
 - réutilisation de fonctions existantes,
issues ou non d'une bibliothèque (telle la libC, libmath,...)

Définition d'une fonction

```
type nom_fonction ( type nom_arg1, type nom_arg2,... )  
{  
  
    /* Déclaration variables locales */  
  
    /* Instructions*/  
  
}
```

Définition d'une fonction

le type de la fonction = le type du résultat retourné

```
type nom_fonction ( type nom_arg1, type nom_arg2, ... )  
{  
  
    /* Déclaration variables locales */  
  
    /* Instructions */  
}
```

Définition d'une fonction

le `type` de la fonction = le `type` du résultat retourné

identifiant ou nom de la fonction :
même règle que pour le nom de variables (pas d'accent....)

```
type nom_fonction ( type nom_arg1, type nom_arg2, ... )  
{  
  
    /* Déclaration variables locales */  
  
    /* Instructions */  
}
```

Définition d'une fonction

le type de la fonction = le type du résultat retourné

identifiant ou nom de la fonction :
même règle que pour le nom de variables (pas d'accent....)

`type nom_fonction (type nom_arg1, type nom_arg2, ...)`

`{`

type et nom formels des arguments/paramètres de la fonction

`/* Déclaration variables locales */`

`/* Instructions*/`

`}`

Définition d'une fonction

le type de la fonction = le type du résultat retourné

identifiant ou nom de la fonction :
même règle que pour le nom de variables (pas d'accent...)

`type` ***nom_fonction*** (`type nom_arg1`, `type nom_arg2, ...`)

type et nom formels des arguments/paramètres de la fonction

```
{
    /* Déclaration variables locales */

    /* Instructions*/
}
```

le corps de la fonction: son bloc d'instruction { }

Définition d'une fonction

le type de la fonction = le type du résultat retourné

identifiant ou nom de la fonction :
même règle que pour le nom de variables (pas d'accent....)

`type nom_fonction (type nom_arg1, type nom_arg2, ...)`

type et nom formels des arguments/paramètres de la fonction

/ Déclaration variables locales */*



En C ANSI, variables déclarées obligatoirement en début de bloc

/ Instructions*/*

le corps de la fonction: son bloc d'instruction { }

Fonction C

```
int carre (int nbr)
{
    int result =0;
    result = nbr*nbr ;    /*mise au carre*/

    return result;
}
```

Fonction C



retourne en résultat/sortie une valeur entière (int)

```
int carre (int nbr)
```

```
{
```

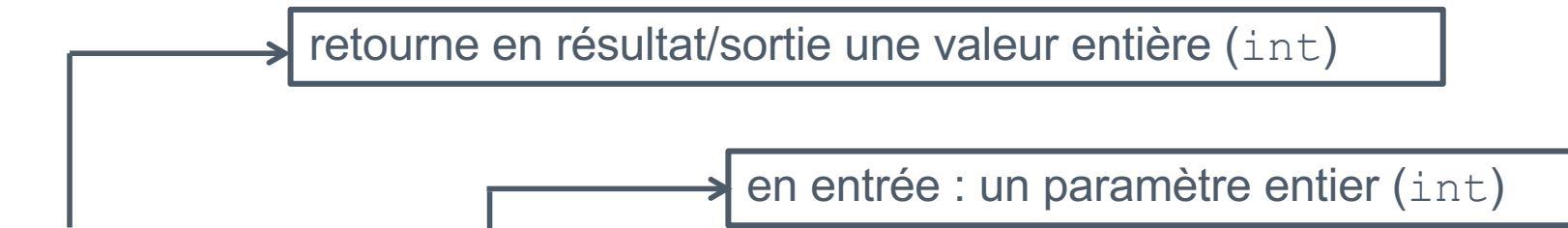
```
    int result =0;
```

```
    result = nbr*nbr ;    /*mise au carre*/
```

```
    return result;
```

```
}
```

Fonction C



```
int carre (int nbr)
{
    int result =0;
    result = nbr*nbr ;    /*mise au carre*/

    return result;
}
```

Fonction C

The diagram illustrates the components of the C function `carre`. Annotations are provided for the return type, the parameter, and the local variables.

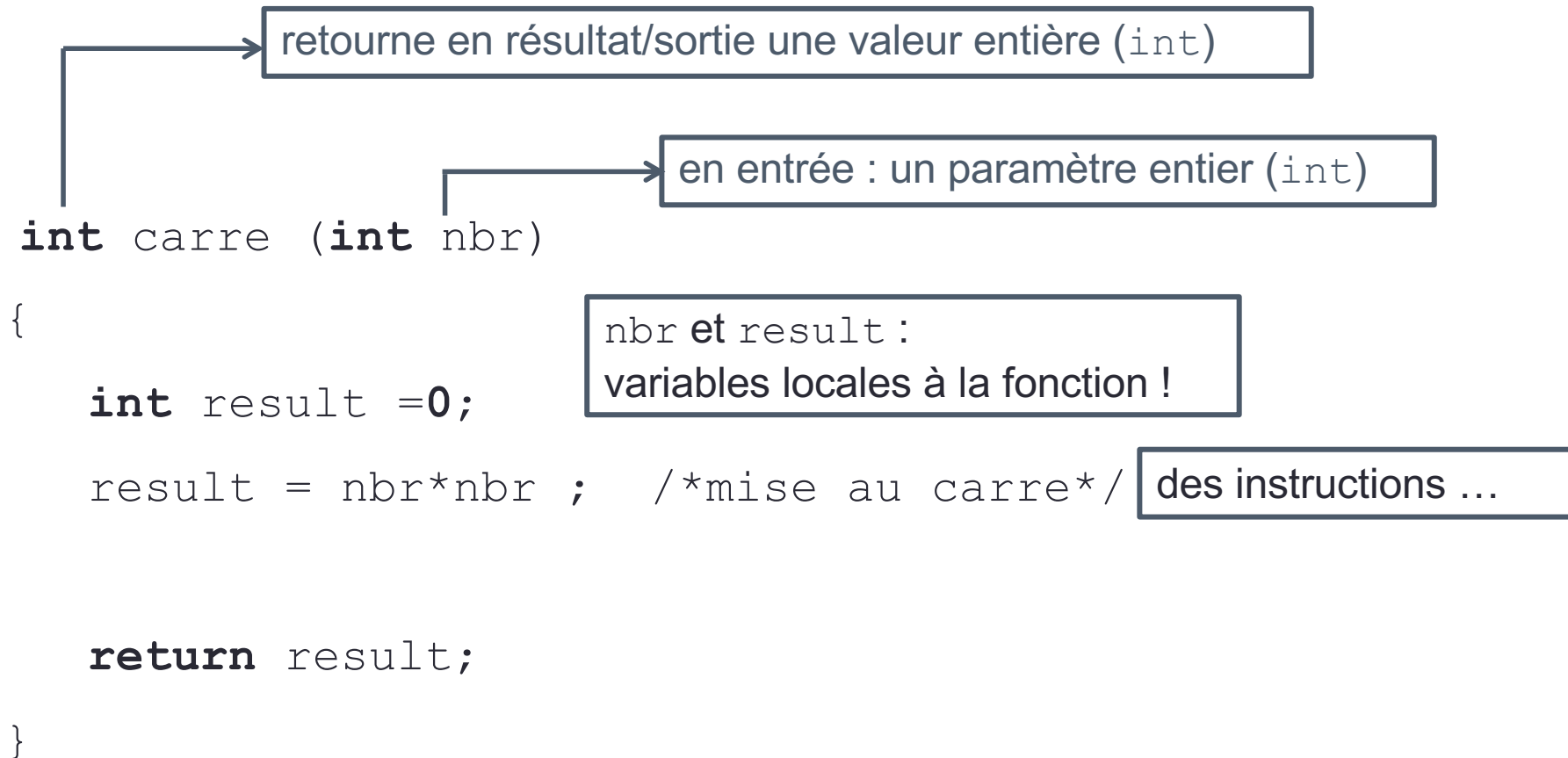
```
int carre (int nbr)
{
    int result =0;
    result = nbr*nbr ; /*mise au carre*/

    return result;
}
```

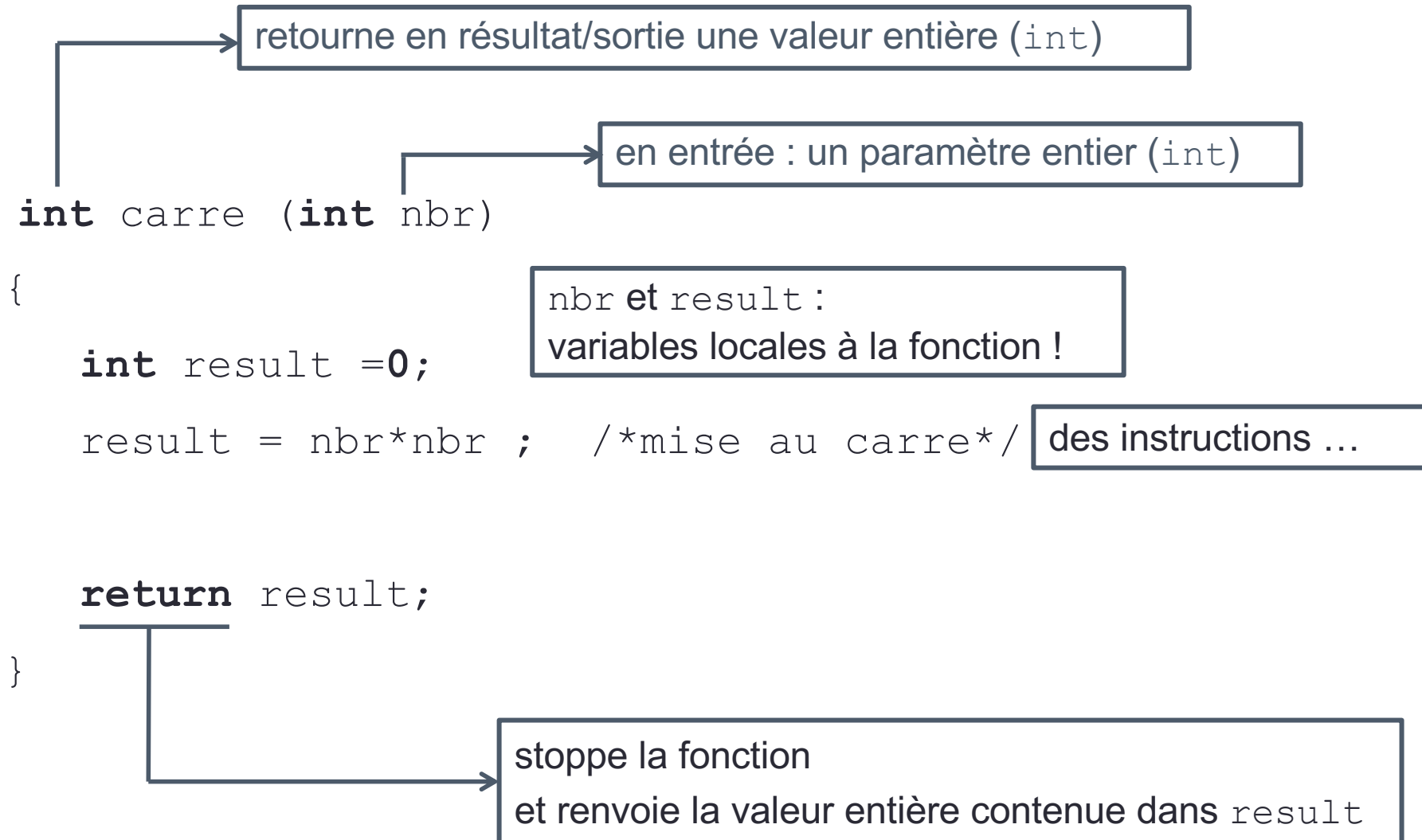
Annotations:

- retourne en résultat/sortie une valeur entière (`int`)
- en entrée : un paramètre entier (`int`)
- `nbr` et `result` : variables locales à la fonction !

Fonction C



Fonction C



Fonction C : variantes de `carre()`

- le paramètre/argument formel `nbr` est une variable locale à la fonction que l'on peut utiliser.

On peut réécrire la fonction précédente ainsi :

```
int carre (int nbr)
{
    nbr *= nbr;      /* idem. nbr=(nbr*nbr) */
    return nbr;
}
```

```
int carre (int nbr)
{
    return nbr*nbr;
}
```

return

retour à la fonction appelante avec la valeur à retourner

- **return** `value`;

ou

- **return** (`value`) ;

ou

- **return**; sans valeur pour une fonction qui ne renvoie rien (type `void`)
Dans ce cas, le return est facultatif !

- Fin d'exécution d'une fonction:
 - au premier **return** rencontré
 - ou si on arrive à la fin de sa définition « } »

return

/ calcul norme d'un vecteur 2D*/*

double norme2d (**double** a , **double** b)

{

double result, square;

 square = a*a+b*b

 if (square < **1e-16**)

 {

return 0;

 }



si condition vérifiée, la fonction se termine
avec 0 pour la valeur retournée.
(le reste du code n'est pas exécuté)

return sqrt (square) ; ———→ retourne la norme non nulle

}

le type `void`

- pour le type de la fonction - si la fonction ne retourne aucune valeur
- pour les arguments – si la fonction n'en prend aucun

```
void insiste (int n)
{
    int i;
    for (i=0; i < n; i++)
        { printf("Non! \n"); }
}
```

```
void bonjour(void)
{
    printf ("Bonjour");
}
```

Appel d'une fonction

```
nom_fonction (arg1, arg2, ...);
```

avec `arg1`, `arg2`, les arguments/paramètres effectifs
des variables (`x`,...) ou des constantes (3.4, "toto",...)

La valeur renvoyée peut être :

- affectée à une variable `x`

```
x=norme2d(a, 2);
```

- utilisée en argument d'une autre fonction

```
printf("cube vaut : %f", cube(a));
```

- utilisée dans un test

```
if ( cube(a) ) {
```

Appel d'une fonction

```
int carre ( int a )  
{  
    a = a*a;  
    return a;  
}
```

```
int main()  
{  
    int num =2,  num_cube=0;  
  
    num_cube = carre( num );  
  
    printf("Le carre de %d est %d.\n",  
           num, num_cube);  
  
    return 0;  
}
```

Appel d'une fonction

```
int carre ( int a )  
{  
    a = a*a;  
    return a;  
}
```

```
int main()  
{  
    int num =2,  num_cube=0;  
    num_cube = carre( num );  
    printf("Le carre de %d est %d.\n",  
           num, num_cube);  
    return 0;  
}
```

appel à `carre()`
avec `num` en argument/paramètre effectif

Appel d'une fonction

```
int carre ( int a )  
{  
    a = a*a;  
    return a;  
}
```

```
int main()  
{  
    int num =2,  num_cube=0;  
    num_cube = carre( num );  
    printf("Le carre de %d est %d.\n",  
           num, num_cube);  
    return 0;  
}
```

appel à `carre()`
avec `num` en argument/paramètre effectif

valeur de retour de `carre()`
affectée à `num_cube`

Appel d'une fonction

```
int carre ( int a )  
{  
    a = a*a;  
    return a;  
}
```

```
int main()  
{  
    int num =2,  num_cube=0;  
    num_cube = carre( num );  
    printf("Le carre de %d est %d.\n",  
           num, num_cube);  
    return 0;  
}
```

appel à `carre()`
avec `num` en argument/paramètre effectif

valeur de retour de `carre()`
affectée à `num_cube`

affiche à l'exécution :
Le carre de 2 est 4.

Appel de fonctions imbriquées

- La fonction `main()` est une fonction qui en appelle d'autres
- Une fonction peut contenir un appel à une autre fonction :

```
int carre ( int nbr )  
{  
    return nbr*nbr ;  
}  
  
double norme2d(int a, int b)  
{  
    double result;  
    result=sqrt( carre(a)+carre(b) );  
    return result;  
}
```

Sommaire chapitre 4

Les fonctions 1

définition	178
passage de paramètres par valeur	187
mécanisme du passage par valeur	191
récurtivité	206
prototype d'une fonction	210

Passage de paramètres par valeur

- En C, le mécanisme de passage des arguments est un passage par valeur.

les variables locales (arguments formels) de la fonction sont initialisées avec **une copie de la valeur des variables** (arguments effectifs) que la fonction appelante lui passe.

→ La fonction travaille sur ses variables, copies locales.

Passage de paramètres par valeur

```
int carre ( int a )  
{  
    a = a*a;  
    return a;  
}
```

```
int main()  
{  
    int num =2,  num_cube=0;  
  
    num_cube = carre( num );  
  
    printf("Le carre de %d est %d.\n",  
           num, num_cube);  
  
    return 0;  
}
```

1. appel à `carre()` avec `num`
en argument/paramètre effectif

Passage de paramètres par valeur

2. la variable locale `a` de `carre()`
est initialisée avec la valeur (2) de `num`

```
int carre ( int a )  
{  
    a = a*a;  
    return a;  
}
```

```
int main()  
{  
    int num =2,  num_cube=0;  
    num_cube = carre( num );  
    printf("Le carre de %d est %d.\n",  
           num, num_cube);  
    return 0;  
}
```

1. appel à `carre()` avec `num`
en argument/paramètre effectif

Passage de paramètres par valeur

```
int carre ( int a )
{
    a = a*a;
    return a;
}
```

2. la variable locale `a` de `carre()` est initialisée avec la valeur (2) de `num`

3. $a=2*2=4$. Valeur de `a` modifiée.
Note: valeur de `num` de la fonction appelante pas modifiée, on travaille sur la variable locale `a`.

```
int main()
{
    int num =2, num_cube=0;

    num_cube = carre( num );

    printf("Le carre de %d est %d.\n",
           num, num_cube);

    return 0;
}
```

1. appel à `carre()` avec `num` en argument/paramètre effectif

Passage de paramètres par valeur

```
int carre ( int a )
{
    a = a*a;
    return a;
}
```

2. la variable locale `a` de `carre()` est initialisée avec la valeur (2) de `num`

3. $a=2*2=4$. Valeur de `a` modifiée.
Note: valeur de `num` de la fonction appelante pas modifiée, on travaille sur la variable locale `a`.

4. retour de la valeur de `a`

```
int main()
{
    int num =2, num_cube=0;

    num_cube = carre( num );

    printf("Le carre de %d est %d.\n",
           num, num_cube);

    return 0;
}
```

1. appel à `carre()` avec `num` en argument/paramètre effectif

Passage de paramètres par valeur

```
int carre ( int a )
{
    a = a*a;
    return a;
}
```

2. la variable locale `a` de `carre()` est initialisée avec la valeur (2) de `num`

3. $a=2*2=4$. Valeur de `a` modifiée.
Note: valeur de `num` de la fonction appelante pas modifiée, on travaille sur la variable locale `a`.

4. retour de la valeur de `a`

```
int main()
{
    int num =2, num_cube=0;

    num_cube = carre( num );

    printf("Le carre de %d est %d.\n",
           num, num_cube);

    return 0;
}
```

1. appel à `carre()` avec `num` en argument/paramètre effectif

5. valeur de retour de `carre()` affectée à `num_cube`

Passage de paramètres par valeur

```
int carre ( int a )
{
    a = a*a;
    return a;
}
```

2. la variable locale `a` de `carre()` est initialisée avec la valeur (2) de `num`

3. $a=2*2=4$. Valeur de `a` modifiée.
Note: valeur de `num` de la fonction appelante pas modifiée, on travaille sur la variable locale `a`.

4. retour de la valeur de `a`

```
int main()
{
    int num =2, num_cube=0;

    num_cube = carre( num );

    printf("Le carre de %d est %d.\n",
           num, num_cube);

    return 0;
}
```

1. appel à `carre()` avec `num` en argument/paramètre effectif

5. valeur de retour de `carre()` affectée à `num_cube`

6. affiche de `num` et `num_cube`
 Le carre de 2 est 4.

Passage de paramètres par valeur

```
void carre ( int a )  
{  
    printf("-recu: %d\n", a);  
    a=a*a;  
    printf("-modifie: %d\n", a);  
}
```

```
int main()  
{  
    int num =2;  
  
    printf("num avant: %d\n", num);  
    carre(num);  
    printf("num apres: %d\n", num);  
    return 0;  
}
```

affichage du programme :

```
num avant: 2  
-recu: 2  
-modifie: 4      ←  
num apres: 2      ←  !
```

Passage de paramètres par valeur

```
void carre ( int a )
{
    printf("-recu: %d\n", a);
    a=a*a;
    printf("-modifie: %d\n", a);
}
```

```
int main()
{
    int num =2;
```

```
    printf("num avant: %d\n", num);
    carre(num);
    printf("num apres: %d\n", num);
    return 0;
```

```
}
```

affichage du programme :

```
num avant: 2
-recu: 2
-modifie: 4      ←
num apres: 2      ← !
```

- la fonction reçoit dans sa variable locale a (argument formel) une copie de la valeur de num
- modifier a ne modifie pas num du main()

Sommaire chapitre 4

Les fonctions 1

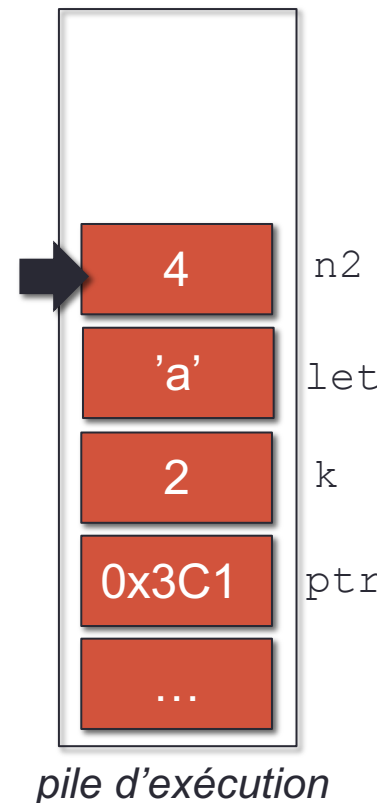
définition	178
passage de paramètres par valeur	187
mécanisme du passage par valeur	191
récurtivité	206
prototype d'une fonction	210

La pile d'exécution

Le nombre de variables d'un programme en exécution (processus) est inconnu à l'avance, il dépend :

- des tests if,
- du nombre d'itérations d'une boucle
- du nombre d'appels récursifs à une fonction

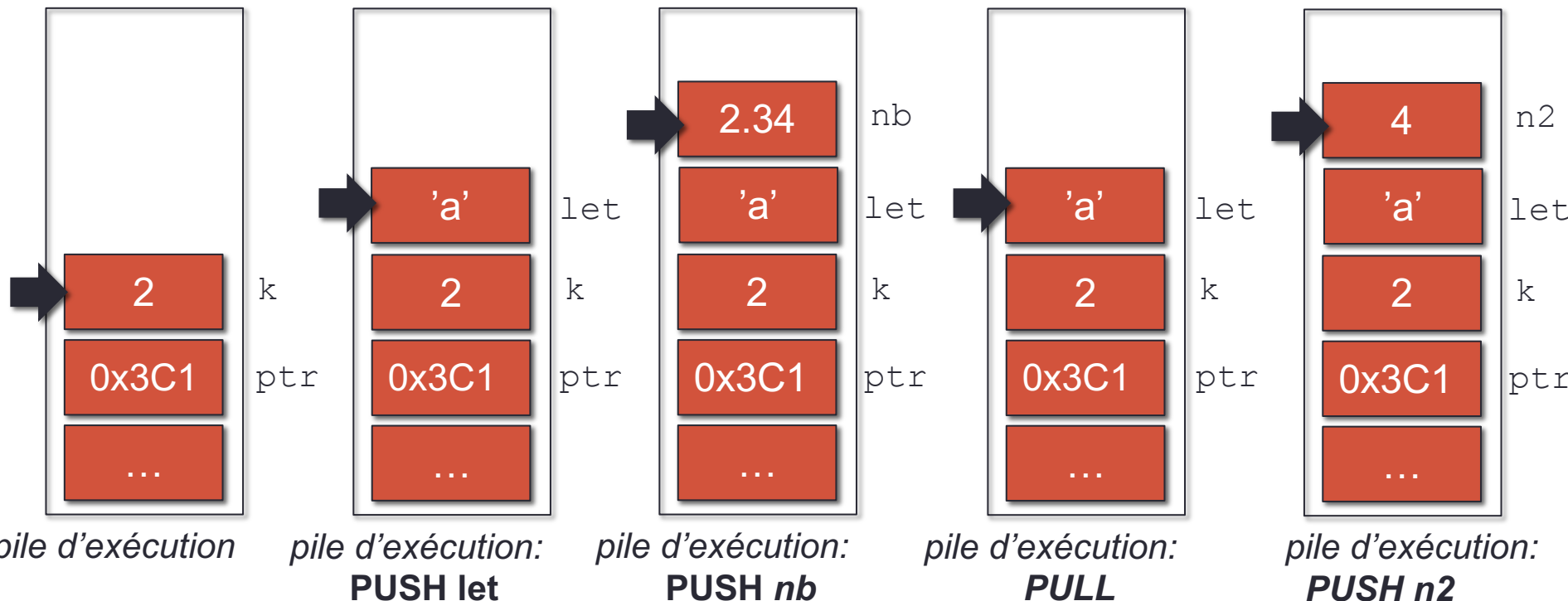
Pour créer de nouvelles variables et les organiser, une zone de la mémoire qui peut croître à la demande est utilisée en interne : la pile d'exécution.



La pile d'exécution

La pile d'exécution :

- toute nouvelle variable est créée (poussée/push) sur le sommet de la pile,
- seule la variable sur le sommet de la pile peut être supprimée (dépilée/pull)

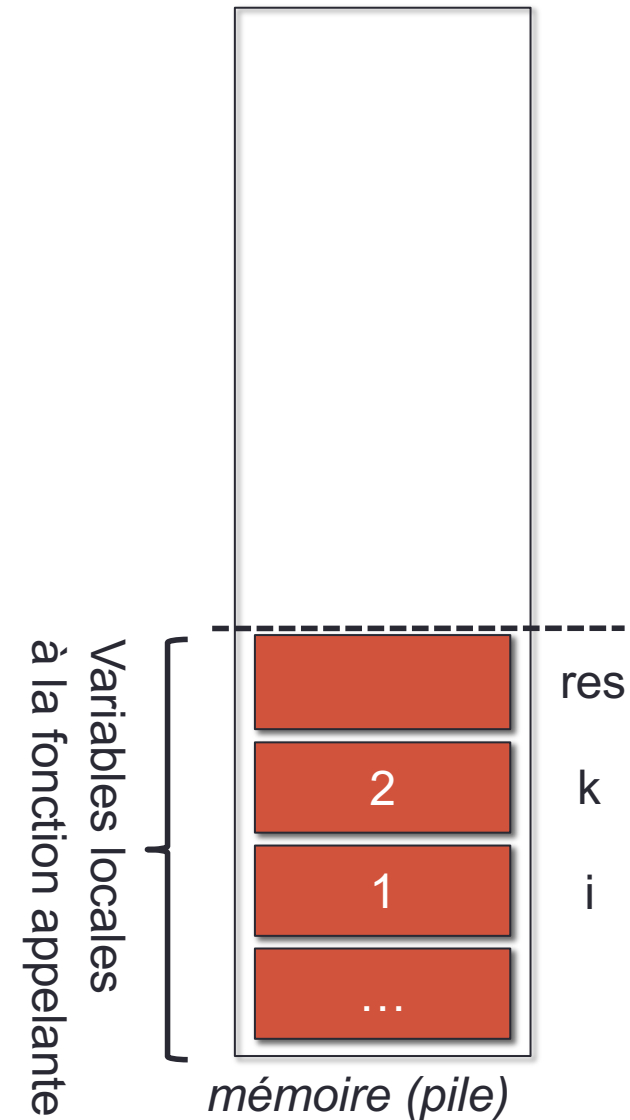


Passage de paramètres par valeur

```
{  
    int i = 1, k = 2;  
    float res;  
    . . .  
    res = norm(k,i);  
    . . .  
}
```

0

```
float norm(int i, int j)  
{  
    float tmp=0;  
    i = i*i;  
    j = j*j;  
    return ( sqrt(i+j) );  
}
```



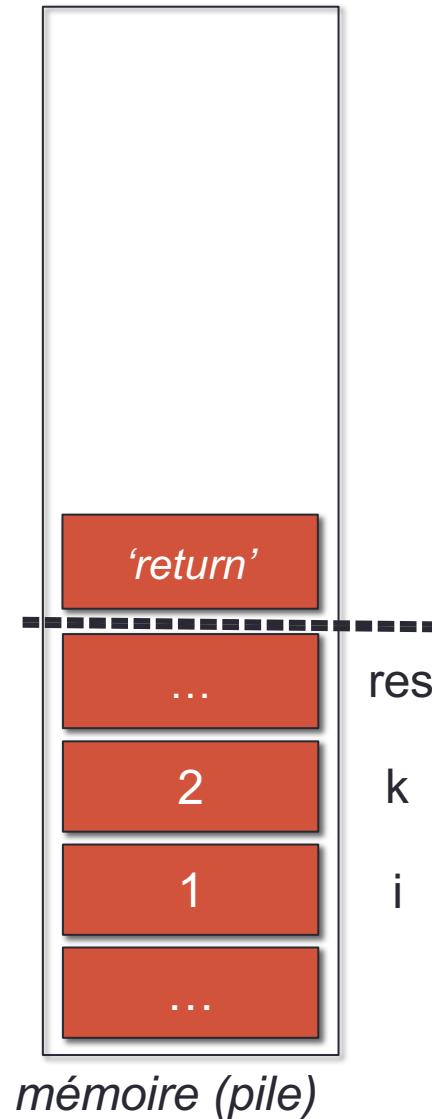
Passage de paramètres par valeur

```
{
    int i = 1, k = 2;
    float res;
    . . .
    res = norm(k, i);
    . . .
}
```

1

```
float norm(int i, int j)
{
    float tmp=0;
    i = i*i;
    j = j*j;
    return ( sqrt(i+j) );
}
```

Variables locales
à la fonction appelante



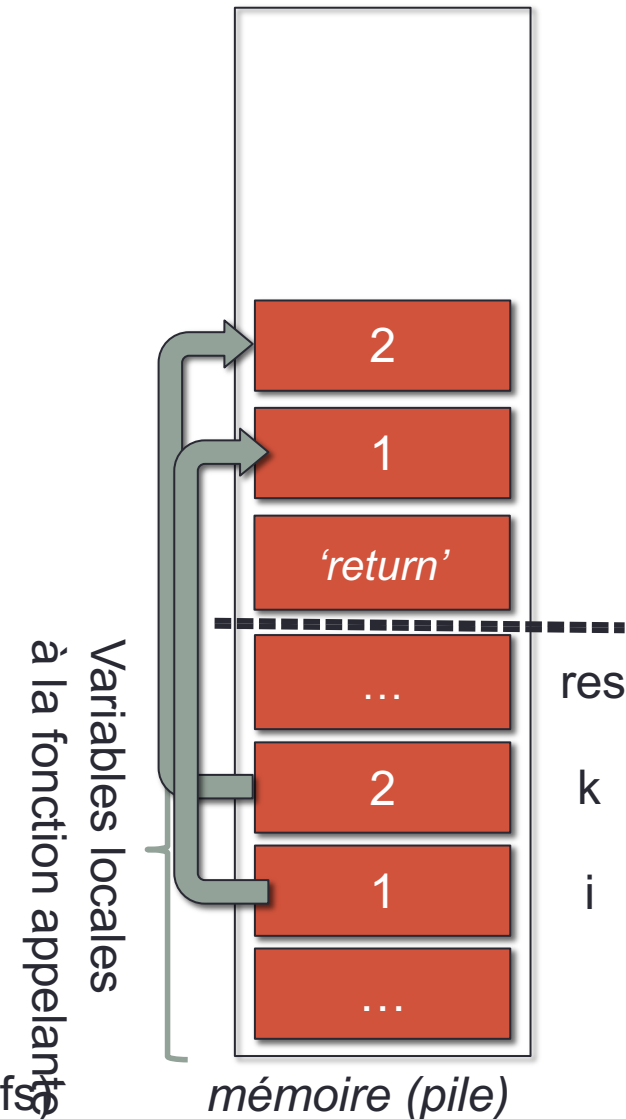
1. Appel fonction:

a) alloue un espace pour le résultat de la fonction ('return')

Passage de paramètres par valeur

```
{
    int i = 1, k = 2;
    float res;
    . . .
    res = norm(k, i);
    . . .
}
```

```
float norm(int i, int j)
{
    float tmp=0;
    i = i*i;
    j = j*j;
    return ( sqrt(i+j) );
}
```



1. Appel fonction:

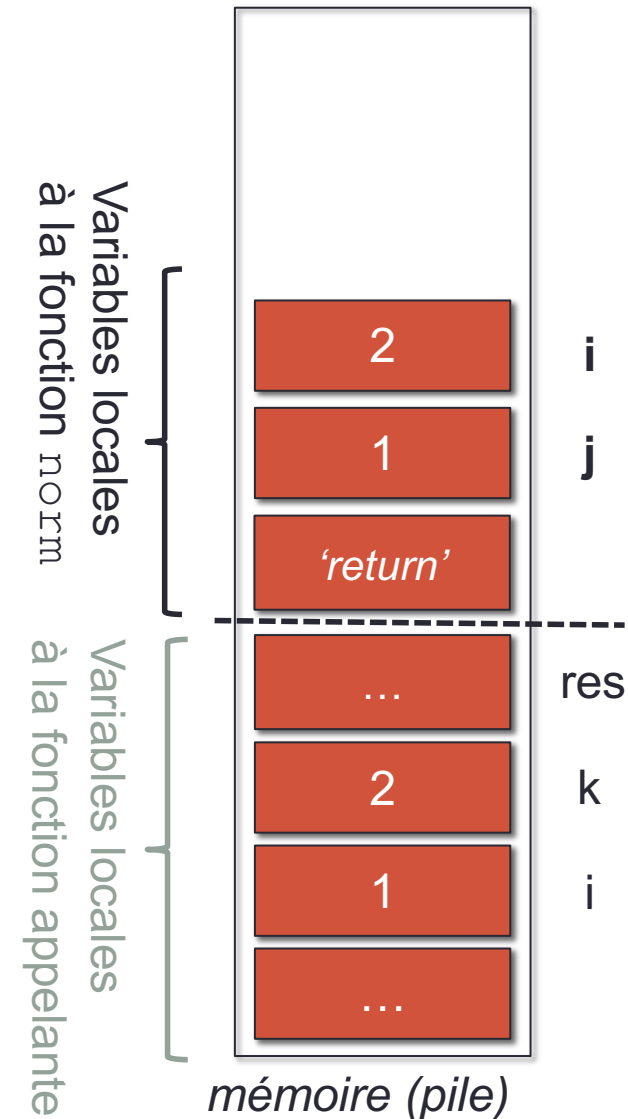
b) **copie de la valeur** des variables `k, i` (arguments effectifs) sur la pile mémoire par la fonction appelante.

Passage de paramètres par valeur

```
{
    int i = 1, k = 2;
    float res;
    . . .
    res = norm(k, i);
    . . .
}
```

2 → `float norm(int i, int j)`

```
{
    float tmp=0;
    i = i*i;
    j = j*j;
    return ( sqrt(i+j) );
}
```



2. Entrée dans la fonction:

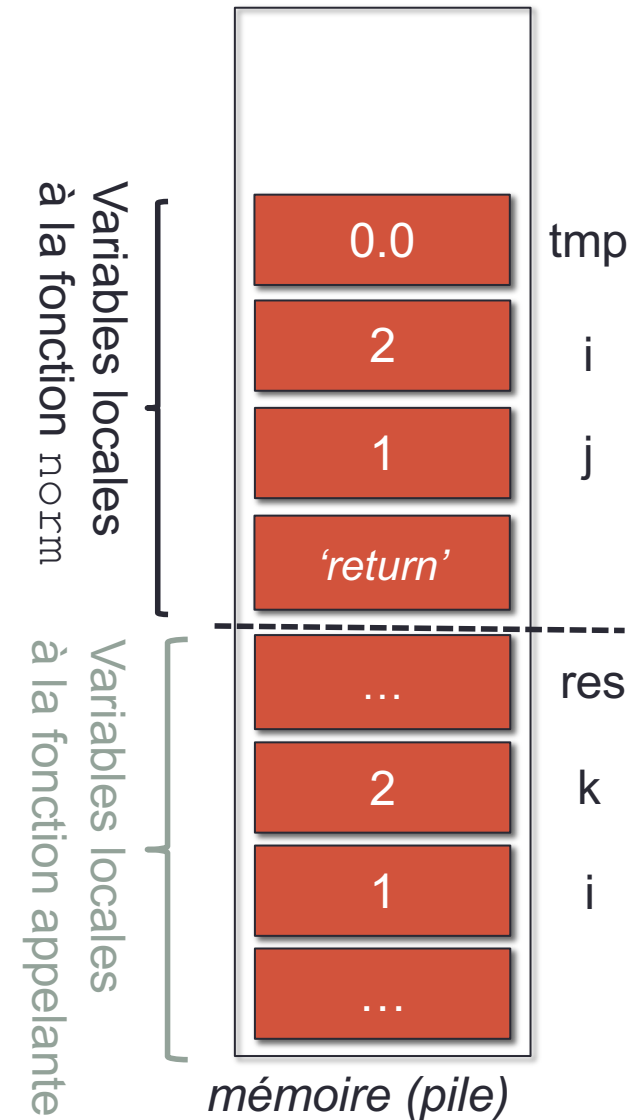
- la fonction trouve la **valeur** de ses variables locales (arguments formels) *i*, *j* sur le sommet de la pile.

Passage de paramètres par valeur

```
{
    int i = 1, k = 2;
    float res;
    . . .
    res = norm(k, i);
    . . .
}
```

3 →

```
float norm(int i, int j)
{
    float tmp=0;
    i = i*i;
    j = j*j;
    return ( sqrt(i+j) );
}
```



3. Exécution de la fonction:

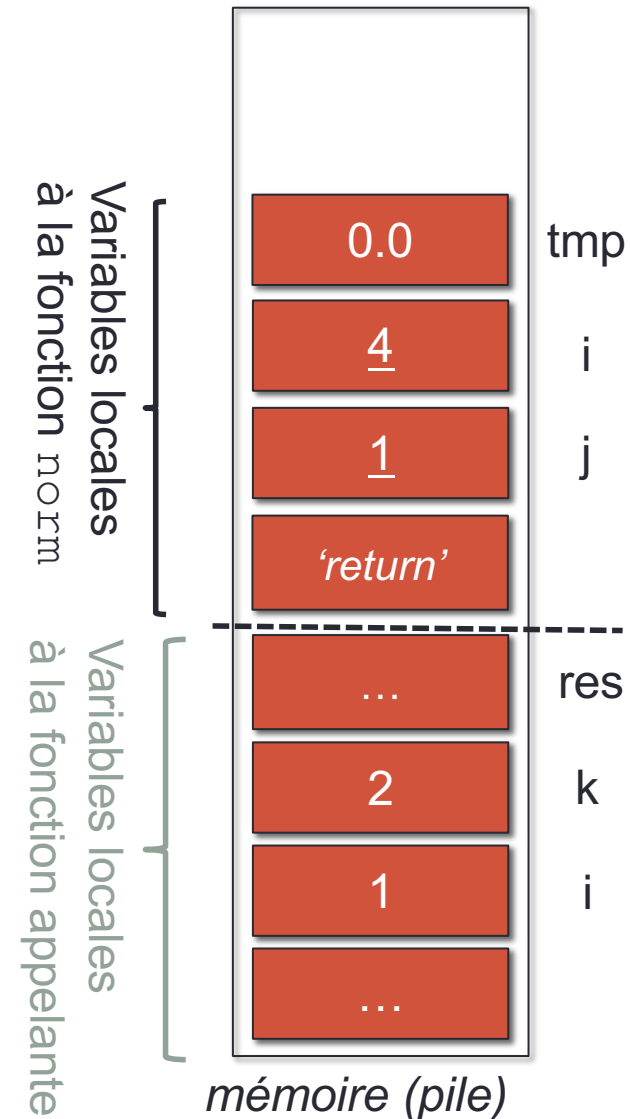
- création des variables locales

Passage de paramètres par valeur

```
{
    int i = 1, k = 2;
    float res;
    . . .
    res = norm(k, i);
    . . .
}
```

3 →

```
float norm(int i, int j)
{
    float tmp=0;
    i = i*i;
    j = j*j;
    return ( sqrt(i+j) );
}
```



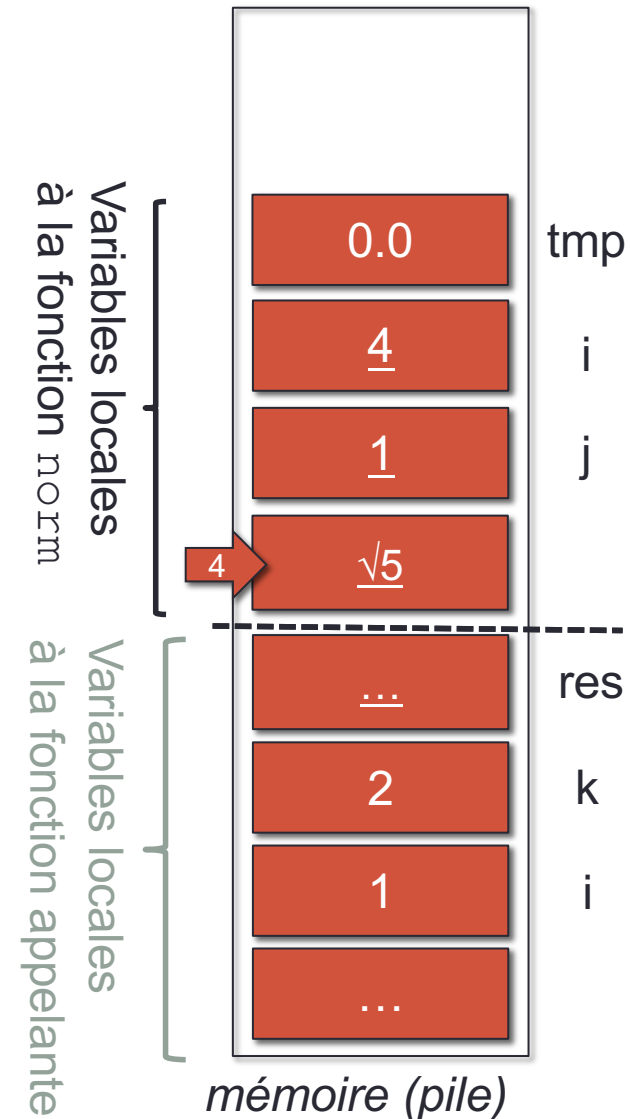
3. Exécution de la fonction:

- création des variables locales
- exécution des instructions

Passage de paramètres par valeur

```
{
    int i = 1, k = 2;
    float res;
    . . .
    res = norm(k, i);
    . . .
}
```

```
float norm(int i, int j)
{
    float tmp=0;
    i = i*i;
    j = j*j;
    return ( sqrt(i+j) );
}
```

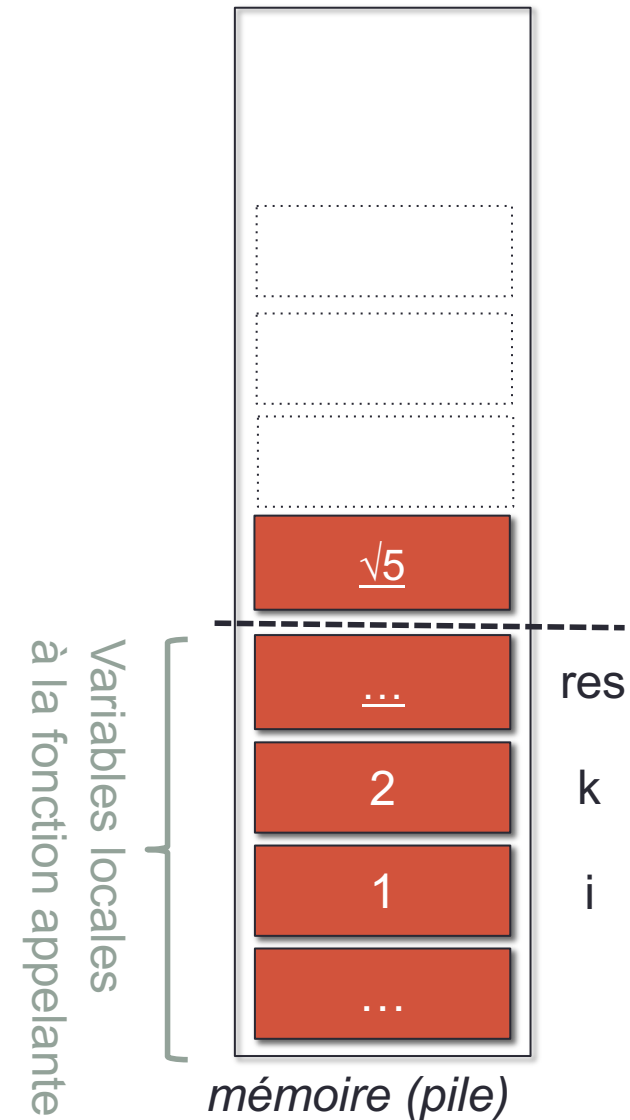


4. Sortie de la fonction :
- a) retour de la valeur

Passage de paramètres par valeur

```
{
    int i = 1, k = 2;
    float res;
    . . .
    res = norm(k, i);
    . . .
}
```

```
float norm(int i, int j)
{
    float tmp=0;
    i = i*i;
    j = j*j;
    return ( sqrt(i+j) );
}
```



4. Sortie de la fonction :

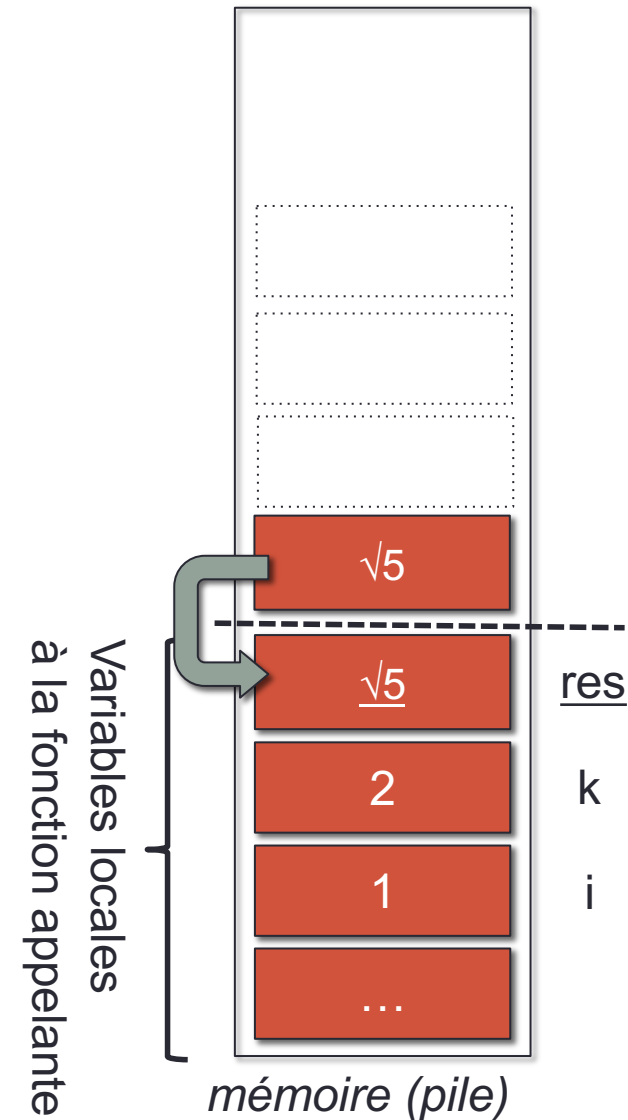
b) suppression des variables locales à la fonction

Passage de paramètres par valeur

```
{
    int i = 1, k = 2;
    float res;
    . . .
    res = norm(k, i);
    . . .
}
```

5

```
float norm(int i, int j)
{
    float tmp=0;
    i = i*i;
    j = j*j;
    return ( sqrt(i+j) );
}
```



5. Fin de l'appel :

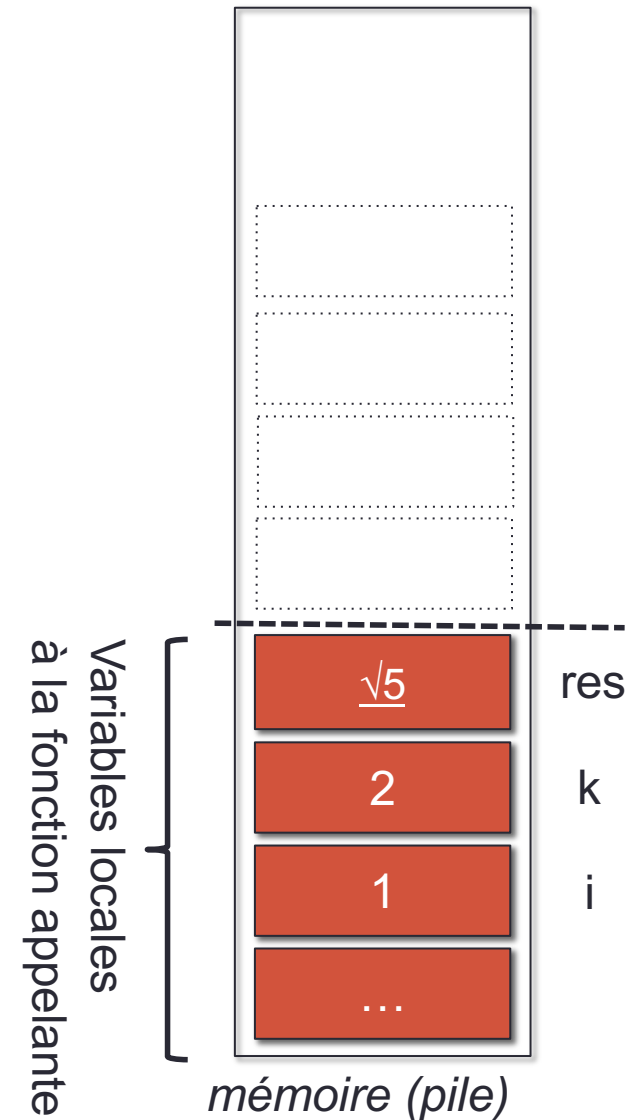
a) affectation du résultat (trouvé sur sommet de la pile)

Passage de paramètres par valeur

```
{
    int i = 1, k = 2;
    float res;
    . . .
    res = norm(k, i);
    . . .
}
```

5

```
float norm(int i, int j)
{
    float tmp=0;
    i = i*i;
    j = j*j;
    return ( sqrt(i+j) );
}
```



5. Fin de l'appel :
- b) poursuite exécution de la fonction appelante
(**k, i inchangés**)

Passage de paramètres par valeur

- Avantage : moins de variables
- les paramètres formels (i, j) de la fonction `norm()` sont des variables locales
 - variables utilisables pour des calculs intermédiaires
- les variables (i, k) de la fonction appelante (`main()` ?) passées à la fonction `norm()` conservent leurs valeurs
 - valeur du vecteur (i, k) protégée et disponible pour la suite du programme

Passage de paramètres par valeur

- Inconvénient :

une fonction ne peut modifier la valeur d'une variable de la fonction appelante

_____ puisqu'elle travaille sur une copie locale



i.e. la valeur de `num` du `main()` sera toujours la même avant et après l'appel à une fonction

Sommaire chapitre 4

Les fonctions 1

définition	178
passage de paramètres par valeur	187
mécanisme du passage par valeur	191
récurtivité	206
prototype d'une fonction	210

Déclaration et Prototype d'une fonction

- Une fonction doit être déclarée avant sa première utilisation, comme pour une variable.

(Sinon le compilateur vous dit ne pas la connaître !)

Deux méthodes pour déclarer une fonction :

1. **[conseillé]** Mettre son **prototype** en début de fichier source, ce qui permet de mettre sa définition n'importe où (même dans un autre fichier).
2. **[déconseillé]** Mettre sa définition complète (son entête et son code) dans le source avant le premier appel à celle-ci.

Ne marche pas si les fonctions s'appellent l'une l'autre (ordre de définition ?) et si la fonction est définie dans un autre fichier !

Prototype d'une fonction

- déclaration du nom de la fonction et son type et nombre d'arguments.

```
type nom_fonction(type nom_arg1, type nom_arg2,...) ;
```

ou

```
type nom_fonction( type, type, ... ) ;
```



le prototype finit avec un point virgule à la fin

Usage d'un prototype 1

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

```
/* prototype ! */
```

```
double aire triangle(double base, double hauteur);
```

```
/* definition ! */
```

```
double aire triangle( double base, double hauteur )
```

```
{  
    return base*hauteur/2 ;  
}
```

```
int main()
```

```
{  
    printf ("Aire d'un triangle de base 3 et hauteur 10 : %f\n",  
            aire triangle(3,10) );  
    return 0;  
}
```

Usage d'un prototype 2

lecture séquentielle
par le compilateur

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

```
/* prototype */
```

```
double aire_triangle(double base, double hauteur);
```

```
int main()
```

```
{
```

```
    printf ("Aire d'un triangle de base 3 et hauteur 10 : %f\n",  
           aire_triangle(3,10) );
```

```
    return 0;
```

```
}
```

```
/* definition*/
```

```
double aire_triangle( double base, double hauteur )
```

```
{
```

```
    return base*hauteur/2 ;
```

```
}
```


Usage d'un prototype 2

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

```
/* prototype */
```

```
double aire_triangle(double base, double hauteur);
```

```
int main()
```

```
{
```

```
    printf ("Aire d'un triangle de base 3 et hauteur 10 : %f\n",  
            aire_triangle(3,10) );
```

```
    return 0;
```

```
}
```

```
/* definition*/
```

```
double aire_triangle( double base, double hauteur )
```

```
{
```

```
    return base*hauteur/2 ;
```

```
}
```

Le compilateur ayant lu le prototype peut vérifier que :

- `aire_triangle` est bien le nom d'une fonction,
- le nombre et le type de ses paramètres est correct (2 double).

Sans prototype, erreur du compilateur : `aire_triangle` inconnu.

Sommaire chapitre 4

Les fonctions 1	
définition	178
passage de paramètres par valeur	187
mécanisme du passage par valeur	191
récursivité	206
prototype d'une fonction	210
 La portée des variables	
locale et globale	215
static	222

Portée

- Portée : région dans laquelle une variable est définie
- la variable déclarée au début d'un bloc { } a une **portée locale** au bloc
- la variable déclarée hors d'une fonction (hors de tout bloc) a une **portée globale**

Variable locale

Variable locale

- portée = le bloc { } au début duquel la variable est déclarée
- dite de classe automatique :
 - **création (mémoire allouée) à l'entrée du bloc de déclaration**
 - **détruite à la sortie du bloc (son emplacement mémoire est libéré)**
- valeur perdue entre 2 exécutions du bloc

Variable locale

```
int cube (int nbr)
{
    int result=0;                /*création en mémoire de result*/
    result=nrb*nbr;

    return result;

}    /* fin du bloc, espace mémoire pour result supprimé */
    /* idem pour la variable local en argument : nbr    */
```

Variable locale

```
int cube (int nbr)
{
    int result=0;    /*création en mémoire de la var.*/
    result=nrb*nbr;

    return result;
}    /*fin de la fonction, result supprimé en mémoire*/
```

```
int main ()
{
    printf("Le cube de 15 vaut : %d", cube(15));
    printf("Le cube de 15 vaut : %d", result);
    return 0;
}
```

↑
Erreur à la compilation !
result n'existe pas dans ce bloc

Variable locale

```
void loop(int j)
{
    int i;
    for (i=0;i<j;i++)
        { printf("o\n"); }
}
```

```
int main()
{
    int N_loop=3;
    loop(N_loop);
    return 0;
}
```



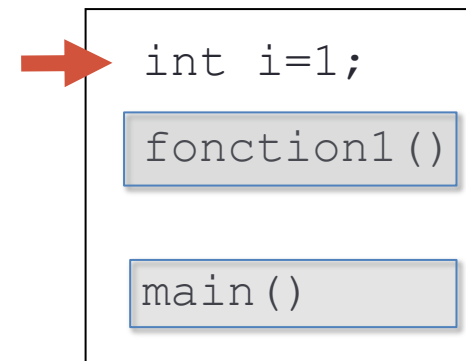
```
void loop(int N_loop)
{
    int i;
    for (i=0;i<N_loop;i++)
        { printf("o\n"); }
}
```

```
int main()
{
    int N_loop=3;
    loop(N_loop);
    return 0;
}
```

2 variables N_loop,
mais chacune locale à sa fonction
→ OK, pas de conflit, ni ambiguïté.

Variable globale

- Variable globale, si définie hors de toute fonction (en général au début du fichier sous les `#include`)
- portée = tous les fichiers sources
 - accessible par n'importe quelle fonction
 - **partagée entre toutes les fonctions**
 - existe pendant toute l'exécution du programme



- Variable globale à éviter si possible :
 - fonctions difficilement réutilisables si dépendante de variables globales
 - interférence possible avec des variables locales de même nom

Variable globale

```
#include <stdio.h>

int compteur=0;           /* VARIABLE GLOBALE */
                           /*compte le nombre d'execution des fonctions*/

int carre (int nbr)
{
    compteur++;
    return (nbr*nbr);
}

int main ()
{
    compteur++;
    printf("Le carre de 15 vaut : %d\n", carre(15));
    return 0;
}
```