

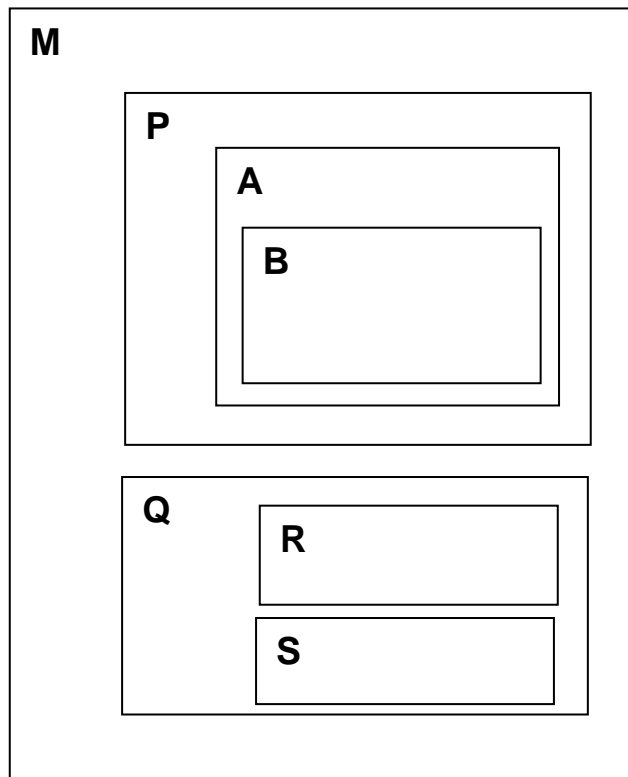
## **COURS 4**

- Compléments sur les variables : portée,
- Pointeurs – arithmétique et opérations sur les adresses
- Fonctions – appels par valeur – gestion de la pile

## 9. Tout sur les variables ...

### *Portée des variables : Variables globales – variables locales*

- Une variable déclarée dans l'entête du programme est *globale*
- Une variable déclarée au début d'un *bloc d'instructions* est *locale* au bloc



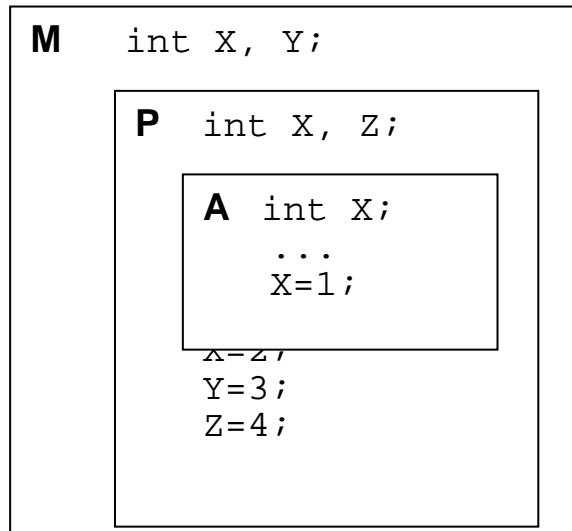
Les objets déclarés dans le bloc	Sont visibles depuis les blocs
M	M, P, A, B, Q, R, S
P	P, A, B
A	A, B
B	B
Q	Q, R, S
R	R
S	S

## Variables : visibilité

Une variable globale est visible (référéncable) depuis tout le programme après sa déclaration.

Une variable locale n'est visible que dans le bloc où elle a été déclarée.

Attention : si on déclare plusieurs variables de même nom dans des blocs imbriqués, c'est la déclaration la plus imbriquée qui prime sur les autres. Les autres déclarations sont masquées.



La déclaration de la variable	du bloc	masque celle de	du bloc
X	P	X	M
X	A	X	P
		X	M

L'instruction	du bloc	référence la variable	du bloc
X=1	A	X	A
X=2	P	X	P
X=5	M	X	M
Y=3	P	Y	M
Z=4	P	Z	M
Z=6	M	provoque	P
			une erreur

## Variables statiques /automatiques

- la *durée de vie*, entre la création (allocation de mémoire) et la destruction (restitution de mémoire).
- la *localisation* : l'espace partagé et commun à tout le programme ou l'espace restreint à un bloc d'instruction.
- le mode d'*instanciation* : une seule instance (la même pour toute la durée du programme) ou plusieurs instances.

	<b>durée de vie</b>	<b>création</b>	<b>destruction</b>	<b>localisation mémoire</b>	<b>instanciation</b>
<b>variable statique</b>	celle du programme	à l'activation du programme	à la fin du programme	mémoire partagée	unique : à l'activation du programme
<b>variable automatique</b>	celle du bloc	à l'activation du bloc	à la fin du bloc	mémoire dynamique (pile)	multiple : une instance nouvelle à chaque activation du bloc

## Variables publiques/privées

Contexte : le programme source est divisé en plusieurs fichiers, compilés séparément. Chaque fichier source contient ses propres déclarations de variables.

- Variable *publique* : visible de tous les fichiers qui composent le programme.
- Variable *privée* : visible seulement dans le fichier où elle est déclarée.

### Par défaut

- Une variable locale est *automatique* et *privée*.
- Une variable globale est *statique* et *publique*.

## Variables - Propriétés

*déclaration\_de\_variable* → *qualifieur mode type var\_init* ⟨, *var\_init*⟩\* ;

*qualifieur* → **auto** | **register** | **static** | **extern** | *rien*

*mode* → **const** | **volatile** | *rien*

### Qualifieur

- **auto** rend la variable automatique. C'est la propriété par défaut des variables locales.
- **static** rend la variable statique. C'est la propriété par défaut des variables globales.
  - déclarer **static** une variable **locale** ne change pas sa visibilité mais change sa durée de vie et le mode d'instanciation : une seule instance de la variable est créée et cette instance est valide pour toutes les activations du bloc où elle est déclarée. A éviter dans les fonctions récursives car risque de gros ennuis.
  - déclarer **static** une variable **globale** change sa visibilité et la rend privée au module où elle est définie.
- **register** permet d'allouer un registre à une variable locale et d'un type simple. A éviter
- **extern** pour une variable publique déclarée dans un autre fichier.

### mode

- **const** : la variable (initialisée) ne changera pas de valeur durant toute sa vie. Très conseillé.
- **volatile** indique une variable susceptible d'être modifiée de manière asynchrone (par interruption, entrée/sortie, etc.).

## ***En résumé ...***

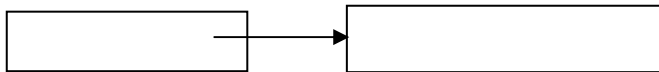
1. Toute variable globale est publique par défaut.
2. Toute variable locale est privée par défaut.
3. L'attribut **static** devant une variable globale la rend invisible en dehors du module où elle est déclarée.
4. Dans l'ensemble des fichiers qui constituent un programme, chaque variable publique :
  - doit avoir été déclarée normalement dans un et un seul des fichiers. Cette définition crée effectivement la variable, lui alloue un espace en mémoire et éventuellement l'initialise.
  - doit avoir été déclarée **extern** et sans initialisation dans les (autres) fichiers qui l'utilisent. Cette déclaration référence la zone mémoire allouée à la variable lors de sa définition.
5. Par précaution et dans tous les fichiers qui composent un programme, rendre privées toutes les variables globales qui peuvent l'être en les rendant **static** lors de leur déclaration. Sinon, des fichiers qui étaient corrects pris séparément risquent de devenir erronés lorsqu'on les rassemble parce qu'ils partagent à tort des variables publiques.

## 10. Pointeurs

*Adressage direct:* Accès au contenu d'une variable par le nom de la variable.

*Adressage indirect:* Accès au contenu d'une variable, en passant par une variable intermédiaire qui contient l'adresse de la variable.

Pointeur : variable dont le contenu est (interprété comme) l'adresse d'une autre variable :



p : variable pointeur    q : variable pointée

A partir de l'une, on dispose d'opérateurs pour passer à l'autre :

- Le référencement : l'opérateur « \* » permet de passer du pointeur à l'objet pointé : si p désigne un pointeur et q l'objet pointé on a par définition  $*p \equiv q$  : l'expression \*p a pour valeur l'objet pointé par p.
- Le déréférencement : l'opérateur « & » permet de passer de l'objet pointé au pointeur: si p désigne un pointeur et q l'objet pointé on a par définition  $\&q \equiv p$  : l'expression &q a pour valeur l'adresse de q.

L'opérateur « & » peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c.-à-d. à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

→ Pour résumer :  $*(\&q) \equiv q$  et  $\&(*p) \equiv p$

## Type pointeur

declaration de

La déclaration d'une variable de type pointeur précise en même temps le type de l'objet pointé.

Par exemple : `int *p`

déclare une variable de nom `p` qui pointe sur un entier : « `*p` », c'est-à-dire l'objet pointé par `p` est un entier.

Le « `int*` » doit être lu comme un tout. En quelque sorte cela introduit un nouveau type : un pointeur sur un objet de type *type* a pour type *\*type*.

### Initialisation des pointeurs

Il y a 4 façons sûres d'initialiser un pointeur :

- Lui affecter la constante `NULL` qui est la valeur conventionnelle d'un pointeur qui ne pointe vers rien du tout.
- Lui affecter l'adresse d'une variable déjà connue. Exemple :

```
int x ;      /* déclaration d'une variable entière x */
```

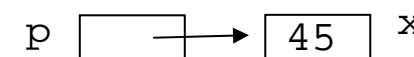
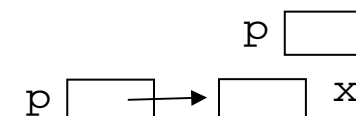
```
int *p ;     /* p est un pointeur sur un entier */
```

```
...
```

```
p = &x ;    /* p contient l'adresse de x */
```

```
*p = 34 ;
```

```
x = 45 ;    /* x et *p désignent le même objet */
```

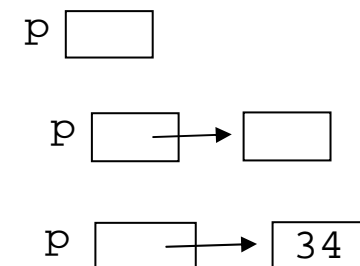


## Initialisation des pointeurs(suite)

- Lui affecter l'adresse d'une variable anonyme allouée dynamiquement

```
int *p;      /* p est un pointeur sur un entier */
...
p=malloc(sizeof(int)); /* alloue une zone mémoire */
                    /* de la taille d'un entier*/
                    /*et met son adresse dans p*/

*p = 34;
```



La différence avec l'exemple précédent est que la zone de mémoire pointée par `p` est anonyme, et donc on ne peut y accéder directement. Toute nouvelle affectation de `p` sans sauvegarde préalable de son contenu rend la zone pointée inaccessible, et donc perdue.

- Lui affecter le résultat d'une expression arithmétique sur les adresses : un ou les deux opérandes d'une expression d'addition ou de soustraction peuvent être des adresses, représentées par des nombres entiers.

## Pointeurs et types

Un pointeur peut référencer (pointer vers) n'importe quoi : une variable simple, un tableau, une fonction, etc.

Dans une déclaration de type, l'opérateur « \* » peut être combiné avec lui-même ou avec d'autres opérateurs comme l'indexation « [ ] » ou l'appel de fonction « ( ) ». Les priorités peuvent être modifiées par des parenthèses.

→ d'où des déclarations parfois complexes et difficiles à comprendre.


`char *p`      `p` est un pointeur vers un caractère (ou : `*p` est un caractère)

`int *p`      `p` est un pointeur vers un entier (ou : `*p` est un entier)

`int *p[N]`     $\equiv$  `int *(p[N])`    `p[i]` est un pointeur vers un entier. Donc `p` est un tableau de `N` pointeurs vers des entiers

`int (*p)[N]`      `(*p)[i]` est un entier. Donc `*p` est un tableau de `N` entiers.

Donc `p` est un pointeur vers un tableau de `N` entiers

`int **p`      `p` est un pointeur vers une variable elle-même pointeur vers un entier :    `p` 

`int *f()`       $\equiv$  `int *(f())` : `f` est une fonction dont le résultat est un pointeur vers un entier.

`int (*f)()`      `*f` est une fonction entière. Donc `f` est un pointeur vers une fonction entière

`int (*f[N])()`    `*f[i]` est une fonction entière. Donc `f[i]` est un pointeur vers une fonction entière et donc `f` est un tableau de `N` pointeurs vers des fonctions entières.

## Pointeurs et types (suite)

Comment s'y prendre ?

**Exemple** : on veut déclarer  $p$ , pointeur vers un tableau de  $N$  entiers

pointeur vers	un tableau de $N$	entiers	$p$
	un tableau de $N$	entiers	$*p$
		entiers	$(*p)[N]$
		→	$\text{int } (*p)[N]$

**Exemple** : on veut déclarer  $p$ , un tableau de  $N$  pointeurs vers des entiers

tableau de $N$	pointeurs vers des	entiers	$p$
	pointeurs vers des	entiers	$p[N]$
		entiers	$*(p[N]) \equiv *p[N]$
		→	$\text{int } *p[N]$

**Exemple** : on veut déclarer  $f$ , tableau de  $N$  pointeurs vers des fonctions entières

tableau de $N$	pointeurs vers des	fonctions à résultat entier	$f$
	pointeurs vers des	fonctions à résultat entier	$f[N]$
		fonctions à résultat entier	$*(f[N]) \equiv *f[N]$
		résultat entier	$(*f[N])()$
		→	$\text{int } (*f[N])()$

## **Arithmétique des adresses**

Dans certains cas, les opérandes d'une expression arithmétique peuvent être des adresses. Ainsi il est permis :

- d'ajouter ou de soustraire un nombre entier à une adresse
- de soustraire deux adresses désignant des objets de même type.
- De même, l'opérande d'une expression d'incrément ou de décrétement peut être une adresse.

Tous les autres opérateurs arithmétiques sont interdits. A chaque fois que l'opérande d'une expression arithmétique est une adresse, il est considéré comme un entier lors de l'évaluation de l'expression.

Attention : cette arithmétique est particulière :

- Si la valeur d'une expression arithmétique  $e$  est l'adresse d'un objet  $O$  de type  $T$ , alors  $e+1$  (resp.  $e-1$ ) désigne l'adresse de l'objet de type  $T$  et situé en mémoire immédiatement après (resp. avant)  $O$ .
- Si  $e1$  et  $e2$  sont les adresses de 2 objets contigus de même type, la valeur de la différence  $e1 - e2$  sera 1.

En conséquence :

- Ajouter (resp. soustraire) 1 à l'adresse d'un objet c'est en réalité ajouter (resp. soustraire) une fois la taille de cet objet à l'entier représentant cette adresse.
- Soustraire une adresse à une autre adresse revient à effectuer la soustraction entre les entiers correspondants, puis diviser le résultat obtenu par la taille des objets désignés (et qui doivent être de même type).
- Tout ce que l'on a dit ci-dessus reste valide avec les opérateurs « ++ », « -- », « += », etc.

## ***Arithmétique des adresses (suite)***

Ainsi, si `p`, `x`, `y`, `z` sont déclarés par :

```
int x,y,z ;  
int *p ;
```

les instructions qui suivent ont l'effet indiqué :

<code>x = 3 ;</code>	<code>x</code> prend la valeur 3
<code>p = &amp;x ;</code>	<code>p</code> pointe sur <code>x</code>
<code>y = *p + 1 ;</code>	<code>y</code> prend la valeur 4
<code>z = *(p + 1) ;</code>	<code>z</code> prend la valeur de l'entier qui suit immédiatement <code>x</code>
<code>*p = 0 ;</code>	<code>x</code> prend la valeur 0
<code>*p += 1 ;</code>	<code>x</code> est incrémenté de 1
<code>(*p)++ ;</code>	<code>x</code> est encore incrémenté de 1

## Pointeurs et tableaux

Similitudes entre les pointeurs et les tableaux

- tous les traitements effectués avec des tableaux peuvent l'être avec des pointeurs.
- le **nom du tableau est une constante dont la valeur est l'adresse de son premier élément**

```
Ex: int a[10] ; int *p ; int i ;
```

définissent :

- un bloc de 10 entiers consécutifs nommés  $a[0]$ ,  $a[1]$  , ... ,  $a[9]$  ,
- un pointeur  $p$  vers un entier
- et un entier  $i$ .

Les expressions suivantes sont équivalentes:

- $a[i]$  et  $*(a+i)$  : désignent la valeur du  $i^{\text{ème}}$  élément du tableau  $a$ .
- $\&a[i]$  et  $a+i$  : désignent l'adresse du  $i^{\text{ème}}$  élément du tableau  $a$ .
- $p = \&a[0]$  et  $p = a$  :  $p$  contient l'adresse de  $a[0]$ , début du tableau  $a$ .
- $p[i]$  et  $*(p+i)$  : désignent la valeur du  $i^{\text{ème}}$  élément du tableau  $a$ .

→ Une indexation et une expression avec pointeur et déplacement sont équivalentes, sauf qu'un nom de tableau est considéré comme une constante.

→ Mais les expressions  $a = p$  ou  $a++$  ou  $p=\&a$  sont interdites,

→ alors que  $p = a$  ou  $p++$  sont autorisées car  $p$  est une variable.

## 11. Fonctions

Fragment de programme (suite d'instructions regroupées ensemble) et auquel on donne un nom.

- pour réaliser un certain calcul : en général, une fonction renvoie un résultat.
- ... qui est répété plusieurs fois à l'intérieur d'un programme et qu'on évite ainsi de dupliquer
- ... qui est paramétré par les arguments de la fonction.

```
#include <stdio.h>

int ENTREE(void)
{int NOMBRE;
 printf("Entrez un nombre entier : ");
 scanf("%d", &NOMBRE);
 return NOMBRE;
}

int MAX(int N1, int N2)
{if (N1>N2) return N1;
 else return N2;
}

main()
{int A, B;
 A = ENTREE();
 B = ENTREE();
 printf("Le maximum est %d\n", MAX(A,B));
}
```

Dans ce programme, la fonction **main** utilise deux fonctions:

- **ENTREE** qui lit un nombre entier au clavier et le fournit comme résultat. La fonction ENTREE n'a pas de paramètres.
- **MAX** qui renvoie comme résultat le maximum de deux entiers fournis comme paramètres.

## Déclaration des fonctions

*déclaration\_de\_fonction* →

*type\_du\_résultat* *nom\_de\_fonction* ( *liste\_de\_paramètres\_formels* ) *bloc* ;

*type\_du\_résultat* → *identificateur*

*nom\_de\_fonction* → *identificateur*

*liste\_de\_paramètres\_formels* → *paramètre\_formel* ⟨ , *paramètre\_formel* ⟩\* | rien

*paramètre\_formel* → *nom\_de\_type* *nom\_de\_variable*

*nom\_de\_variable* → *identificateur*

- Le type de la fonction est `void` : aucun résultat n'est retourné.
  - Le type de la fonction est absent : par défaut le résultat est de type `int` (à éviter !).
  - La liste de paramètres formels est vide : la fonction n'a pas d'arguments.
- paramètres formels : une construction comme `: int x, y` est interdite. On écrira `: int x, int y`.
- Le corps de la fonction est un bloc d'instructions
- Pas de point-virgule derrière la définition des paramètres de la fonction.
- Si le nom de la fonction existe déjà dans une bibliothèque, notre fonction **cache** la fonction prédéfinie.

## ***Pré-déclaration des fonctions***

Une fonction doit en principe être déclarée avant d'être activée. Cas de 2 fonctions qui s'appellent mutuellement ?

1) Déclaration implicite : tout identificateur non déclaré suivi d'une parenthèse est considéré comme un nom de fonction de type entier.

→ Lorsque l'appel d'une fonction constitue la 1<sup>ère</sup> apparition de cette fonction dans le programme, tout se passe comme si la fonction est implicitement déclarée de type entier et avec des paramètres entiers.

2) Déclaration en 2 temps :

▪ une pré-déclaration qui précise le type et le nom de la fonction:

```
pré_déclaration_de_fonction → type_du_résultat nom_de_fonction(liste_de_types_des_par_formels) ;  
liste_de_types_des_par_formels → nom_de_type < , nom_de_type >* | rien
```

▪ plus loin dans le programme, la déclaration complète

→ La pré-déclaration ne comporte que les types des paramètres formels

→ Leurs noms et le corps de fonction figureront dans la déclaration elle-même.

→ Si les types des paramètres formels sont absents, il s'agit d'entiers par défaut.

→ Les séquences suivantes engendrent une erreur à la compilation :

```
...  
x = f(y) ;  
...  
float f(int n) {...}      ou   int f(float n) {...}
```

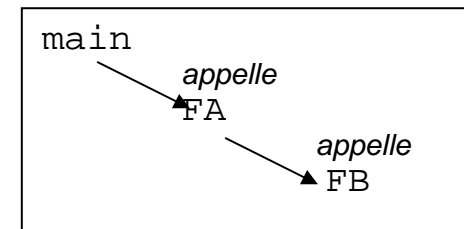
## Déclaration vs. pré-déclaration des fonctions

### 1. Pré-déclarations locales des fonctions et déclarations 'top-down'

```
/* Définition de main */
main()
{ /* pré-déclaration locale de FA */
  int FA (int X, int Y);
  ...
  /* Appel de FA */
  I = FA(2, 3);
  ...
}

/* déclaration de FA */
int FA(int X, int Y)
{ /* Pré déclaration locale de FB */
  int FB (int N, int M);
  ...
  J = FB(20, 30); /* Appel de FB */
  ...
}

/* déclaration de FB */
int FB(int N, int M)
{ ... }
```



La définition 'top-down' suit la hiérarchie des fonctions: Nous commençons par la définition de la fonction principale **main**, suivie des sous-programmes FA et FB. Nous devons déclarer explicitement FA et FB car leurs définitions suivent leurs appels.

En passant du haut vers le bas dans le fichier on retrouve toutes les dépendances des fonctions simplement en se référant aux déclarations locales. S'il existe beaucoup de dépendances dans un programme, le nombre de déclarations locales peut quand même s'accroître beaucoup !

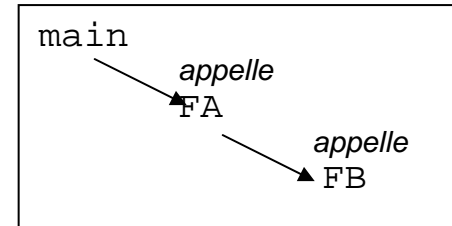
## Déclaration vs. pré-déclaration des fonctions (suite)

### 2. Définition 'bottom-up' sans déclarations

```
/* déclaration de FB */
int FB(int N, int M)
{ ...
}

/* déclaration de FA */
int FA(int X, int Y)
{ ...
  /* Appel de FB */
  J = FB(20, 30);
  ...
}

/* déclaration de main */
main()
{ ...
  /* Appel de FA */
  I = FA(2, 3);
  ...
}
```



La définition 'bottom-up' commence en bas de la hiérarchie:  
La fonction **main** se trouve à la fin du fichier. Les fonctions qui traitent les détails du problème sont déclarées en premier lieu.

Comme les fonctions sont déclarées avant leur appel, le texte du programme est allégé, mais il est beaucoup plus difficile de retrouver les dépendances entre les fonctions.

## Déclaration vs. pré-déclaration des fonctions (fin)

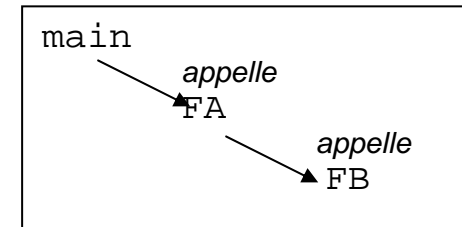
### 3. Pré-déclaration globale des fonctions et déclaration 'top-down'

```
/* Pré-déclaration globale de FA et FB */
int FA (int X, int Y);
int FB (int N, int M);

/* déclaration de main */
main()
{ ...
  I = FA(2, 3); /* Appel de FA */
  ...
}

/* déclaration de FA */
int FA(int X, int Y)
{ ...
  J = FB(20, 30); /* Appel de FB */
  ...
}

/* déclaration de FB */
int FB(int N, int M)
{ ...
}
```



En déclarant toutes les fonctions globalement au début du texte du programme, nous ne sommes pas forcés de nous occuper de la dépendance entre les fonctions. Cette solution est la plus simple et la plus sûre pour des programmes complexes contenant une grande quantité de dépendances. Il est quand même recommandé de définir les fonctions selon l'ordre de leur hiérarchie:

## ***Appel des fonctions et passage des paramètres (cas général)***

1. substituer position pour position les paramètres formels par les arguments ou paramètres effectifs
2. exécuter les instructions du corps de la fonction jusqu'à rencontrer `return` ou terminer ce bloc.
3. revenir au programme appelant en renvoyant le résultat

**Appel par valeur** : les paramètres effectifs sont évalués et ces valeurs sont affectées aux paramètres formels.

```
fonction f(x :entier) : entier ;
y : entier ;
début
  x=x+1 ;
  y=2*x ;
  retour y ;
fin ;
```

et la séquence :

```
i=2 ; j=f(i) ;
```

a le même effet que :

- |                                  |  |
|----------------------------------|--|
| (1) <code>x=i ;</code>           | évaluation des paramètres effectifs et initialisation des paramètres formels |
| (2) <code>x=x+1 ; y=2*x ;</code> | exécution du corps de la fonction  |
| (3) <code>j=y ;</code>           | résultat et retour à l'appelant  |

Avant la ligne (1) la variable `i` a pour valeur 2. La variable `x` reçoit la valeur 2 puis est incrémentée. Après la ligne (2) `x` vaut 3 et `y` vaut 6. Après la ligne (3), la variable `j` a la valeur 6, `i` est inchangée et a la valeur 2.

**Appel par nom (ou par adresse, ou par référence)** : chaque paramètre formel est remplacé par le paramètre effectif correspondant, position pour position et nom pour nom..

Avec un passage de paramètres par nom, l'appel de fonction de l'exemple précédent a le même effet que l'exécution des instructions suivantes :

(1)  $i=i+1$  ;  $y=2*i$  ; exécution du corps de la fonction  
(2)  $j=y$  ; résultat et retour à l'appelant

Avant la ligne (1) la variable  $i$  a pour valeur 2 puis est incrémentée. Après la ligne (1)  $i$  vaut 3 et  $y$  vaut 6. Après la ligne (2),  $j$  a la valeur 6, la valeur de  $i$  a changé et est passée à 2.

### **En résumé...**

- dans un appel par valeur, la fonction travaille sur une copie des paramètres effectifs. Ceux-ci gardent la valeur qu'ils avaient avant l'appel.
- dans un appel par nom, la fonction travaille sur les paramètres effectifs eux-mêmes. Ceux-ci peuvent être modifiés par la fonction et perdre la valeur qu'ils avaient avant l'appel.
- si on veut que la fonction modifie les paramètres effectifs, il faut employer un appel par nom

## **Appel des fonctions et passage des paramètres (cas de C)**

*appel\_de\_fonction* → *nom\_de\_fonction*( *liste\_de\_paramètres\_effectifs* )  
*liste\_de\_paramètres\_effectifs* → *paramètre\_effectif* ⟨ ,*paramètre\_effectif* ⟩\* | rien  
*paramètre\_effectif* → *expression*

Dans tous les cas :

- il doit y avoir autant de paramètres effectifs que de paramètres formels
- le type du  $k^{\text{ième}}$  paramètre effectif doit être compatible pour l'affectation avec le type du  $k^{\text{ième}}$  paramètre formel.
- l'ordre d'évaluation des paramètres effectifs n'est pas spécifié par la norme C.

→ se méfier des effets de bords éventuels comme dans l'exemple suivant :  $x = f(i++, i++)$ .

### **Appel par valeur en C**

- C'est le seul mode d'appel en C. Donc tout appel de fonction est « par valeur »

### **Appel par nom en C**

- Le passage d'arguments par nom doit être explicitement pris en charge par le programmeur.

*A voir plus tard après la notion de pointeur*

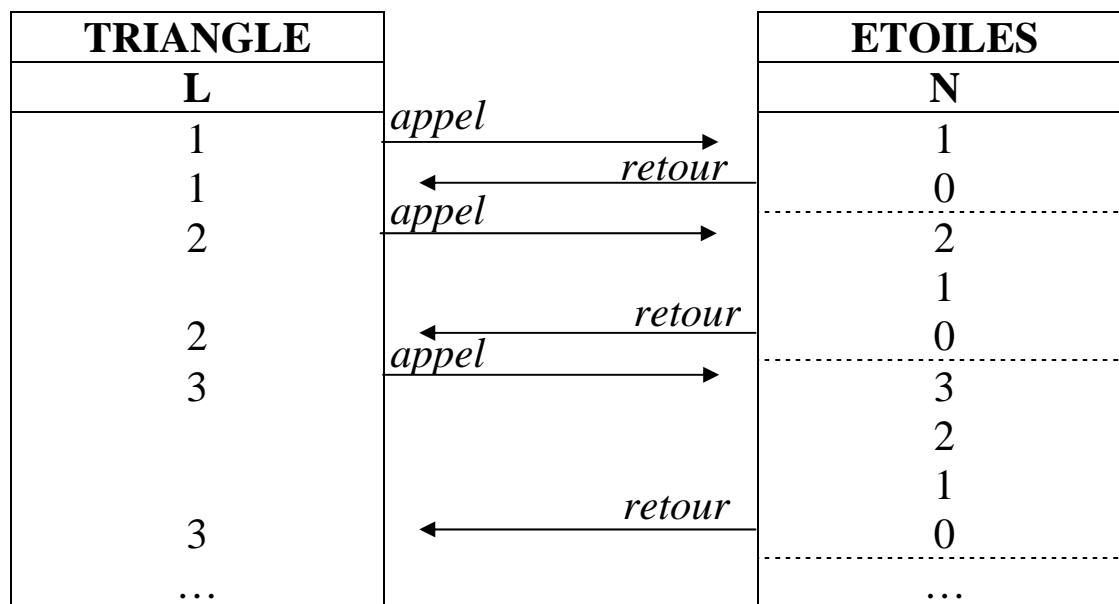
## Appel par valeur : exemple en C

```
void ETOILES(int N)
{while (N>0)
  {printf("*");
   N--;}
 printf("\n");
}

void TRIANGLE(void)
{int L;
 for(L=1; L<10; L++) ETOILES(L);
}
```

La fonction ETOILES dessine une ligne de N étoiles. Le paramètre N est modifié à l'intérieur de la fonction. Le passage par *valeur* permet d'utiliser les paramètres comme des variables locales bien initialisées. De cette façon, on peut utiliser N comme compteur et

La fonction TRIANGLE, appelle la fonction ETOILES en utilisant la variable L comme paramètre. A l'appel, la *valeur* de L est copiée dans N. N peut donc être décrétementée à l'intérieur de ETOILES, sans influencer la valeur originale de L.



## Renvoi du résultat

- Toutes les fonctions fournissent un résultat d'un type qui doit être déclaré.
- Une fonction peut renvoyer une valeur d'un type simple ou l'adresse d'une variable ou d'un tableau.

*return expression ;*

- évaluation de *expression*
- conversion automatique du résultat de l'expression dans le type de la fonction
- renvoi du résultat
- fin de la fonction

```
#include <math.h> /* pour sin et cos */
```

```
double TAN(double X)
{
    if (cos(X) != 0)
        return sin(X)/cos(X);
    else
        printf("Erreur !\n");
}
```

```
printf("La tangente de %f est %f \n", X, TAN(X));
...
COT = 1/TAN(X);
```

la fonction TAN calcule la tangente d'un réel X à l'aide des fonctions **sin** et de **cos** de la bibliothèque *<math>*.

Un appel à la fonction TAN peut être intégré dans des calculs ou des expressions:

## ***Retour ignoré ou void***

```
void LIGNE(int L)
{
  /* Déclarations des variables locales */
  int I;
  /* Traitements */
  for (I=0; I<L; I++)printf("*");
  printf("\n");
}
```

L'utilisateur est libre d'accepter le résultat d'une fonction ou de l'ignorer. Ainsi la fonction **scanf** renvoie comme résultat le nombre de données correctement reçues et on peut utiliser ce résultat comme contrôle:

```
int JOUR, MOIS, ANNEE;
int RES;
do
{
  printf("Entrez la date actuelle : ");
  RES = scanf("%d %d %d", &JOUR,&MOIS,&ANNEE);
}
while (RES != 3);
```

## Renvoi du résultat (fin)

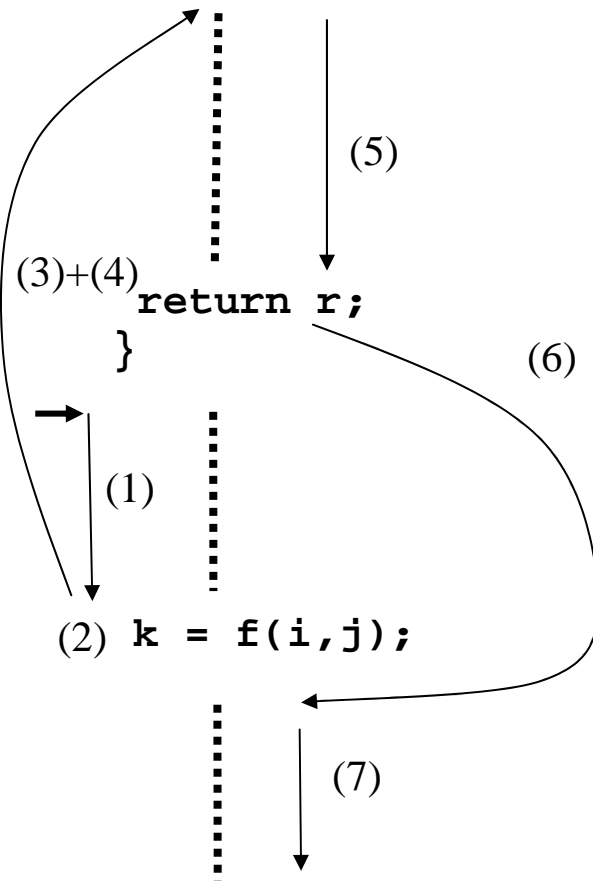
- les programmes renvoient la valeur zéro comme code d'erreur s'ils se terminent avec succès. Des valeurs différentes de zéro indiquent un arrêt fautif ou anormal.
- Si on quitte une fonction (d'un type différent de **void**) sans renvoyer de résultat à l'aide de **return**, la valeur transmise à la fonction appelante est indéfinie. Le résultat d'une telle action est imprévisible si une erreur fatale s'est produite à l'intérieur d'une fonction.
- On peut utiliser la fonction **exit** qui est définie dans la bibliothèque `<stdlib>`. **exit** nous permet d'interrompre l'exécution du programme en fournissant un code d'erreur à l'environnement.
- Il est dangereux de déclarer la fonction TAN comme nous l'avons fait plus haut: Le cas d'une division par zéro, est bien intercepté et reporté par un message d'erreur, mais l'exécution du programme continue 'normalement' avec des valeurs incorrectes.

```
#include <math.h>

double TAN(double X)
{if (cos(X) != 0)
    return sin(X)/cos(X);
  else {printf("\aFonction TAN:\n"
              "Erreur: Division par zéro !\n");
        exit(-1); /* Code erreur -1 */
      }
}
```

## Gestion de la pile et appel de fonction

```
int f(int a, int b)
{int x;
```



1. Exécuter les instructions dans l'ordre
2. Interrompre momentanément l'exécution des instructions de la séquence en cours,
3. Passer les paramètres :  
 $a \leftarrow i$  et  $b \leftarrow j$
4. Se dérouter vers le point d'entrée de la fonction
5. Exécuter les instructions du bloc de la fonction . Lorsque l'instruction `return` est rencontrée, sauvegarder le résultat  $r$  et terminer la fonction
6. Revenir à la séquence appelante, et récupérer le résultat :  $k \leftarrow r$
7. Reprendre l'exécution des instructions de la séquence appelante à partir de celle qui suit immédiatement l'appel de la fonction.

Sauvegarder l'adresse de l'instruction du programme qui suit l'appel de fonction (adresse de retour) pour pouvoir y revenir en (6)

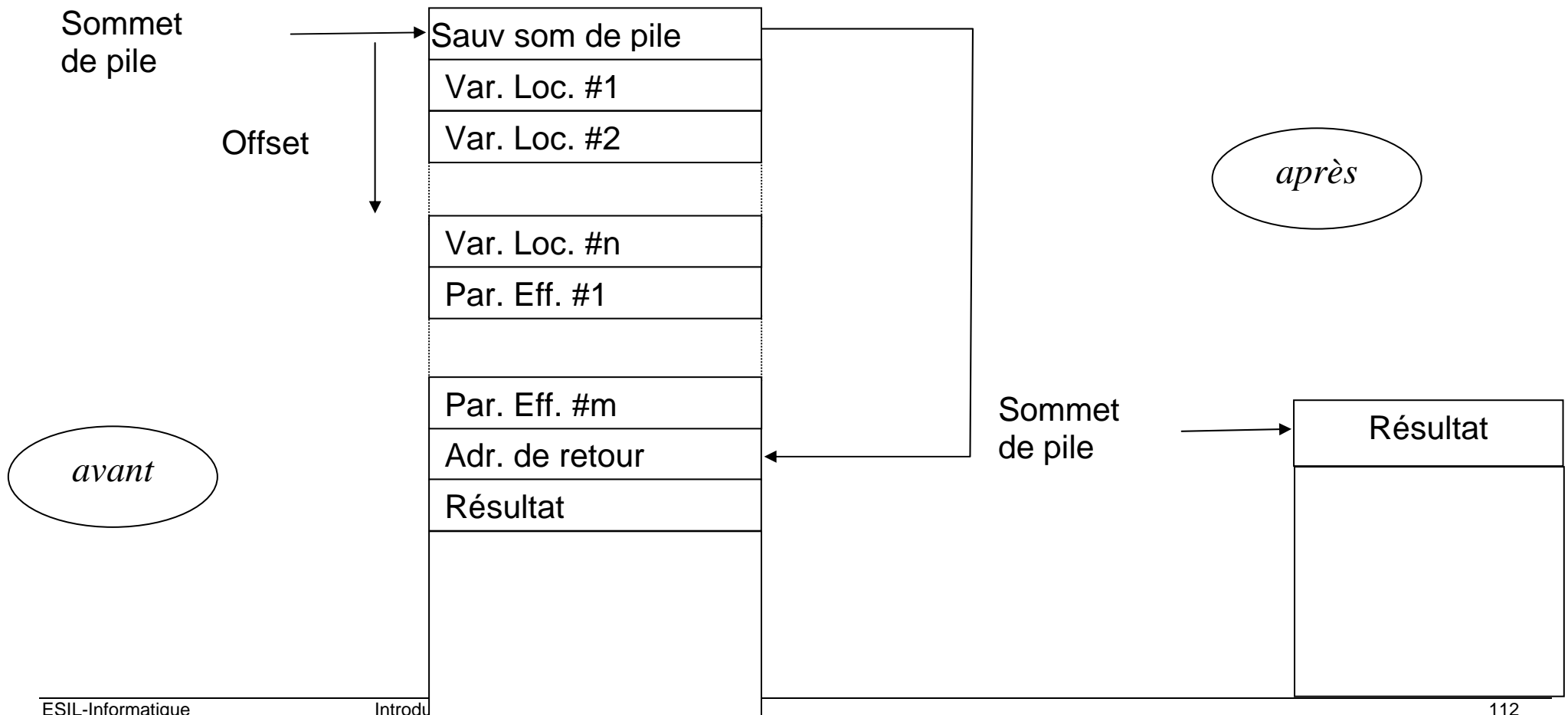
Disposer d'une zone de mémoire suffisante pour y loger les paramètres formels et les variables locales de la fonction

Disposer d'une zone de mémoire suffisante pour y loger le résultat.

## Gestion de la pile et appel de fonction (suite)

- appels de fonctions imbriqués, une fonction pouvant rappeler une autre fonction (ou elle-même).
- gérer dynamiquement des instances du processus ci-dessus

→ utiliser une pile pour stocker et gérer toutes ces informations.



```

FB(int U, int V)
{...⑤...
  return 28}

```

```

FA(int M, int N)
{int K ;
  ...③...
  K = FB(I,J) ;
  ...④ ...}

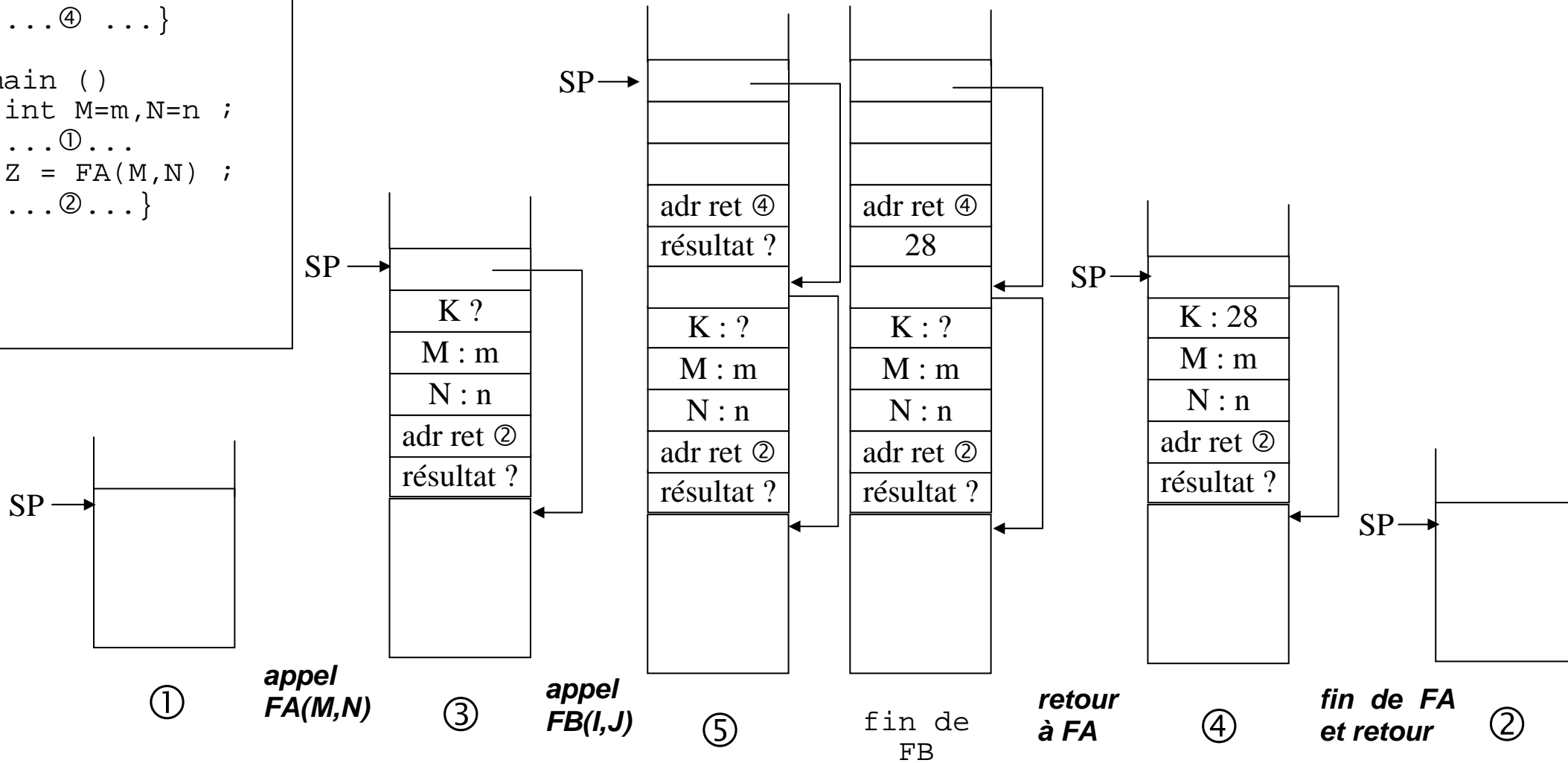
```

```

main ()
{int M=m,N=n ;
  ...①...
  Z = FA(M,N) ;
  ...②...}

```

### Gestion de la pile et appel de fonction (fin)



## **Fonctions mathématiques standard : (option `-lm` à la compilation)**

```
#include <math.h>
```

**Type des données** : arguments et résultats de ces fonctions sont du type **double**.

<b>exp(X)</b>	fonction exponentielle : $e^X$
<b>log(X)</b>	logarithme neperien : $\ln(X)$ , $X > 0$
<b>log10(X)</b>	logarithme à base 10 : $\log_{10}(X)$ , $X > 0$
<b>pow(X,Y)</b>	X exposant Y : $X^Y$
<b>sqrt(X)</b>	racine carrée de X $\sqrt{X}$ pour $X > 0$
<b>fabs(X)</b>	valeur absolue de X : $ X $
<b>floor(X)</b>	arrondi à l'entier inférieur ou égal
<b>ceil(X)</b>	arrondi à l'entier supérieur ou égal
<b>fmod(X,Y)</b>	reste rationnel de X/Y (signe de X $X \neq 0$ )
<b>sin(X) cos(X) tan(X)</b>	
<b>asin(X) acos(X) atan(X)</b>	
<b>sinh(X) cosh(X) tanh(X)</b>	