Structures et Récursivité

Pierre-Alain FOUQUE
Département d'Informatique
École normale supérieure

Tableaux

Les tableaux permettent

- de définir sous un nom unique un groupe d'objets de même type
- d'accéder à chaque objet par sa position dans le tableau

Tableaux

Les tableaux permettent

- de définir sous un nom unique un groupe d'objets de même type
- d'accéder à chaque objet par sa position dans le tableau

Comment faire pour regrouper des objets de types différents ?

Structures

Les structures permettent de regrouper des objets de types différents

⇒ nouveau type : **`struct bloc`**

Structures

Les structures permettent de regrouper des objets de types différents

⇒ nouveau type : **`struct bloc`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

Structures

Les structures permettent de regrouper des objets de types différents

⇒ nouveau type : **`struct bloc`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

Chaque objet de ce type contient

Structures

Les structures permettent de regrouper des objets de types différents

⇒ nouveau type : **`struct bloc`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

- Chaque objet de ce type contient
  - un champ entier, appelé **`nombre`**

Structures

Les structures permettent de regrouper des objets de types différents

⇒ nouveau type : **`struct bloc`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

Chaque objet de ce type contient

- un champ entier, appelé **`nombre`**
- un champ flottant, appelé **`valeur`**

Utilisation

`struct bloc bl;`
définit une variable **`bl`**
de type **`struct bloc`**

Le champ **`nombre`** est alors accessible par **`bl.nombre`**

et le champ **`valeur`** par **`bl.valeur`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

Utilisation

`struct bloc bl;`
définit une variable **`bl`**
de type **`struct bloc`**

Le champ **`nombre`** est alors accessible par **`bl.nombre`**

et le champ **`valeur`** par **`bl.valeur`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

```
struct bloc bl;

bl.nombre = 10;
bl.valeur = 3.2;
```

Nouveau type avec `typedef`

Le nouveau type créé s'appelle
`struct bloc`
⇒ lourd à utiliser

```
typedef struct bloc sbloc;
```
définit un synonyme : **`sbloc`**

Nouveau type avec `typedef`

Le nouveau type créé s'appelle
`struct bloc`
⇒ lourd à utiliser

`typedef struct bloc sbloc;`
définit un synonyme : **`sbloc`**

```
struct bloc {
   int nombre;
   float valeur;
};

typedef struct bloc sbloc;
```

Nouveau type avec `typedef`

Le nouveau type créé s'appelle
`struct bloc`
⇒ lourd à utiliser

`typedef struct bloc sbloc;`
définit un synonyme : **`sbloc`**

```
struct bloc {
   int nombre;
   float valeur;
};

typedef struct bloc sbloc;
```

ou

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

Nouveau type avec typedef

Le nouveau type créé s'appelle
struct bloc
⇒ lourd à utiliser

```
sbloc bl;

bl.nombre = 10;
bl.valeur = 3.2;
```

typedef struct bloc sbloc;
définit un synonyme : **sbloc**

```
struct bloc {
   int nombre;
   float valeur;
};

typedef struct bloc sbloc;
```

ou

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

Exemple I

On peut définir un point du plan par ses coordonnées :

Exemple I

On peut définir un point du plan par ses coordonnées :

```
typedef struct {
   float abscisse;
   float ordonnée;
} point2D;
```

Exemple I

On peut définir un point du plan par ses coordonnées :

```
typedef struct {
   float abscisse;
   float ordonnée;
} point2D;
```

On peut ensuite déclarer et initialiser un tel objet :

Exemple I

On peut définir un point du plan par ses coordonnées :

```
typedef struct {
   float abscisse;
   float ordonnée;
} point2D;
```

On peut ensuite déclarer et initialiser un tel objet :

```
   point2D P;

   P.abscisse = 2.5;
   P.ordonnee = 4.3;
```

Exemple I – suite

Une structure est alors un type comme les autres :

Exemple I – suite

Une structure est alors un type comme les autres :

```
point2D translation (point2D a, point2D b)
{
   point2D c;
   c.abscisse = a.abscisse + b.abscisse;
   c.ordonnee = a.ordonnee + b.ordonnee;
   return c;
}
```

Déclaration et initialisation

```
sbloc bl;
```

déclare une variable **bl** de type **sbloc**

Son initialisation peut se faire champ par champ, ou globalement :

```
sbloc bl2 = { 10, 3.2 };
```

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

Déclaration et initialisation

`sbloc bl;`

déclare une variable **`bl`** de type **`sbloc`**

Son initialisation peut se faire champ par champ, ou globalement :

`sbloc bl2 = { 10, 3.2 };`

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

```
sbloc bl2 = { 10, 3.2 };

sbloc bl;
bl.nombre = 10;
bl.valeur = 3.2;
```

Exemple II - complexes

On peut définir le type complexe par :

Exemple II – complexes

On peut définir le type complexe par :

```
typedef struct {
   float reel;
   float im;
} complexe;
```

Exemple II – complexes

On peut définir le type complexe par :

```
typedef struct {
   float reel;
   float im;
} complexe;
```

On peut ensuite déclarer et initialiser un tel objet :

Exemple II – complexes

On peut définir le type complexe par :

```
typedef struct {
   float reel;
   float im;
} complexe;
```

On peut ensuite déclarer et initialiser un tel objet :

```
complexe c;

c.reel = 2.5;
c.im = 4.3;
```

Exemple II – suite

On peut alors définir l'addition :

Exemple II – suite

On peut alors définir l'addition :

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.reel = c1.reel + c2.reel;
   c.im   = c1.im   + c2.im;
   return c;
}
```

Exemple II – suite

On peut alors définir l'addition :

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.reel = c1.reel + c2.reel;
   c.im   = c1.im   + c2.im;
   return c;
}
```

```
   complexe c,d;

   c.reel = 2.5;
   c.im = 4.3;
   d = addition(c,c);
```

Exemple II – exécution

Exemple II – exécution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.reel = c1.reel + c2.reel;
   c.im   = c1.im   + c2.im;
   return c;
}
```

Exemple II – exécution

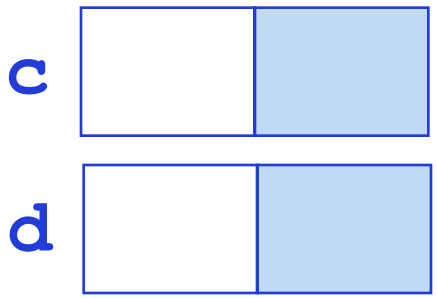
```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.reel = c1.reel + c2.reel;
   c.im   = c1.im   + c2.im;
   return c;
}
```

```
complexe c,d;

c.reel = 2.5;
c.im = 4.3;
d = addition(c,c);
```

Exemple II – exécution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.reel = c1.reel + c2.reel;
   c.im   = c1.im   + c2.im;
   return c;
}
```

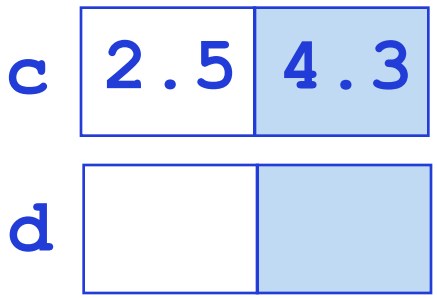
```
complexe c,d;

c.reel = 2.5;
c.im = 4.3;
d = addition(c,c);
```



Exemple II – exécution

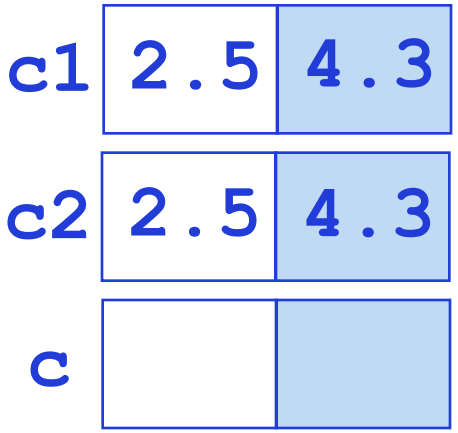
```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.reel = c1.reel + c2.reel;
   c.im   = c1.im   + c2.im;
   return c;
}
```

```
complexe c,d;

c.reel = 2.5;
c.im = 4.3;
d = addition(c,c);
```

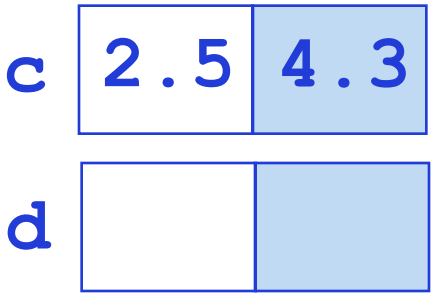


Exemple II – exécution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.reel = c1.reel + c2.reel;
   c.im   = c1.im   + c2.im;
   return c;
}
```

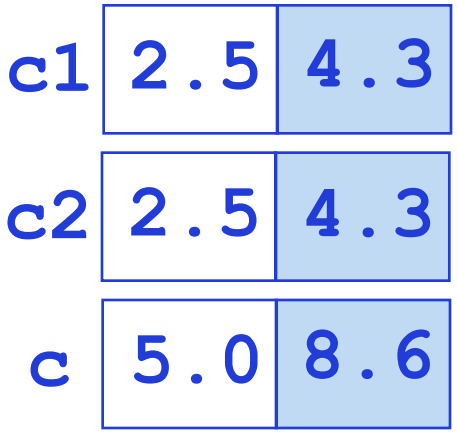



```
complexe c,d;

c.reel = 2.5;
c.im = 4.3;
d = addition(c,c);
```

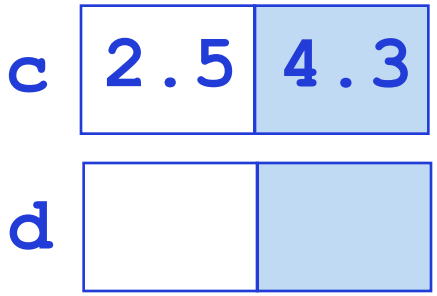


Exemple II – exécution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.reel = c1.reel + c2.reel;
   c.im   = c1.im   + c2.im;
   return c;
}
```



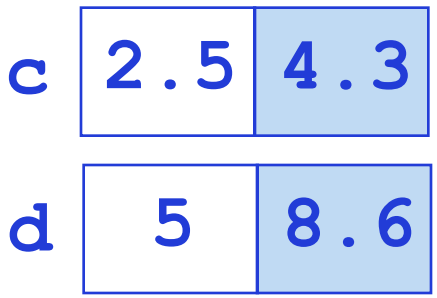

```
complexe c,d;

c.reel = 2.5;
c.im = 4.3;
d = addition(c,c);
```



Exemple II – exécution

```
complexe addition (complexe c1, complexe c2)
{
   complexe c;
   c.reel = c1.reel + c2.reel;
   c.im   = c1.im   + c2.im;
   return c;
}
```

```
complexe c,d;

c.reel = 2.5;
c.im = 4.3;
d = addition(c,c);
```



Affichage

Pour afficher une structure,
il faut afficher les champs un à un
(comme les cases d'un tableau)

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

Affichage

Pour afficher une structure,
il faut afficher les champs un à un
(comme les cases d'un tableau)

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

```
void affiche (sbloc bl) {
   printf(``%d - %f \n '',
            bl.nombre,
            bl.valeur);
}
```

Exemple II – affichage

On rappelle le type complexe :

```
typedef struct {
   float reel;
   float im;
} complexe;
```

Exemple II – affichage

On rappelle le type complexe :

```
typedef struct {
   float reel;
   float im;
} complexe;
```

On peut afficher un tel objet :

Exemple II – affichage

On rappelle le type complexe :

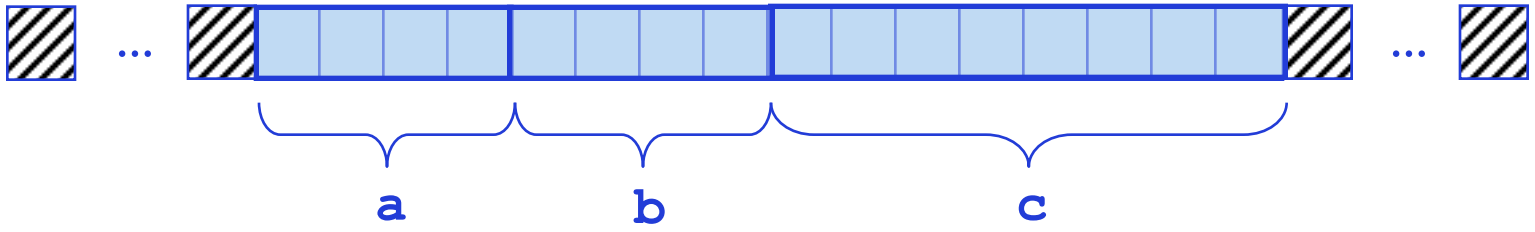
```
typedef struct {
   float reel;
   float im;
} complexe;
```

On peut afficher un tel objet :

```
void affiche (complexe c) {
  printf(``%f + i * %f \n '',
          c.reel, c.im);
}
```

Pointeurs

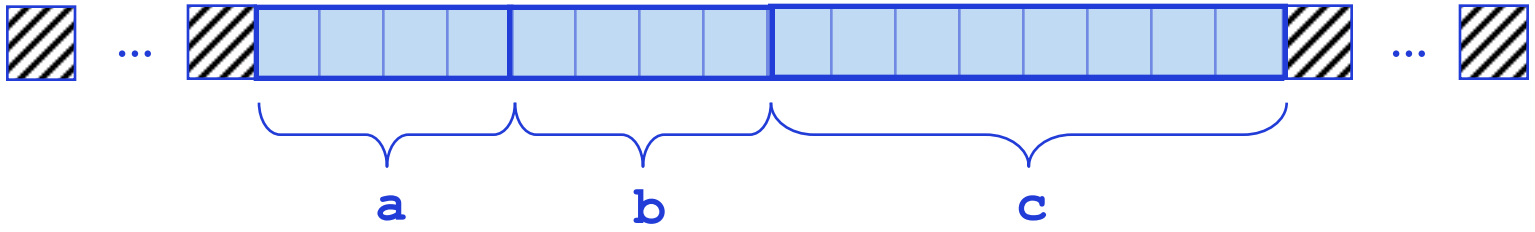
```
int a,b;
long long c;
```



Pointeurs

Une variable est stockée dans une zone mémoire réservée lors de la déclaration

```
int a,b;
long long c;
```

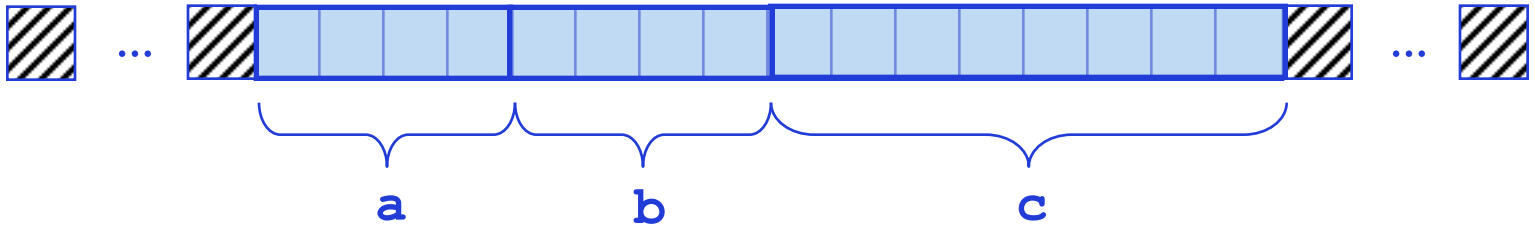


Pointeurs

Une variable est stockée dans une zone mémoire réservée lors de la déclaration

`&a` indique l'adresse de cette zone

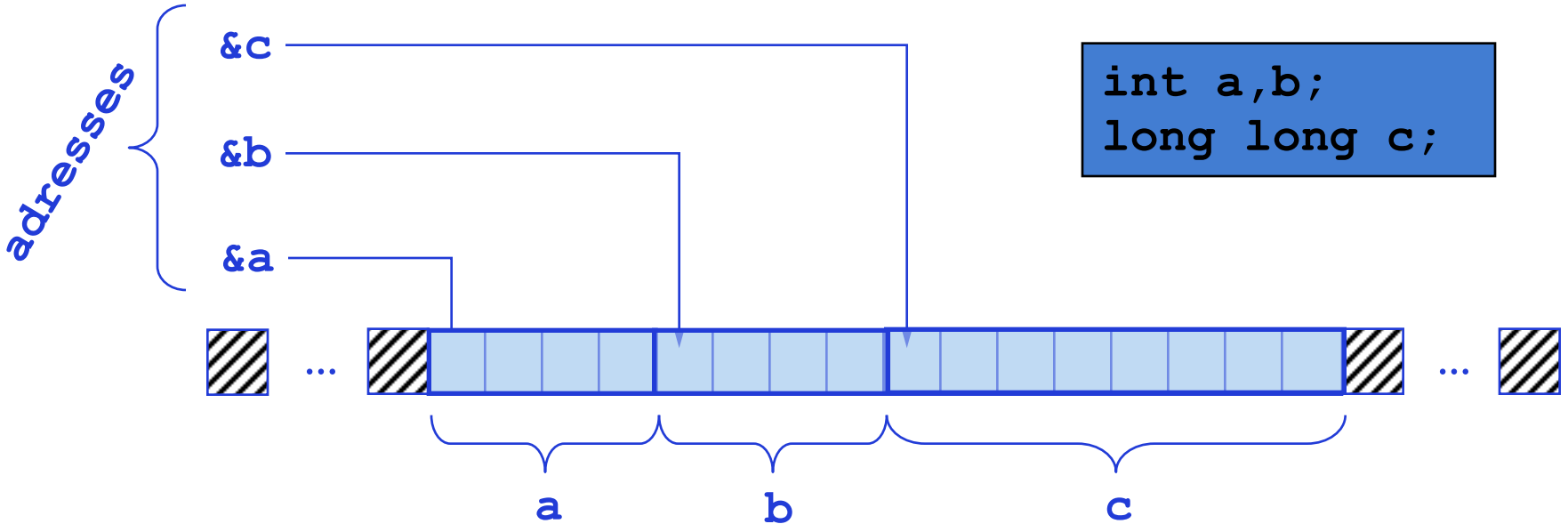
```
int a,b;
long long c;
```



Pointeurs

Une variable est stockée dans une zone mémoire réservée lors de la déclaration

&a indique l'adresse de cette zone



Pointeurs et structures

```
int *Pt;
 int T;
Pt = &T;
```

Pointeurs et structures

```
int *Pt;
 int T;
Pt = &T;
```

définit un pointeur **`Pt`**
et le fait pointer sur **`T`**.

Pointeurs et structures

```
int *Pt;
 int T;
Pt = &T;
```

définit un pointeur **`Pt`**
et le fait pointer sur **`T`**.
Alors **`*Pt`** désigne **`T`**

Pointeurs et structures

```
int *Pt;
 int T;
Pt = &T;
```

*** et & réciproques**

définit un pointeur **Pt**
et le fait pointer sur **T**.
Alors ***Pt** désigne **T**

Pointeurs et structures

```
int *Pt;
int T;
Pt = &T;
```

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

*** et & réciproques**

définit un pointeur **Pt**
et le fait pointer sur **T**.
Alors ***Pt** désigne **T**

Pointeurs et structures

```
int *Pt;
int T;
Pt = &T;
```

*** et & réciproques**

définit un pointeur **Pt**
et le fait pointer sur **T**.
Alors ***Pt** désigne **T**

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

```
sbloc bl;
sbloc *Psb = &bl;

(*Psb).nombre = 10;
(*Psb).valeur = 3.2;
```

Pointeurs et structures

```
int *Pt;
int T;
Pt = &T;
```

définit un pointeur **Pt** et le fait pointer sur **T**.
Alors ***Pt** désigne **T**

*** et & réciproques**

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

```
sbloc bl;
sbloc *Psb = &bl;

(*Psb).nombre = 10;
(*Psb).valeur = 3.2;
```

ou

```
sbloc bl;
sbloc *Psb = &bl;

Psb->nombre = 10;
Psb->valeur = 3.2;
```

Pointeurs et structures

```
int *Pt;
int T;
Pt = &T;
```

*** et & réciproques**

définit un pointeur **Pt**
et le fait pointer sur **T**.
Alors ***Pt** désigne **T**

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

```
sbloc bl;
sbloc *Psb = &bl;

(*Psb).nombre = 10;
(*Psb).valeur = 3.2;
```

ou

```
sbloc bl;
sbloc *Psb = &bl;

Psb->nombre = 10;
Psb->valeur = 3.2;
```

Pointeurs et structures

```
int *Pt;
int T;
Pt = &T;
```

*** et & réciproques**

définit un pointeur **Pt**
et le fait pointer sur **T**.
Alors ***Pt** désigne **T**

```
typedef struct {
   int nombre;
   float valeur;
} sbloc;
```

```
sbloc bl;
sbloc *Psb = &bl;

(*Psb).nombre = 10;
(*Psb).valeur = 3.2;
```

ou

```
sbloc bl;
sbloc *Psb = &bl;

Psb->nombre = 10;
Psb->valeur = 3.2;
```

. ou -> ?

`struct bloc bl;`

définit une variable **b**

de type **`struct blo`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

`struct bloc *Pbl;`

définit un pointeur sur un objet de type **`struct bloc`**

. ou -> ?

`struct bloc bl;`
définit une variable **b**
de type **`struct blo`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

`struct bloc *Pbl;`
définit un pointeur sur un objet de type **`struct bloc`**

Le champ **`nombre`** de **`bl`** : **`bl.nombre`**

. ou -> ?

`struct bloc bl;`
définit une variable **b**
de type **`struct blo`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

`struct bloc *Pbl;`
définit un pointeur sur un objet de type **`struct bloc`**

Le champ **`nombre`** de **`bl`** : **`bl.nombre`**

Le champ **`valeur`** de **`*Pbl`** : **`Pbl->valeur`**

. ou -> ?

`struct bloc bl;`
définit une variable **b**
de type **`struct blo`**

```
struct bloc {
   int nombre;
   float valeur;
};
```

`struct bloc *Pbl;`
définit un pointeur sur un objet de type **`struct bloc`**

Le champ **`nombre`** de **`bl`** : **`bl.nombre`**

Le champ **`valeur`** de **`*Pbl`** : **`Pbl->valeur`**
Si **`Pbl`** pointe vers une telle structure !!

Récursivité

La puissance peut être exprimée récursivement :

$$a^e = a * a^{e-1}$$
$$a^0 = 1$$

Récursivité

La puissance peut être exprimée récursivement :

$$a^e = a * a^{e-1}$$
$$a^0 = 1$$

$\Rightarrow$ programmation récursive :

Récursivité

La puissance peut être exprimée récursivement :

$$a^e = a * a^{e-1}$$
$$a^0 = 1$$

⇒ programmation récursive :
la fonction puissance s'appelle elle-même

Récursivité

La puissance peut être exprimée récursivement :

$$a^e = a * a^{e-1}$$
$$a^0 = 1$$

Critère d'arrêt

⇒ programmation récursive :
la fonction puissance s'appelle elle-même

Exemple I

La fonction **puissance** peut être programmée de la façon suivante :

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Si **e=0** le résultat est **1** ⇒ **return 1;**

sinon, le résultat est $a^{e-1} * a$

⇒ **return puissance(a,e-1)*a;**

Exemple I

La fonction **puissance** peut être programmée de la façon suivante :

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Critère d'arrêt

Si **e=0** le résultat est **1** ⇒ **return 1;**

sinon, le résultat est $a^{e-1} * a$

⇒ **return puissance(a,e-1)*a;**

Exemple I – analyse

À l'appel de la fonction **`puissance`**

- le **`double '2.0'`** est stocké dans **`a`**
- et l'**`int '2'`** est stocké dans **`e`**

```
double y;

y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Exemple I – exécution

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Exemple I – exécution

`puissance(2.0,2)`

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Exemple I – exécution

```
puissance(2.0,2)
    a ← 2.0
```

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Exemple I – exécution

```
puissance(2.0,2)
    a ← 2.0
    e ← 2
```

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Exemple I – exécution

```
puissance(2.0,2)
    a ← 2.0
    e ← 2
← puissance(2.0,1)*2.0
```

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Exemple I – exécution

```
puissance(2.0,2)
      a ← 2.0
      e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
```

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Exemple I – exécution

```
puissance(2.0,2)
    a ← 2.0
    e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
    a ← 2.0
```

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Exemple I – exécution

```
puissance(2.0,2)
    a ← 2.0
    e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
    a ← 2.0
    e ← 1
```

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

Exemple I – exécution

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

```
puissance(2.0,2)
     a ← 2.0
     e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
     a ← 2.0
     e ← 1
← puissance(2.0,0)*2.0
```

Exemple I – exécution

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

```
puissance(2.0,2)
    a ← 2.0
    e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
    a ← 2.0
    e ← 1
← puissance(2.0,0)*2.0
  puissance(2.0,0)
```

Exemple I – exécution

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

```
puissance(2.0,2)
     a ← 2.0
     e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
     a ← 2.0
     e ← 1
← puissance(2.0,0)*2.0
  puissance(2.0,0)
     a ← 2.0
```

Exemple I – exécution

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

```
puissance(2.0,2)
     a ← 2.0
     e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
     a ← 2.0
     e ← 1
← puissance(2.0,0)*2.0
  puissance(2.0,0)
     a ← 2.0
     e ← 0
```

Exemple I – exécution

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

```
puissance(2.0,2)
     a ← 2.0
     e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
     a ← 2.0
     e ← 1
← puissance(2.0,0)*2.0
  puissance(2.0,0)
     a ← 2.0
     e ← 0
     ← 1
```

Exemple I – exécution

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

```
puissance(2.0,2)
    a ← 2.0
    e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
    a ← 2.0
    e ← 1
← puissance(2.0,0)*2.0
  puissance(2.0,0)
    a ← 2.0
    e ← 0
    ← 1
← 1 * 2.0 = 2.0
```

Exemple I – exécution

```
   double y;

   y = puissance(2.0, 2);
```

```
double puissance(double a, int e)
{
   if (e == 0) return 1;
   return puissance(a,e-1)*a;
}
```

```
puissance(2.0,2)
    a ← 2.0
    e ← 2
← puissance(2.0,1)*2.0
  puissance(2.0,1)
    a ← 2.0
    e ← 1
← puissance(2.0,0)*2.0
  puissance(2.0,0)
    a ← 2.0
    e ← 0
    ← 1
← 1 * 2.0 = 2.0
← 2.0 * 2.0 = 4.0
```

Variables locales

```
double puissance(double a, int e)
```

chaque exécution de la fonction **`puissance`** possède ses propres variables locales **`a`** et **`e`**, détruites à la fin de la fonction

Exemple II

Une programmation équivalente serait :

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Rappel : le **`return x`** quitte la fonction en retournant le contenu de la variable **`x`**

Exemple II

Une programmation équivalente serait :

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Critère d'arrêt

Rappel : le **`return x`** quitte la fonction en retournant le contenu de la variable **`x`**

Exemple II – exécution

```
puissance(2.0,2)
```

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

```
puissance(2.0,2)
```

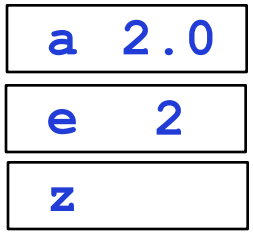
```
puissance(2.0,2)
```

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

puissance(2.0,2)

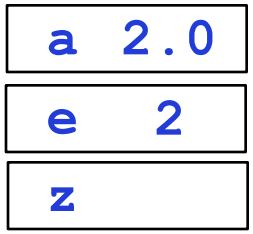



```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

puissance(2.0,2)



puissance(2.0,1)

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

puissance(2.0,2)

a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

puissance(2.0,2)

a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	

puissance(2.0,0)

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

puissance(2.0,2)

a	2.0
e	2
z	

puissance(2.0,1)

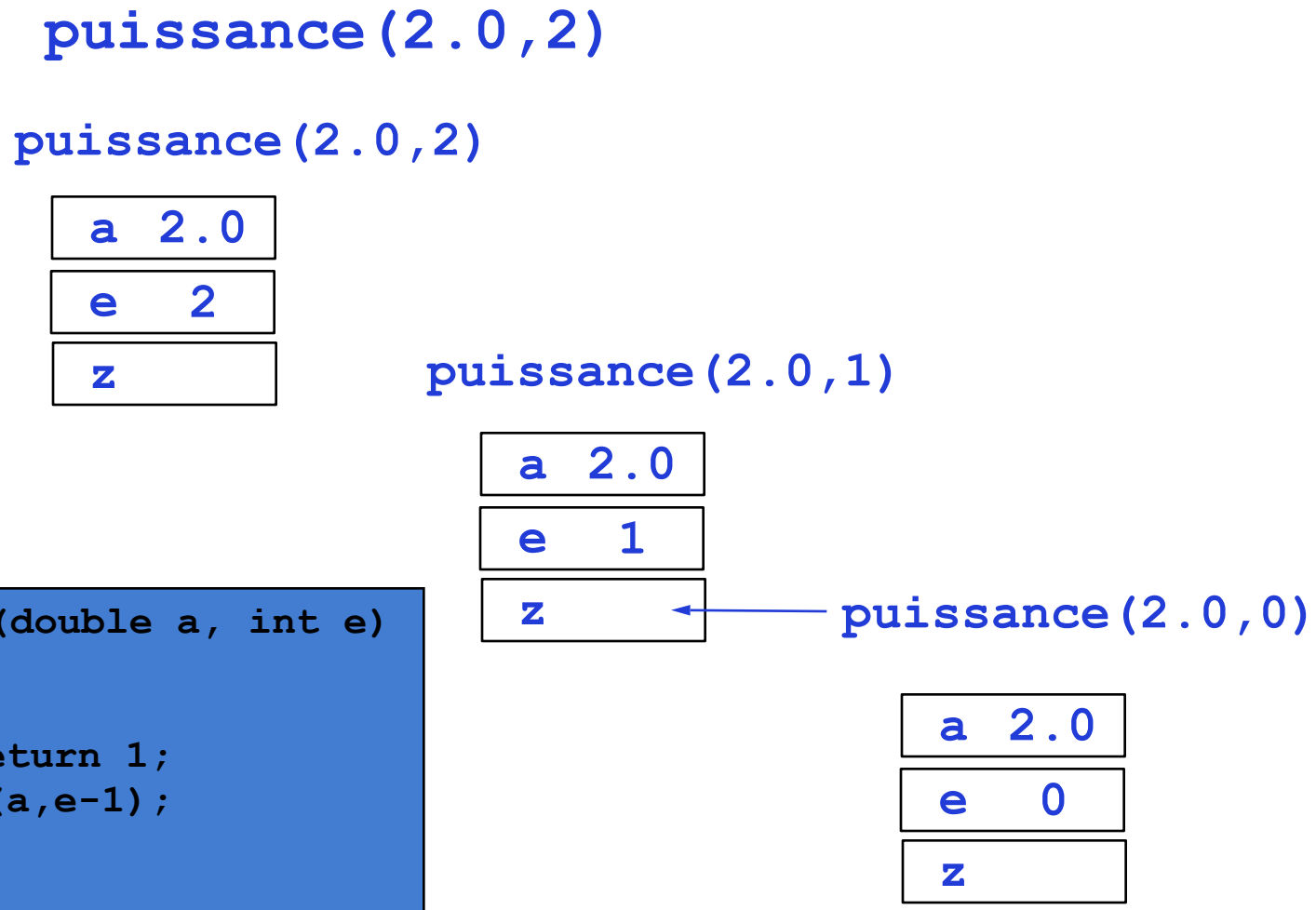
a	2.0
e	1
z	

puissance(2.0,0)

a	2.0
e	0
z	

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution




```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

puissance(2.0,2)

a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	

puissance(2.0,0)

a	2.0
e	0
z	

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

puissance(2.0,2)

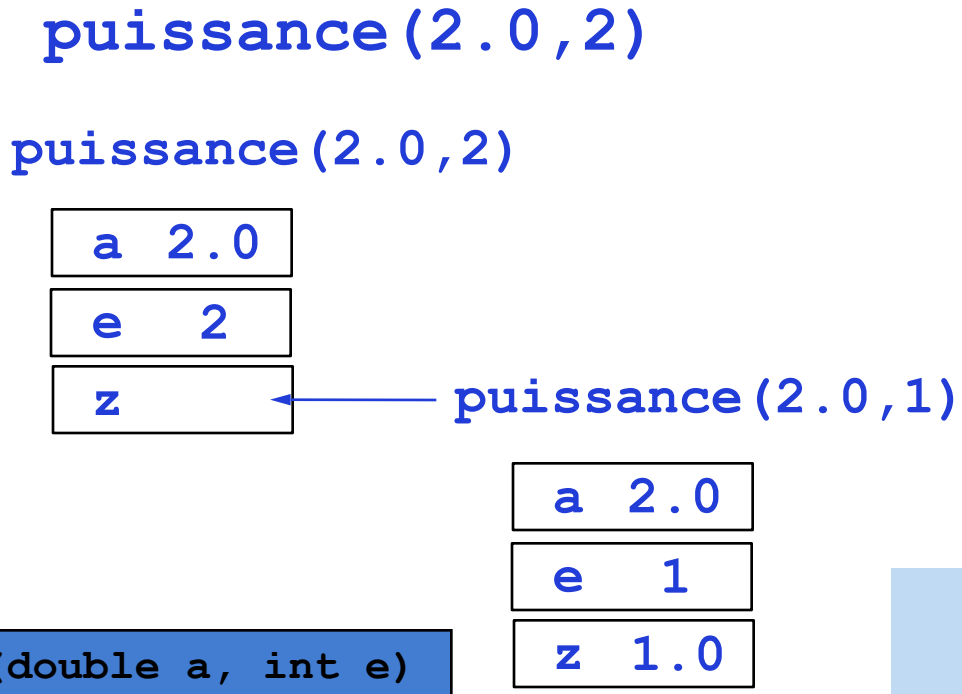
a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	1.0

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution




```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

puissance(2.0,2)

a	2.0
e	2
z	

puissance(2.0,1)

a	2.0
e	1
z	1.0

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

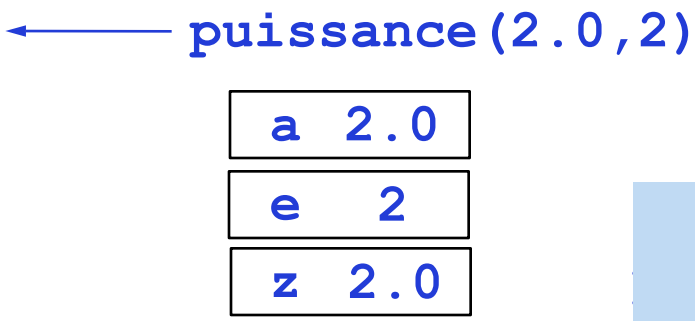
puissance(2.0,2)

a	2.0
e	2
z	2.0

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)




```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

puissance(2.0,2)

puissance(2.0,2)

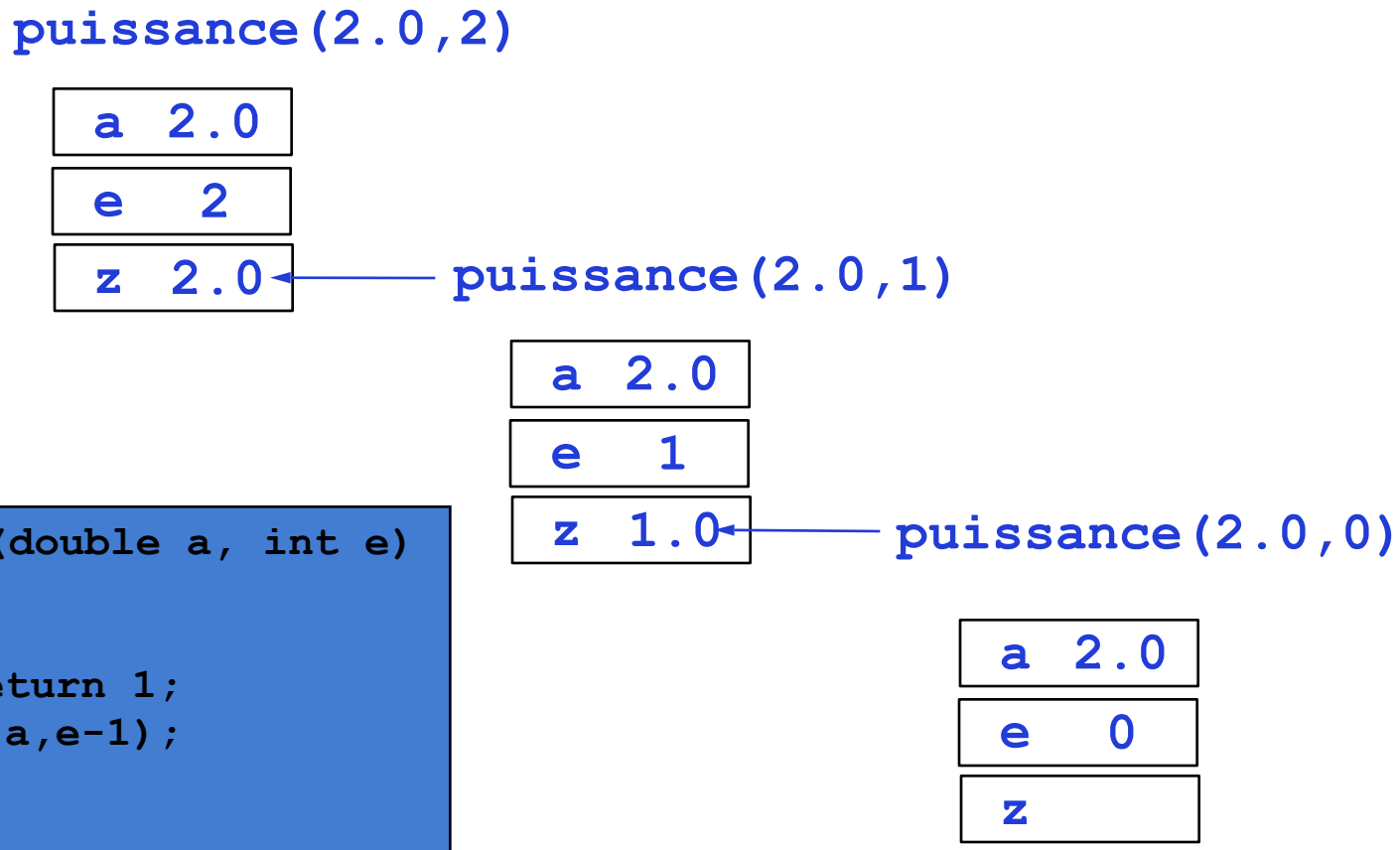
a	2.0
e	2
z	2.0

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – exécution

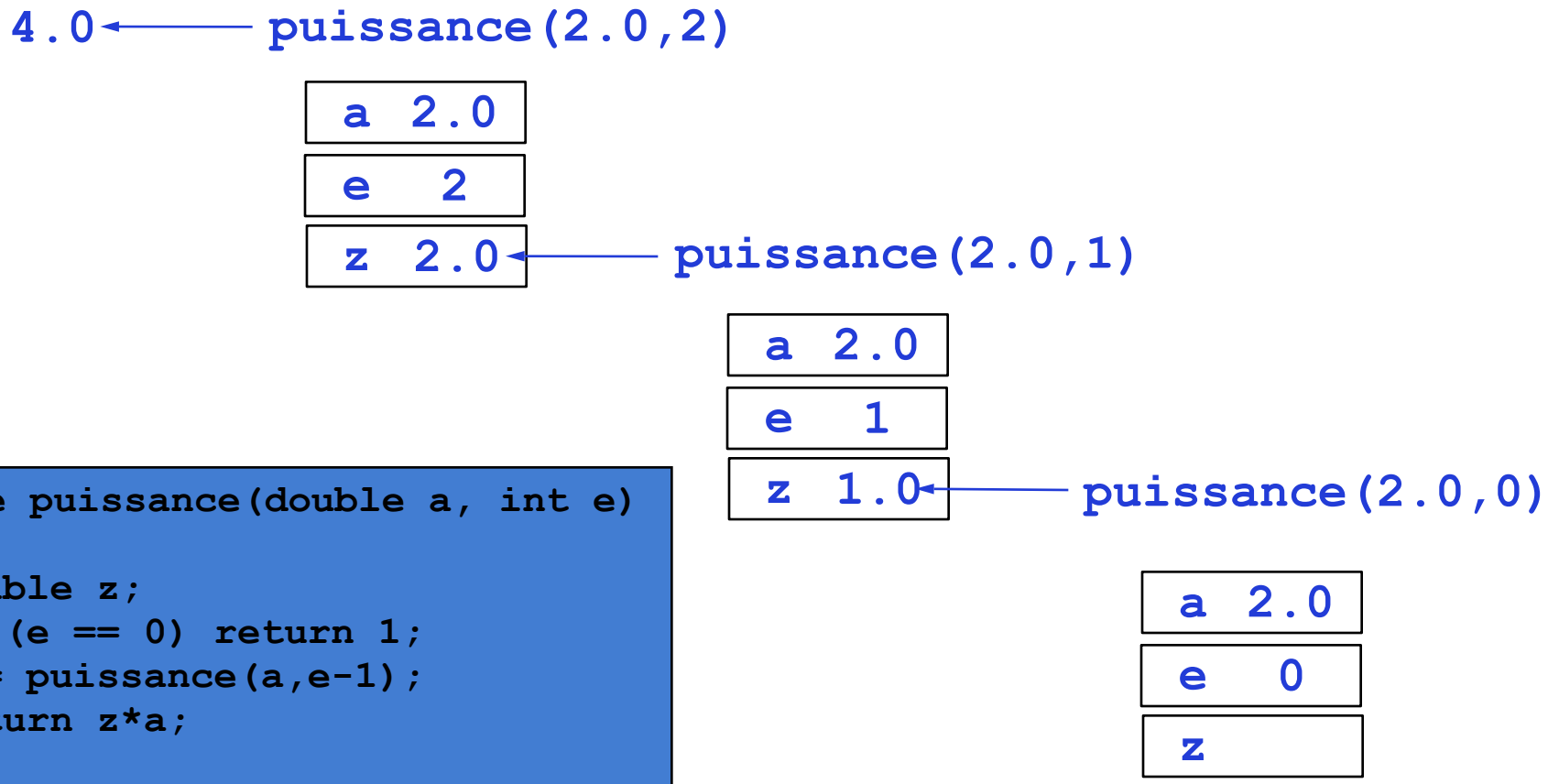

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – pile




```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Exemple II – pile




```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Récursivité non terminale

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

Récursivité non terminale

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

L'appel récursif n'est pas la dernière instruction de la fonction

Récursivité non terminale

```
double puissance(double a, int e)
{
   double z;
   if (e == 0) return 1;
   z = puissance(a,e-1);
   return z*a;
}
```

L'appel récursif n'est pas la dernière instruction de la fonction

$\Rightarrow$ l'opération est effectuée au « dépilement », en remontant

Exemple III

L'addition entière peut être reprogrammée avec seulement des incrémentations et décrémentations :

```
a + b = (a+1) + (b-1)
a + 0 = a
```

Exemple III

L'addition entière peut être reprogrammée avec seulement des incrémentations et décrémentations :

```
a + b = (a+1) + (b-1)
a + 0 = a
```

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III

L'addition entière peut être reprogrammée avec seulement des incrémentations et décrémentations :

```
a + b = (a+1) + (b-1)
a + 0 = a
```


Critère d'arrêt

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

```
addition(3,2)
```

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

```
addition(3,2)
```

```
addition(3,2)
```

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

addition(5,0)

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

addition(5,0)

a	5
b	0

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

addition(5,0)

a	5
b	0

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

addition(5,0)

a	5
b	0

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

5

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

← addition(4,1)

a	4
b	1

5

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

addition(4,1)

a	4
b	1

5

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

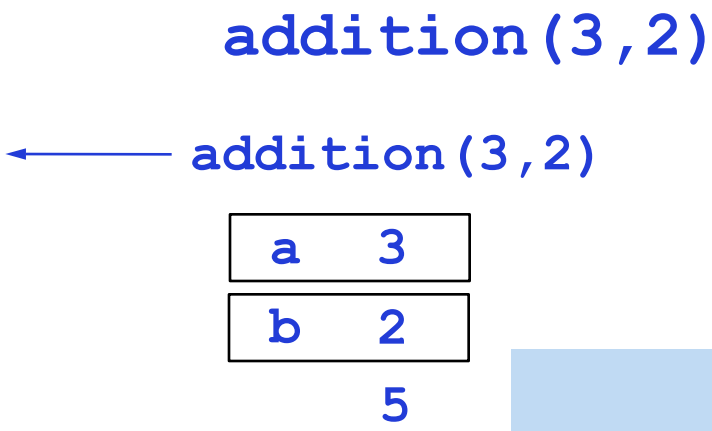
addition(3,2)

a	3
b	2

5

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution




```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

addition(3,2)

addition(3,2)

a	3
b	2

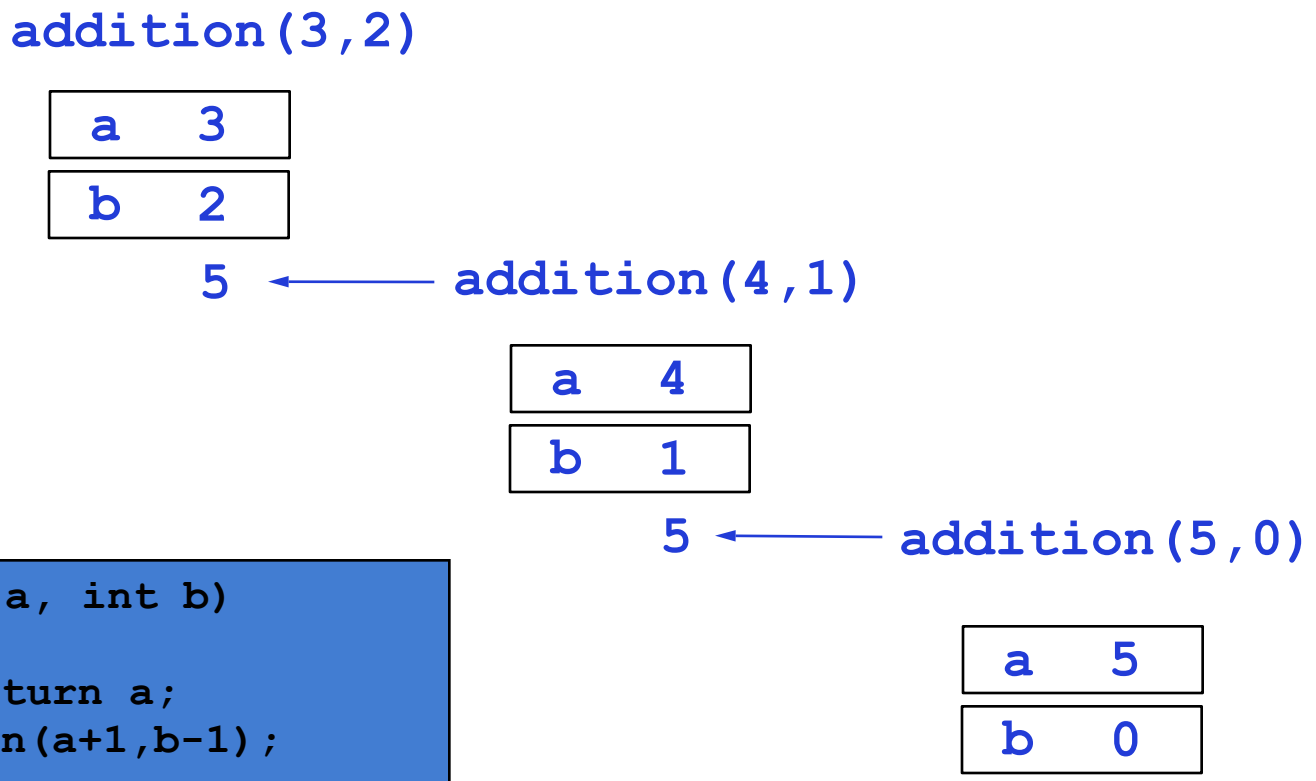
5

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – exécution

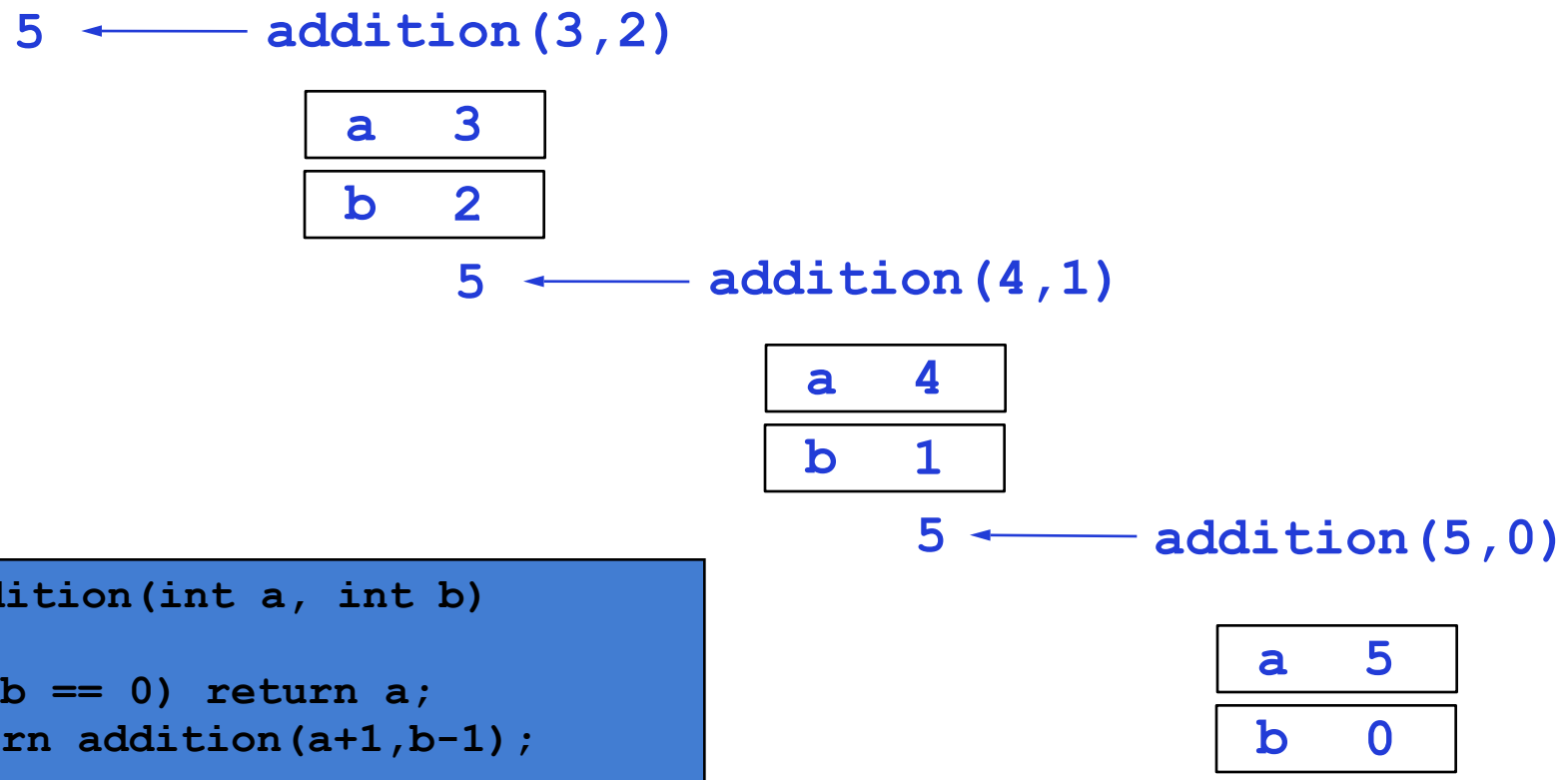

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – pile




```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Exemple III – pile




```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Récursivité terminale

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

Récursivité terminale

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

L'appel récursif est la dernière instruction de la fonction

Récursivité terminale

```
int addition(int a, int b)
{
   if (b == 0) return a;
   return addition(a+1,b-1);
}
```

L'appel récursif est la dernière instruction de la fonction

⇒ une fois le résultat obtenu (lorsque **b=0**)

le résultat est propagé au « dépilement »

Récursivité multiple

Pour s'évaluer, une fonction peut faire plusieurs fois appel à elle-même
Ex : Fibonacci

```
F(n) = F(n-2) + F(n-1)
F(0) = 1
F(1) = 1
```

Récursivité multiple

Pour s'évaluer, une fonction peut faire plusieurs fois appel à elle-même
Ex : Fibonacci

```
F(n) = F(n-2) + F(n-1)
F(0) = 1
F(1) = 1
```

Critères d'arrêt

Exemple IV

La suite de Fibonacci peut donc être programmée de la façon suivante :

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV

La suite de Fibonacci peut donc être programmée de la façon suivante :

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Critère d'arrêt

Exemple IV – exécution

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution

`fibonacci(4)`

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution

`fibonacci(4)`

`fibonacci(2)`

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution

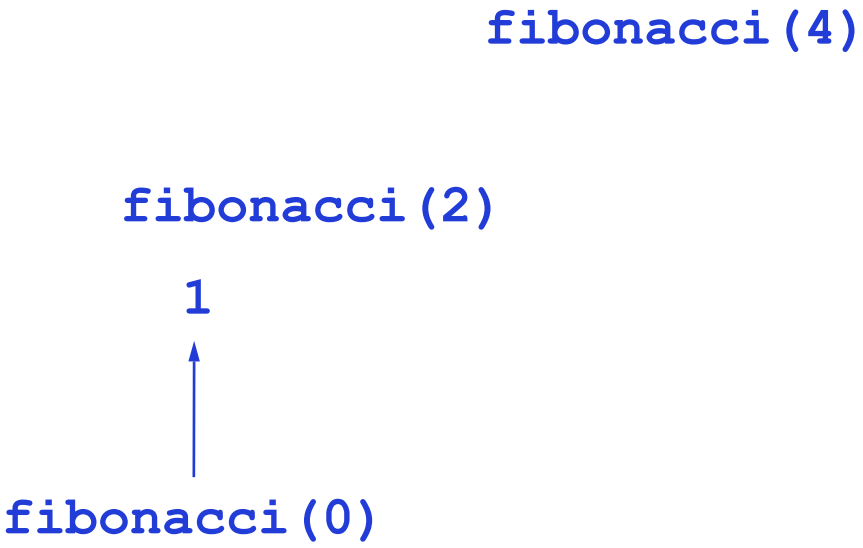
`fibonacci(4)`

`fibonacci(2)`

`fibonacci(0)`

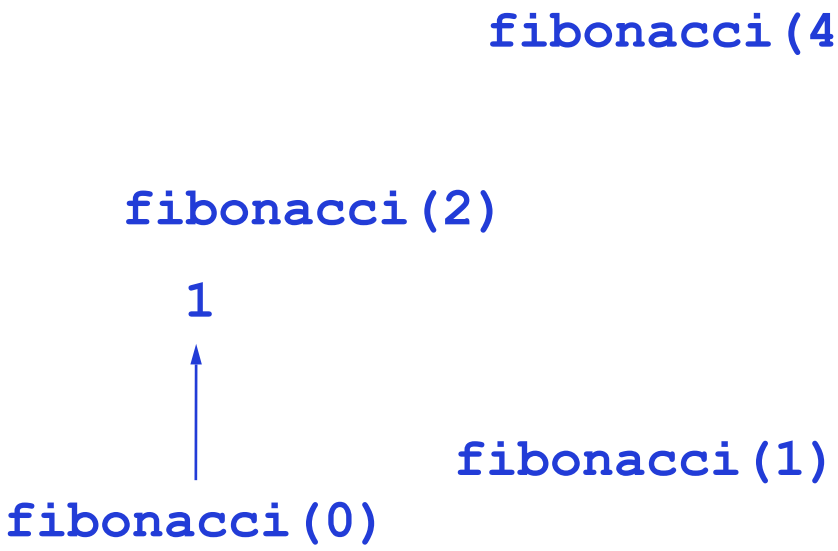
```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



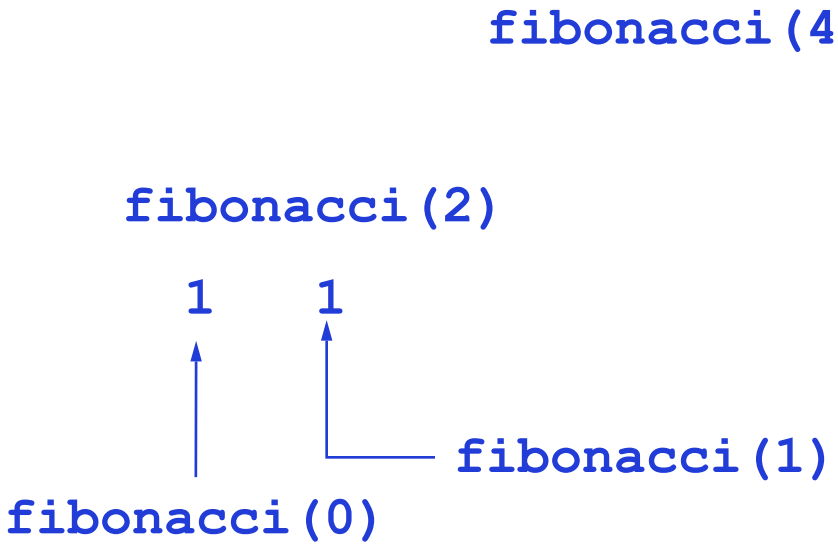

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



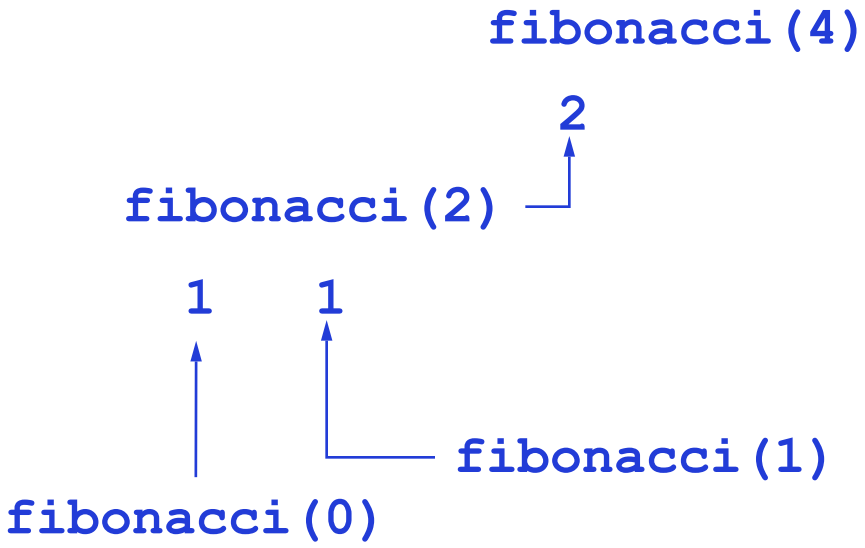

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



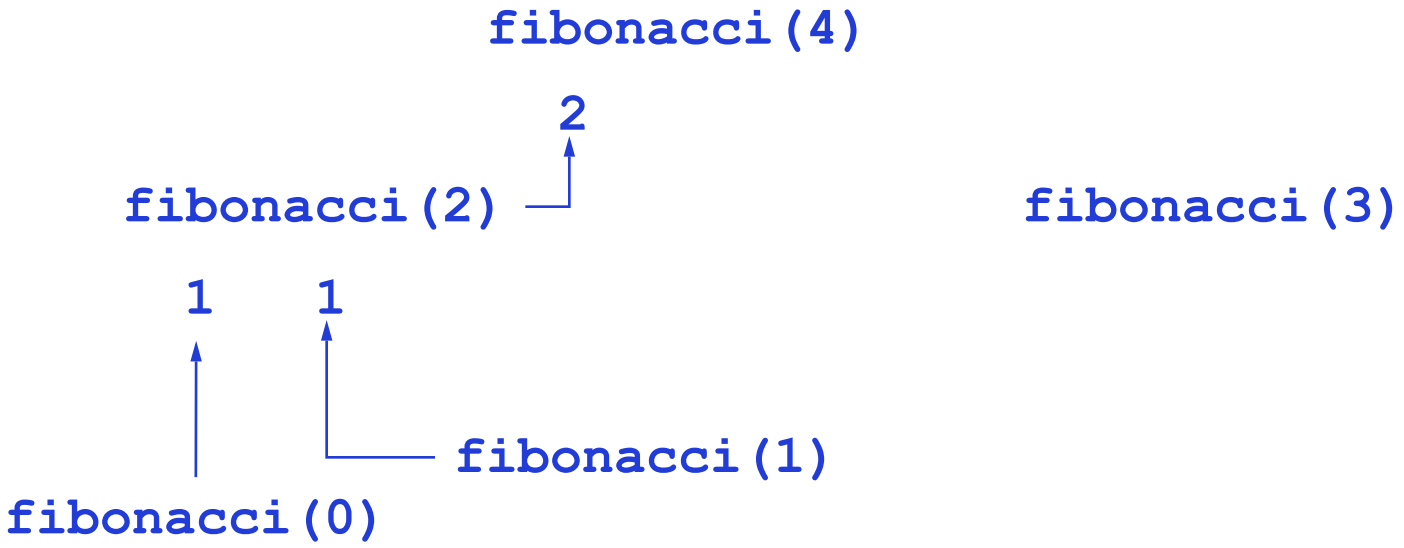

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



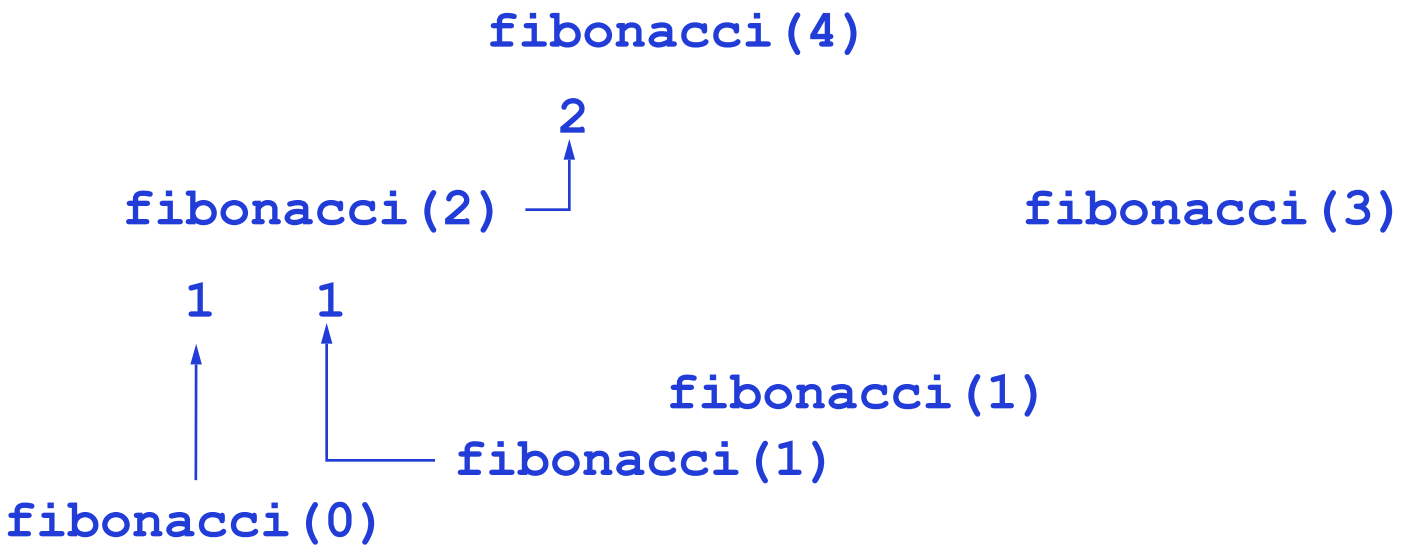

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



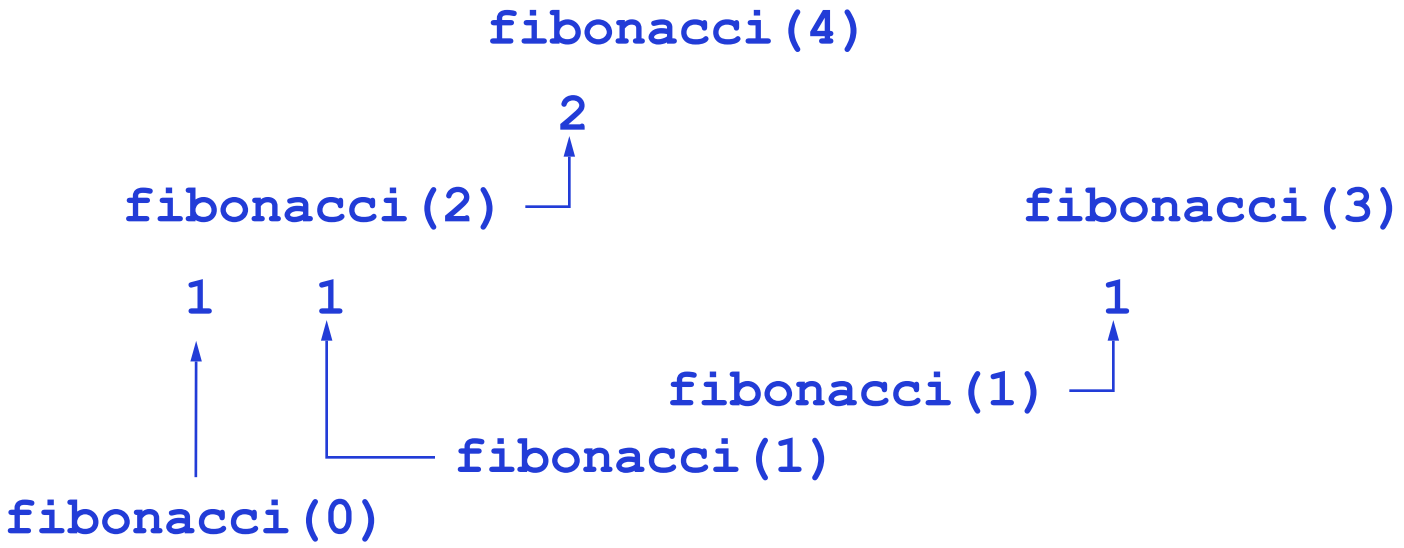

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



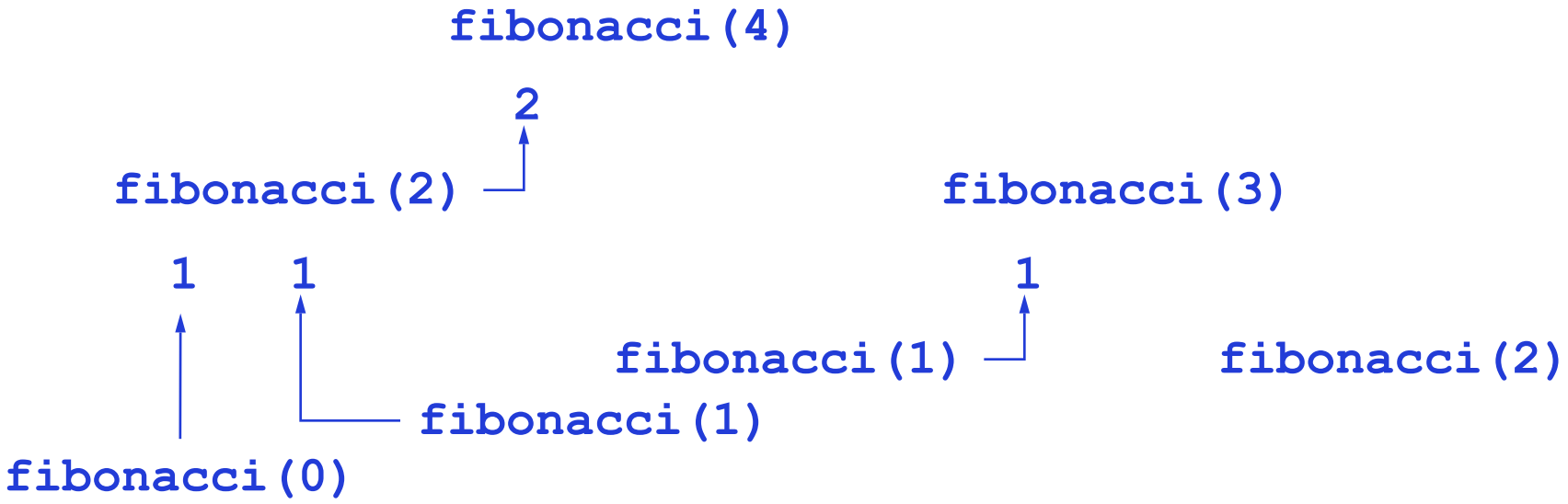

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



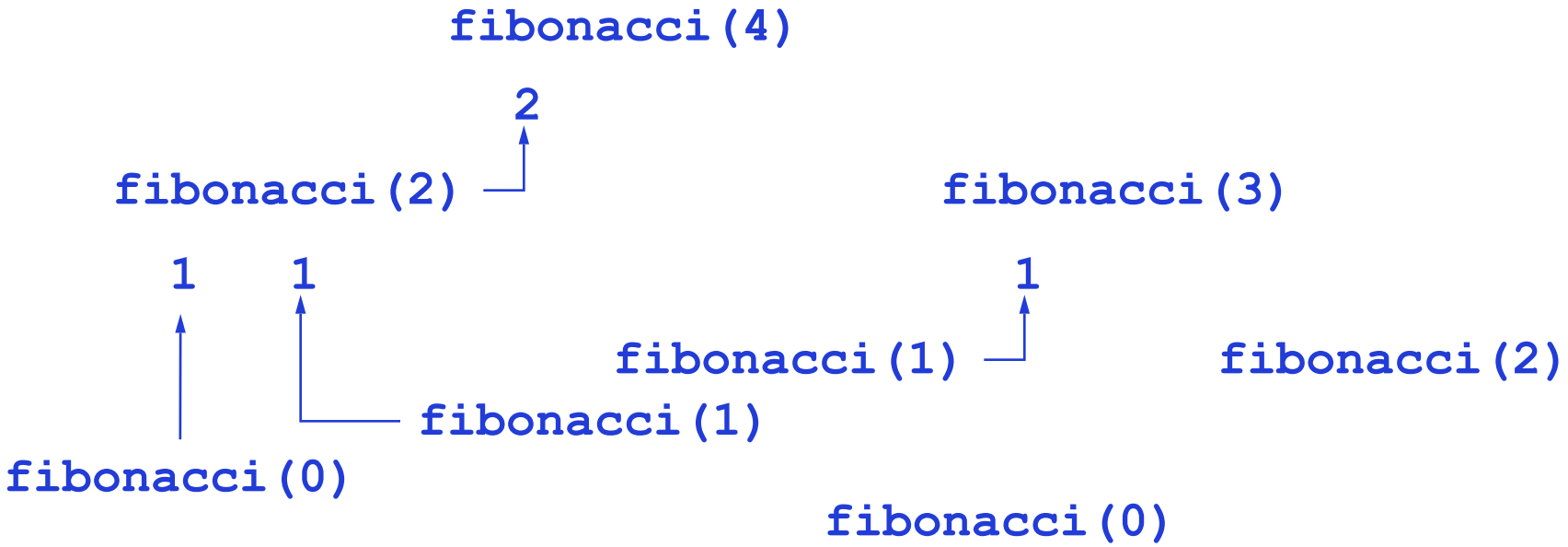

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



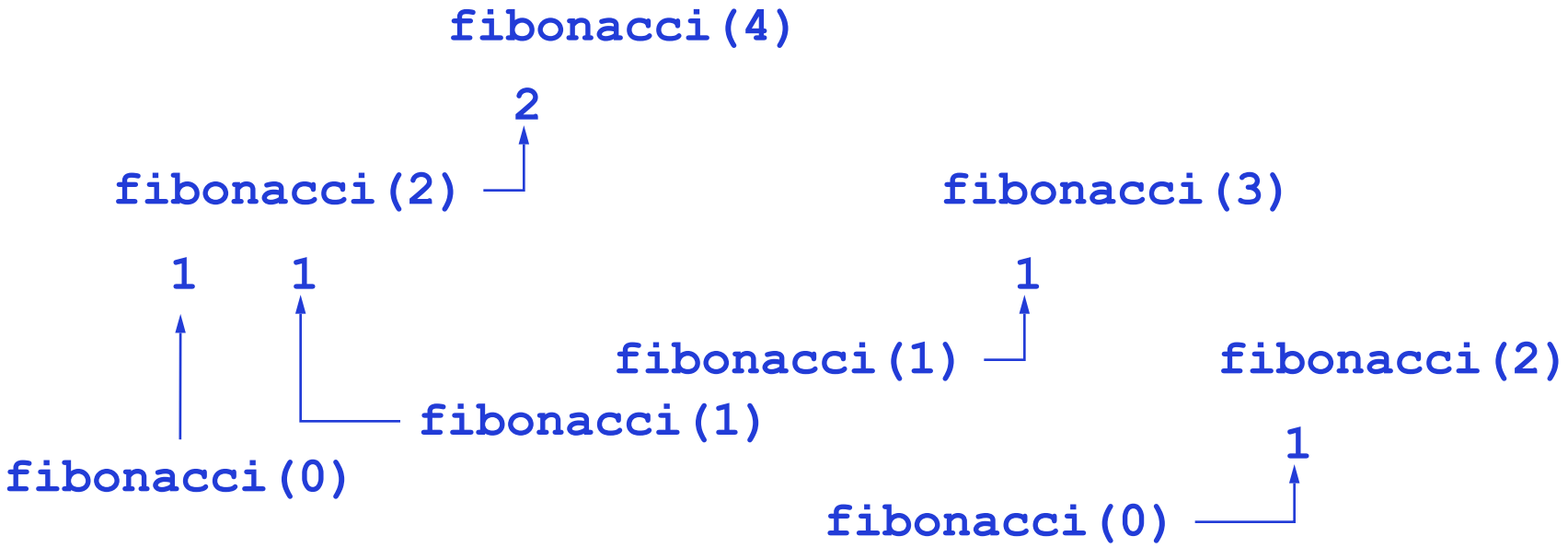

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



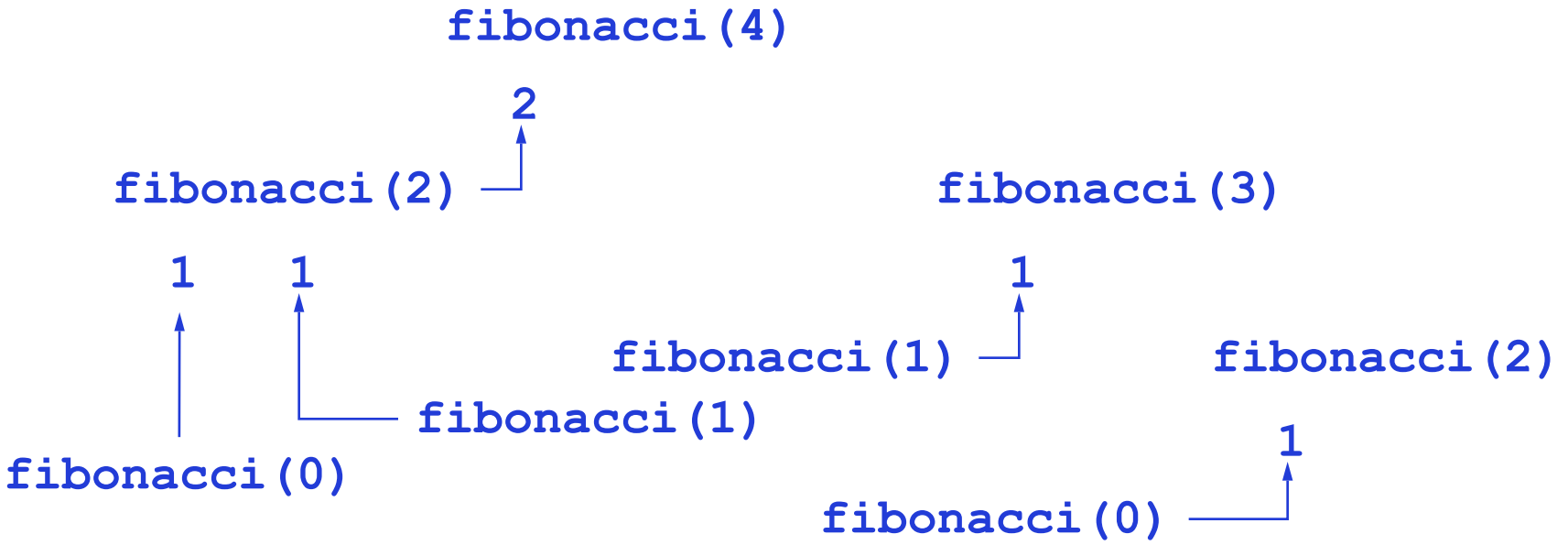

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution




```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

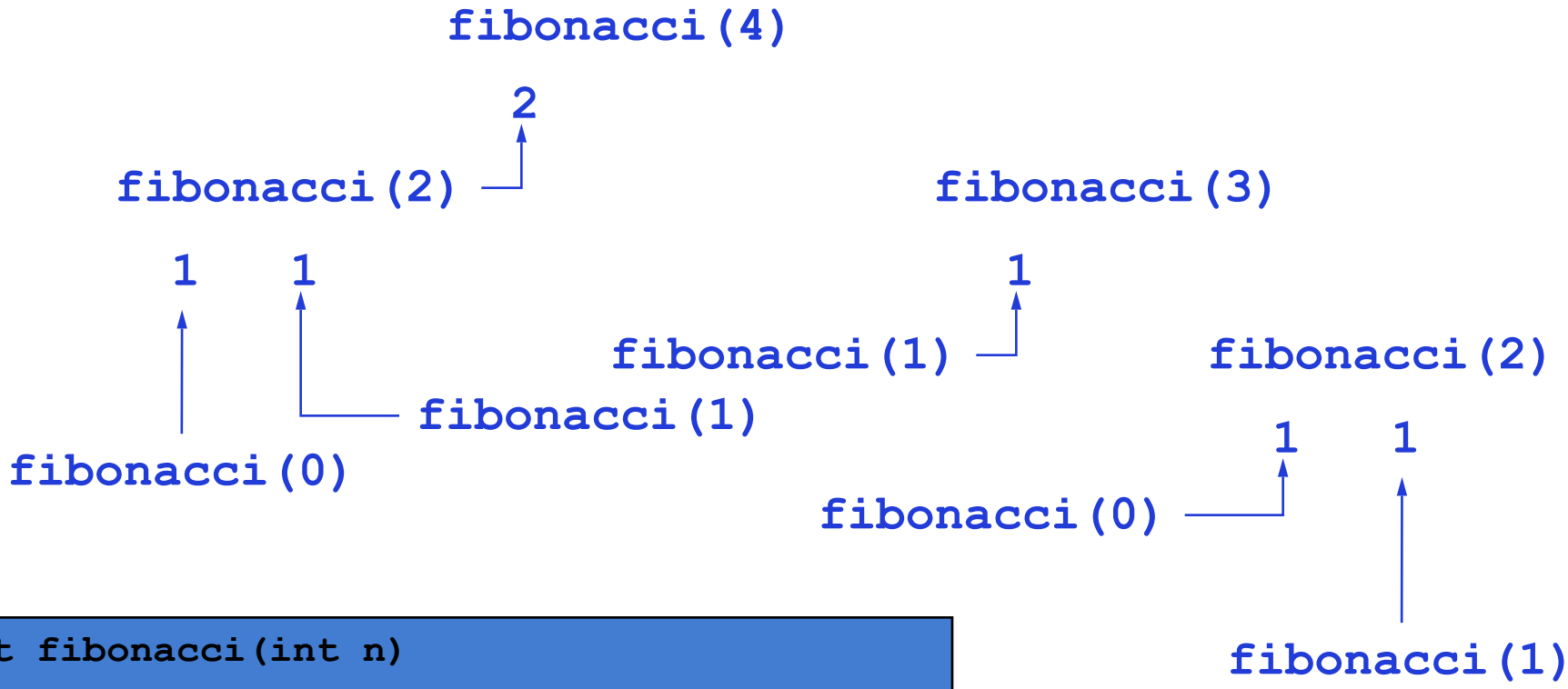
Exemple IV – exécution




```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

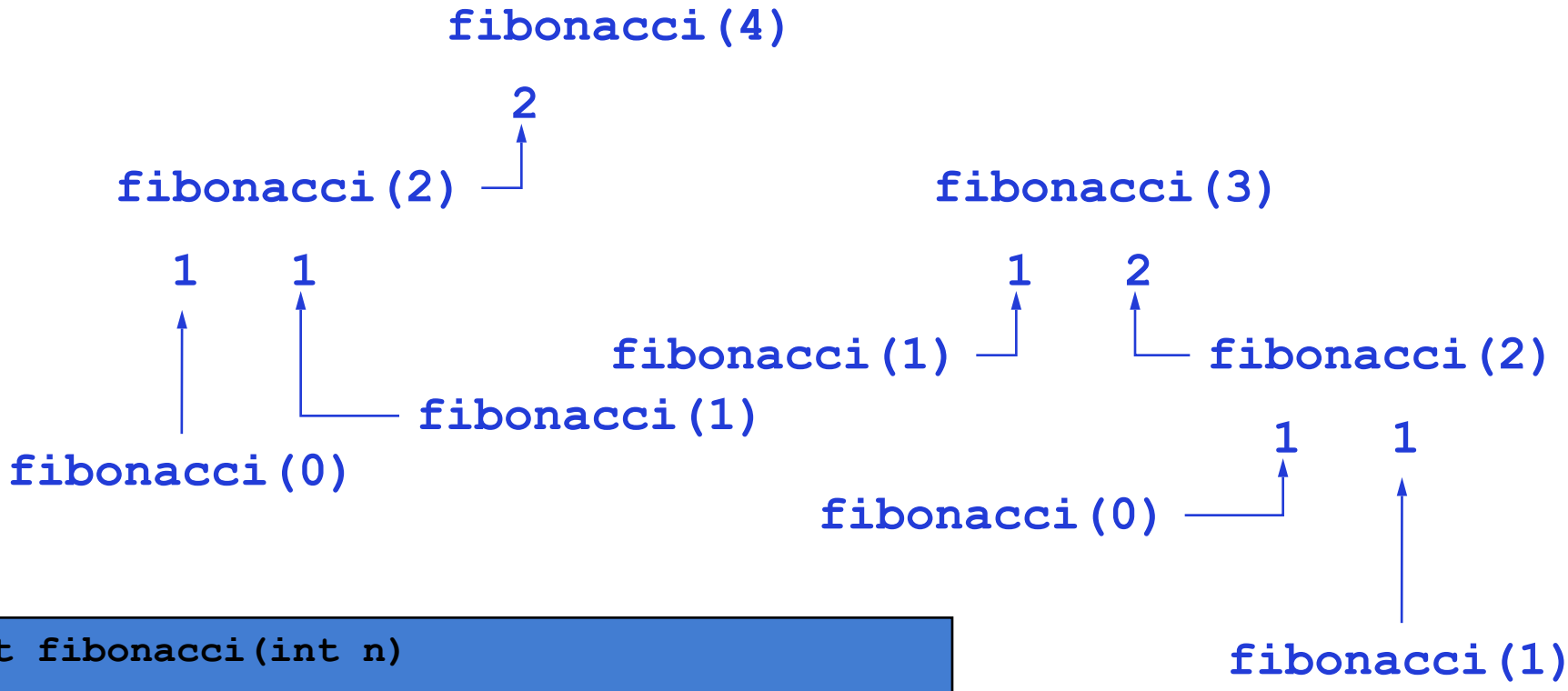
fibonacci(1)

Exemple IV – exécution



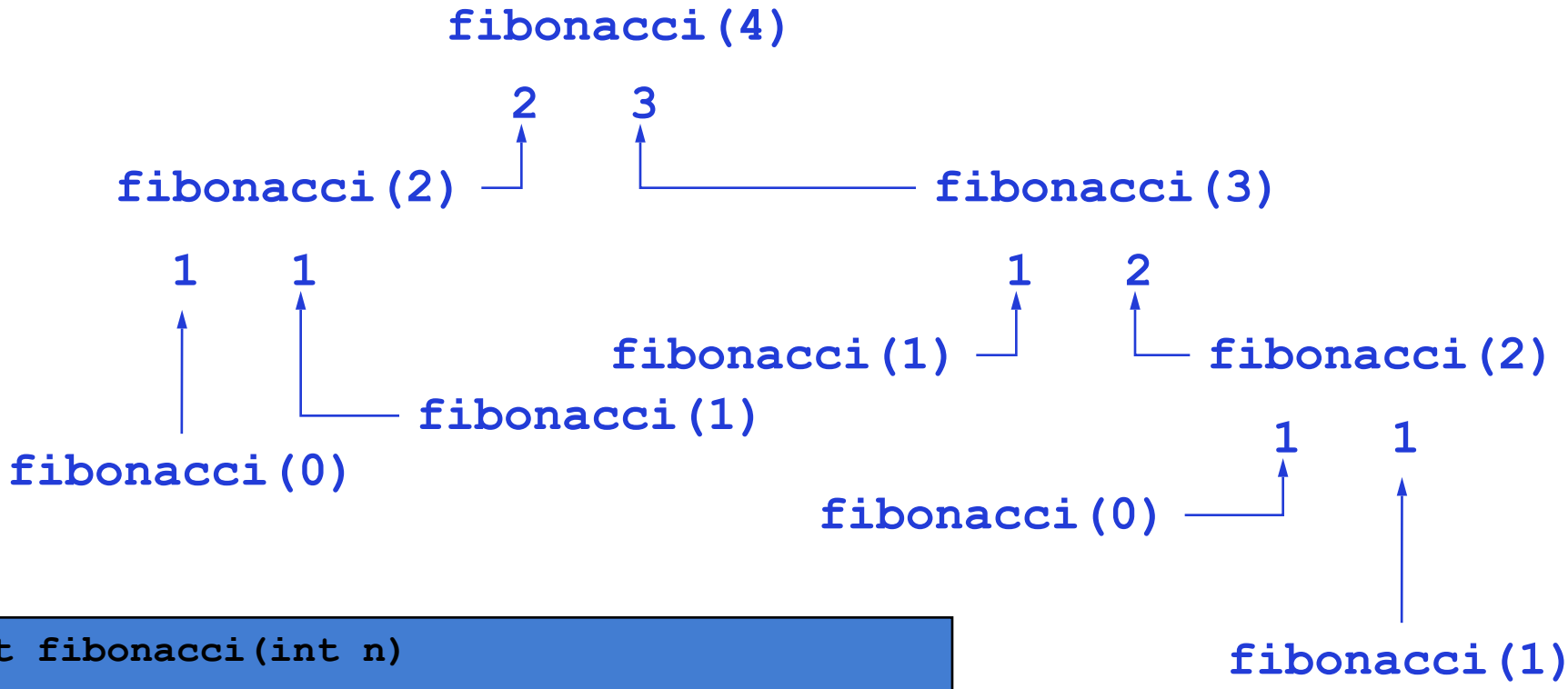

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



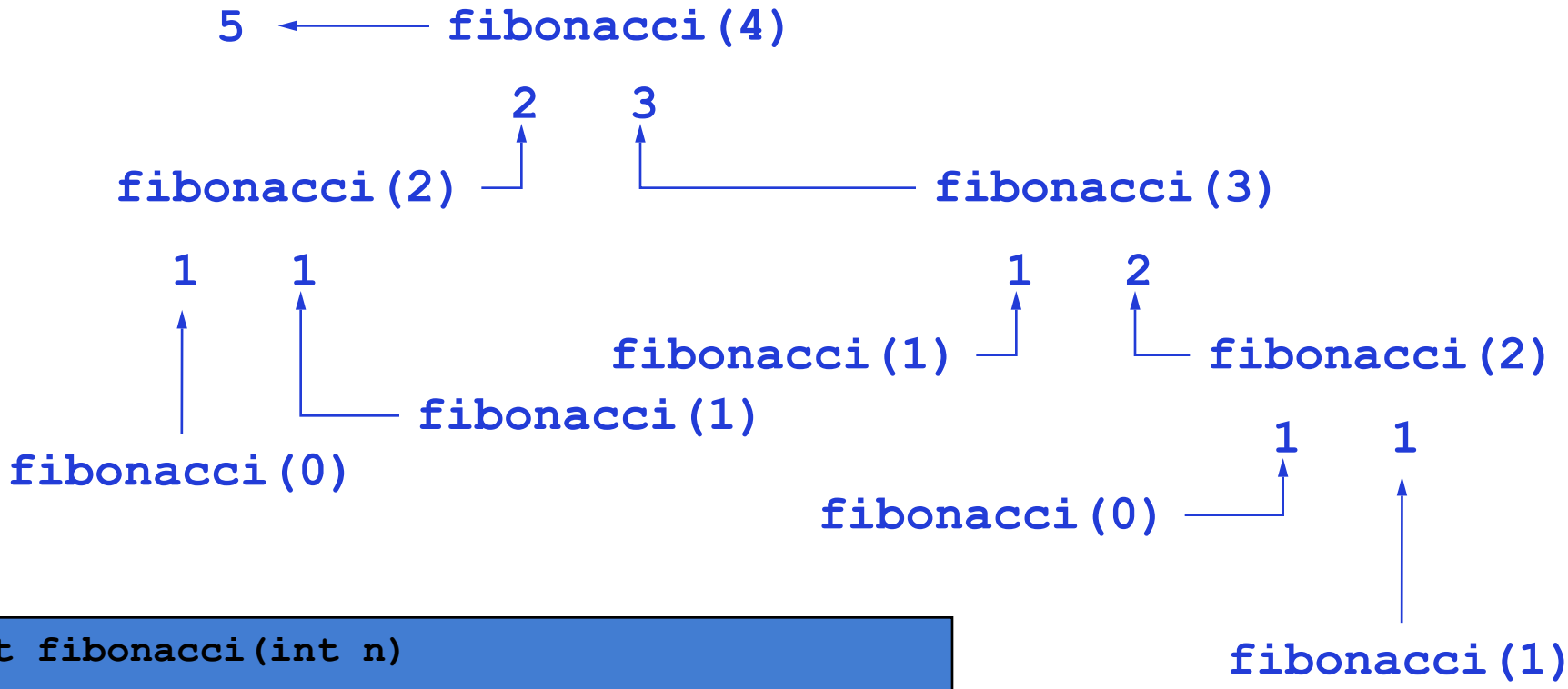

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution



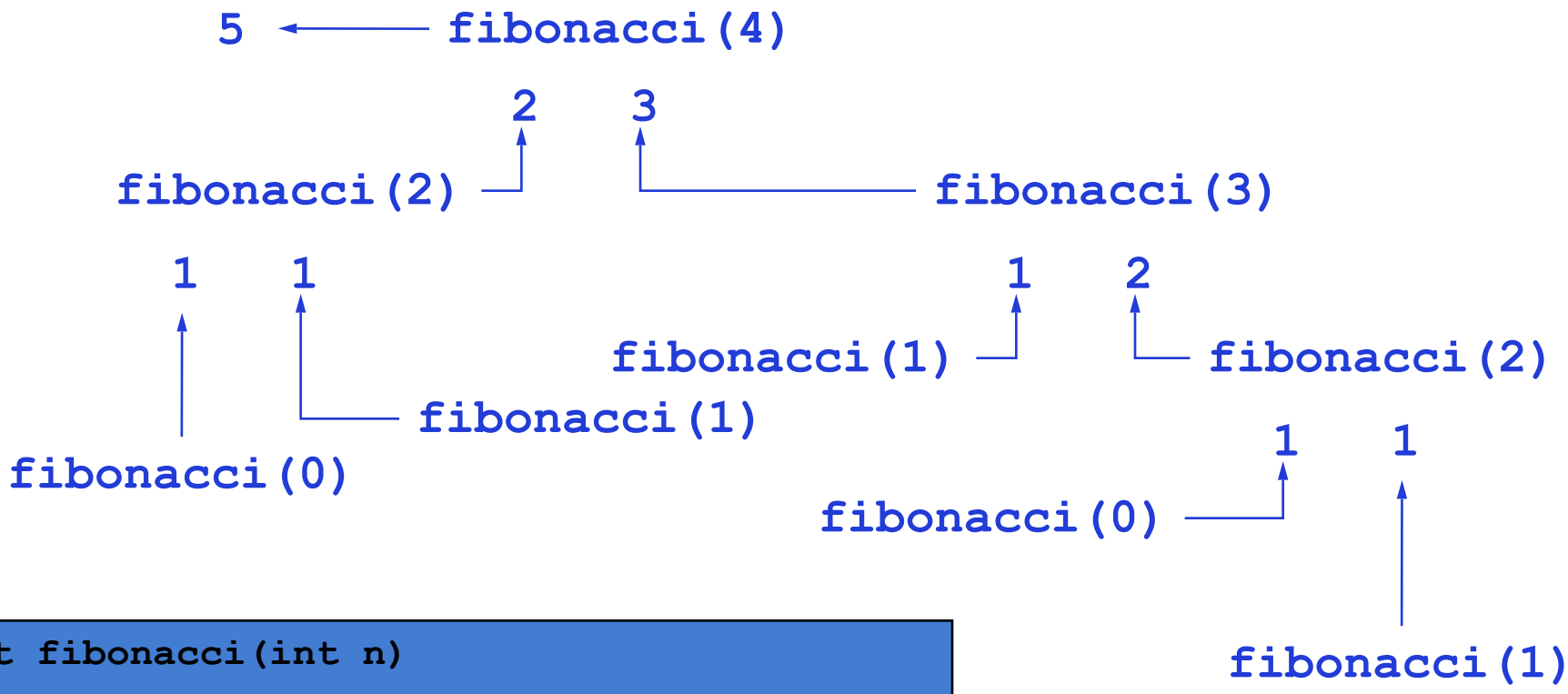

```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution




```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

Exemple IV – exécution




```
int fibonacci(int n)
{
   if (n < 2) return 1;
   return fibonacci(n-2) + fibonacci(n-1);
}
```

⇒ pas efficace !

Récursivité : avantages

La programmation récursive
est très proche de la définition
mathématique du problème
$\Rightarrow$ très facile à implémenter

Mais, ne pas oublier les **critères d'arrêt**,
au risque d'une boucle sans fin !

Conclusion

Structures :

- permettent de regrouper dans un même type des champs différents
- définissent de nouveaux types, utilisables comme les autres