

Chapitre 6

Les fonctions : première approche

Plan

1. Notion de fonction

1. Appel de fonction

2. Définition de fonction

3. Déclaration de fonction

4. Valeurs retournées par une fonction

5. Paramètres d'une fonction

2. Variables locales/globales

3. Organisation de la mémoire

Exemple

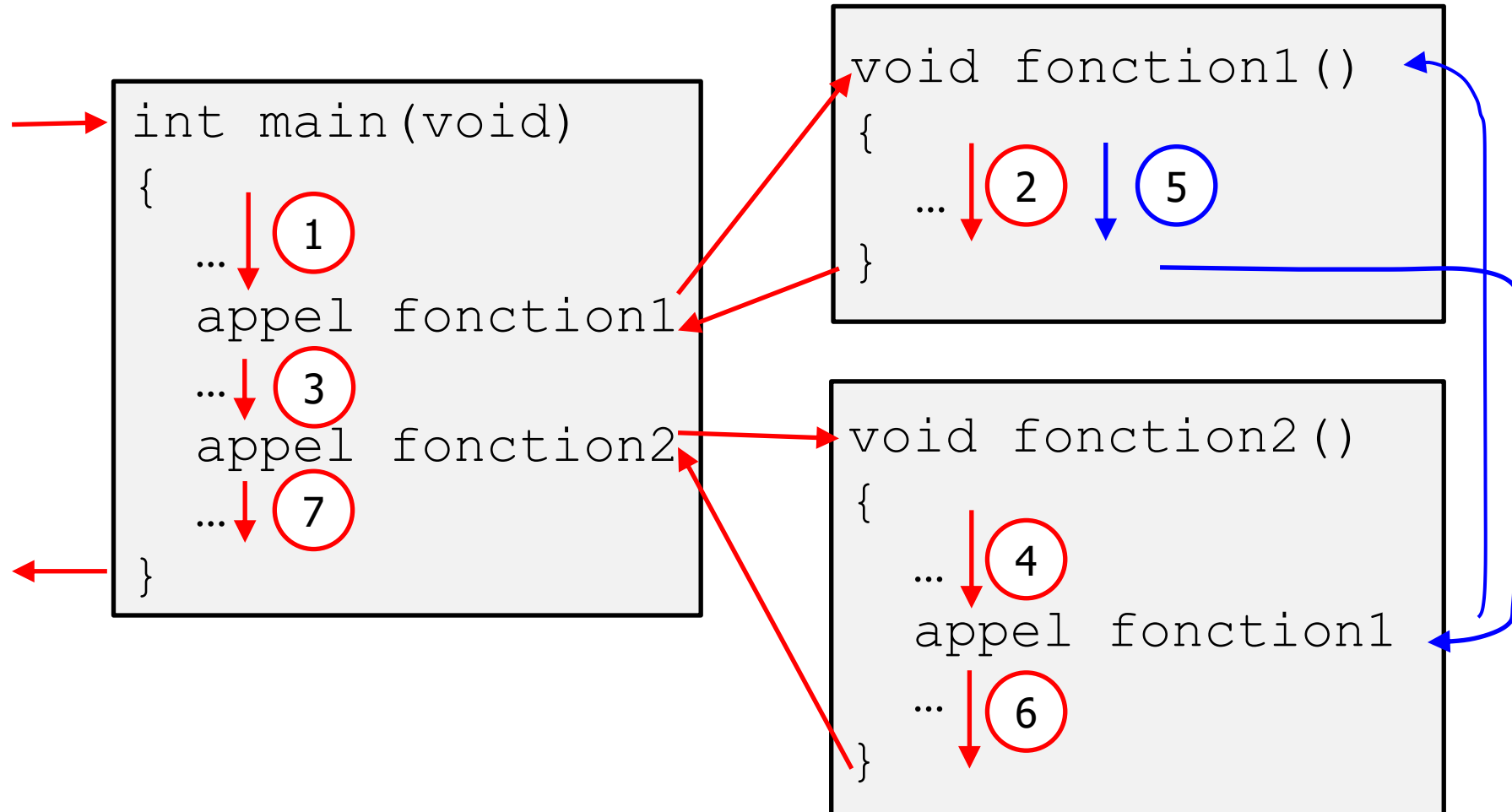
```
int f(int x)
{
    return(x*x + 3*x - 25);
}
```

$$f : x \rightarrow x^2 + 3x - 25$$
$$f(x) = x^2 + 3x - 25$$

```
int main(void)
{
    int val = 3;
    int y = f(val); // int y = val*val+3*val-25;
    y = f(34);      // y = 34*34+3*34-25;

    return 0;
}
```

Décomposition en fonctions



Avantages

Décomposition de longs programmes en sous-programmes

Factorisation : rassemblement de **suites d'instructions identiques** dispersées dans un programme, en une fonction

Meilleure lisibilité des programmes

Facilitent la maintenance du code

Les fonctions peuvent être **réutilisées** dans d'autres programmes

Appel de fonction

Fonction **sin(x)**, prédéfinie dans une bibliothèque

```
#include <math.h>
printf( "%lf", sin( 45 * M_PI/180 ) );
```

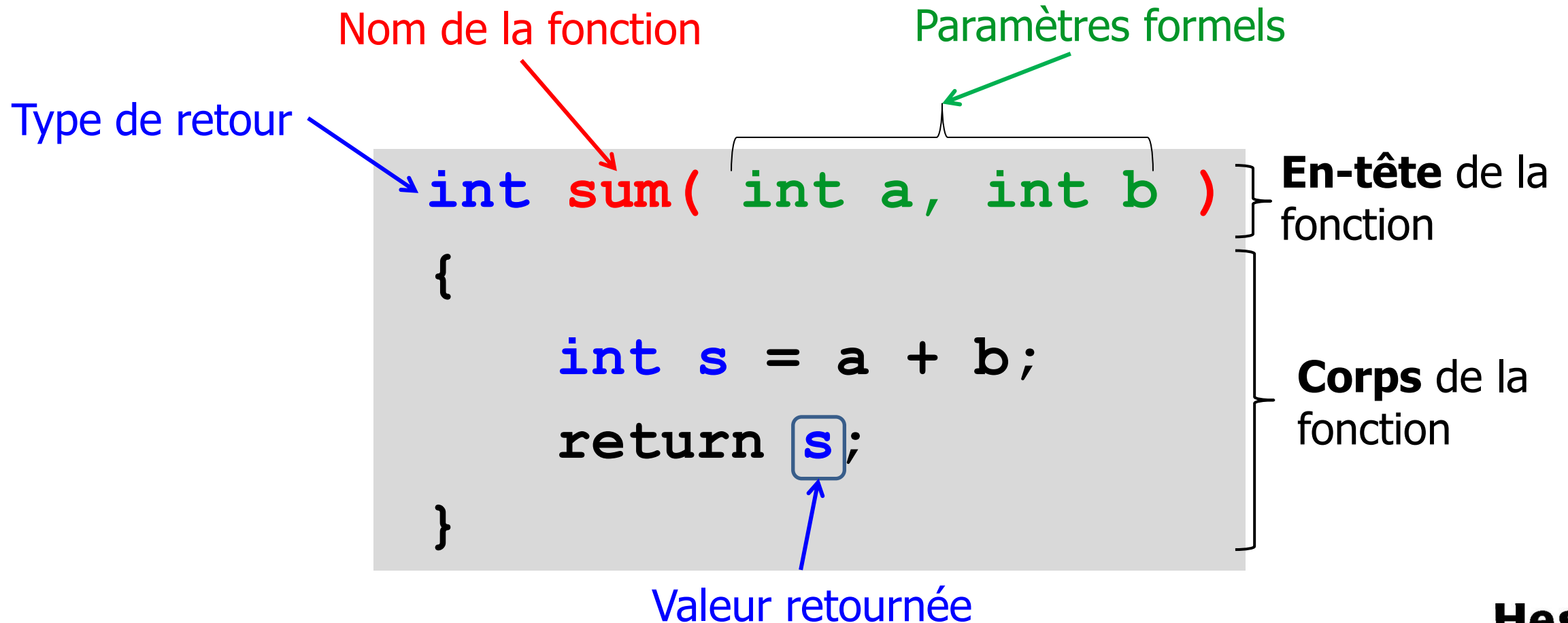
Fonction **sum(x,y)**, définie par l'utilisateur

```
int n = 4, result = 0;
result = sum( n, 37 );
```

Nom de la fonction

Paramètres effectifs

Définition de fonction



Où définir les fonctions ?

IMPORTANT

Une fonction doit être **déclarée**, et donc son en-tête doit être connue par le compilateur, avant son premier appel

Une fonction peut être **définie**

A) Dans le fichier où elle est appelée

1) Avant le premier appel

2) Après le premier appel

B) Dans un fichier différent (voir chapitre 07 sur la modularisation)

Où définir les fonctions ?

A) Dans le même fichier, la fonction peut être définie

Avant son appel

Après son appel

déclaration
et
définition

```
int f( int nb )  
{  
    ...;  
}
```

```
int main(void)  
{  
    int x=f(4); //APPEL  
    return 0;  
}
```

```
int f( int nb );
```

```
int main(void)  
{  
    int x=f(4); //APPEL  
    return 0;  
}
```

```
int f( int nb )  
{  
    ...;  
}
```

déclaration

définition

Où définir les fonctions ?

B) Dans un fichier différent

main.c

```
#include "foo.h"

int main(void)
{
    int x=f(4); //APPEL
    return 0;
}

...
```

foo.h

```
int f( int nb );
```

Déclaration

foo.c

```
#include "foo.h"
```

```
int f( int nb )
{
    ...;
}
```

Définition

Déclaration de fonction

On **déclare une fonction** avec son **prototype**

Syntaxe d'un prototype

```
TypeDeRetour nomFonction ( Type1 [nomParametre1] ,  
                             Type2 [nomParametre2] ,  
                             ...  
                             TypeN [nomParametreN]  
                             ) ;
```

Les **noms des paramètres sont facultatifs** dans le prototype.

Le compilateur vérifie l'adéquation entre le prototype et les appels.

Le nom de fonction doit être unique. On ne peut pas définir deux fonctions avec le même nom : pas de surcharge.

Valeur retournée

Instruction : **return**

⚠ s'il manque une instruction **return**, la valeur de retour est indéterminée.

L'exécution de toute fonction se **termine** soit

- à l'exécution d'une instruction **return**
- à la fin du bloc d'instructions, marqué par **}**
- par un **exit** (⚠ le programme complet s'arrête)

Remarque : une fonction peut avoir plusieurs **return**

Type de retour

Fonction qui **retourne une valeur**

```
int inc( int nb )  
{  
    return nb+1;  
}
```

```
int main(void)  
{  
    int x = inc(4);  
    return 0;  
}
```

Fonction qui **ne retourne rien**

```
void show( int nb )  
{  
    printf("%d",nb);  
}
```

```
int main(void)  
{  
    show(4);  
    return 0;  
}
```

Type de retour

```
TypeDeRetour FunctionName (...)
```

Exemples

```
void show (...)  
int inc (...)
```

TypeDeRetour : entier, réel, pointeur, **struct**, **void**

TypeDeRetour doit être compatible avec la valeur retournée par **return**

⚠ **Toujours indiquer le type de retour.** Sinon, c'est un entier par défaut

Paramètres formels

```
FuncName (type1 nomP1, type2 nomP2, ...) ;
```

Fonction sans paramètre formel → **void**

```
showMenu (void) ;
```

Les paramètres peuvent être **de n'importe quel type**

```
interest(float money, double rate, int time) ;
```

Remarque : **printf**, **scanf** acceptent un nombre **variable** de paramètres (voir le chapitre 8)

Paramètres effectifs

```
nomFonction( <Liste paramètres effectifs> )
```

Lors de l'appel → paramètres effectifs

Doivent être de type compatible avec les paramètres formels

Peuvent être des expressions, des constantes ou des variables

```
somme ( a + b , pow ( 2 , n ) ) ;
```



Si la fonction retourne une valeur, elle peut être :

affectée à une autre variable

```
x = sum ( nb1 , nb2 ) ;
```

utilisée dans une expression

```
x = y + sum ( nb1 , nb2 ) + z ;
```

```
x = sqrt ( sum ( nb1 , nb2 ) ) ;
```


Plan

1. Notion de fonction

1. Appel de fonction
2. Définition de fonction
3. Déclaration de fonction
4. Valeurs retournées par une fonction
5. Paramètres d'une fonction

2. Variables locales / globales

3. Organisation de la mémoire

Visibilité des variables

Les variables sont visibles dans le bloc où elles sont déclarées.

```
int x,y,z; // Variable globales

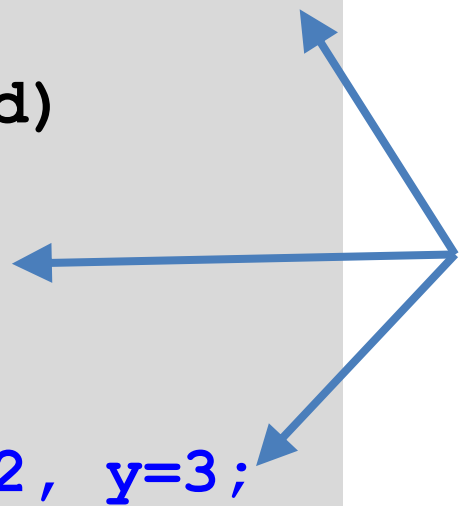
void f1(void)
{
    int x,z; // Variables locales
}

void f(void)
{
    int y,z; // Variables locales
}
```

```
int x=3,y=2,z=10;

void f(void)
{
    int x=1;
    {
        int x=2, y=3;
    }
}
```

x?, y?, z?



Visibilité des variables

Variables globales

déclarées hors de toute fonction → on parle de portée globale
sont visibles partout
existent jusqu'à la fin du programme

Variables locales

déclarées dans un bloc → corps d'une fonction
sont **visibles** dans le bloc où elles sont déclarées
masquent les variables déclarées hors du bloc
leur **durée de vie** s'étend de leur déclaration jusqu'à la sortie du bloc :
} ou **return**

Dangerosité des variables globales

```
void print_stars(int);

int main(void)
{
    int i; // Variable locale
    for(i=0; i<5; i++)
        print_stars(5);
    return 0;
}

void print_stars(int n)
{
    int i; // Variable locale
    for(i=0; i<n; i++)
        printf("*");
    printf("\n");
}
```

```
void print_stars(int);
int i; // Variable globale

int main(void)
{
    for(i=0; i<5; i++)
        print_stars(5);
    return 0;
}

void print_stars(int n)
{
    for(i=0; i<n; i++)
        printf("*");
    printf("\n");
}
```

Visibilité des variables

Une variable **locale** déclarée avec le mot clé **static**

n'est visible que dans la fonction où elle est déclarée

→ comme une variable locale

Conserve son contenu entre deux appels

→ comme une variable globale

est initialisée à 0 par défaut à sa 1^{ère} utilisation

→ comme une variable globale

Les variables statiques permettent de gérer la persistance tout en ayant un statut privé

```
void f1(void)
{
    static int x=15;
    printf("%d ", ++x);
}

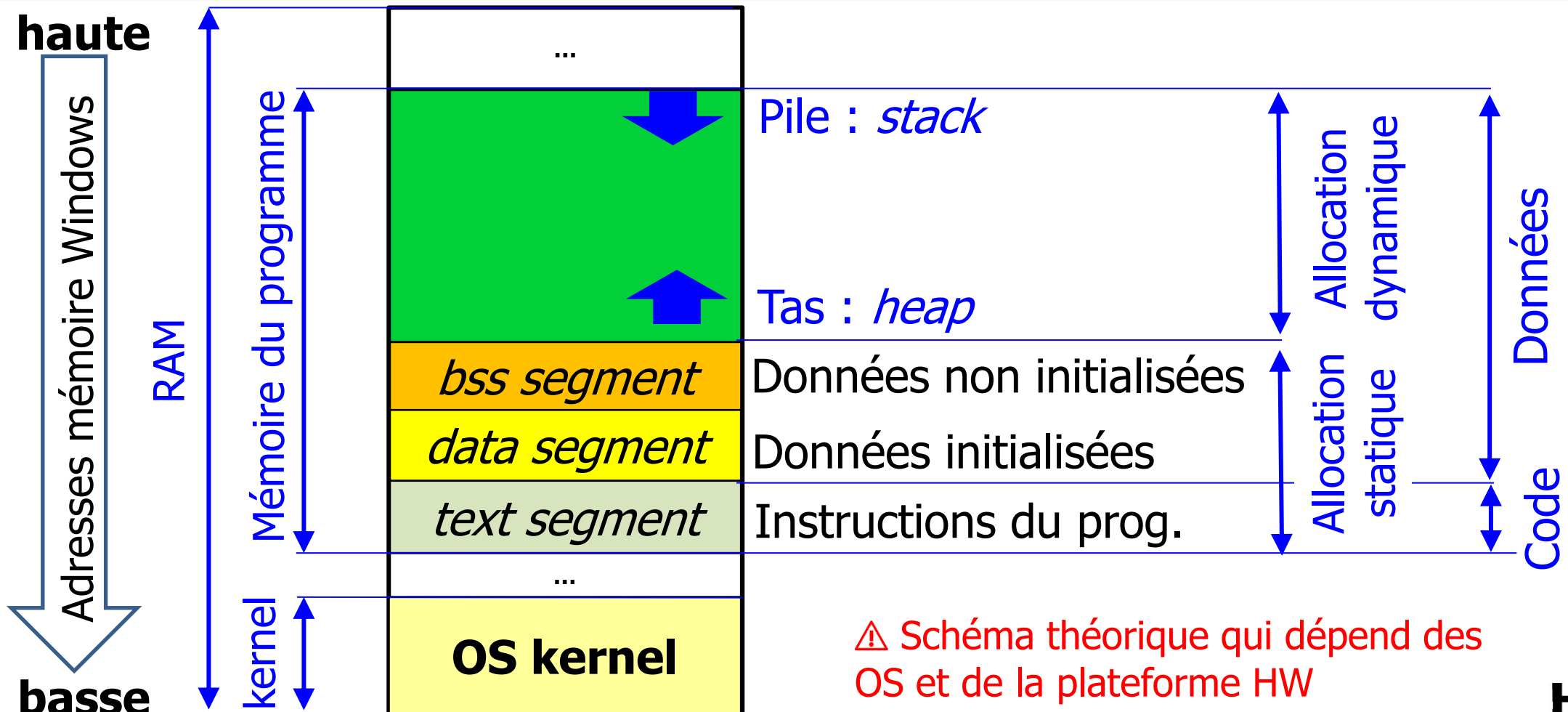
int main(void)
{
    f1(); f1(); f1();
    return 0;
}
```

16 17 18

Plan

1. Notion de fonction
 1. Appel de fonction
 2. Définition de fonction
 3. Déclaration de fonction
 4. Valeurs retournées par une fonction
 5. Paramètres d'une fonction
2. Variables locales / globales
- 3. Organisation de la mémoire**

Organisation mémoire



⚠ Schéma théorique qui dépend des OS et de la plateforme HW

Variables globales & statiques

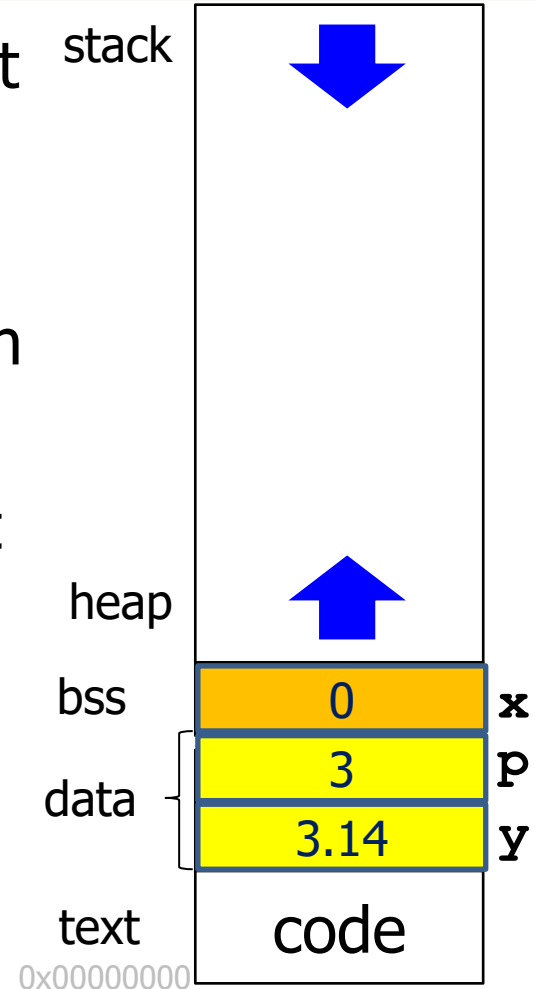
La **mémoire statique** (segments *data* et *bss*) contient
les **variables globales**
les **variables locales static**

La mémoire statique est attribuée lors de la compilation

Une **variable globale initialisée** va dans **data**

Une **variable globale non initialisée** va dans **bss** et
prend la **valeur 0**

```
int x;  
float y=3.14; int p=3;
```



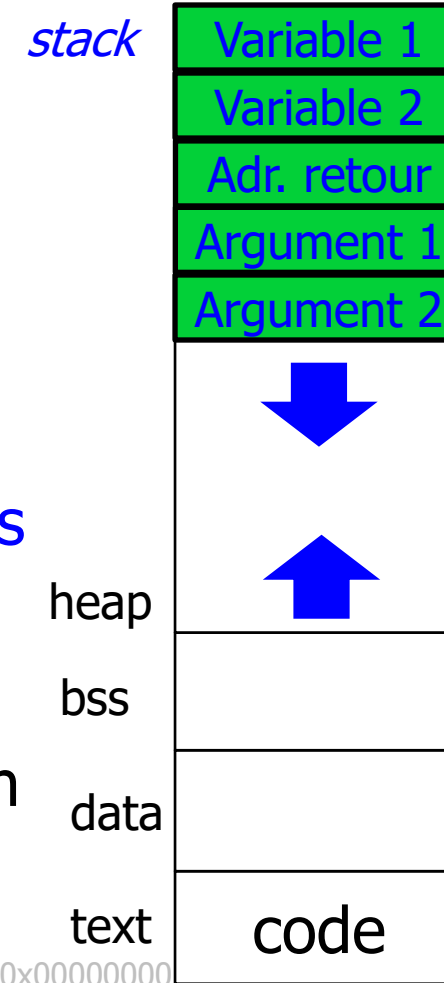
Variables locales (automatiques)

La **pile "stack"** contient

- les **variables locales**
- les **adresses de retour des fonctions**
- les **arguments des fonctions**

Si les variables locales **ne sont pas initialisées** explicitement, alors elles ont une valeur indéterminée

L'allocation mémoire change à chaque appel de fonction

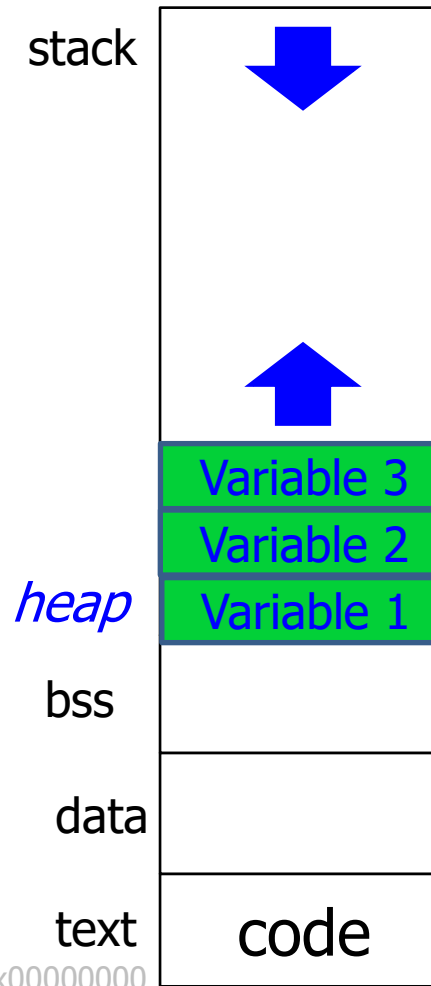


Variables **dynamiques**

La **durée de vie** de ces variables est **gérée par le programmeur**

Leur existence commence grâce à la fonction **malloc** et se termine avec la fonction **free**

La zone d'allocation s'appelle le tas (**heap**)
Cette zone mémoire est une réserve dans laquelle le programmeur peut puiser durant l'exécution

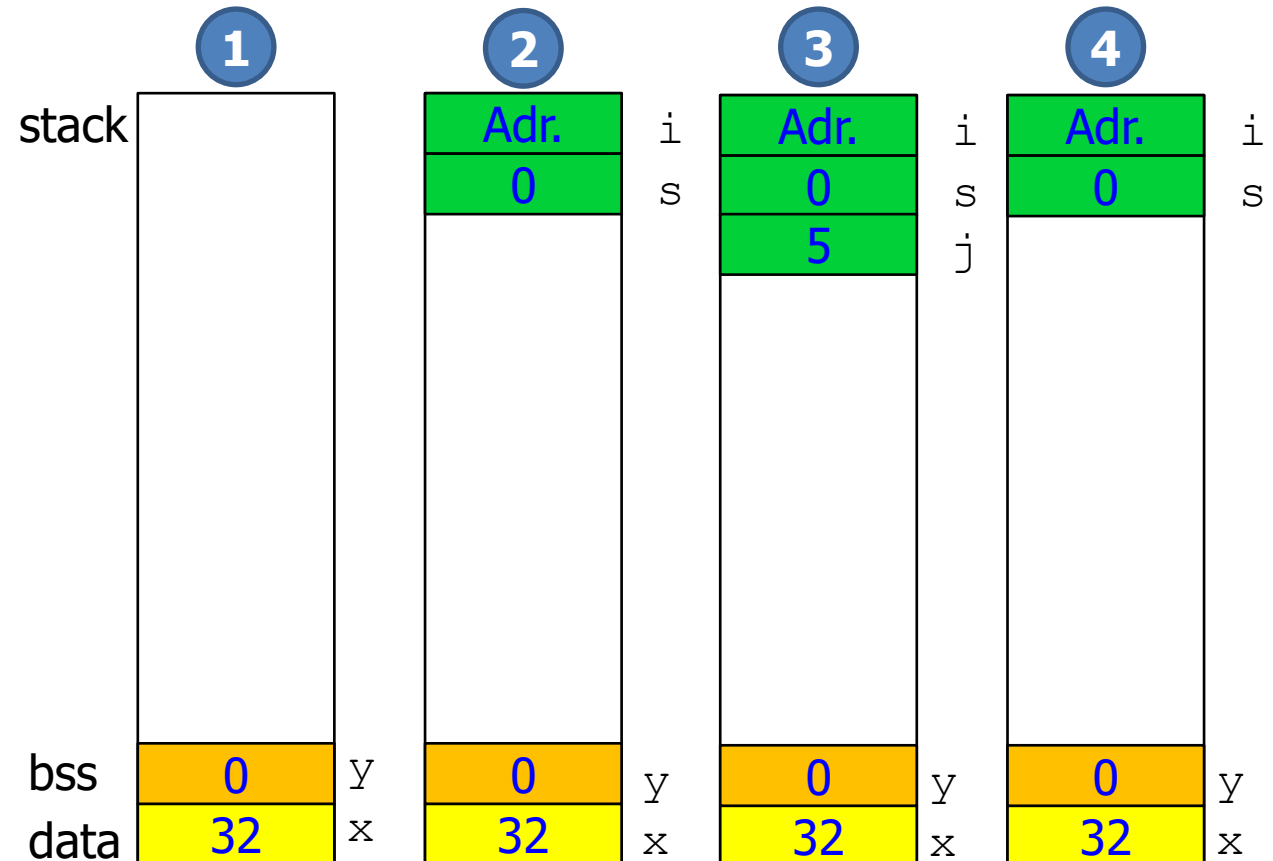


0x00000000

Allocation mémoire

Exemple : variables locales, globales

```
int y;  
int x=32;  
int main(void)  
{  
  ① int i;  
  int s=0;  
  if(s==0) ②  
  {  
    int j=5; ③  
  } ④  
  return 0;  
}
```

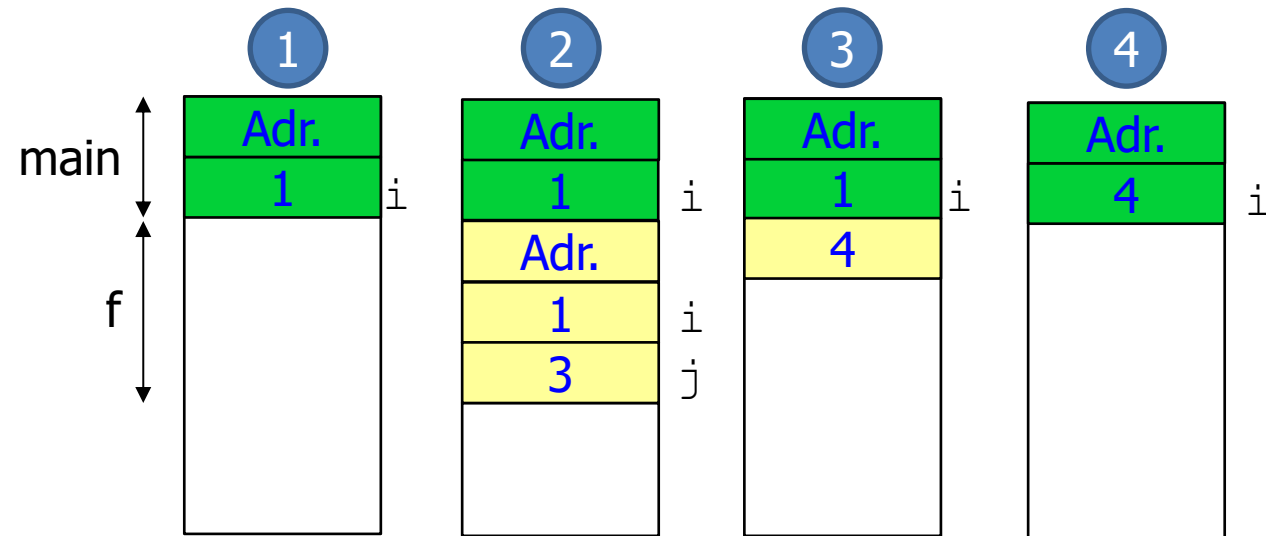


Allocation mémoire

Exemple : fonctions

```
int main(void)
{
    int i=1;
    i=f(i,3);
    return 0;
}
```

```
int f(int i,int j)
{
    return i+j;
}
```



Passage de paramètres

Deux modes de passage de paramètres

1) Passage par **valeur**

en donnée / en entrée – *i.e.* le compilateur passe la valeur à la fonction

2) Passage par **adresse**

en donnée / résultat, en entrée/sortie, *i.e.* le compilateur passe l'adresse de la variable. Donc la fonction peut lire et modifier la valeur de la variable

En C, le passage des paramètres ne se fait que par valeur

Passage par adresse : Voir chapitre Fonction II

Exercices



Exercices du chapitre 06