Programmation en C
Fonctions récursives

Régis Barbanchon

L1 Info-Math, Semestre 2

Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) {
  Say ("Hi",  name);
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann");
  Visit ("Bob");
}
```

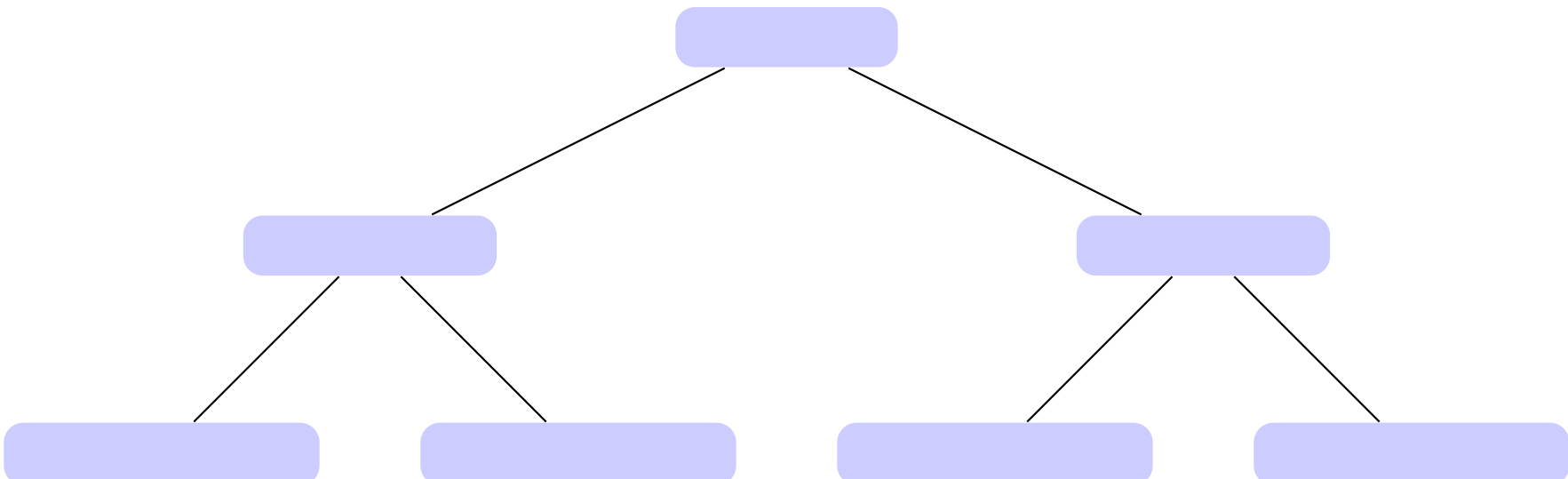
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) {
  Say ("Hi",  name);
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann");
  Visit ("Bob");
}
```



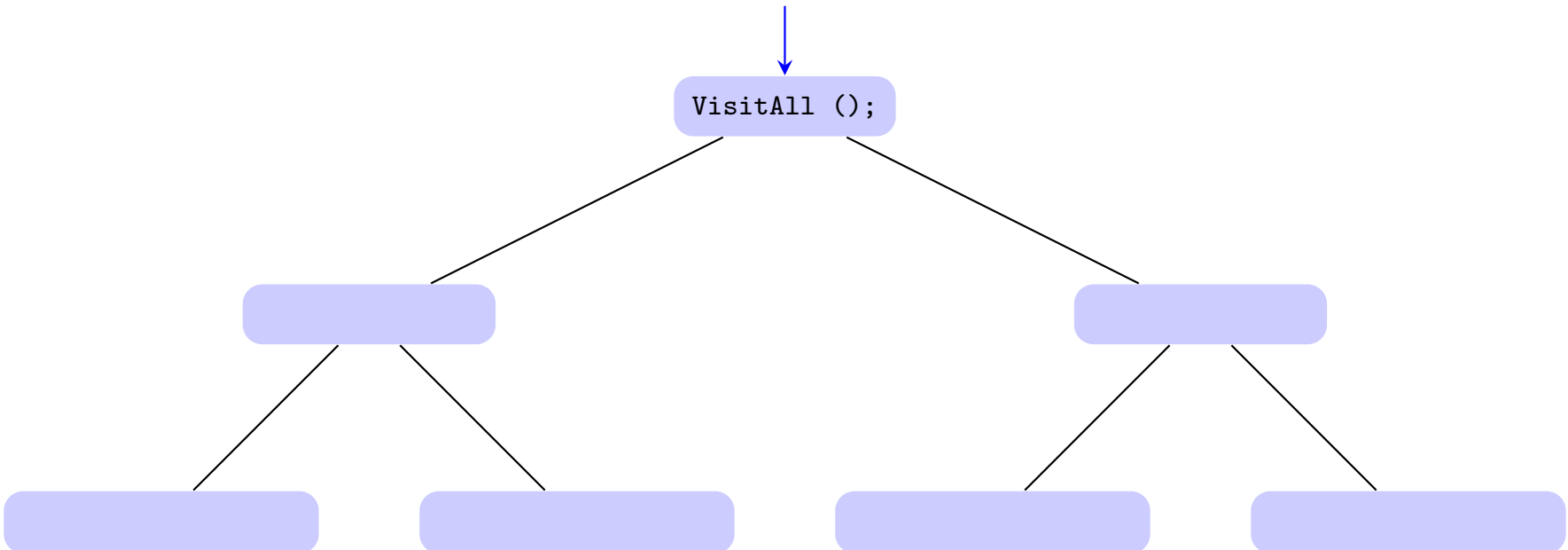
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) {
  Say ("Hi",  name);
  Say ("Bye", name);
}

void VisitAll () { •
  Visit ("Ann");
  Visit ("Bob");
}
```



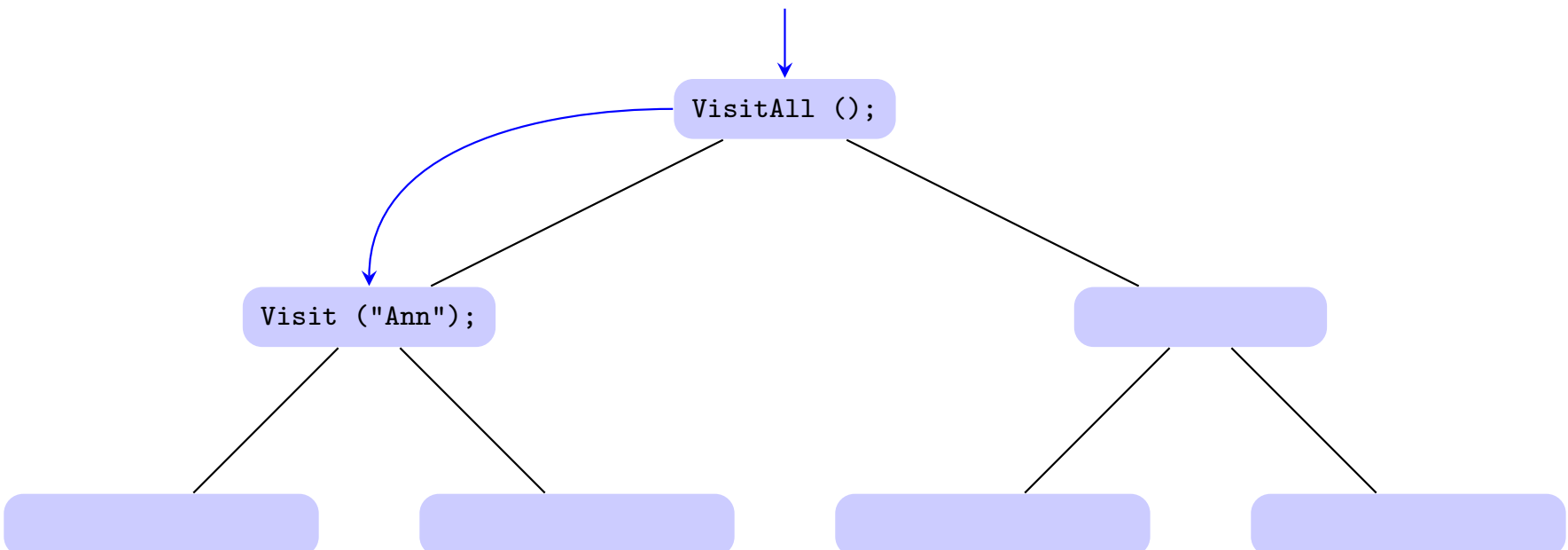
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) { •
  Say ("Hi",  name);
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann"); ⟸
  Visit ("Bob");
}
```



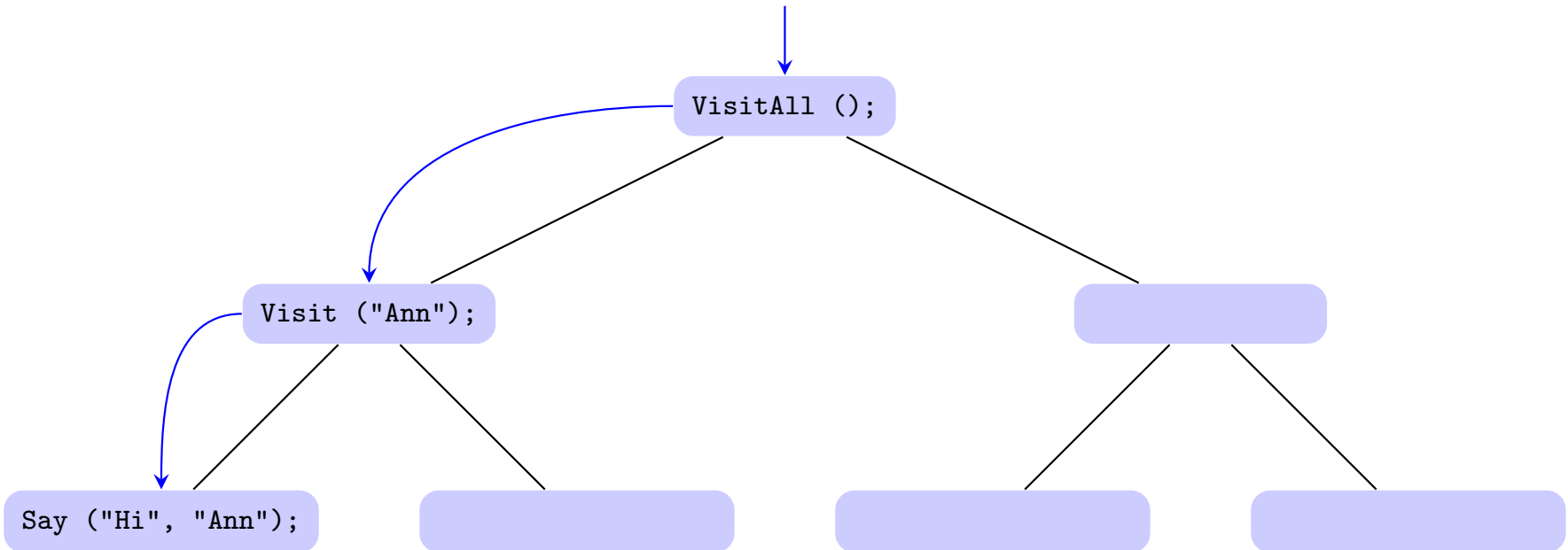
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);  •
}

void Visit (char const name[]) {
  Say ("Hi",  name); ⟸
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann"); ⟸
  Visit ("Bob");
}
```



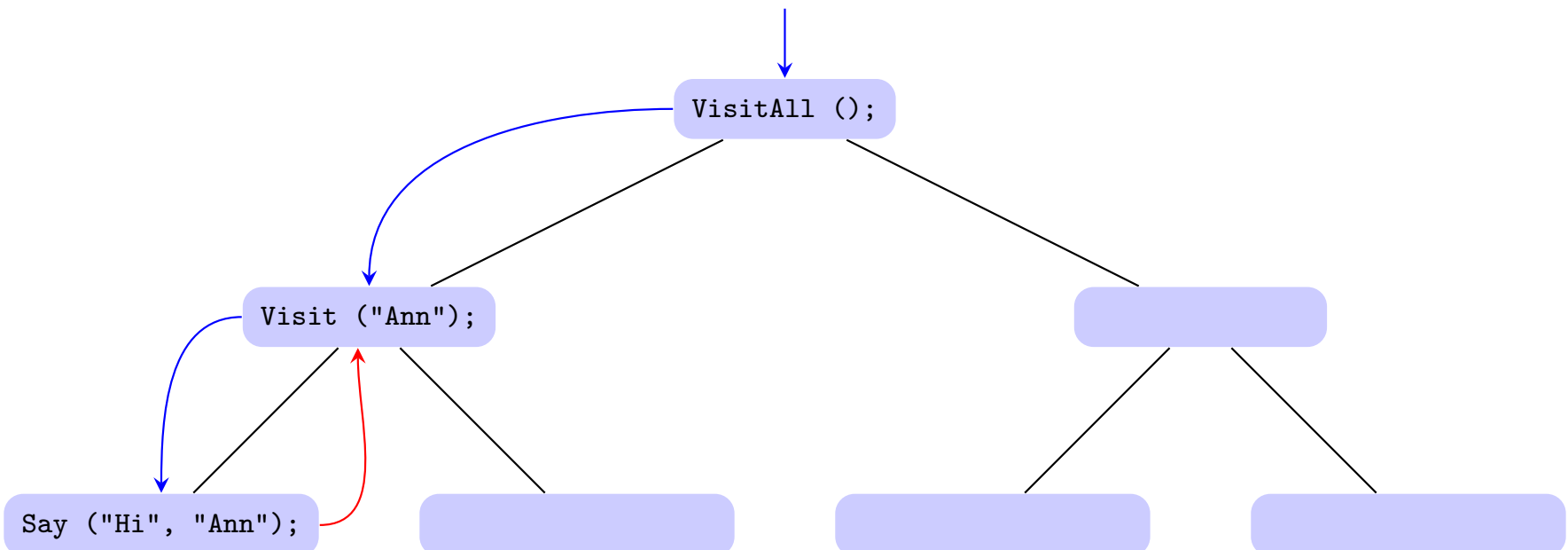
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) {
  Say ("Hi",  name); •
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann"); ⟸
  Visit ("Bob");
}
```



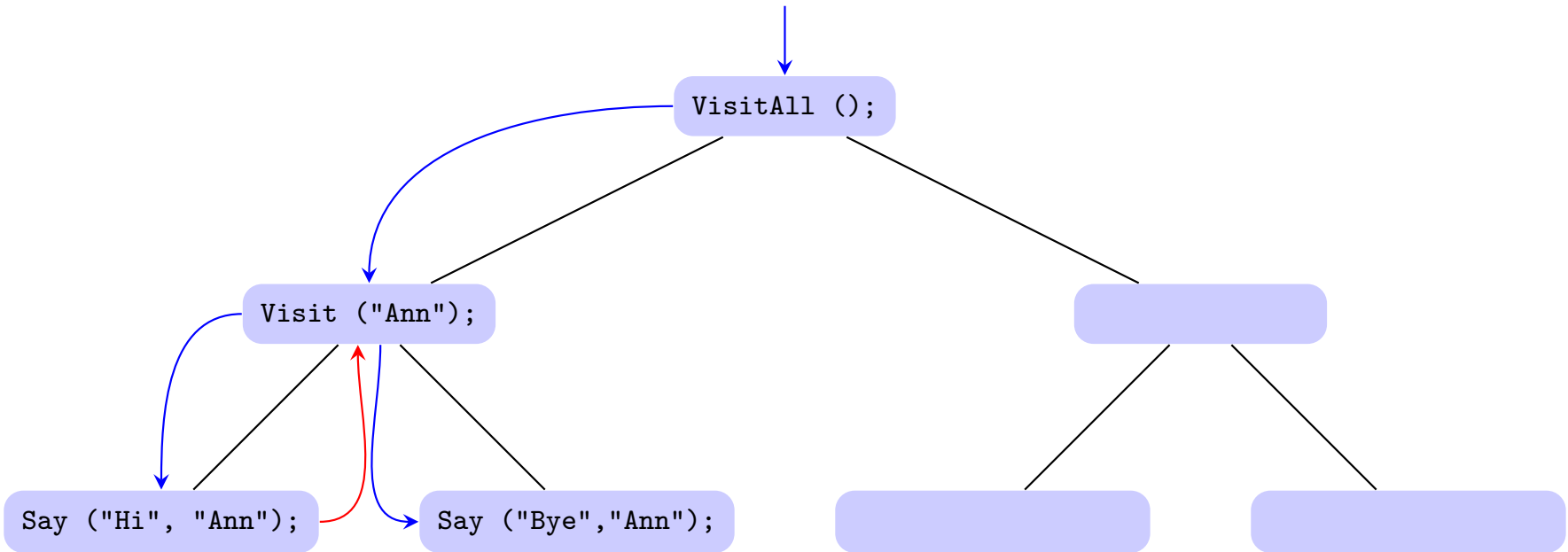
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);  •
}

void Visit (char const name[]) {
  Say ("Hi",  name);
  Say ("Bye", name); ⟸
}

void VisitAll () {
  Visit ("Ann"); ⟸
  Visit ("Bob");
}
```



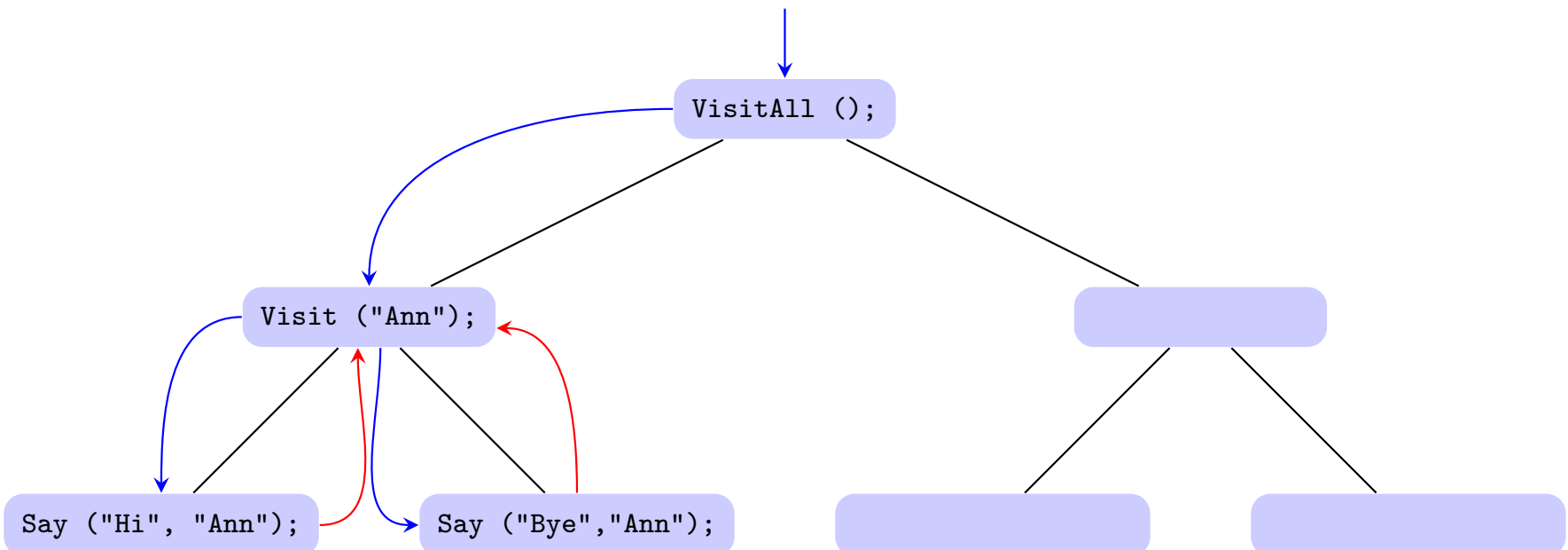
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) {
  Say ("Hi",  name);
  Say ("Bye", name); •
}

void VisitAll () {
  Visit ("Ann"); ⟸
  Visit ("Bob");
}
```



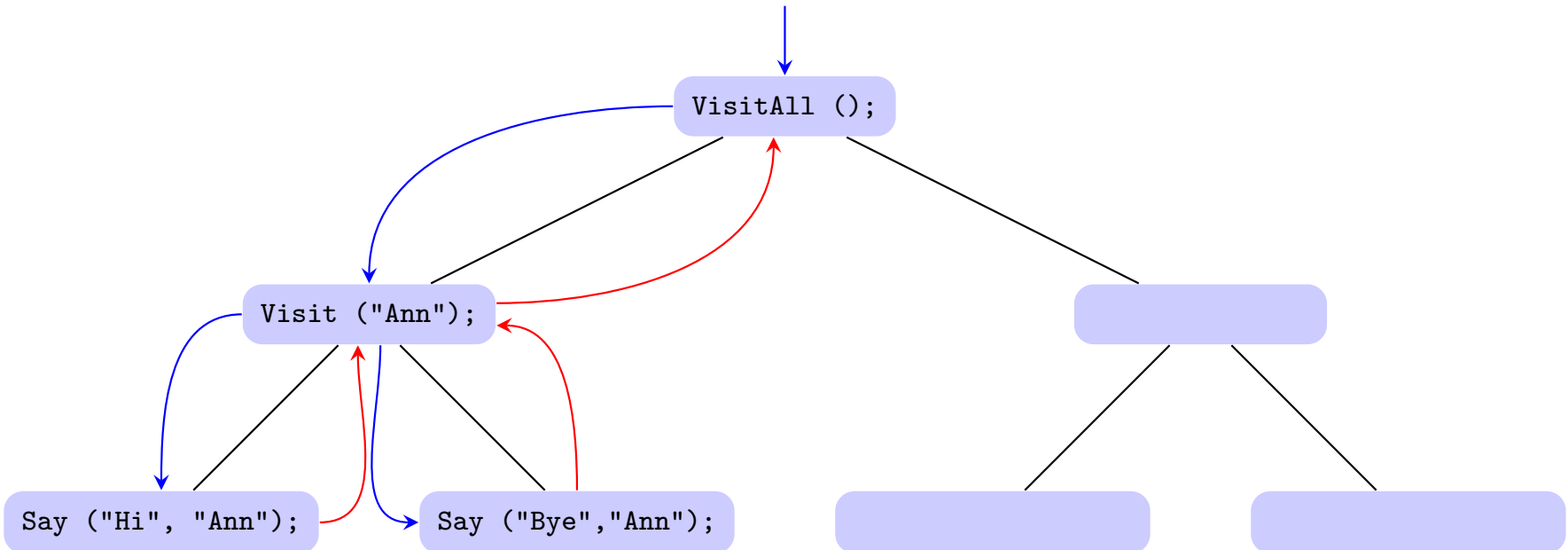
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) {
  Say ("Hi",  name);
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann"); ●
  Visit ("Bob");
}
```



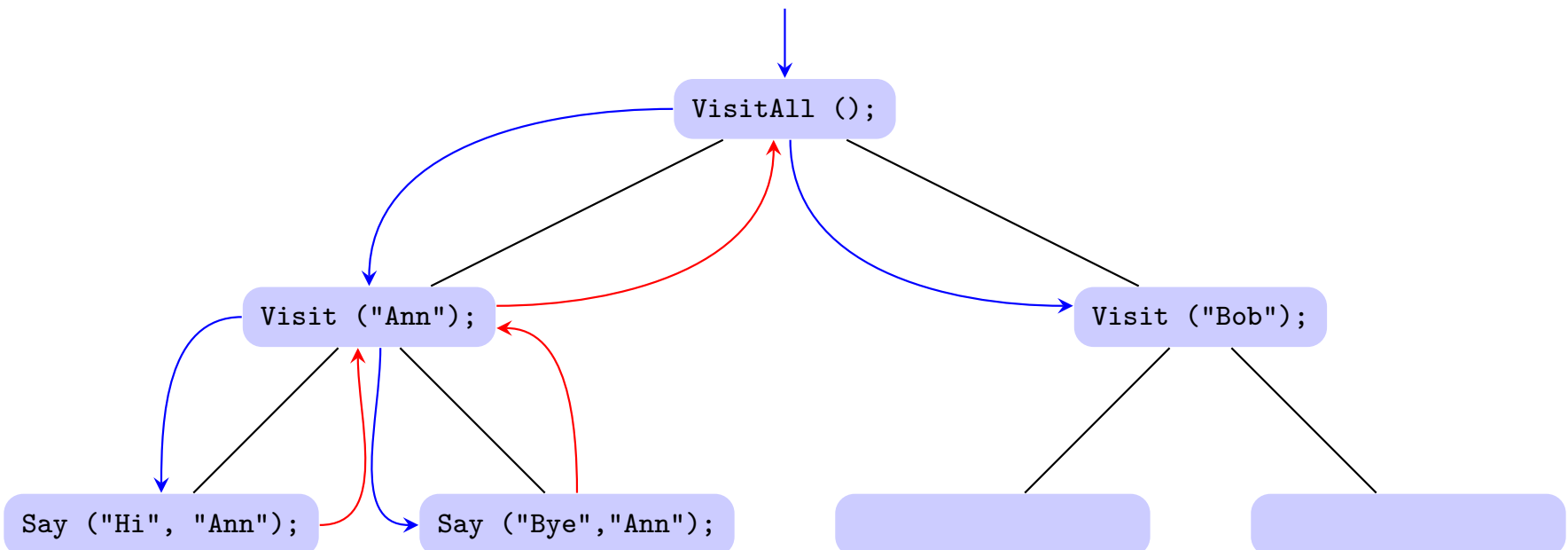
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) { •
  Say ("Hi",  name);
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann");
  Visit ("Bob"); ⟸
}
```



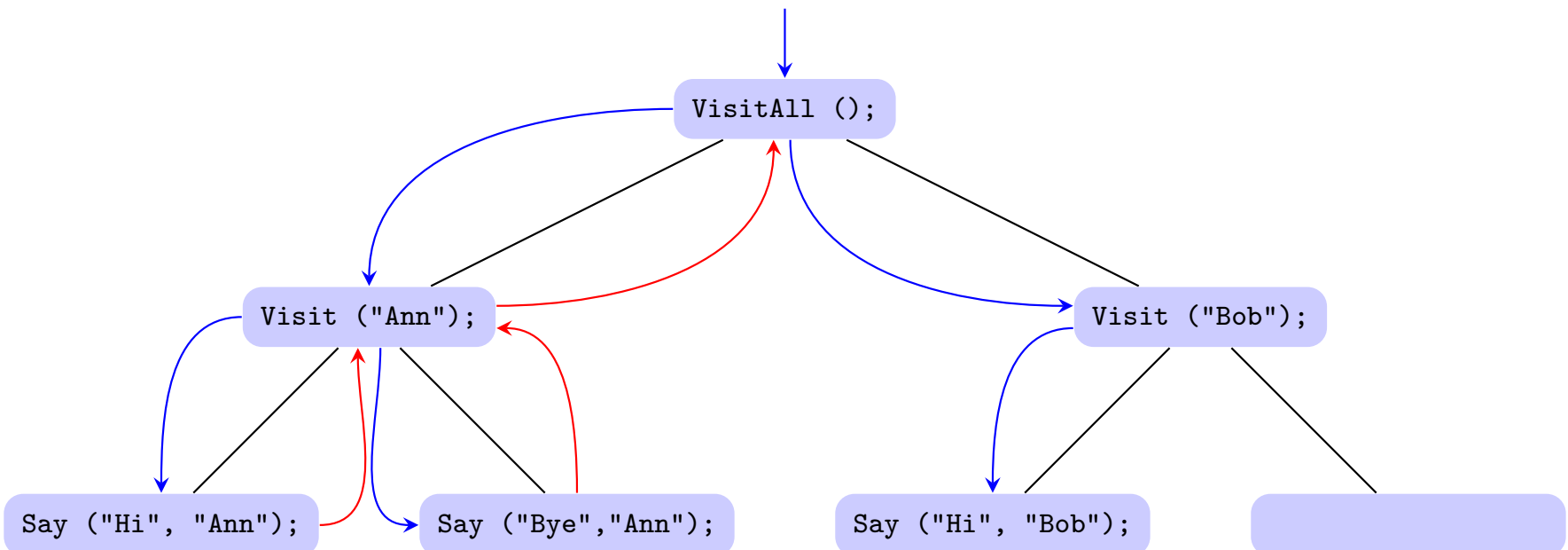
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);  •
}

void Visit (char const name[]) {
  Say ("Hi",  name); ⟸
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann");
  Visit ("Bob"); ⟸
}
```



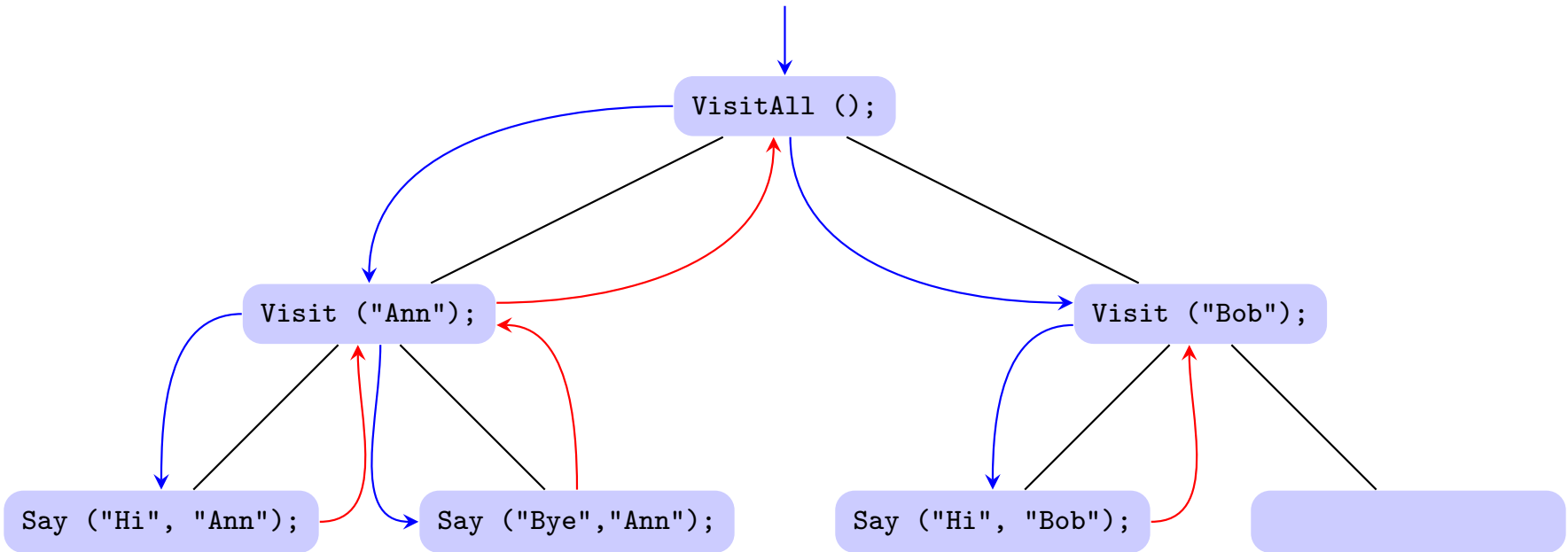
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) {
  Say ("Hi",  name); •
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann");
  Visit ("Bob"); ⟸
}
```



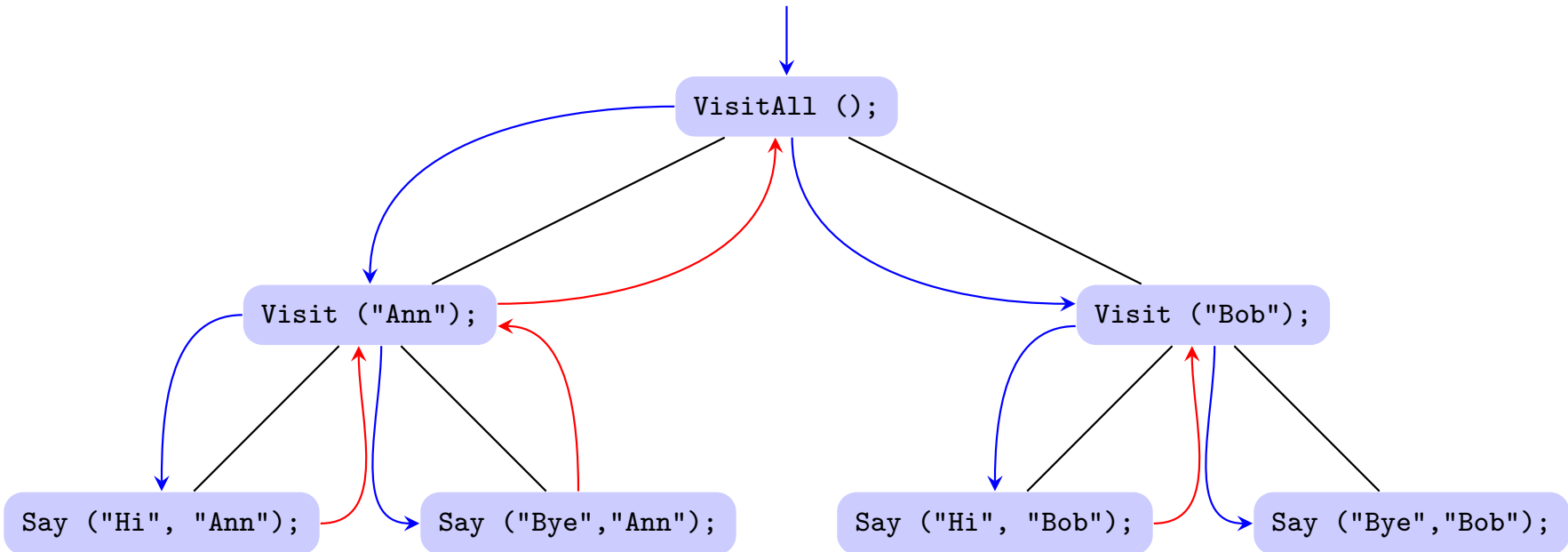
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);  •
}

void Visit (char const name[]) {
  Say ("Hi",  name);
  Say ("Bye", name); ⟸
}

void VisitAll () {
  Visit ("Ann");
  Visit ("Bob"); ⟸
}
```



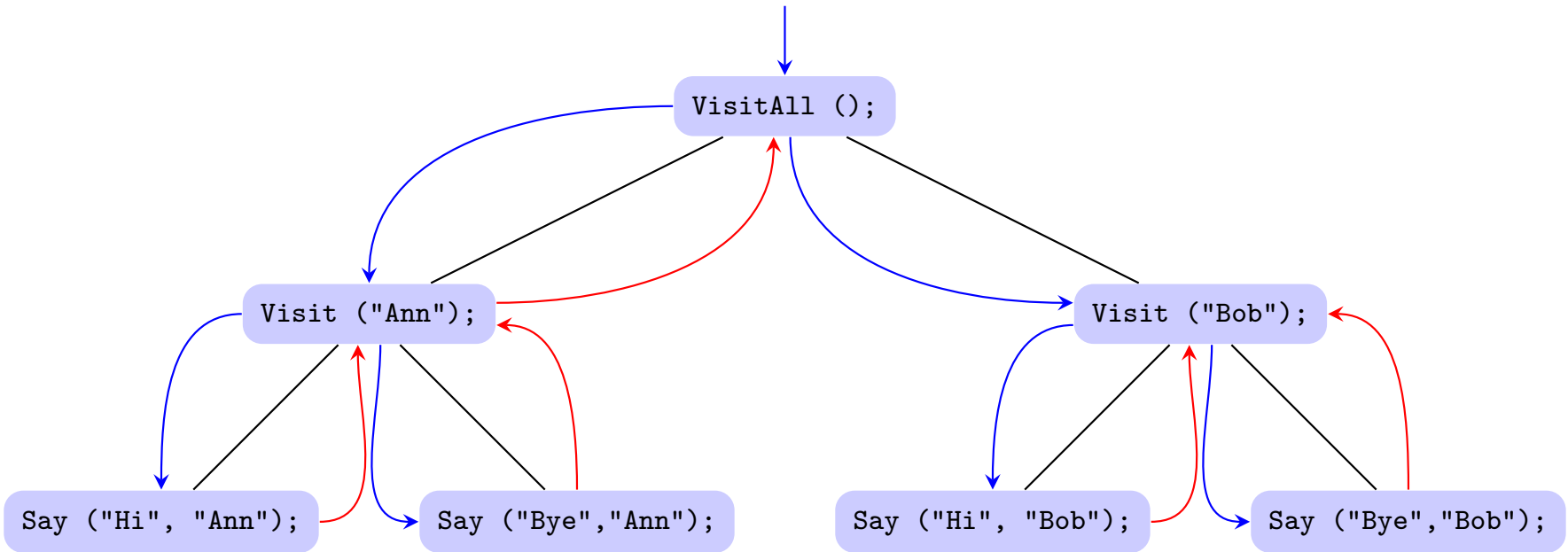
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) {
  Say ("Hi",  name);
  Say ("Bye", name); •
}

void VisitAll () {
  Visit ("Ann");
  Visit ("Bob"); ⟸
}
```



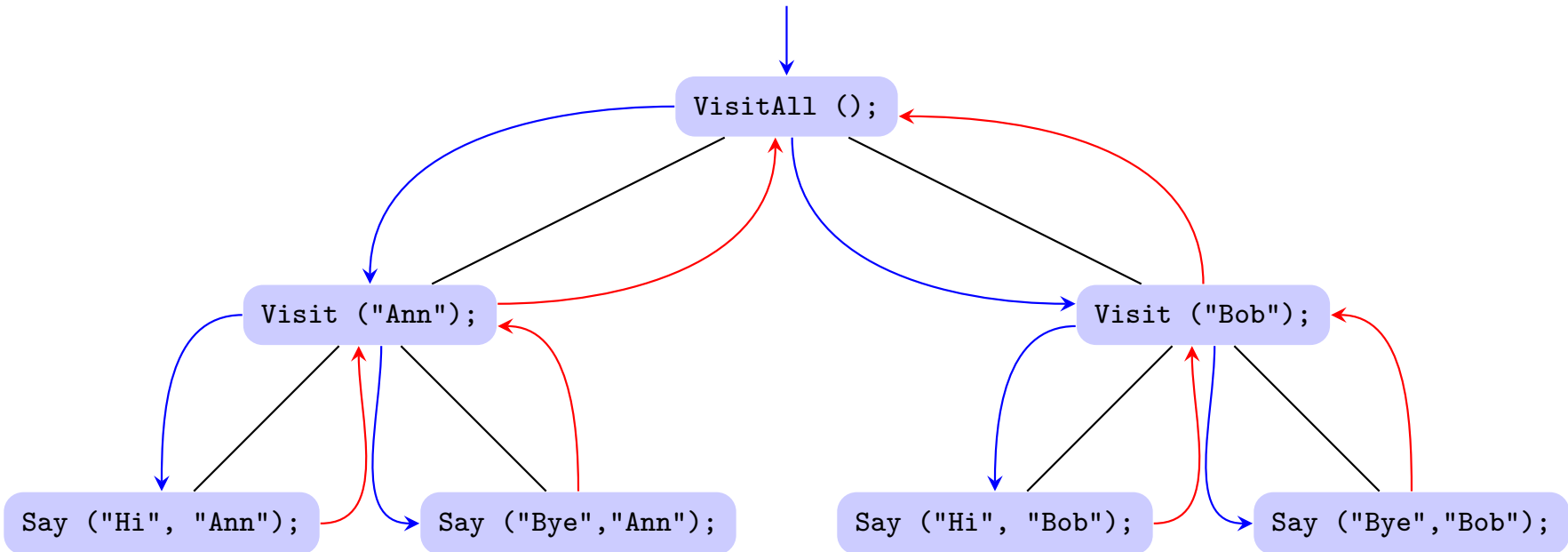
Invocation de fonctions

Traçons la séquence d'invocation de fonctions dans `VisitAll()`.

```
void Say (char const message[], char const name[]) {
  printf ("%s %s!\n", message, name);
}

void Visit (char const name[]) {
  Say ("Hi",  name);
  Say ("Bye", name);
}

void VisitAll () {
  Visit ("Ann");
  Visit ("Bob"); ●
}
```



Fonction récursive (1/5)

Une fonction récursive est une fonction qui s'invoque elle-même.
Une auto-invocation par une telle fonction s'appelle une **récursion**.

En voici une qui s'invoque à l'infini,
en comptant eten affichant les récursions au fur et à mesure :

```
void PrintRecursions (int nb_recursions) {
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1);
  printf ("leaving recursion %d\n", nb_recursions); // code never reached
}
```

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

```
clang -std=c99 -W -Wall -pedantic print-recs.c -o print-recs
```

```
warning: all paths through this function will call itself [-Winfinite-recursion]
void PrintRecursions (int nb_recursions) {
                                         ^
1 warning generated.
```

Fonction récursive (2/5)

Le dernier `printf("leaving...")` n'est en fait jamais atteint, car on ne sort jamais de l'appel récursif qui le précède :

```
void PrintRecursions (int nb_recursions) {
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1);
  printf ("leaving recursion %d\n", nb_recursions); // code never reached
}
```

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

À chaque appel de fonction, le point de l'appel est empilé sur la pile système pour pouvoir y revenir au retour de la fonction. Ici, le programme doit empiler les points d'appel à l'infini, sans jamais pouvoir les dépiler au retour de l'appel, et la pile système finit par être saturée (**stack overflow**) :

```
$ print-recs
entering recursion 0
entering recursion 1
...
entering recursion 261837
Segmentation fault (core dumped)
```

Fonction récursive (3/5)

Si l'on supprime le dernier `printf("leaving...")`,
plus aucun calcul ne suit l'appel récursif.
La fonction devient alors **récursive terminale (tail-recursive)** :

```
void PrintRecursions (int nb_recursions) {
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1);
  // function is now tail-recursive
}
```

Il faut compiler au moins avec le niveau d'optimisation `-O1`
pour que `clang` détecte la récursivite terminale :

```
clang -std=c99 -W -Wall -pedantic -O1 print-recs.c -o print-recs
```

Les points d'appel ne sont alors plus empilés sur la pile système,
et la fonction peut alors tourner à l'infini sans saturer la pile :

```
$ print-recs
entering recursion 0
entering recursion 1
...
entering recursion 1000000
...
```

Fonction récursive (4/5)

Remetttons le dernier `printf("leaving...")`,
et imposons une limite au nombre de récursions,
via un paramètre supplémentaire `max_recursions`.
Le **cas d'arrêt** est testé par une **clause de garde** :

```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```

Fixons une limite de 3 appels récursifs dans le `main()` :

```
int main (void) {
  PrintRecursions (0, 3);
  return 0;
}
```

On obtient la sortie suivante :

```
$ print-recs
entering recursion 0
entering recursion 1
entering recursion 2
leaving recursion 2
leaving recursion 1
leaving recursion 0
```

Fonction récursive (5/5)

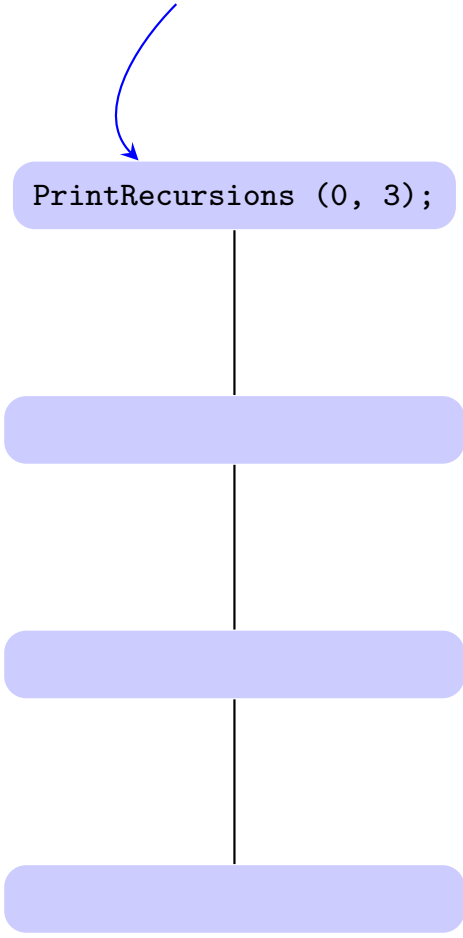
```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```

Fonction récursive (5/5)

```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```

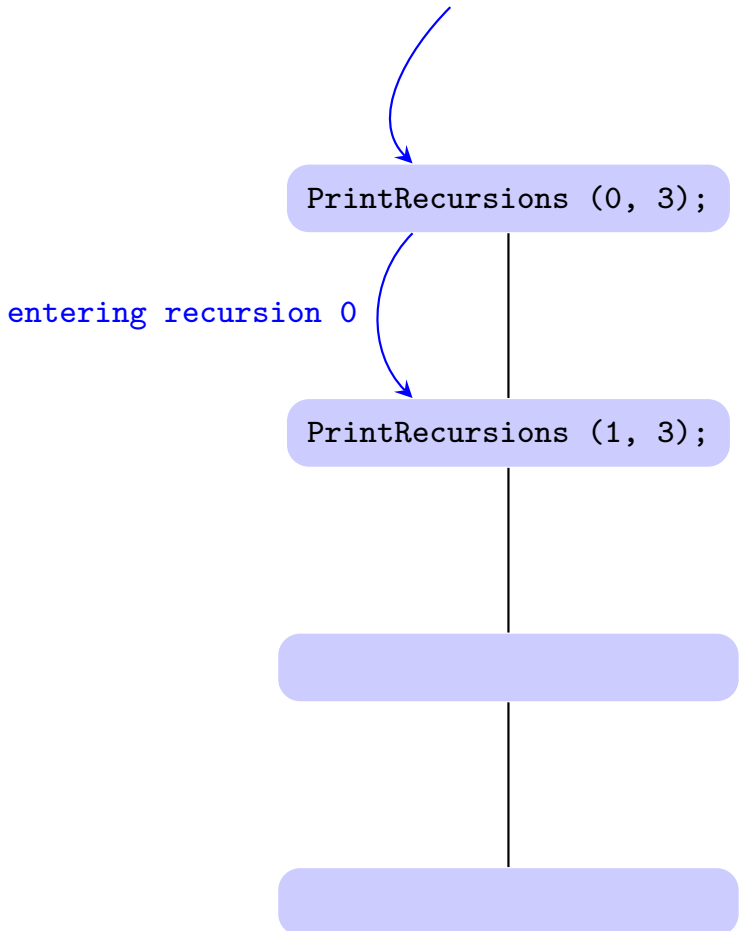
Fonction récursive (5/5)

```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```



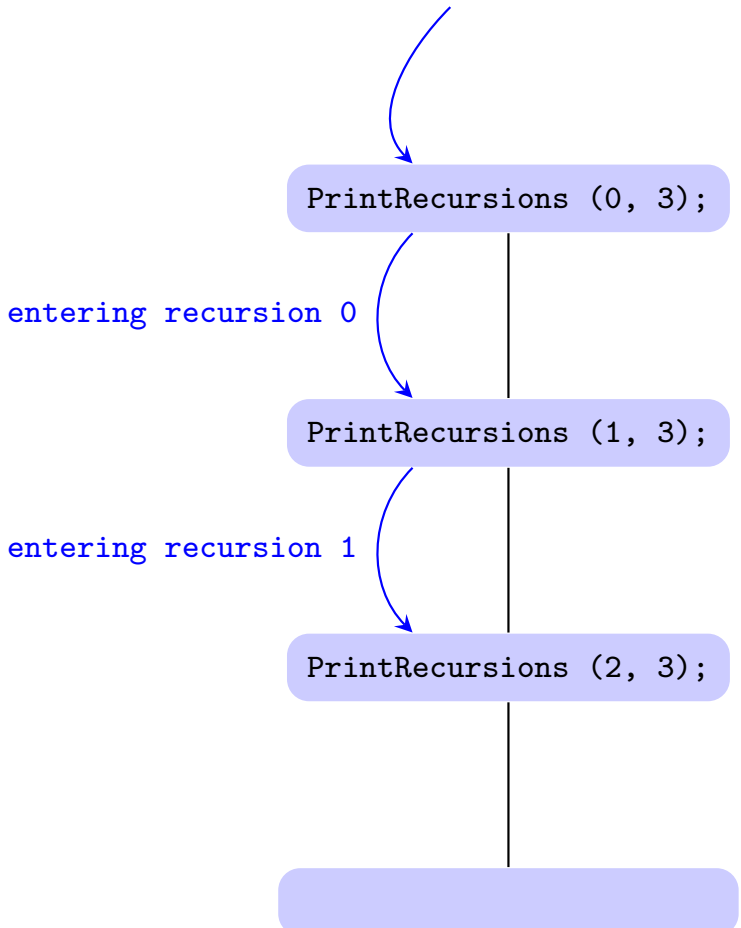
Fonction récursive (5/5)

```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```



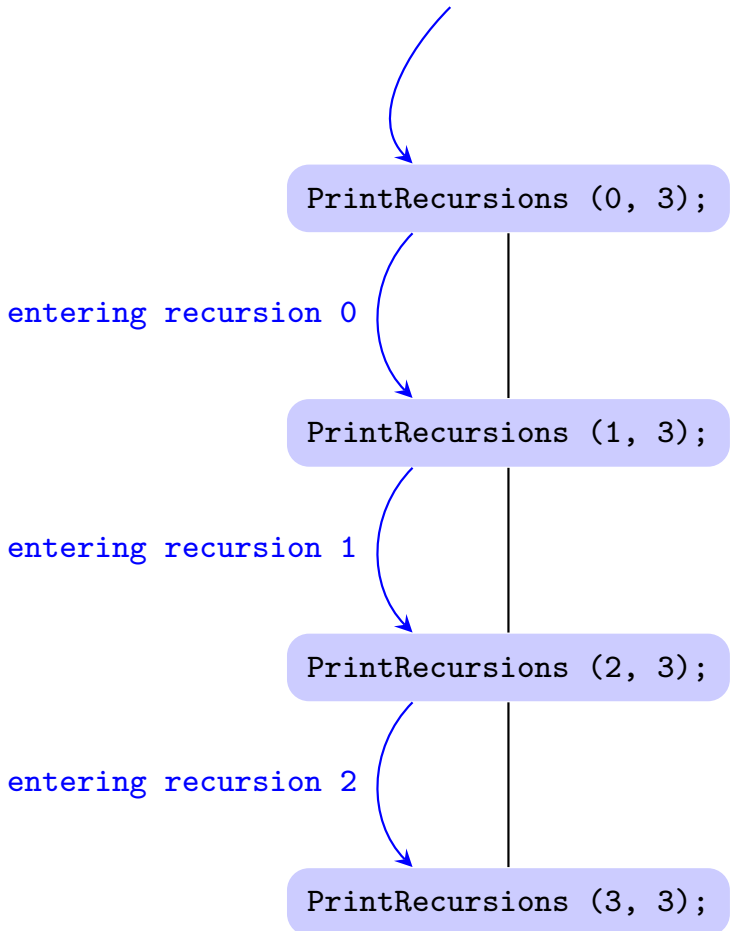
Fonction récursive (5/5)

```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```



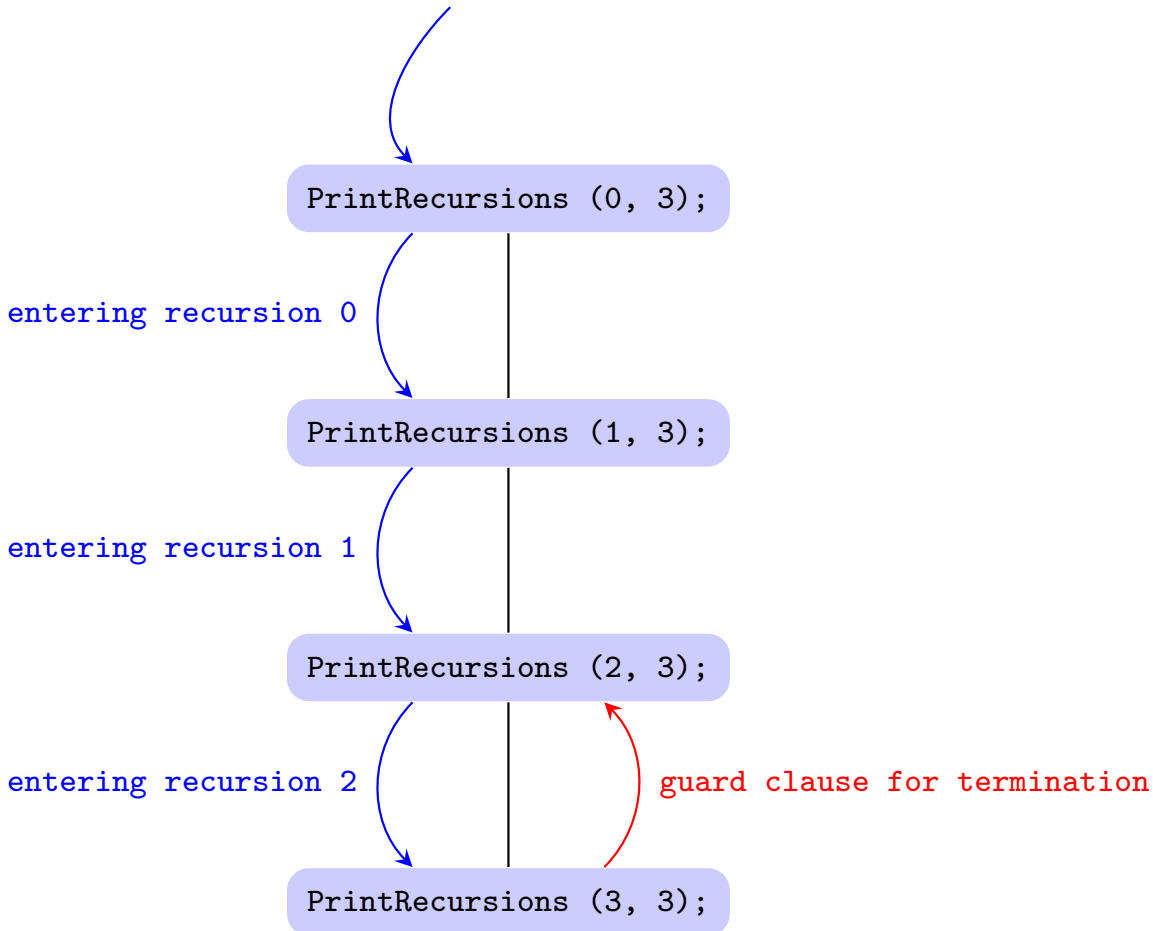
Fonction récursive (5/5)

```c
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```



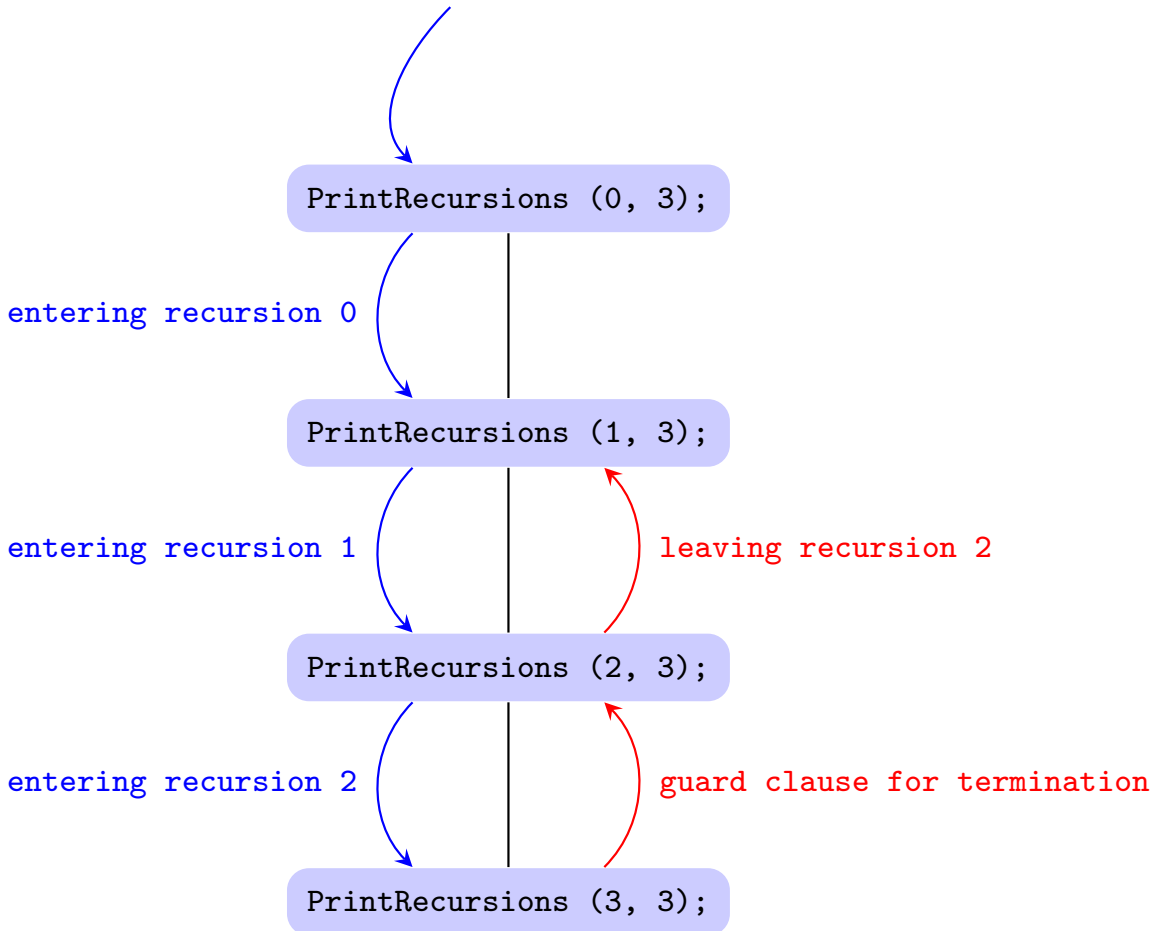
Fonction récursive (5/5)

```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```



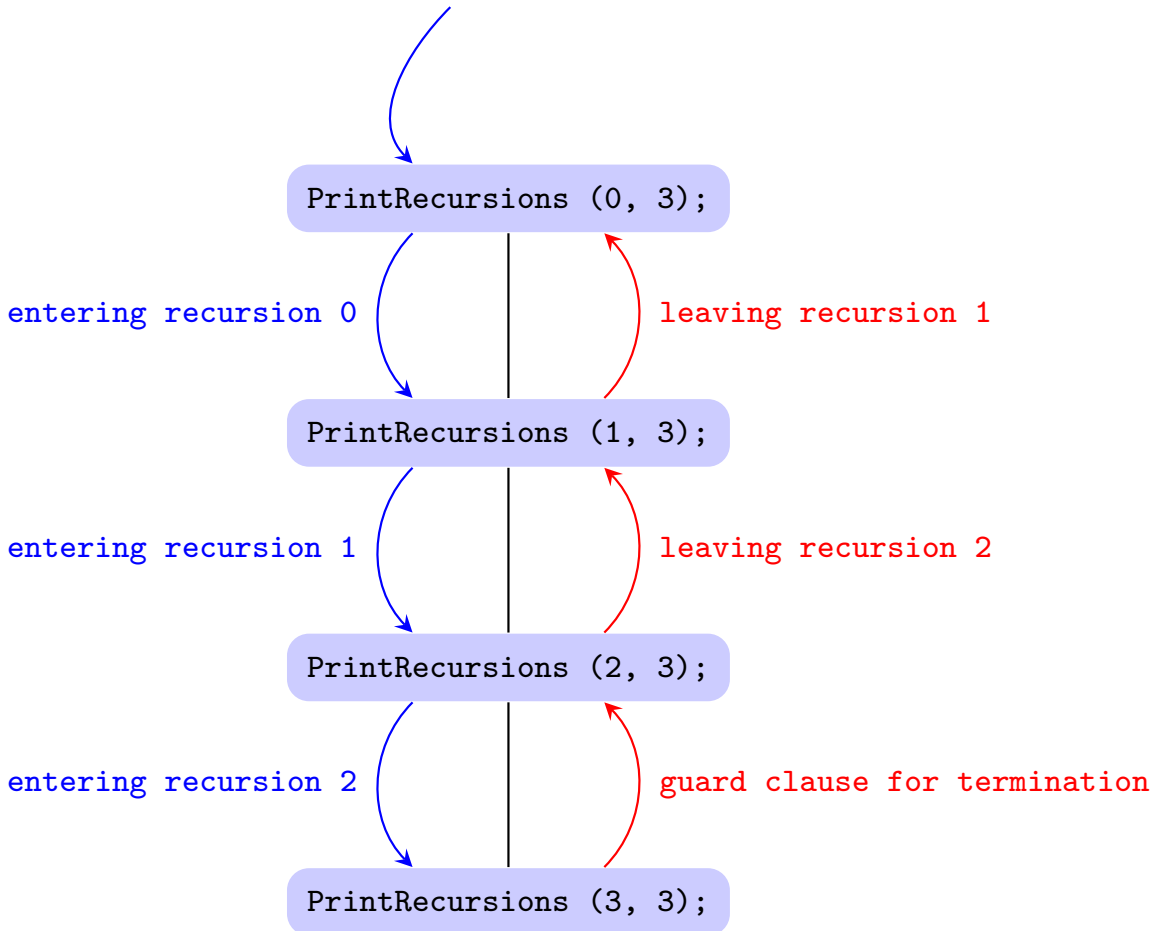
Fonction récursive (5/5)

```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```



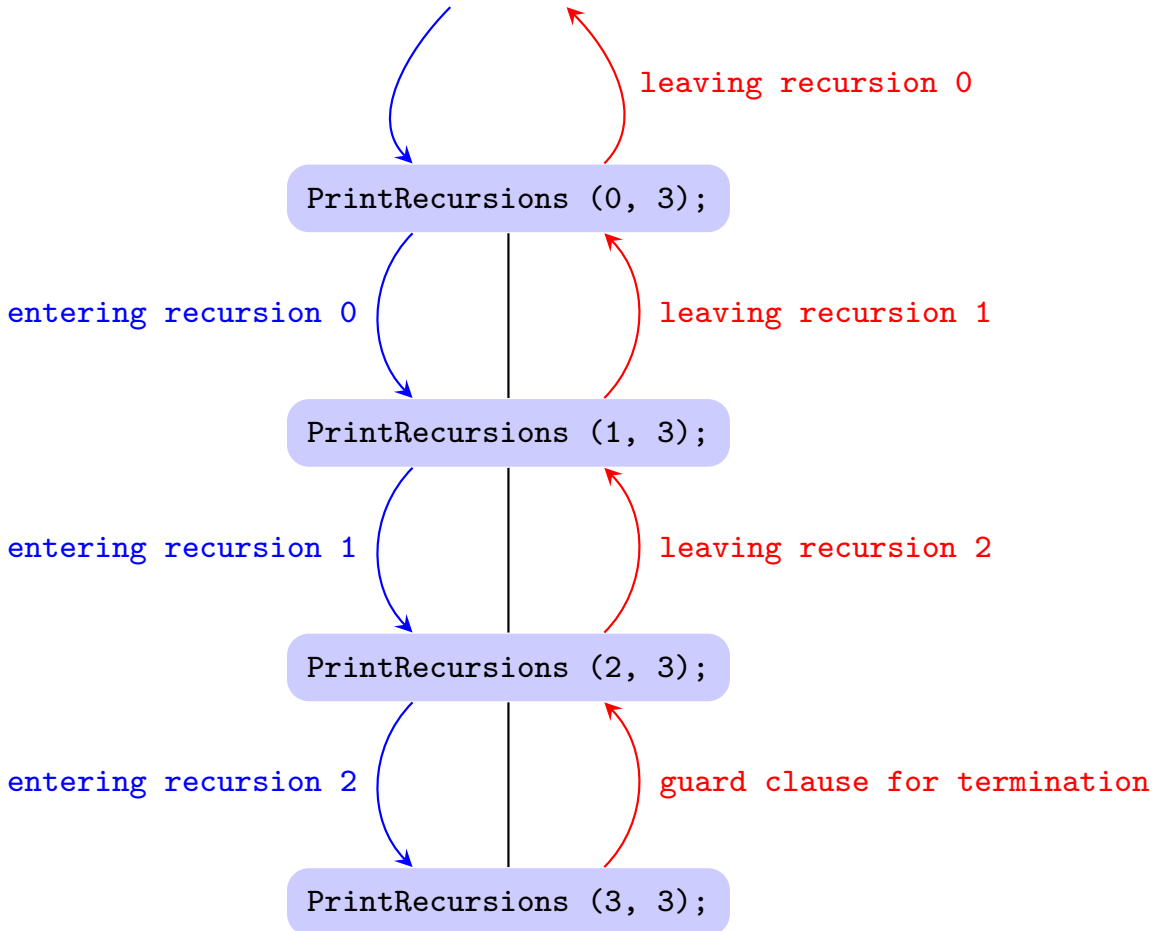
Fonction récursive (5/5)

```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```



Fonction récursive (5/5)

```
void PrintRecursions (int nb_recursions, int max_recursions) {
  if (nb_recursions >= max_recursions) return; // guard clause for termination
  printf ("entering recursion %d\n", nb_recursions);
  PrintRecursions (nb_recursions + 1, max_recursions);
  printf ("leaving recursion %d\n", nb_recursions);
}
```



Somme récursive (1/2) de la gauche vers la droite

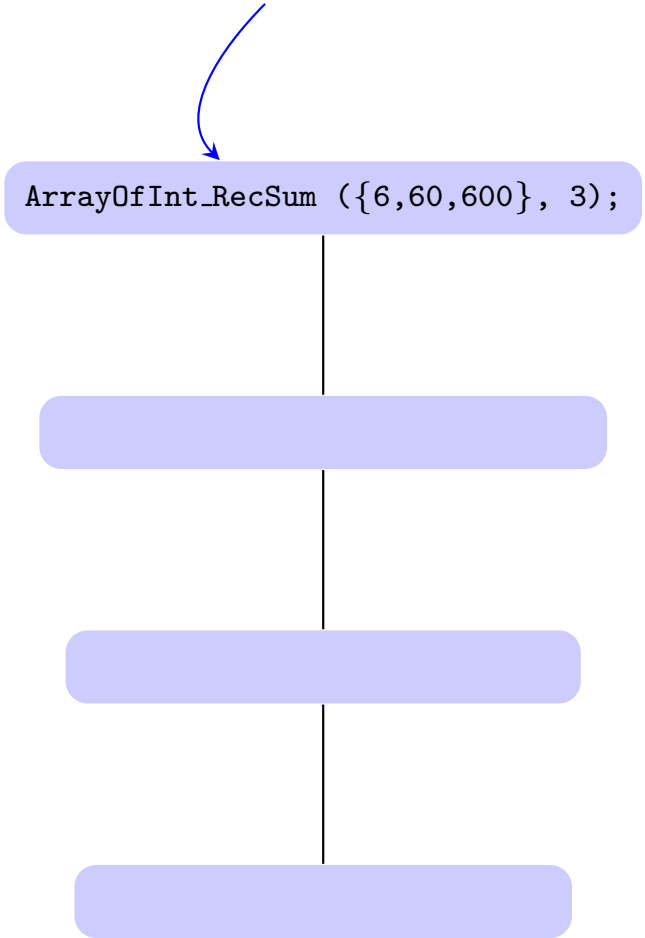
```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```

Somme récursive (1/2) de la gauche vers la droite

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```

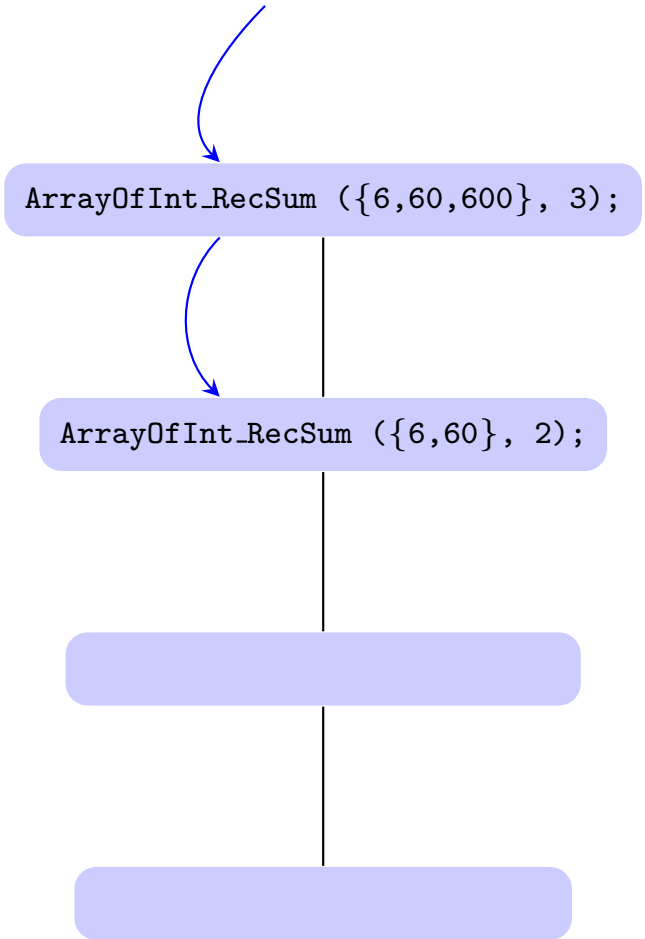
Somme récursive (1/2) de la gauche vers la droite

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```



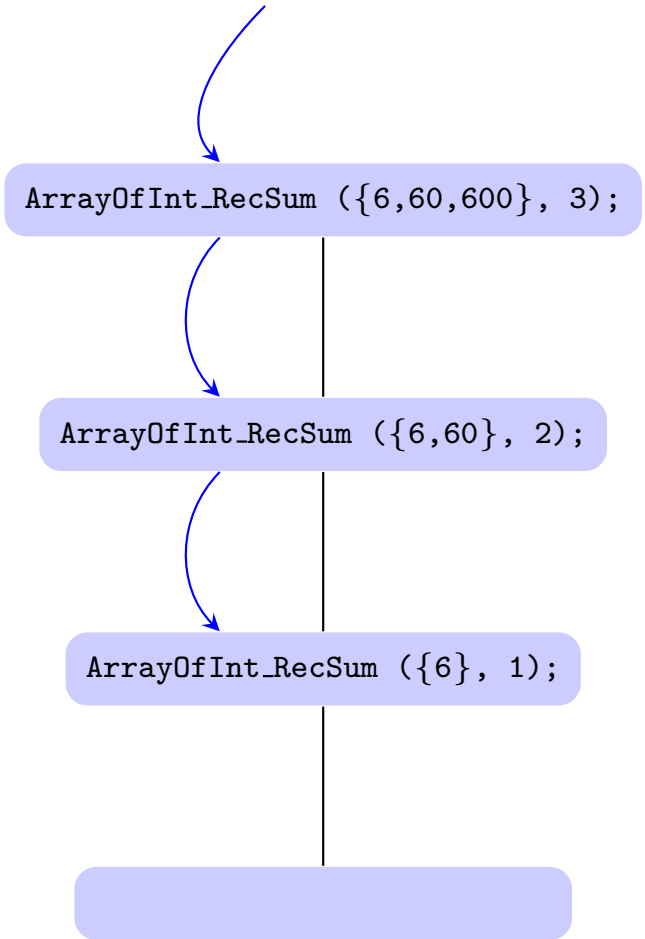
Somme récursive (1/2) de la gauche vers la droite

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```



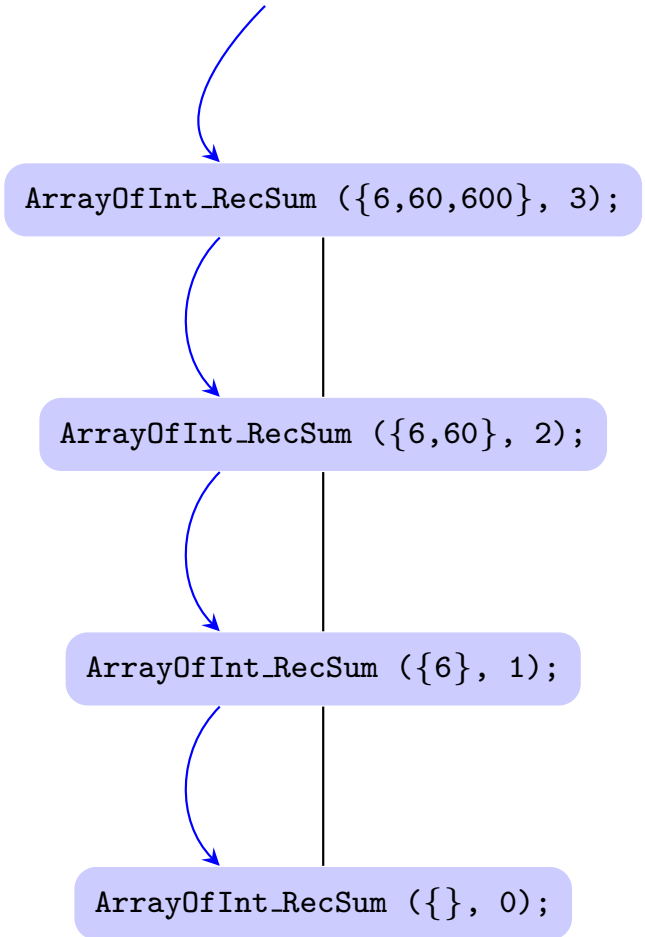
Somme récursive (1/2) de la gauche vers la droite

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```



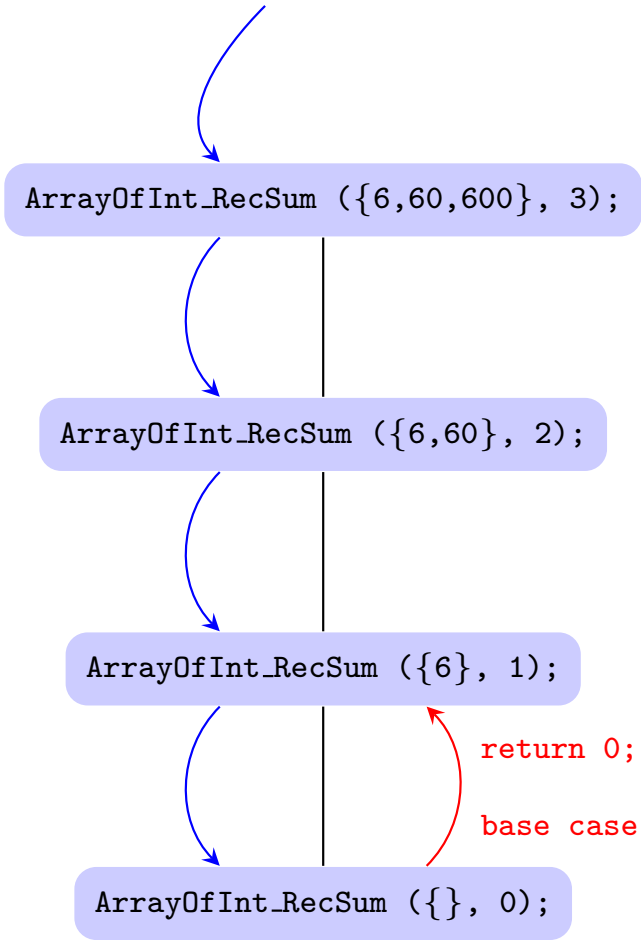
Somme récursive (1/2) de la gauche vers la droite

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```



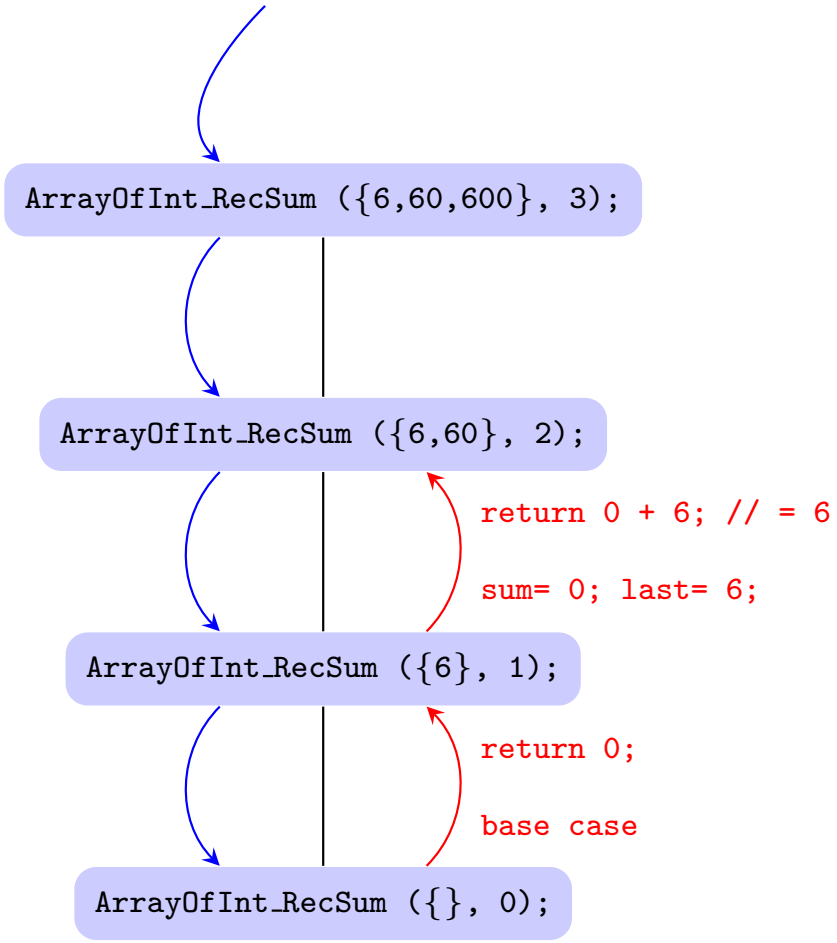
Somme récursive (1/2) de la gauche vers la droite

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```



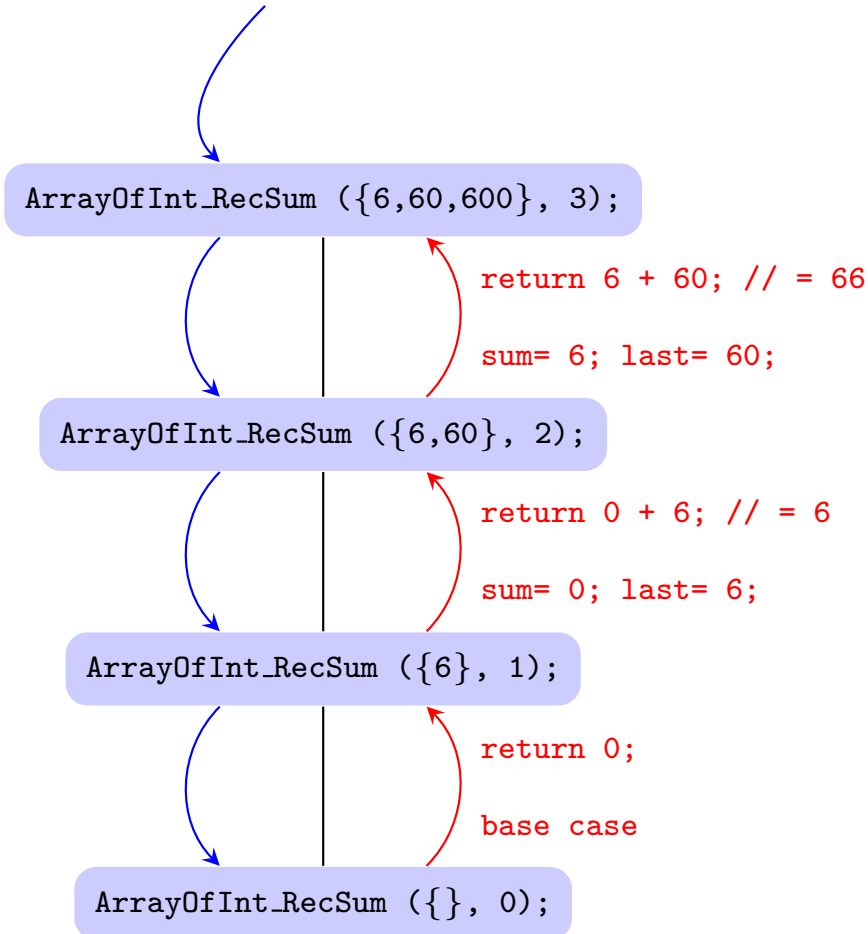
Somme récursive (1/2) de la gauche vers la droite

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```



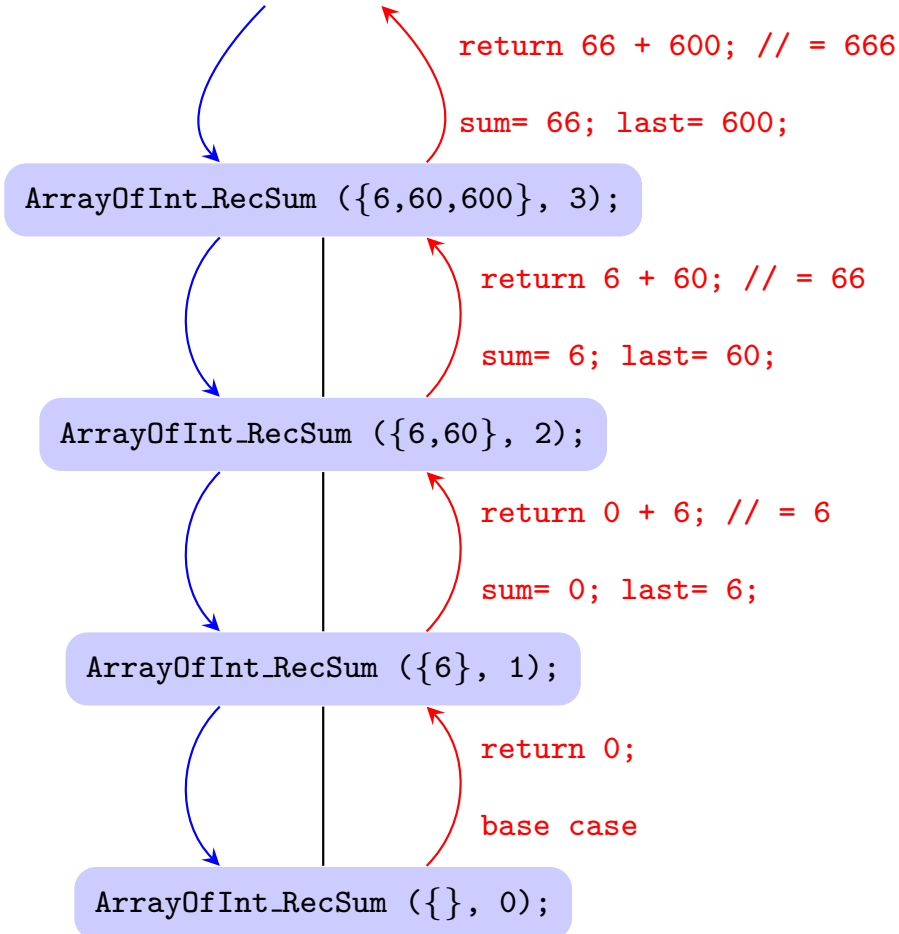
Somme récursive (1/2) de la gauche vers la droite

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```



Somme récursive (1/2) de la gauche vers la droite

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int sum= ArrayOfInt_RecSum (array, length-1);
  int last= array[length-1];
  return sum + last; // general case
}
```



Somme récursive (2/2) de la droite vers la gauche

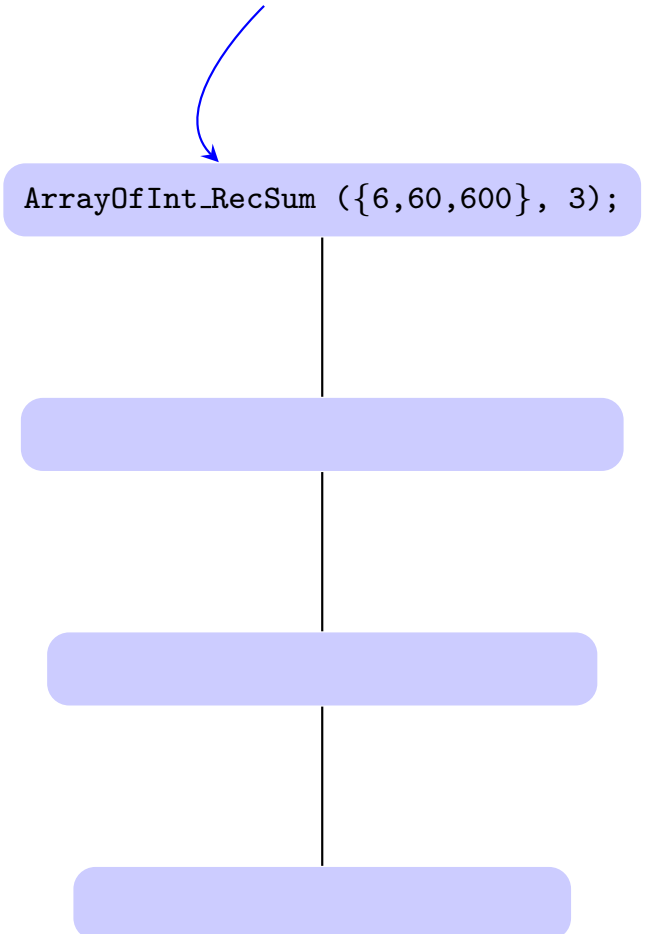
```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```

Somme récursive (2/2) de la droite vers la gauche

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```

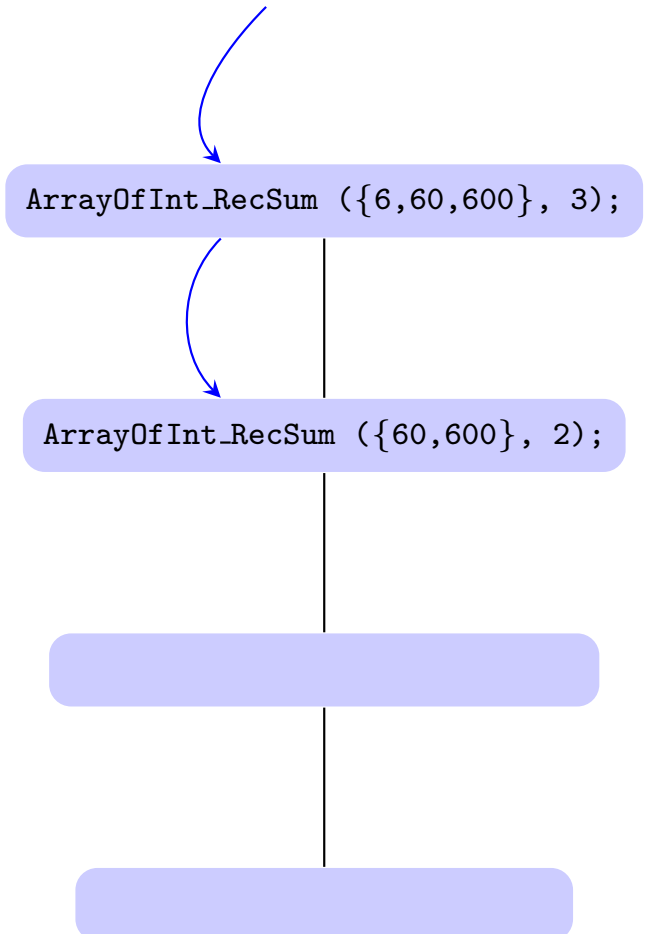
Somme récursive (2/2) de la droite vers la gauche

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```



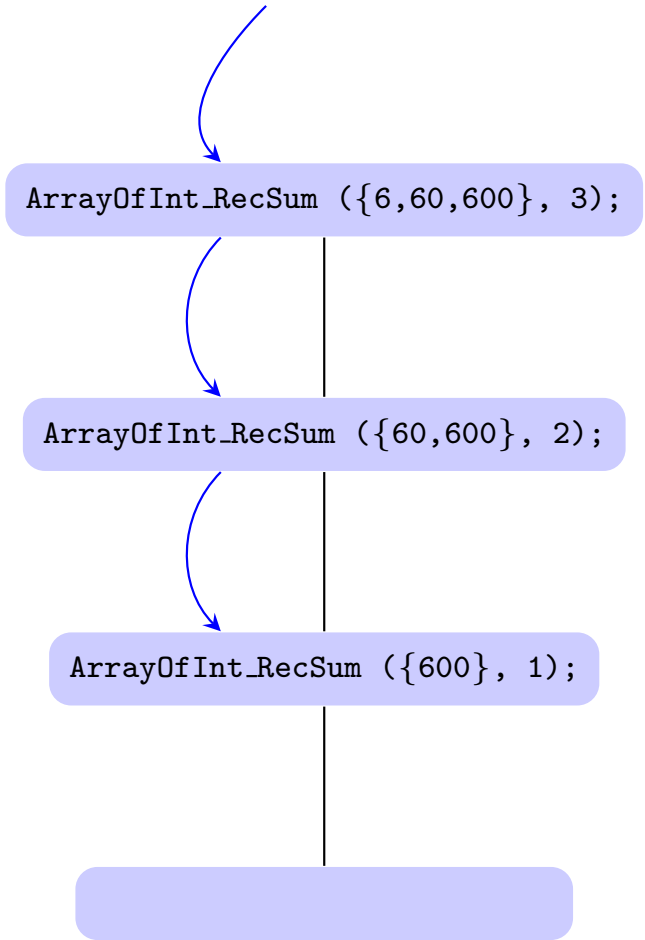
Somme récursive (2/2) de la droite vers la gauche

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```



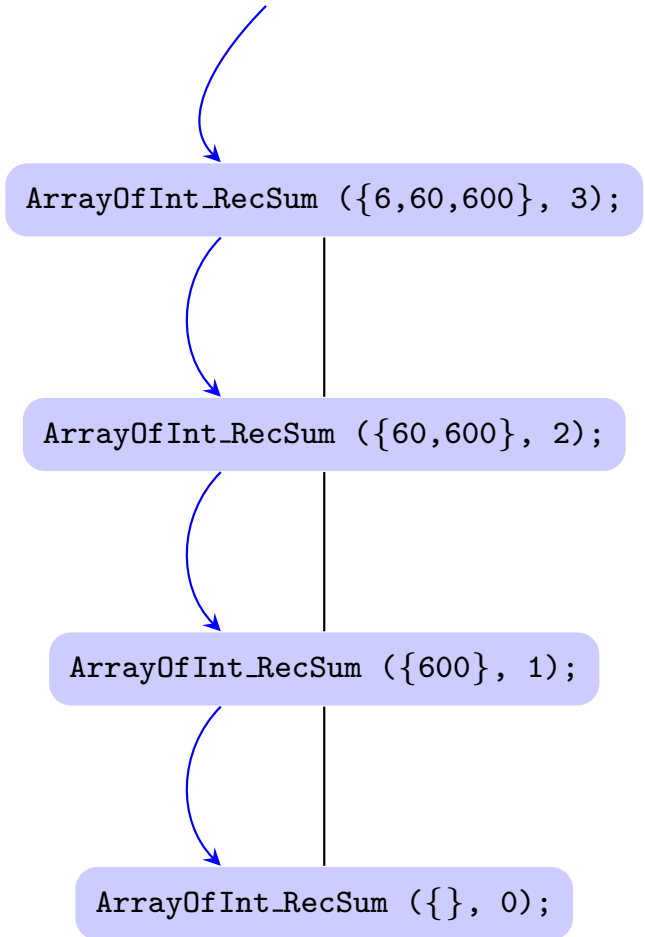
Somme récursive (2/2) de la droite vers la gauche

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```



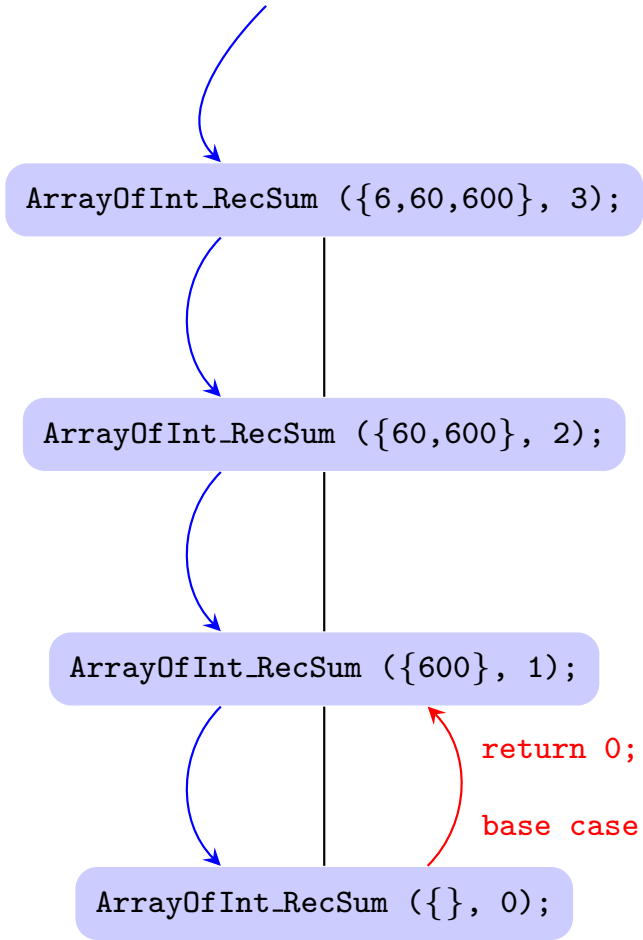
Somme récursive (2/2) de la droite vers la gauche

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```



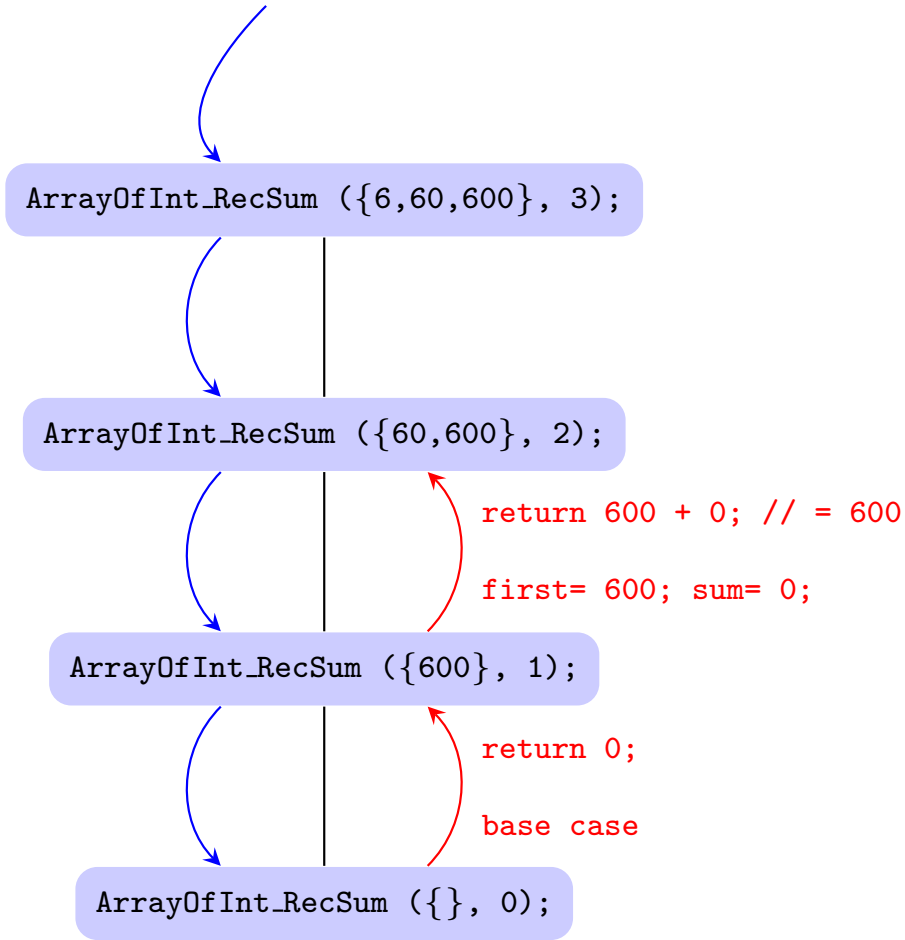
Somme récursive (2/2) de la droite vers la gauche

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```



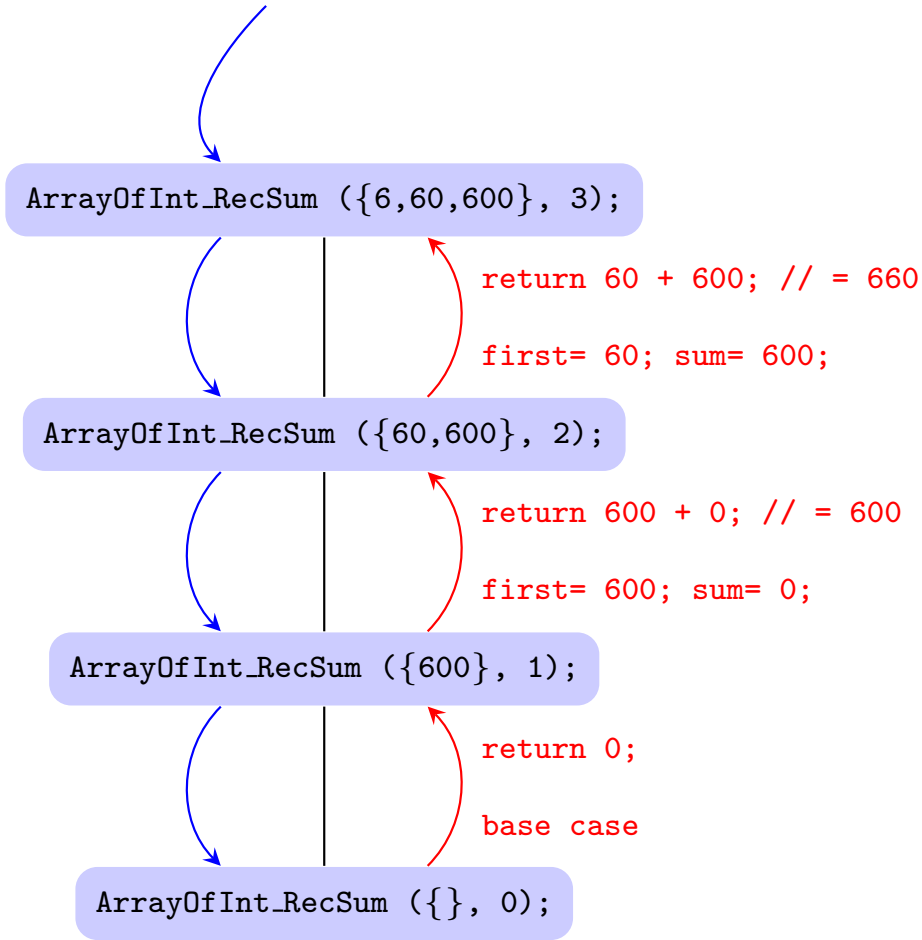
Somme récursive (2/2) de la droite vers la gauche

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```



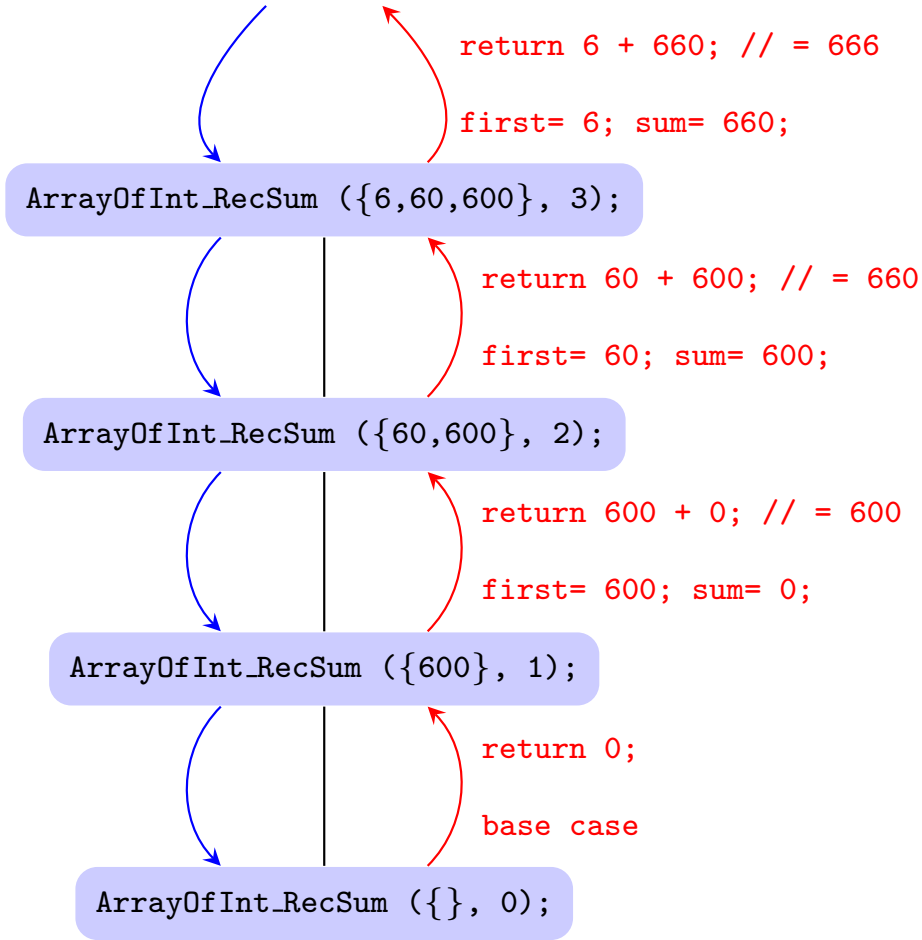
Somme récursive (2/2) de la droite vers la gauche

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```



Somme récursive (2/2) de la droite vers la gauche

```
int ArrayOfInt_RecSum (int const array[], int length) {
  if (length == 0) return 0; // guard clause for base case
  int first= array[0];
  int sum= ArrayOfInt_RecSum (array + 1, length-1);
  return first + sum; // general case
}
```



Somme réc. terminale (1/2) de la gauche vers la droite

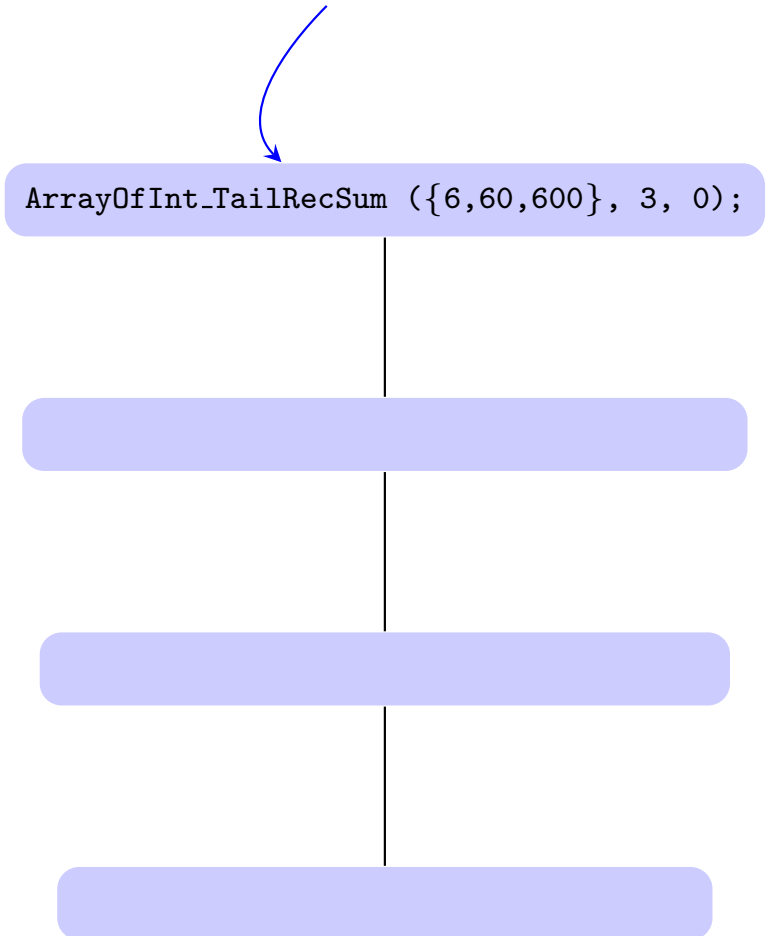
```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```

Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```

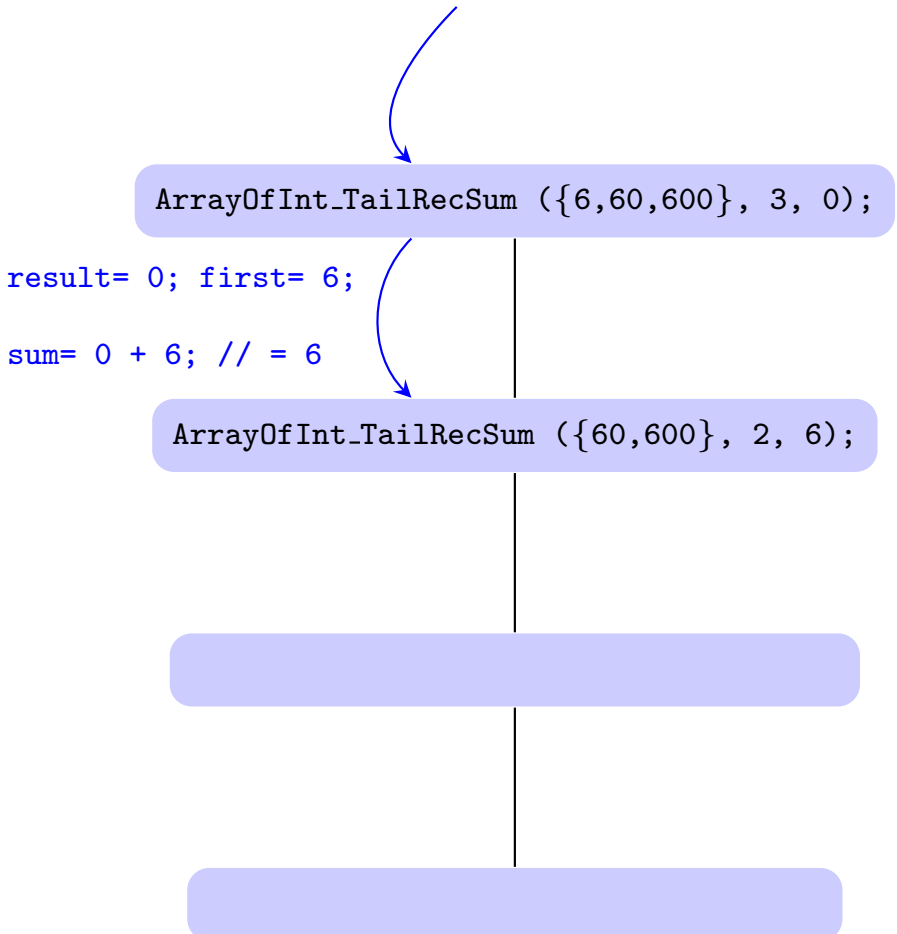
Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```



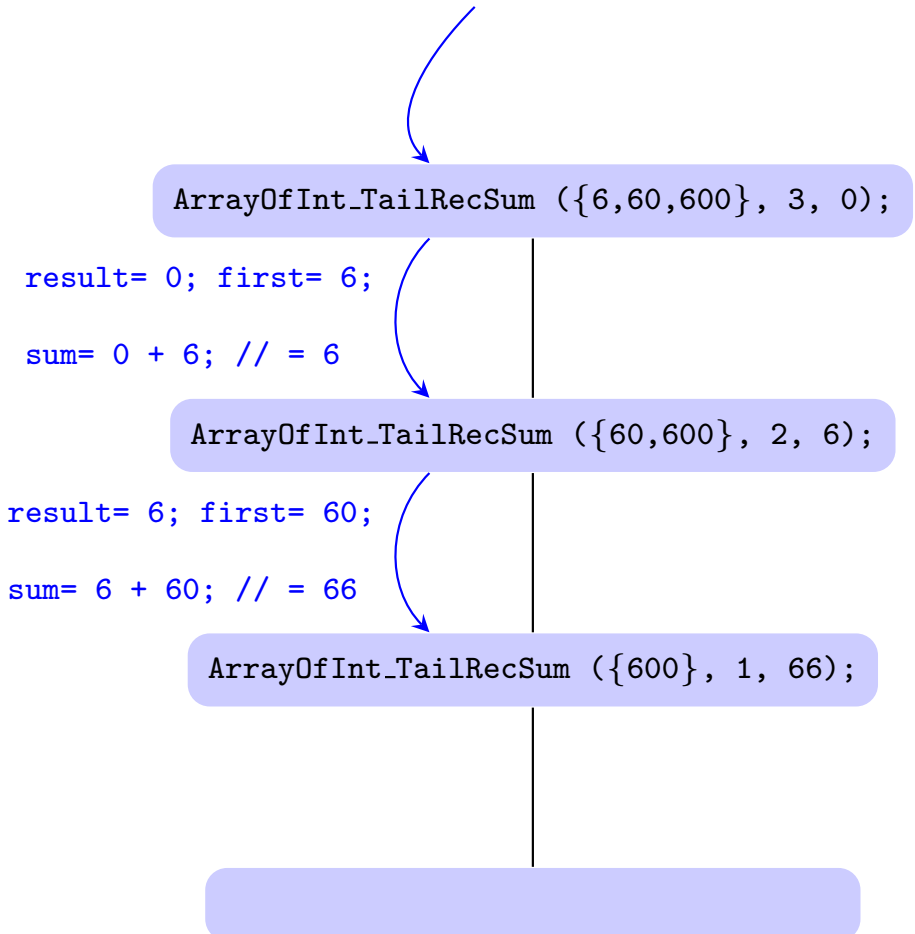
Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```



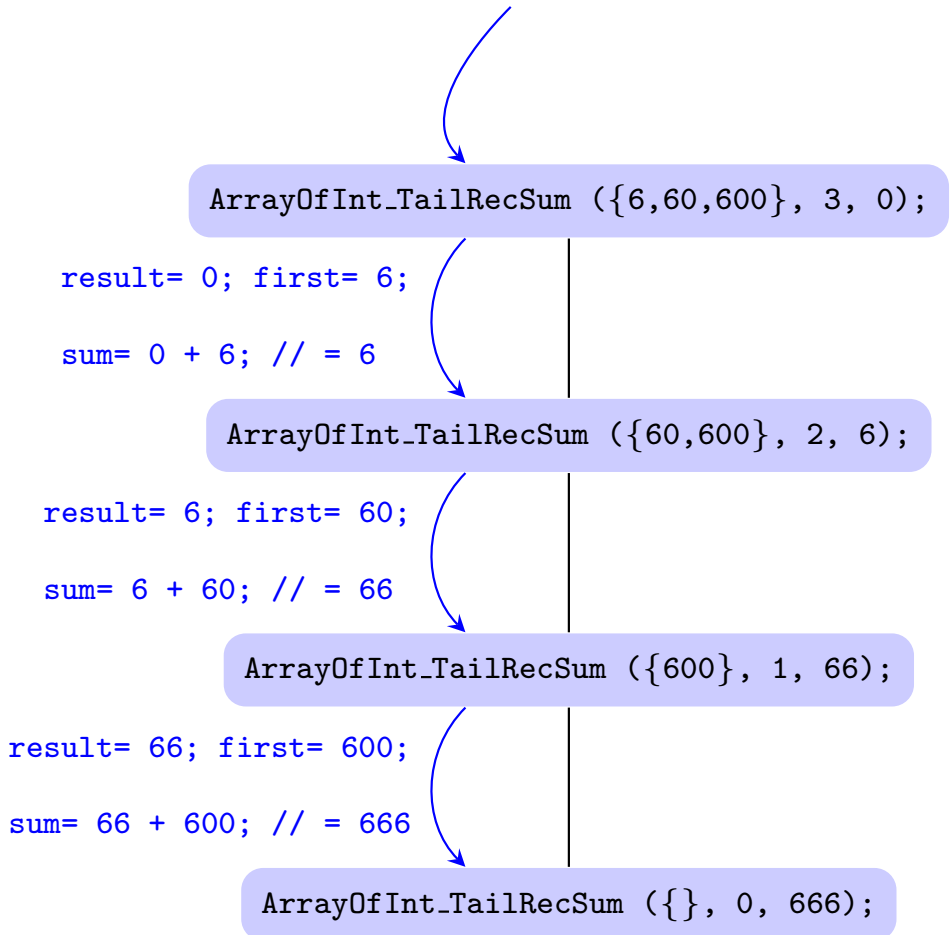
Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```



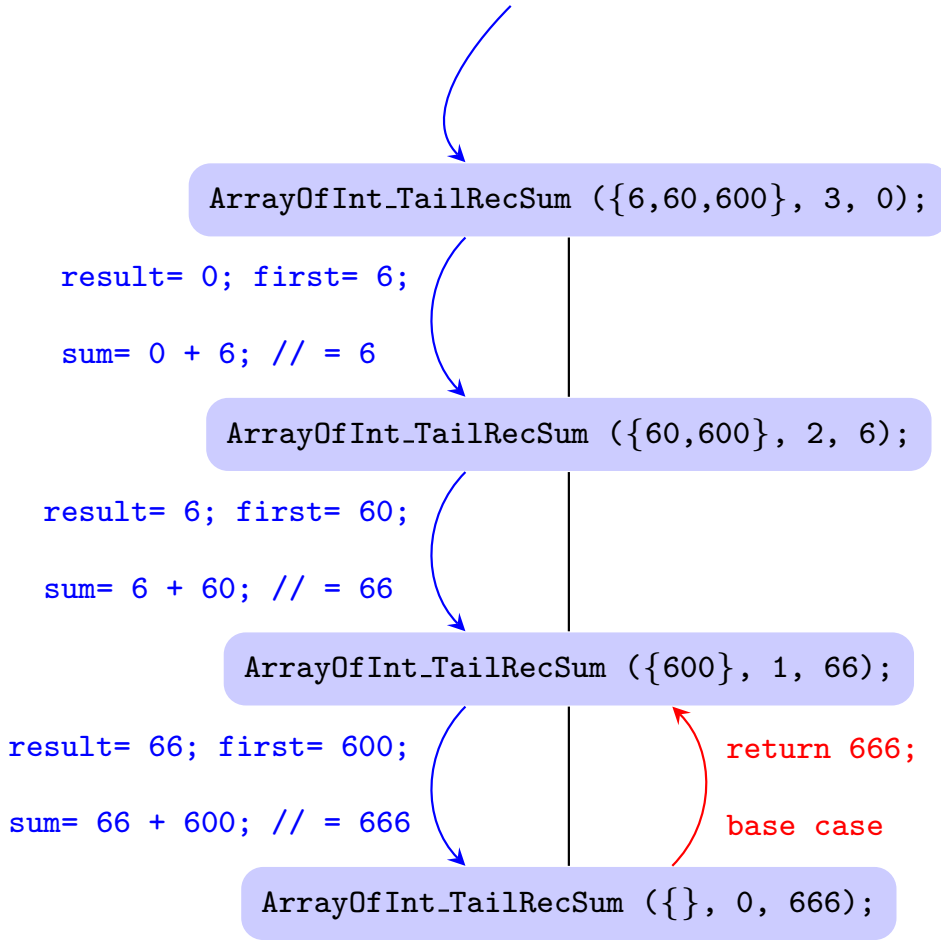
Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```



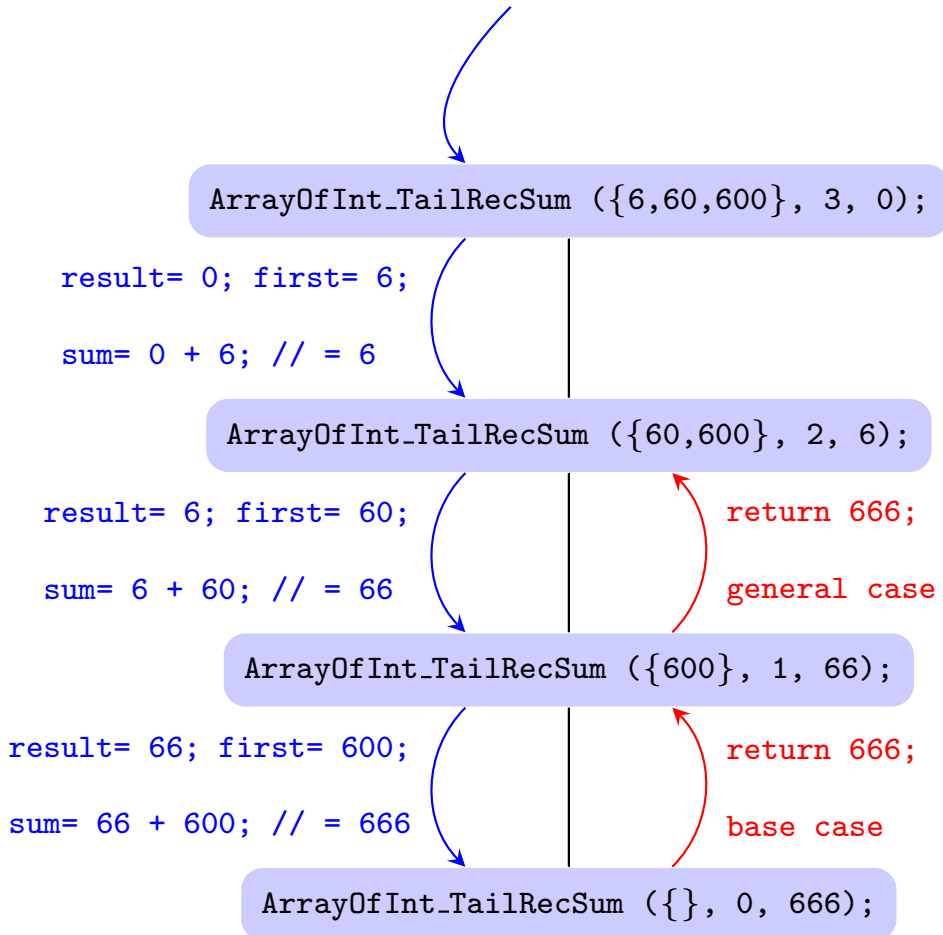
Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```



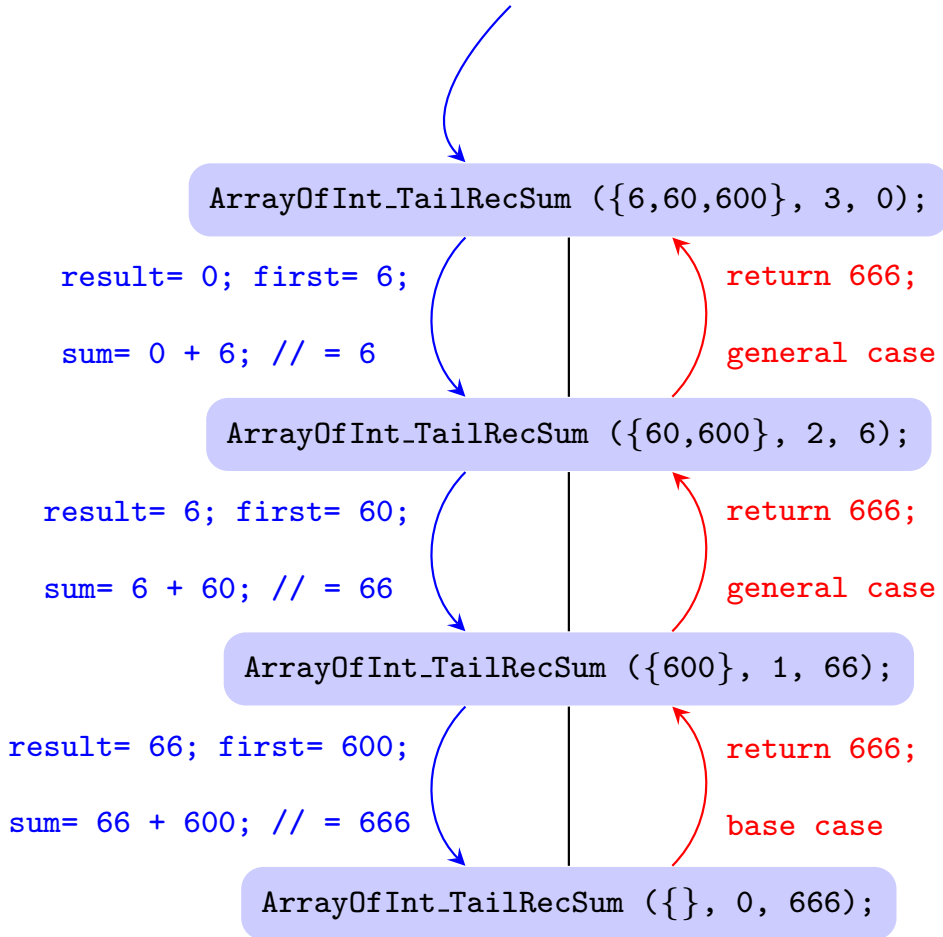
Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```



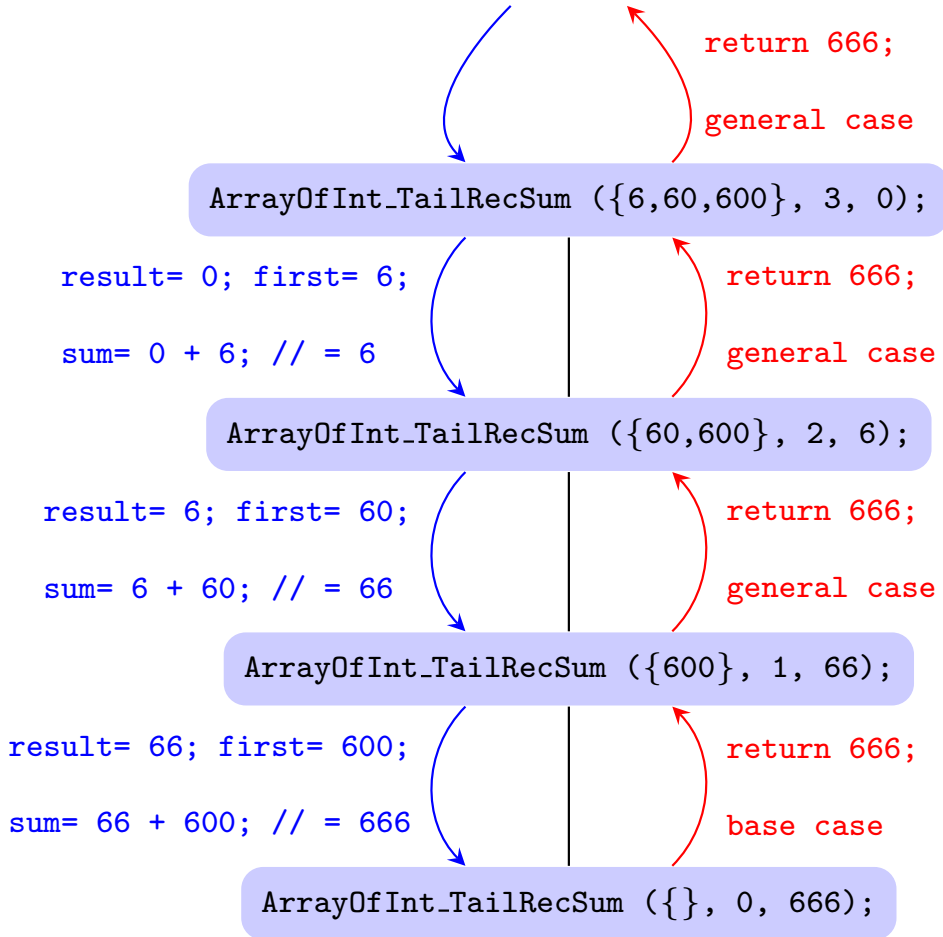
Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```



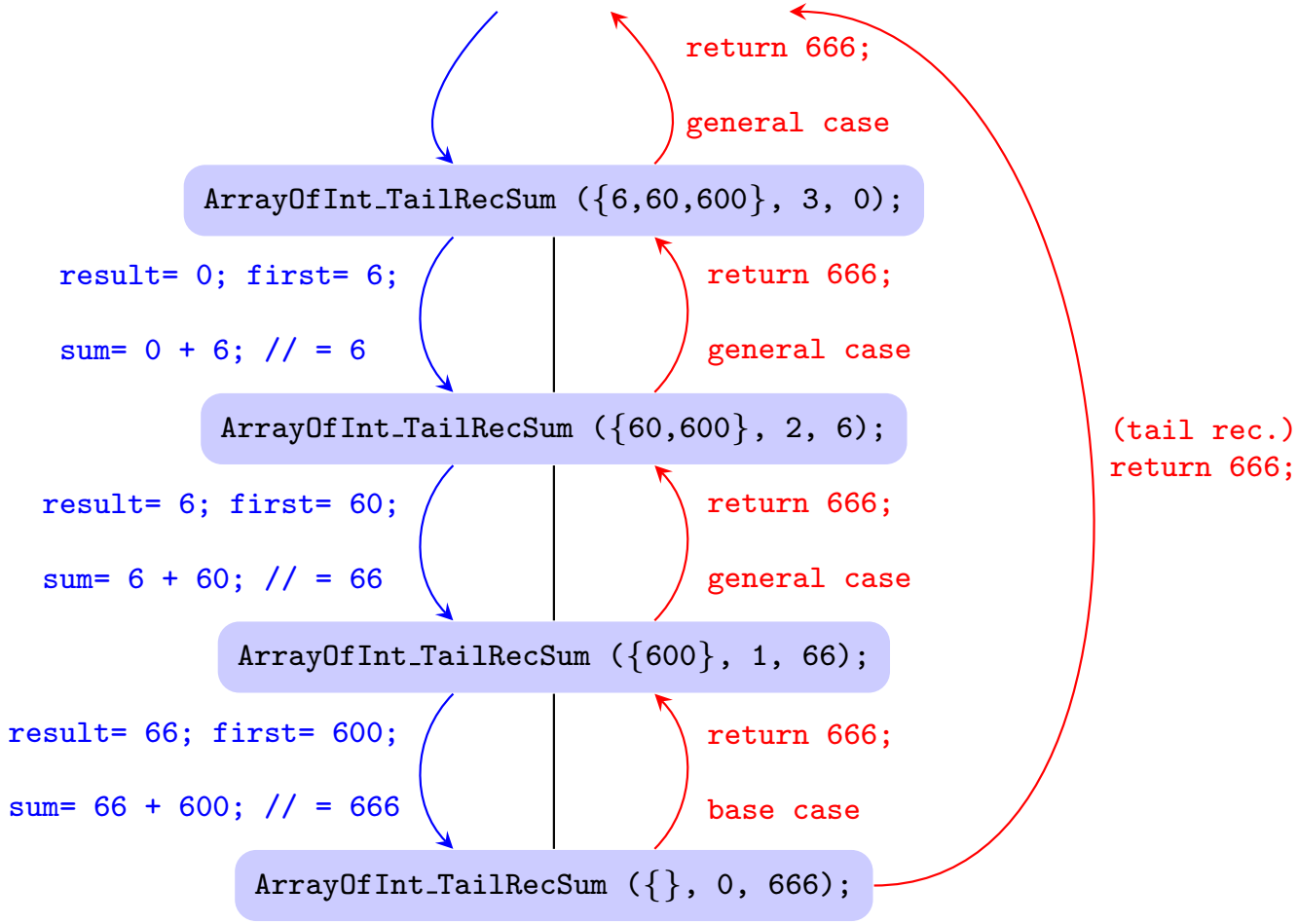
Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```



Somme réc. terminale (1/2) de la gauche vers la droite

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```



Somme réc. terminale (2/2) de la droite vers la gauche

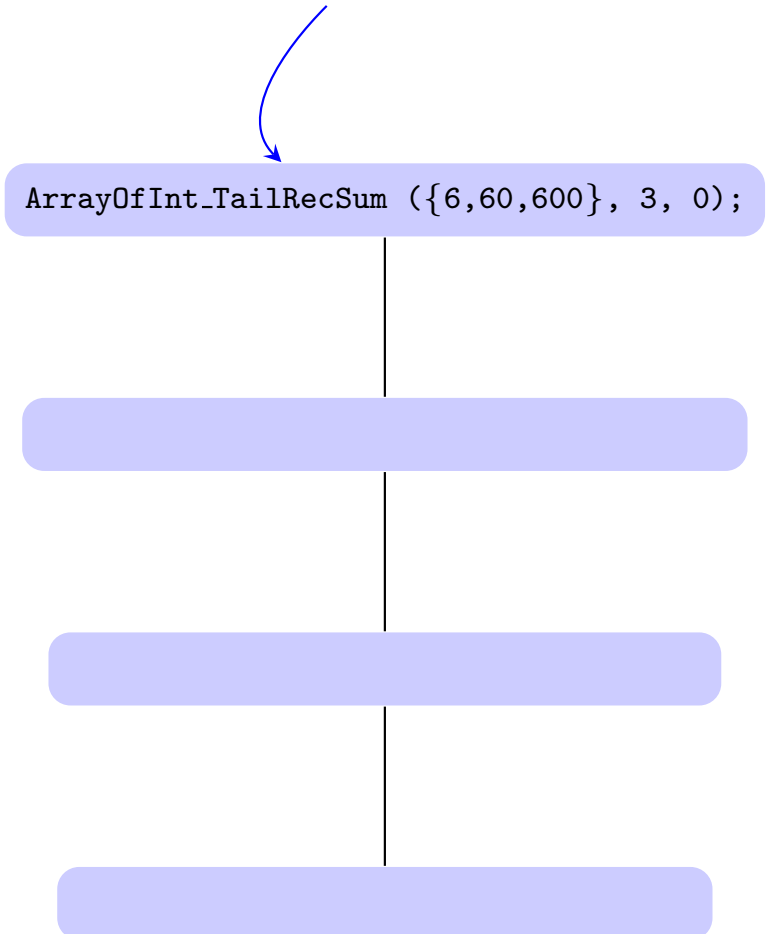
```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```

Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```

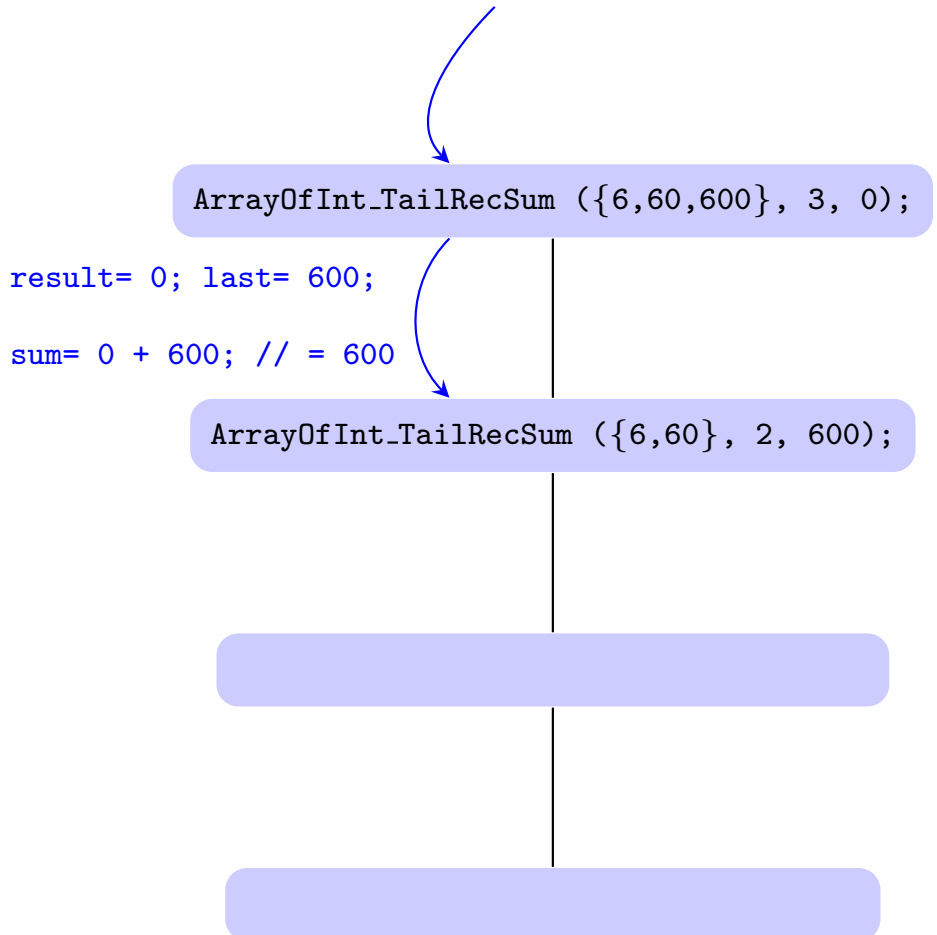
Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```



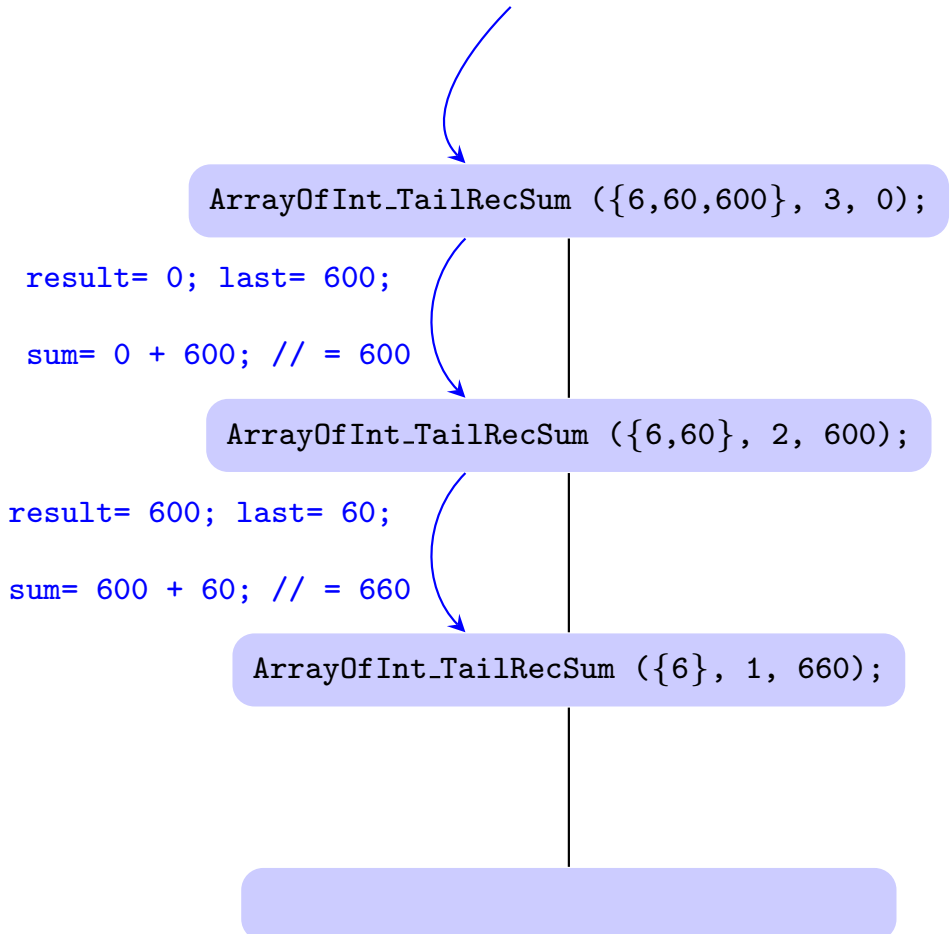
Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```



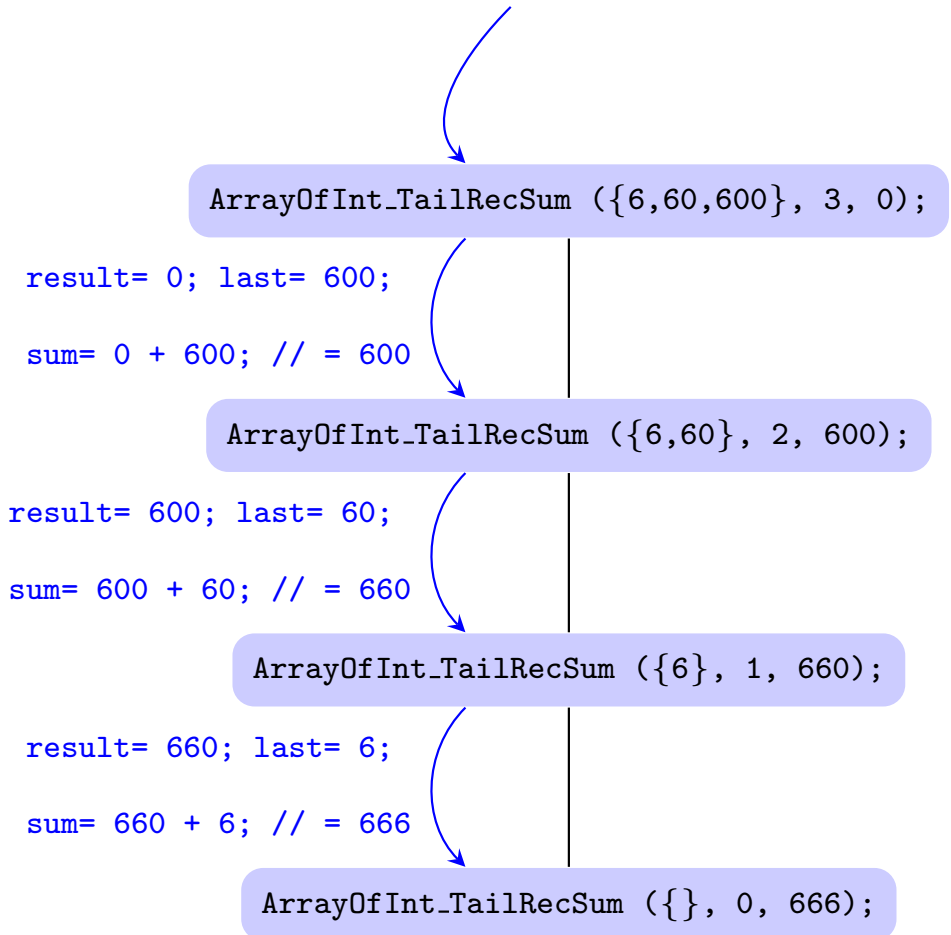
Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```



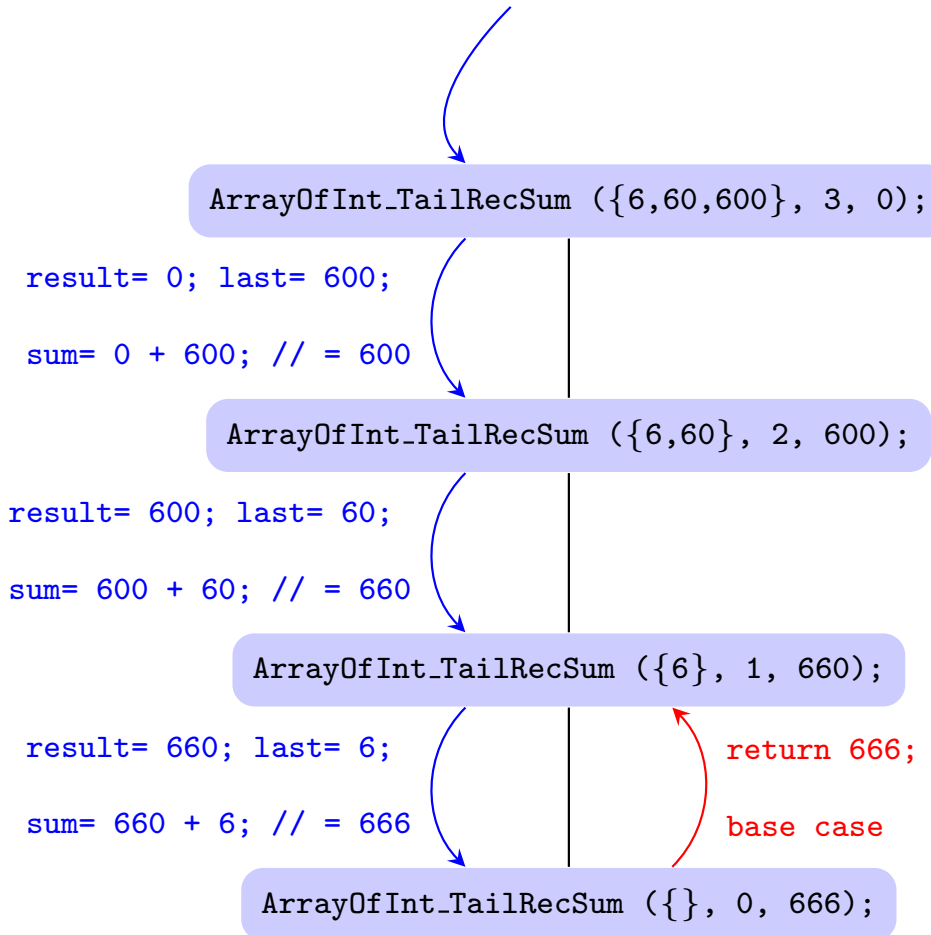
Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```



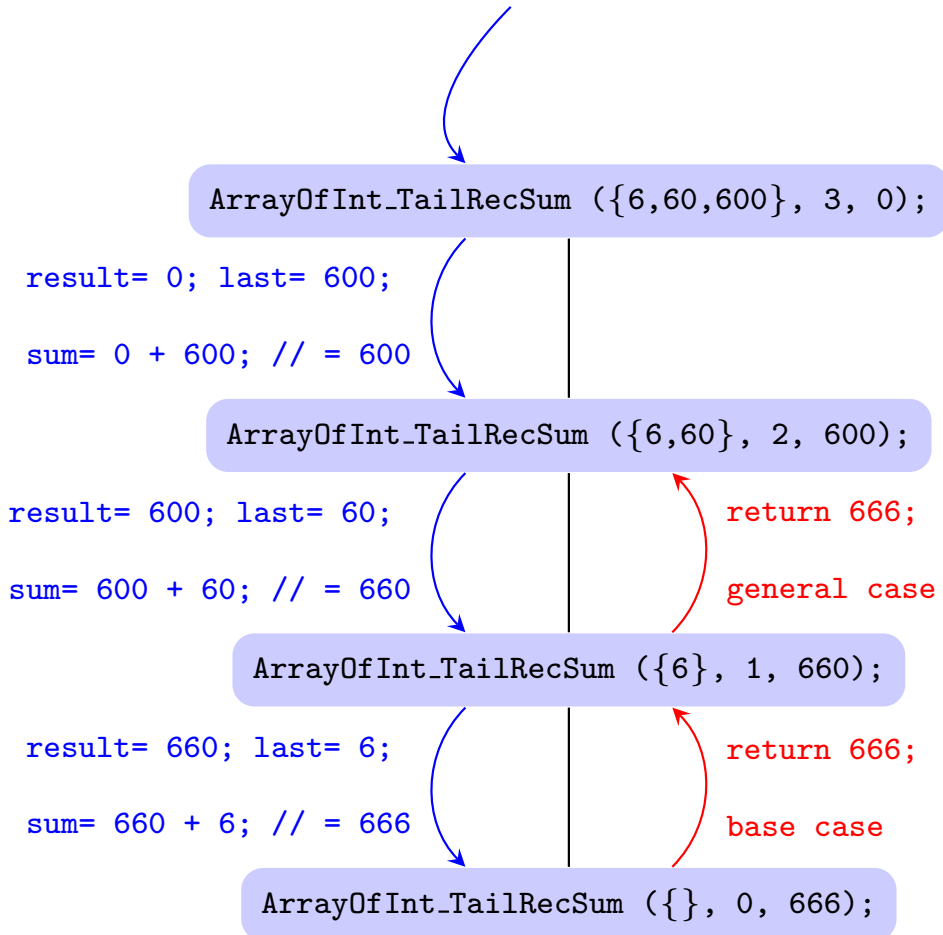
Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```



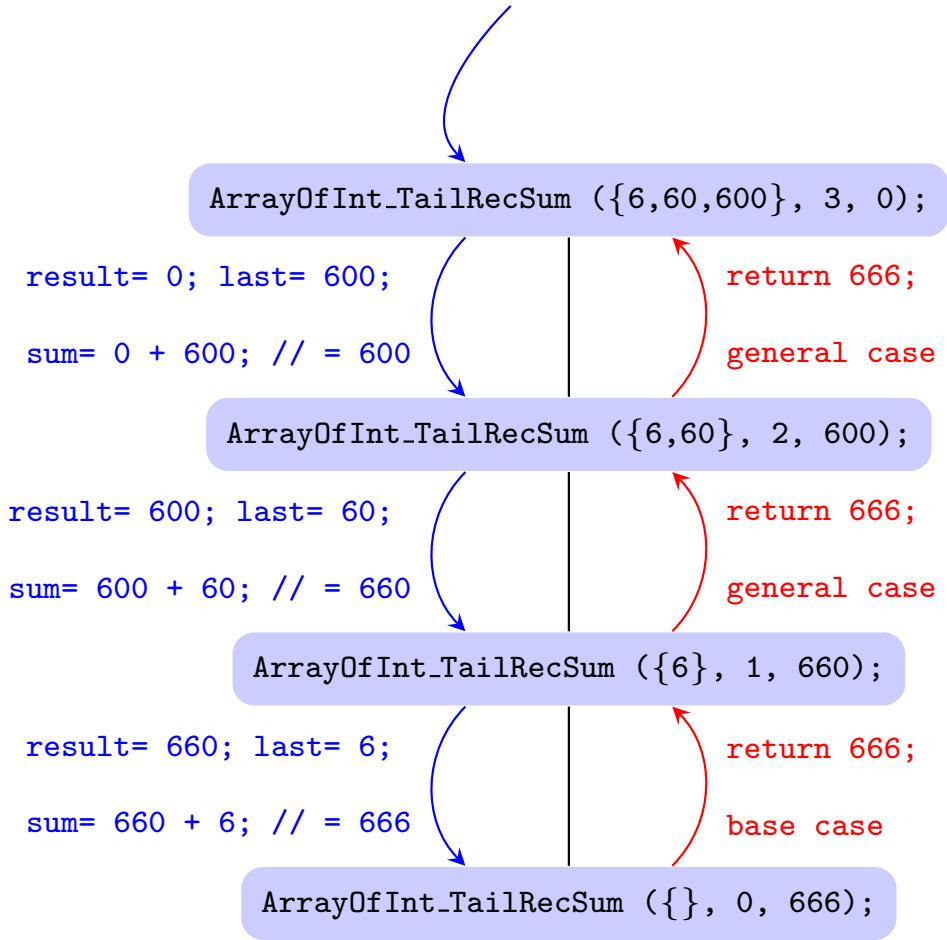
Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```



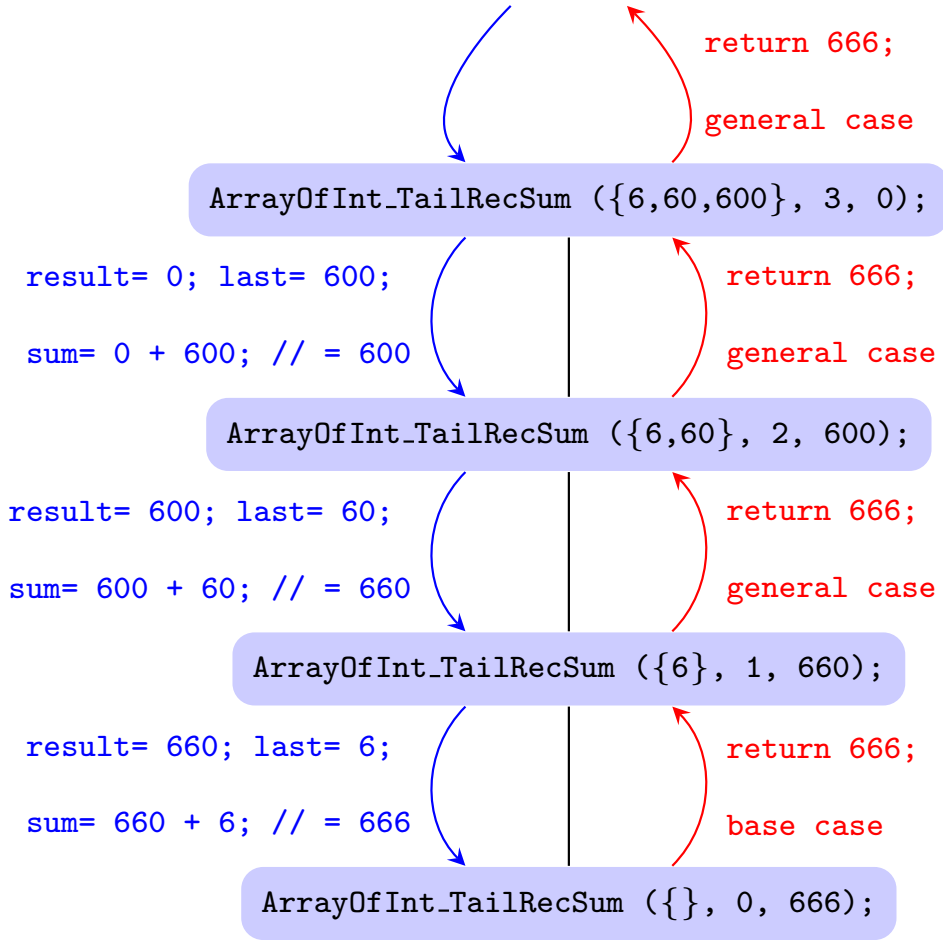
Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```



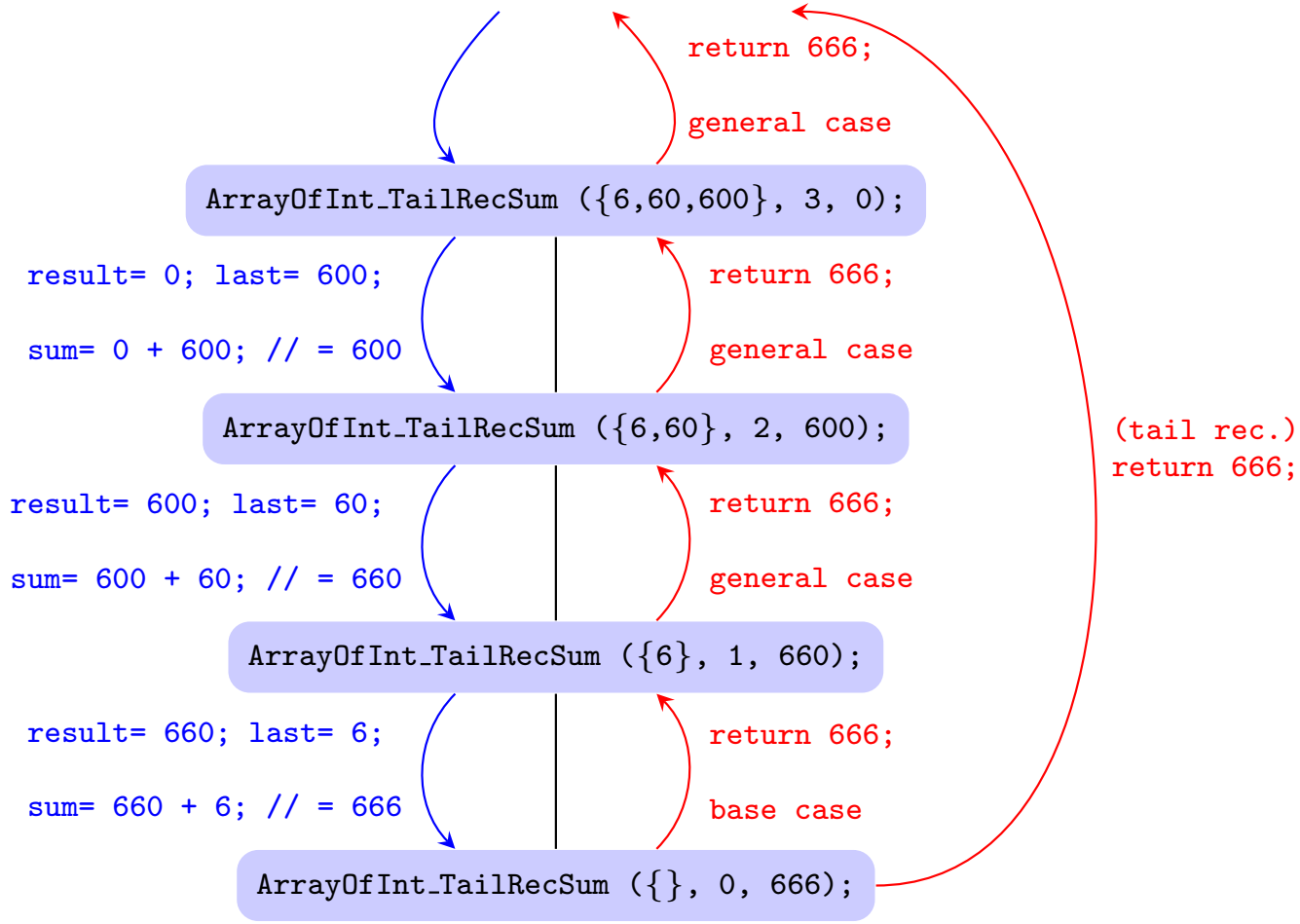
Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```



Somme réc. terminale (2/2) de la droite vers la gauche

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```



Wrapper pour la somme récursive terminale

Que la version récursive terminale de la somme additionne les termes de gauche à droite comme ci-dessous. . .

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int first= array[0];
  int sum= result + first;
  return ArrayOfInt_TailRecSum (array+1, length-1, sum); // general case
}
```

ou qu'elle les additionne de droite à gauche comme ci-dessous. . .

```
int ArrayOfInt_TailRecSum (int const array[], int length, int result) {
  if (length == 0) return result; // guard clause for base case
  int last= array[length-1];
  int sum= result + last;
  return ArrayOfInt_TailRecSum (array, length-1, sum); // general case
}
```

on peut dans les deux cas écrire le wrapper suivant pour cacher le paramètre accumulateur `result` :

```
int ArrayOfInt_Sum (int const array [], int length)
{
  return ArrayOfInt_TailRecSum (array, length, 0);
}
```

Index du max récursif (1/2), de la gauche vers la droite

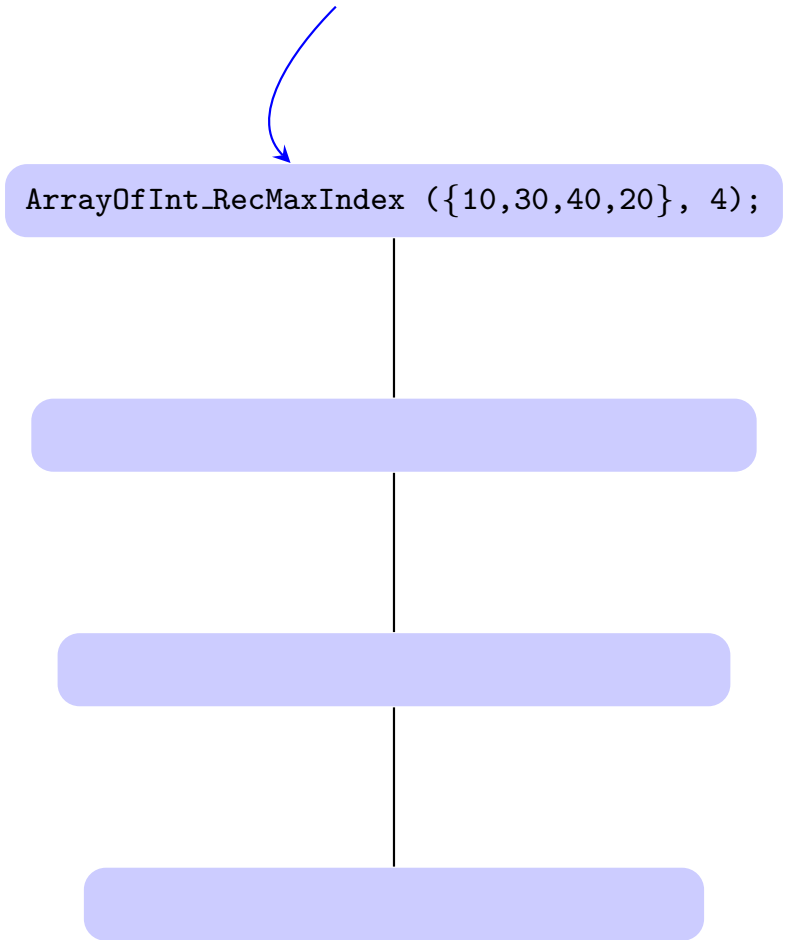
```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```

Index du max récursif (1/2), de la gauche vers la droite

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```

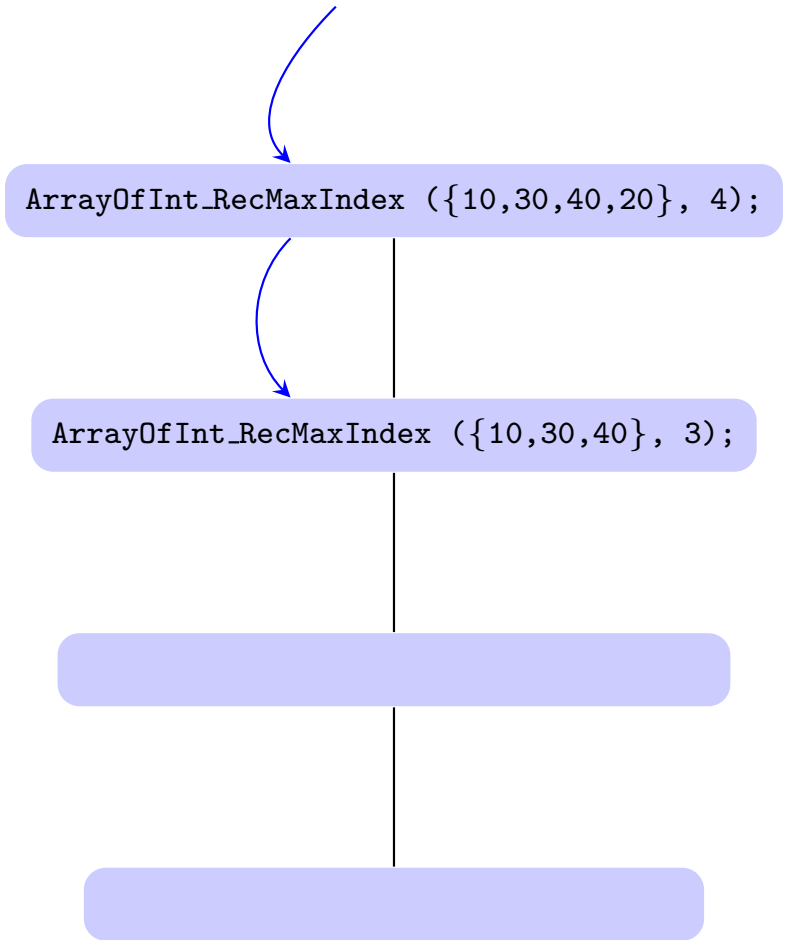
Index du max récursif (1/2), de la gauche vers la droite

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```



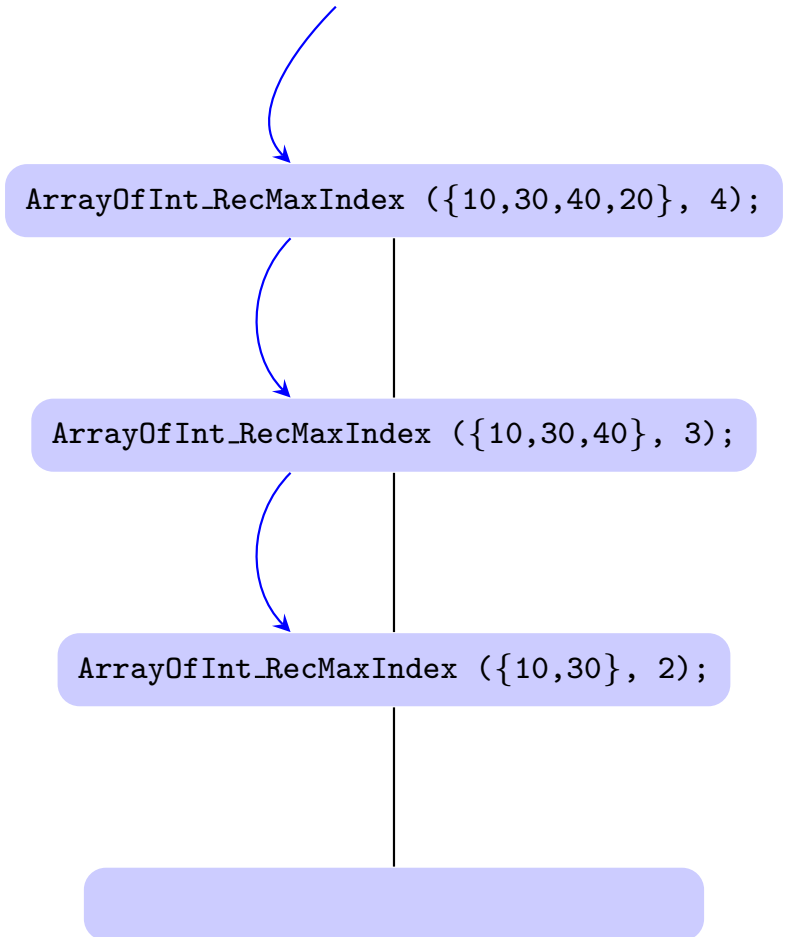
Index du max récursif (1/2), de la gauche vers la droite

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```



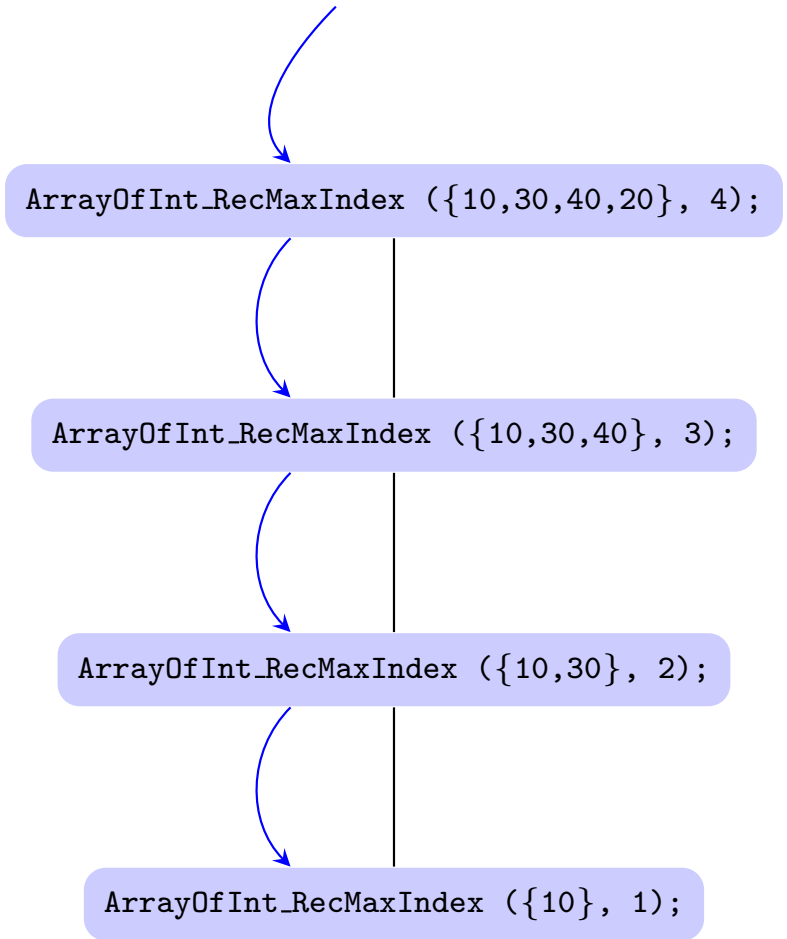
Index du max récursif (1/2), de la gauche vers la droite

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```



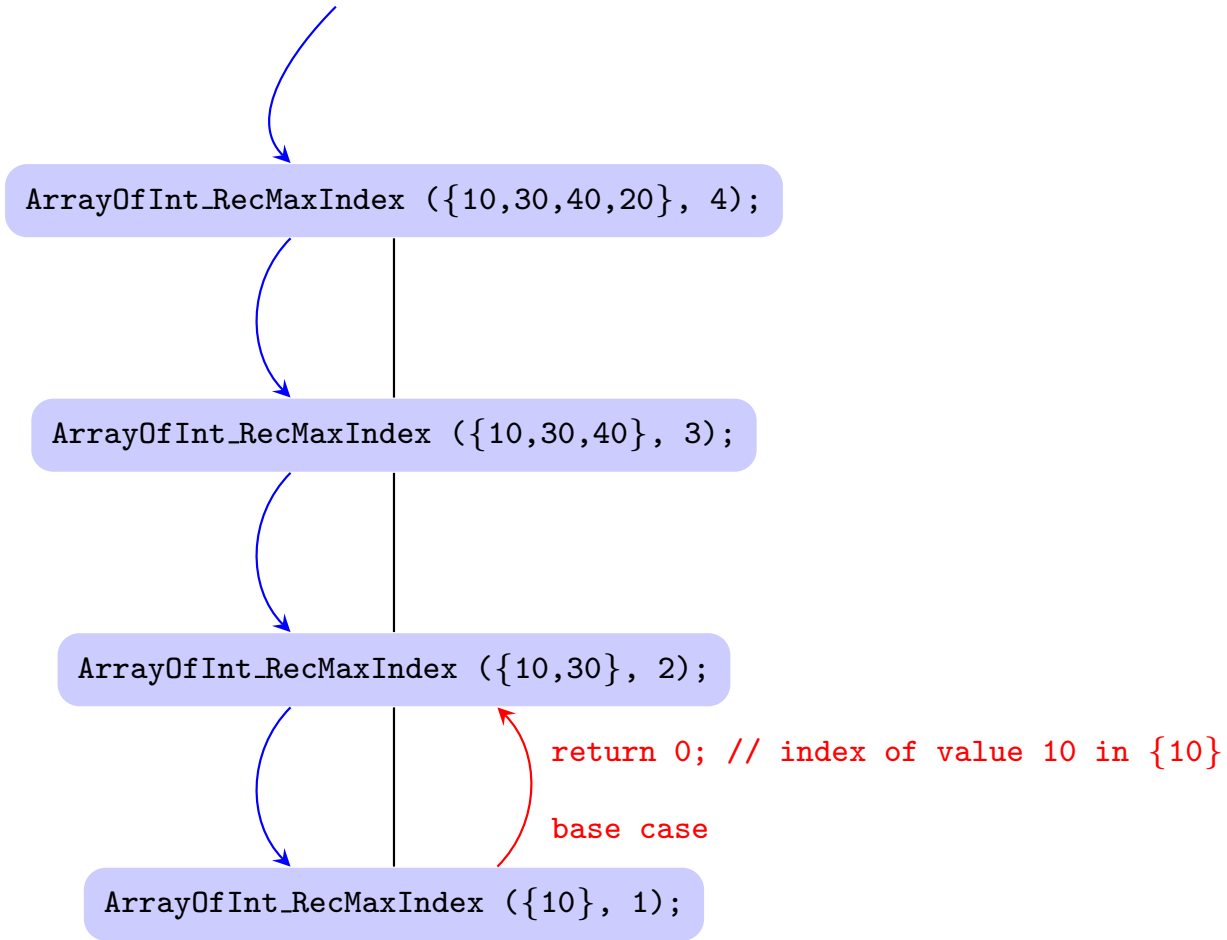
Index du max récursif (1/2), de la gauche vers la droite

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```



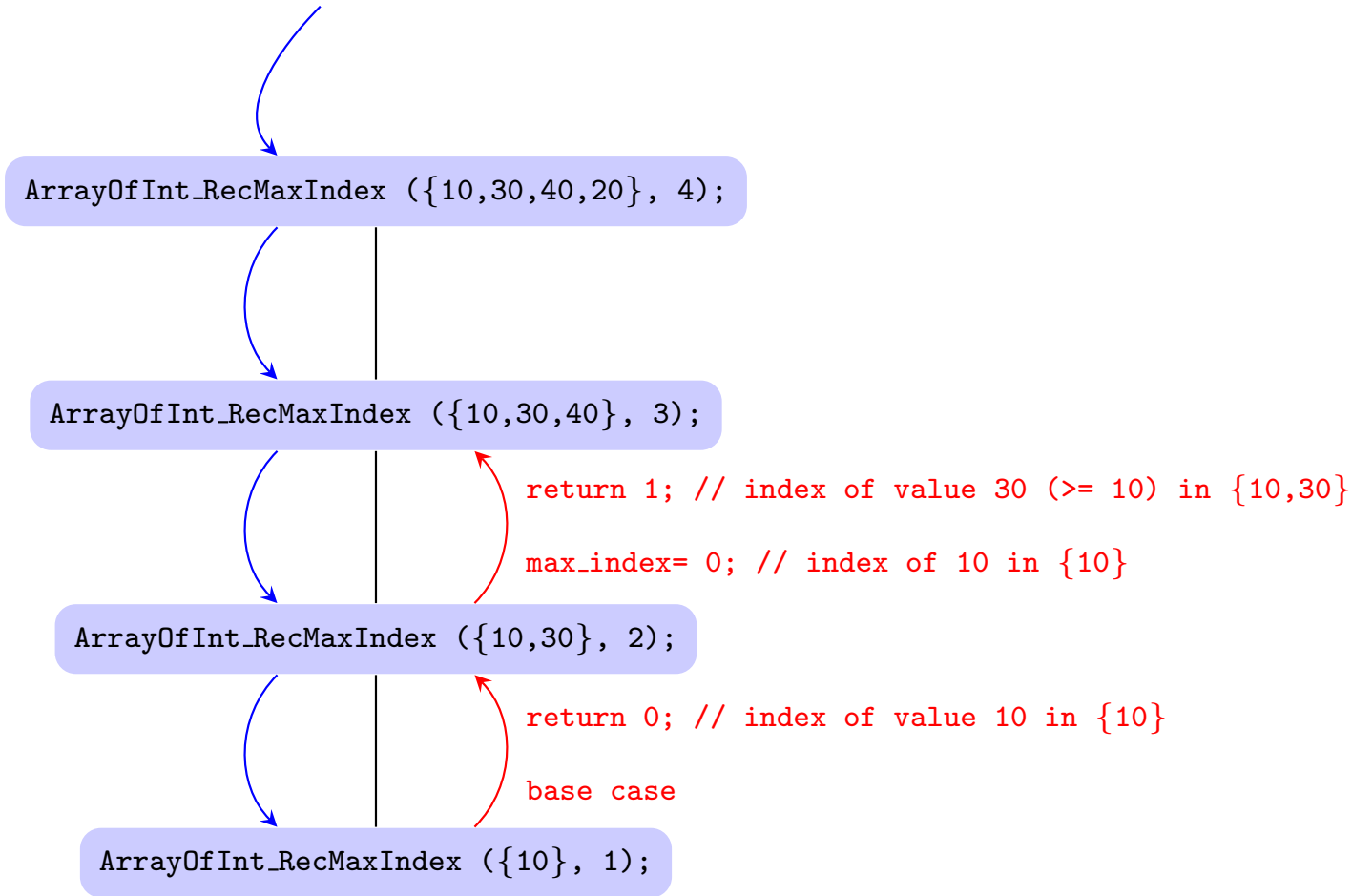
Index du max récursif (1/2), de la gauche vers la droite

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```



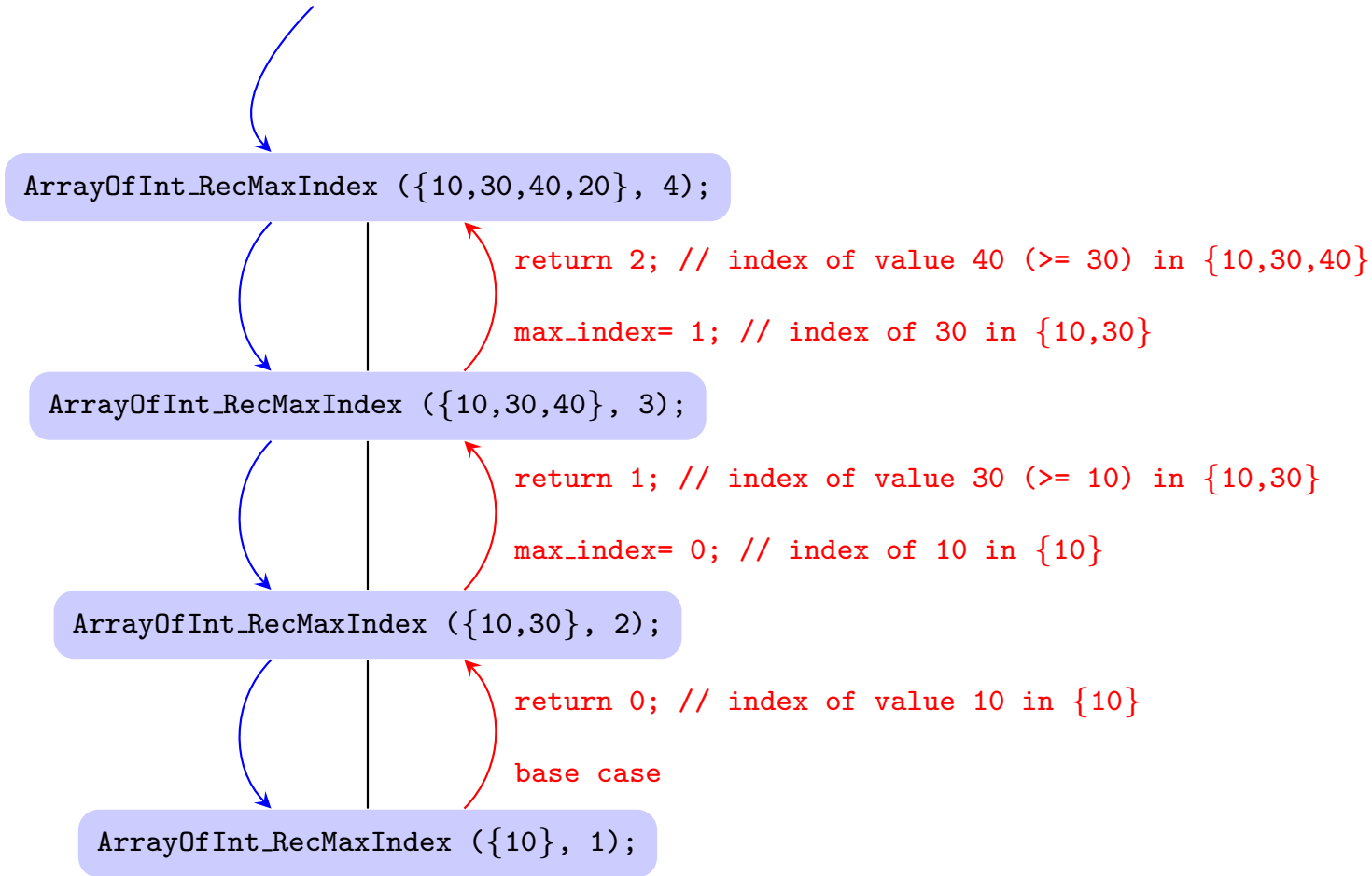
Index du max récursif (1/2), de la gauche vers la droite

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```



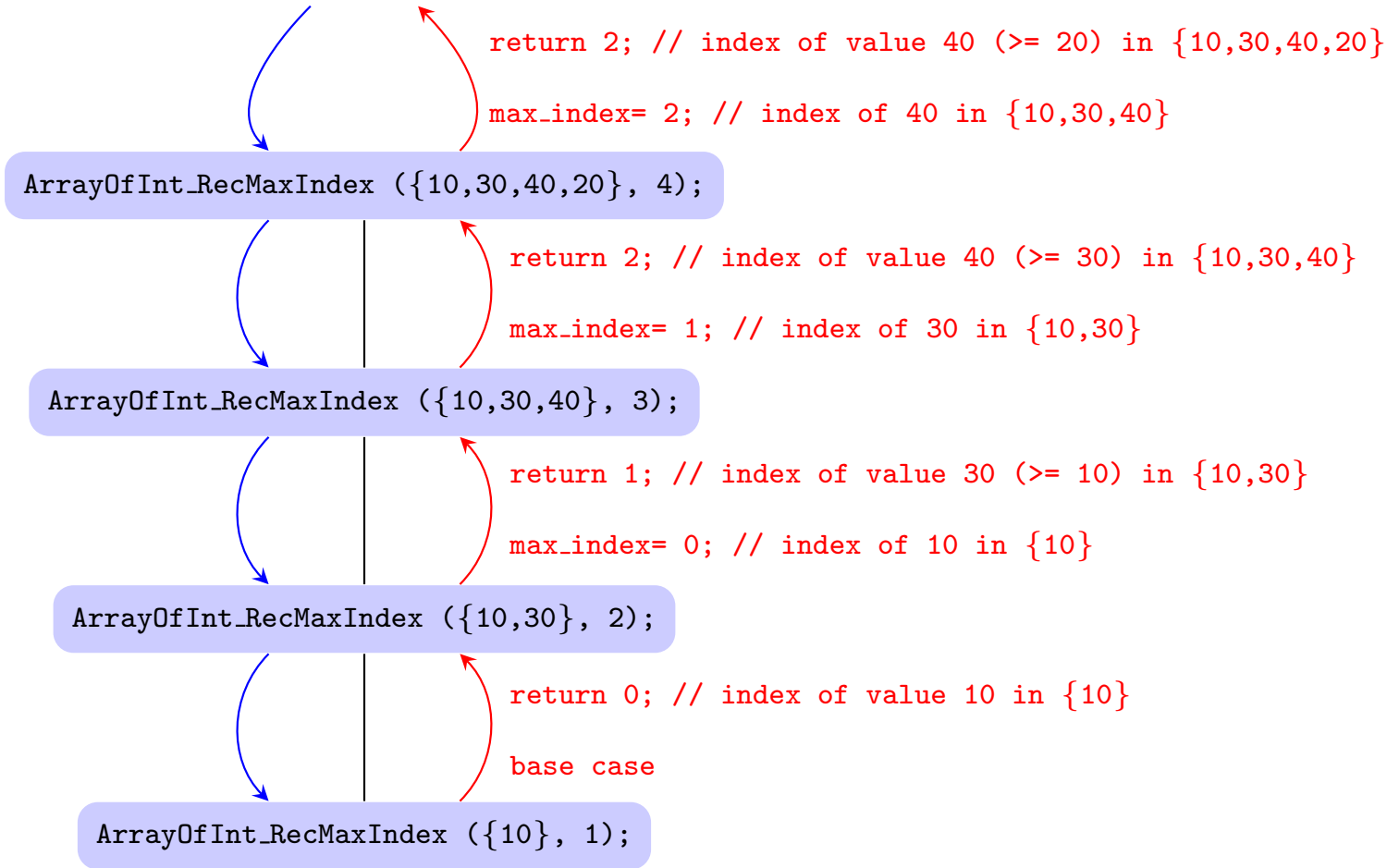
Index du max récursif (1/2), de la gauche vers la droite

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```



Index du max récursif (1/2), de la gauche vers la droite

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0; // guard clause for base case
  int max_index= ArrayOfInt_RecMaxIndex (array, length-1);
  return (array [max_index] >= array [length-1]) ? max_index : length-1;
}
```



Index du max récursif (2/2), de la droite vers la gauche

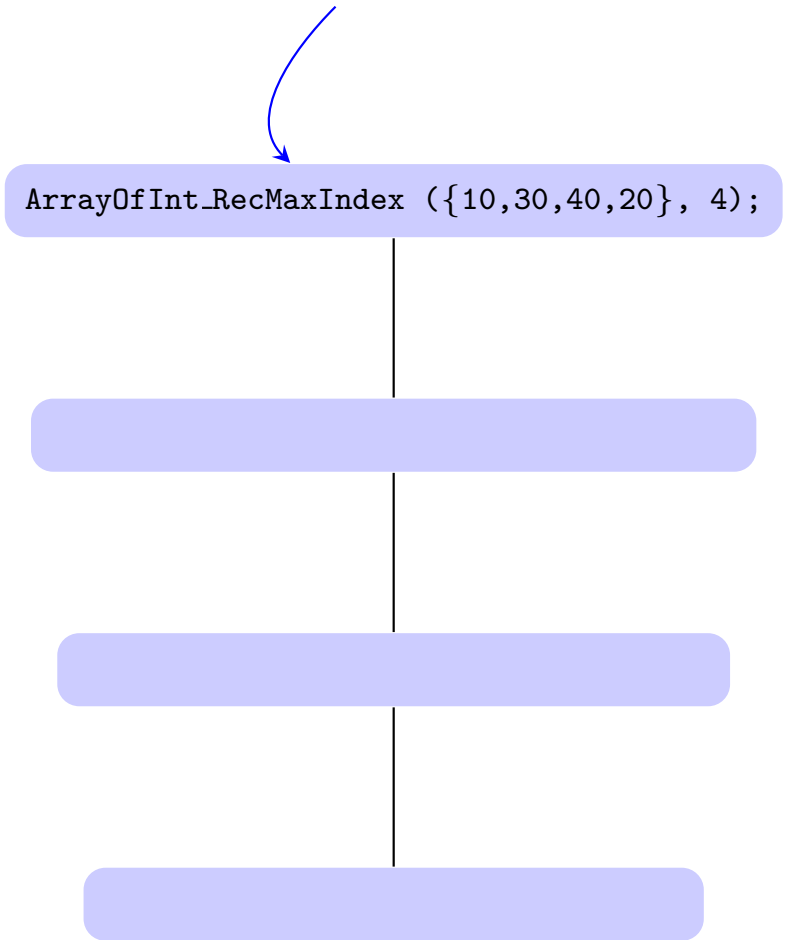
```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```

Index du max récursif (2/2), de la droite vers la gauche

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```

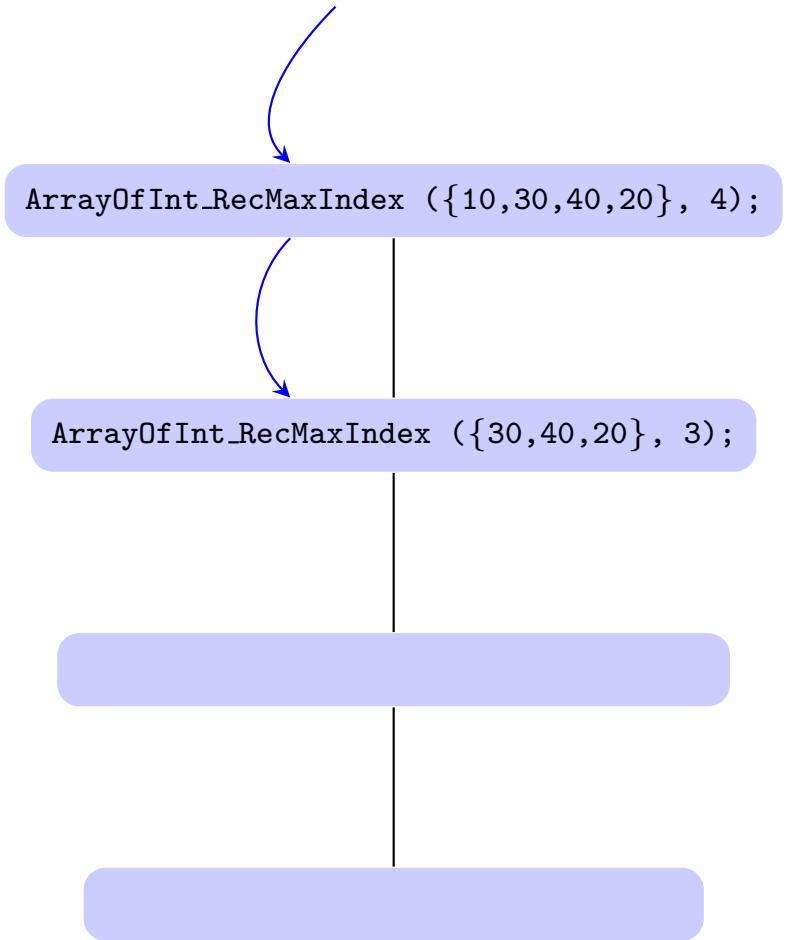
Index du max récursif (2/2), de la droite vers la gauche

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```



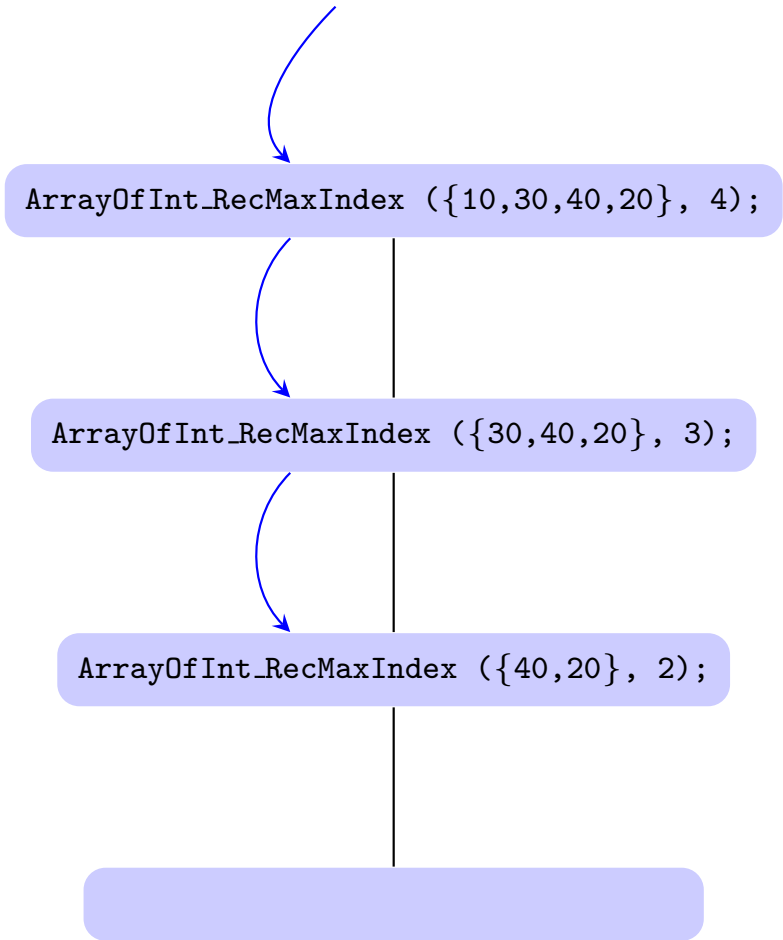
Index du max récursif (2/2), de la droite vers la gauche

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```



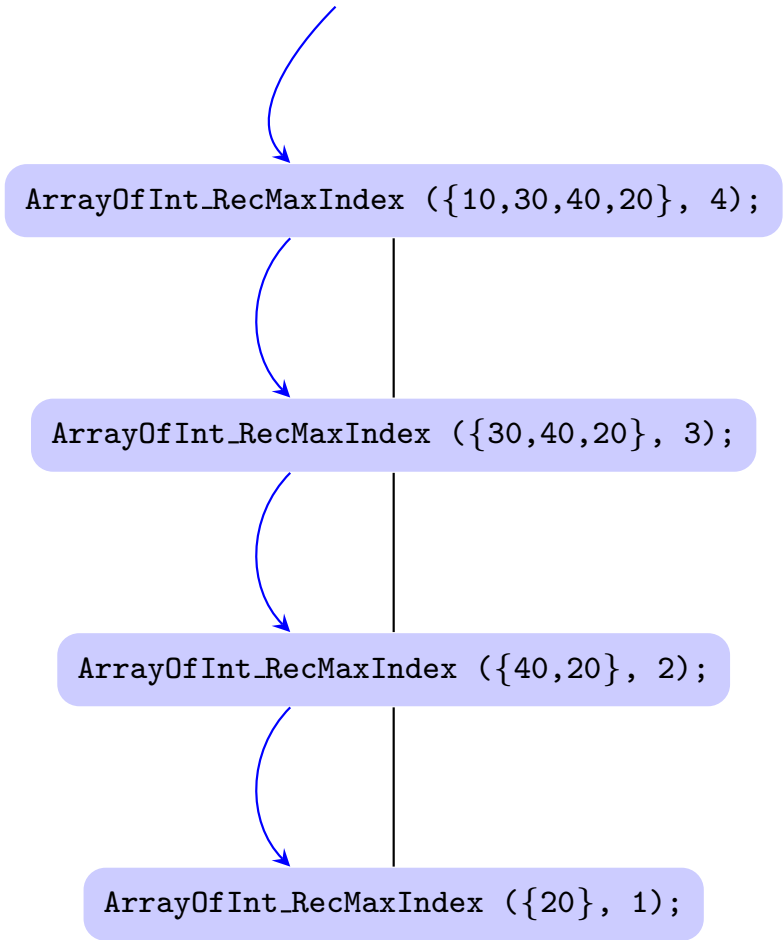
Index du max récursif (2/2), de la droite vers la gauche

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```



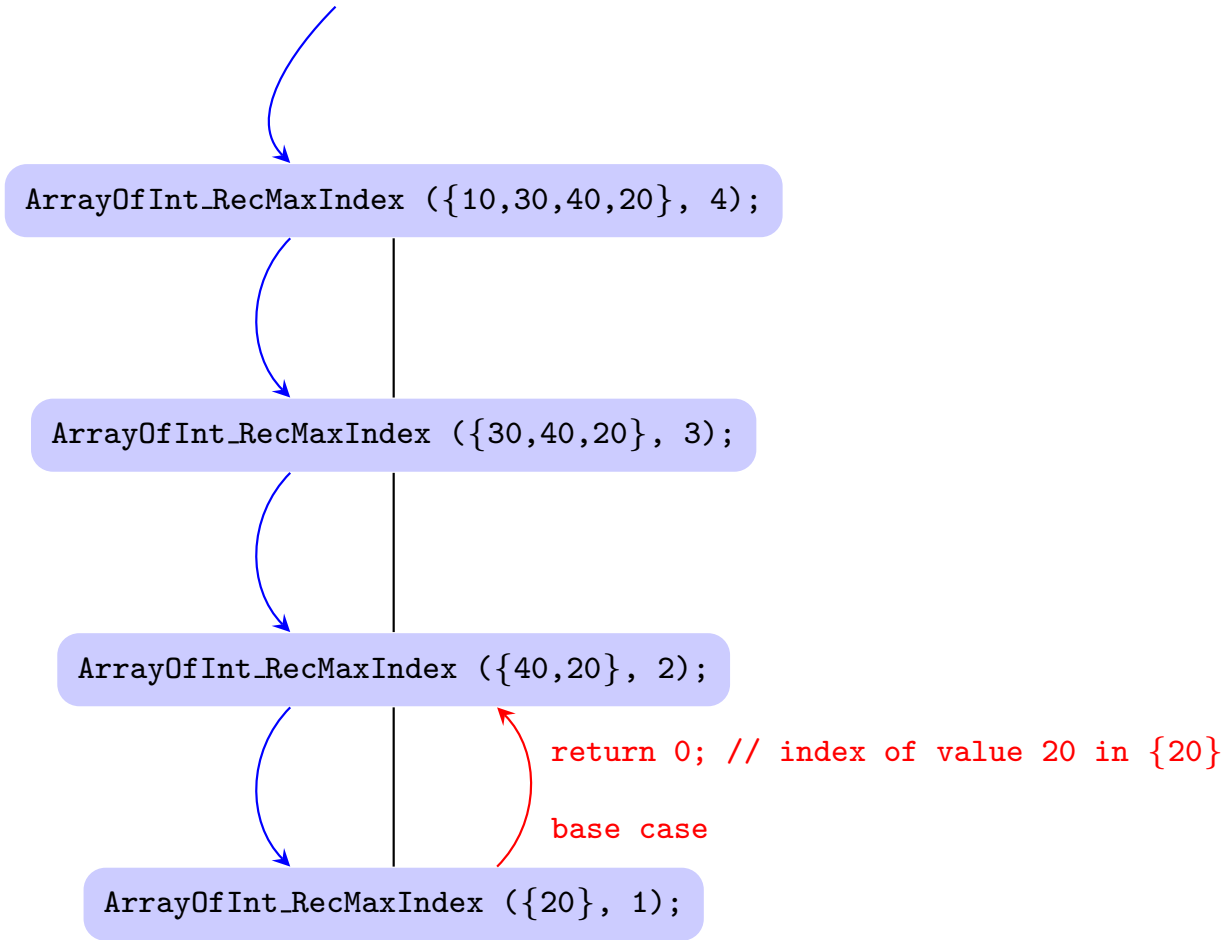
Index du max récursif (2/2), de la droite vers la gauche

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```



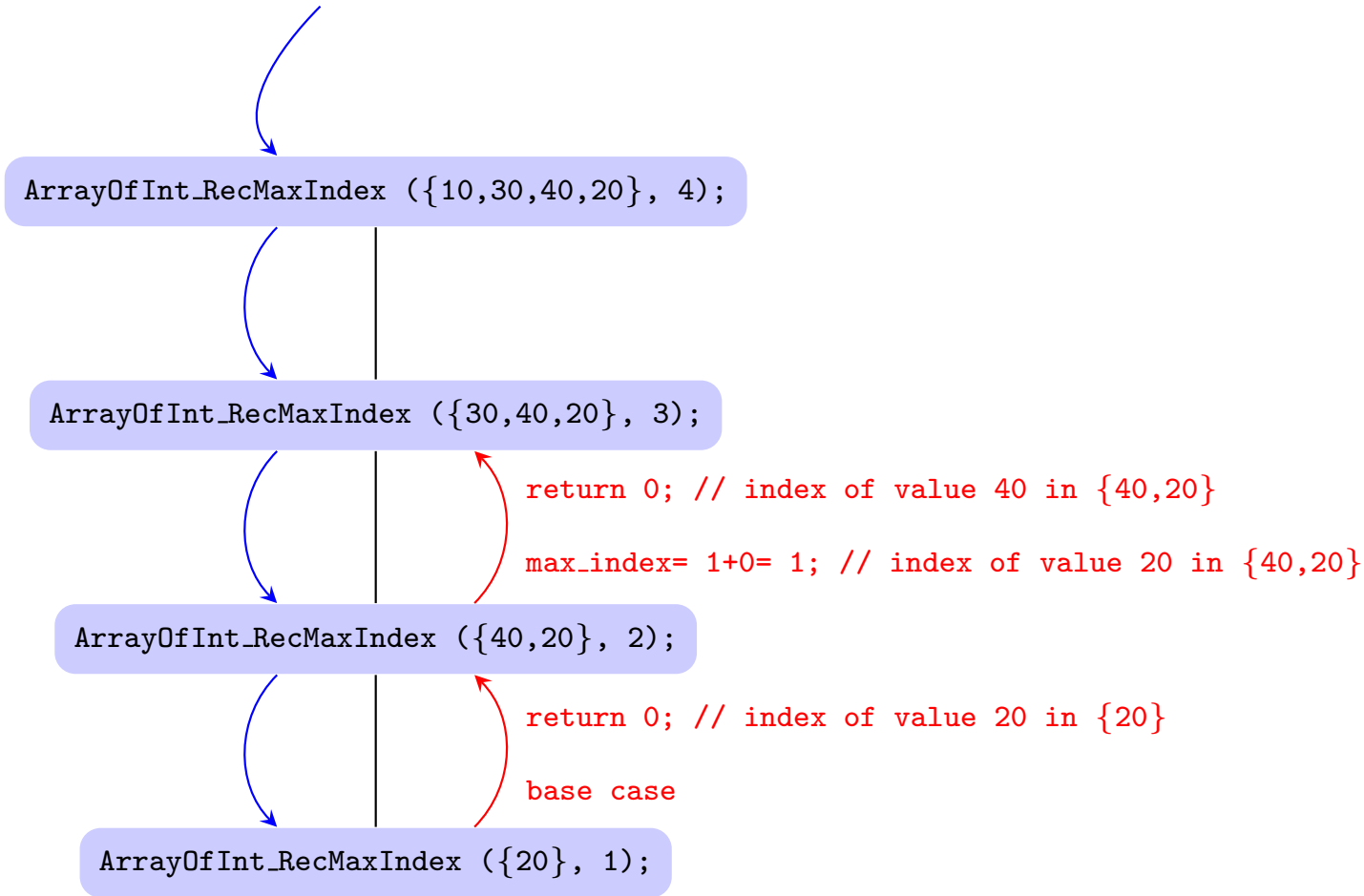
Index du max récursif (2/2), de la droite vers la gauche

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```



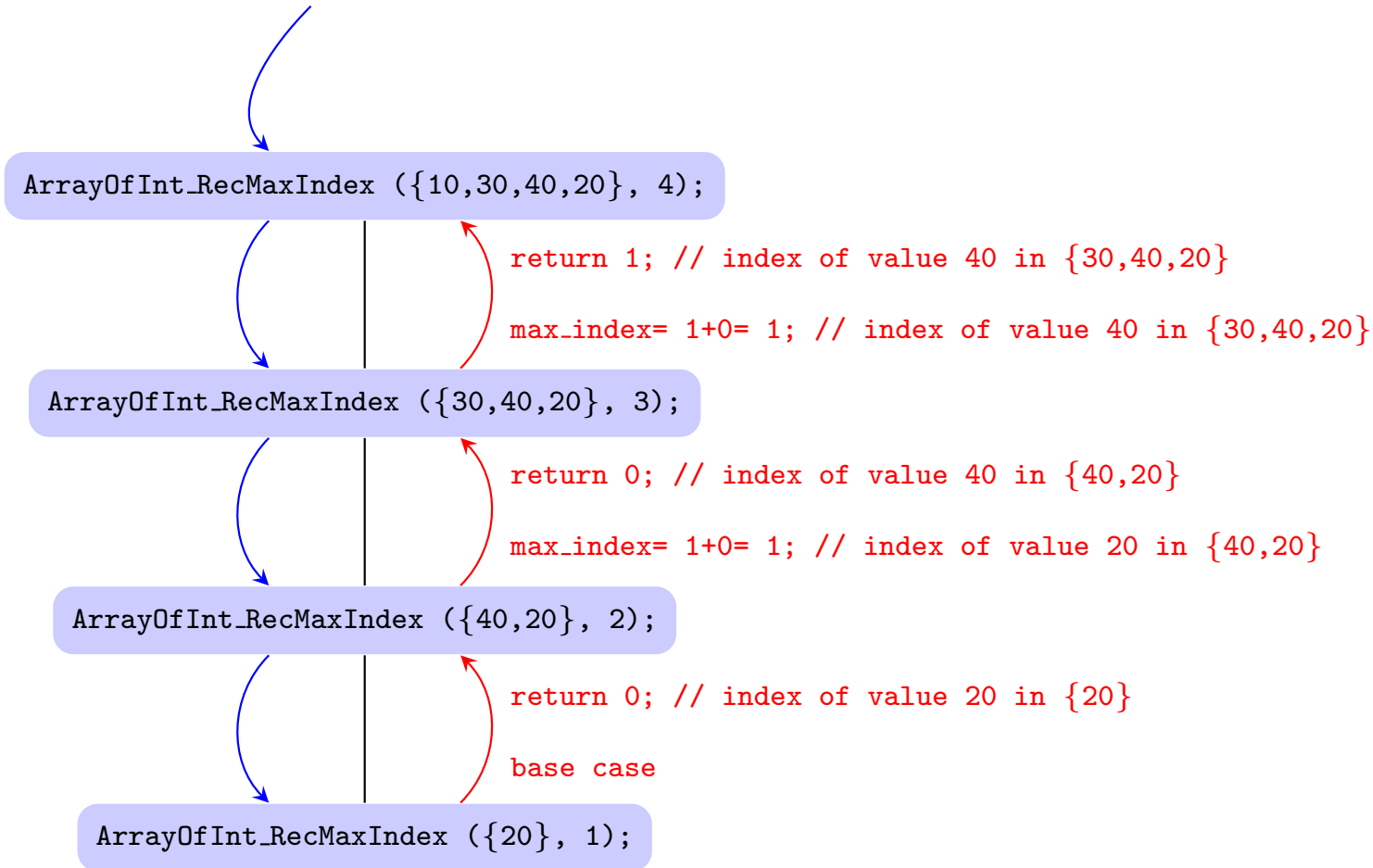
Index du max récursif (2/2), de la droite vers la gauche

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```



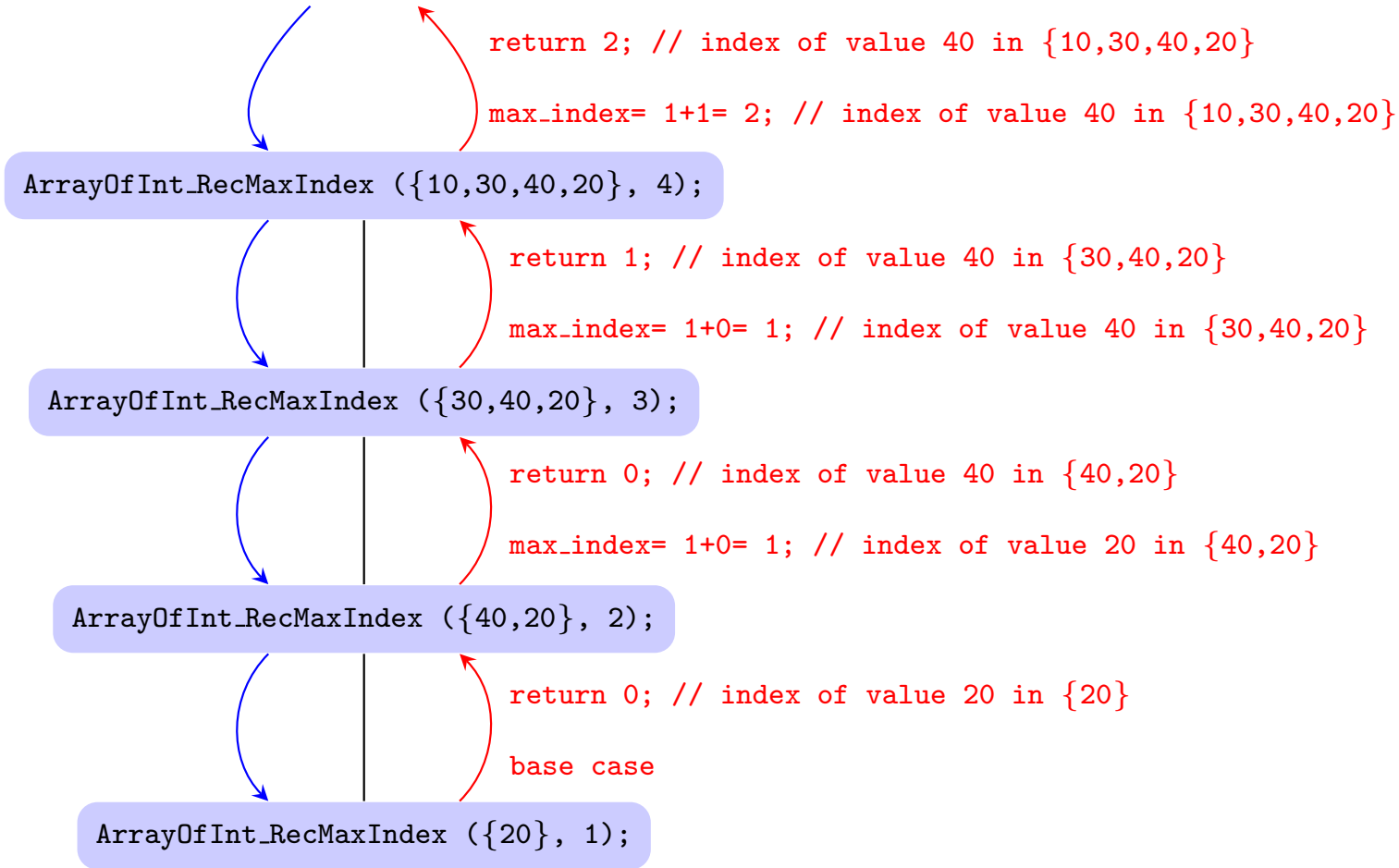
Index du max récursif (2/2), de la droite vers la gauche

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```



Index du max récursif (2/2), de la droite vers la gauche

```
int ArrayOfInt_RecMaxIndex (int const array [], int length) {
  if (length == 1) return 0;
  int max_index= 1 + ArrayOfInt_MaxIndex (array+1, length-1);
  return (array [max_index] >= array [0]) ? max_index : 0;
}
```



Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```

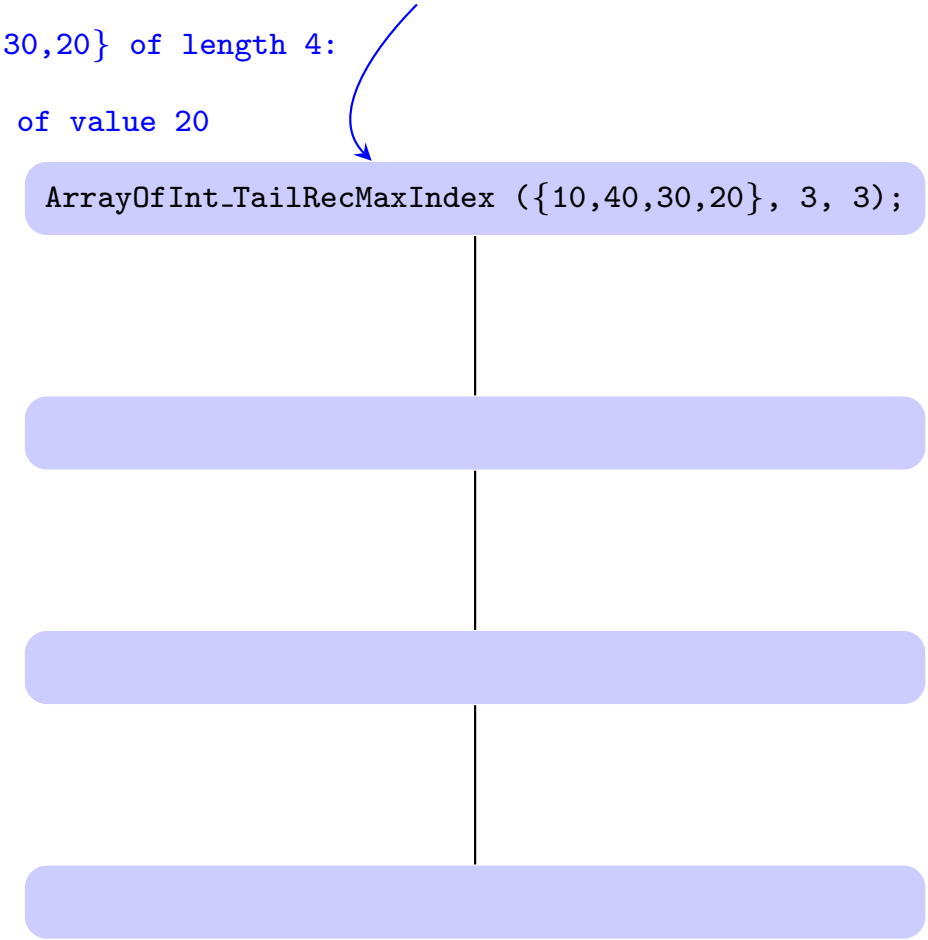
Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```

Index du max réc. terminal (1/1)

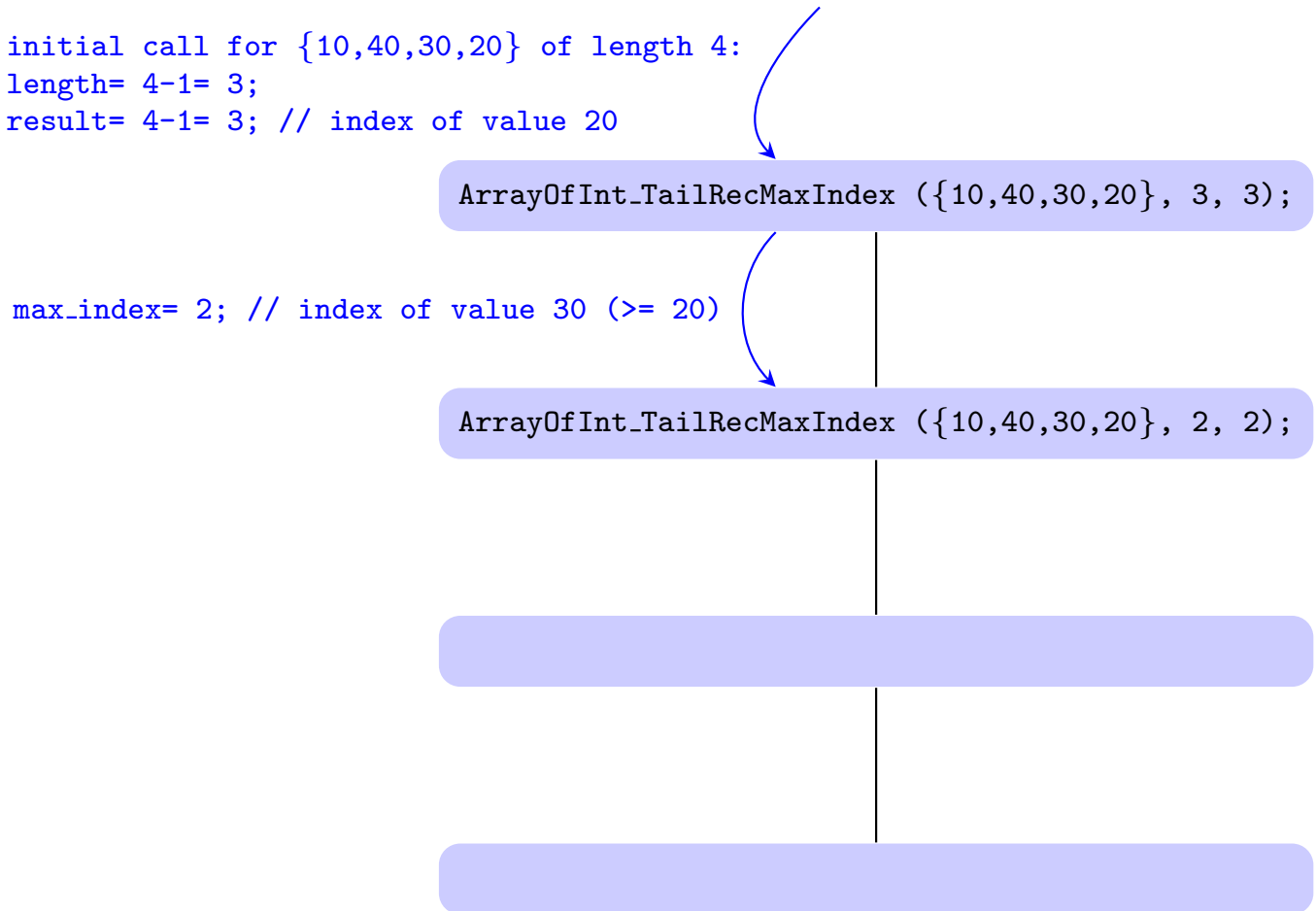
```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```

initial call for {10,40,30,20} of length 4:
length= 4-1= 3;
result= 4-1= 3; // index of value 20



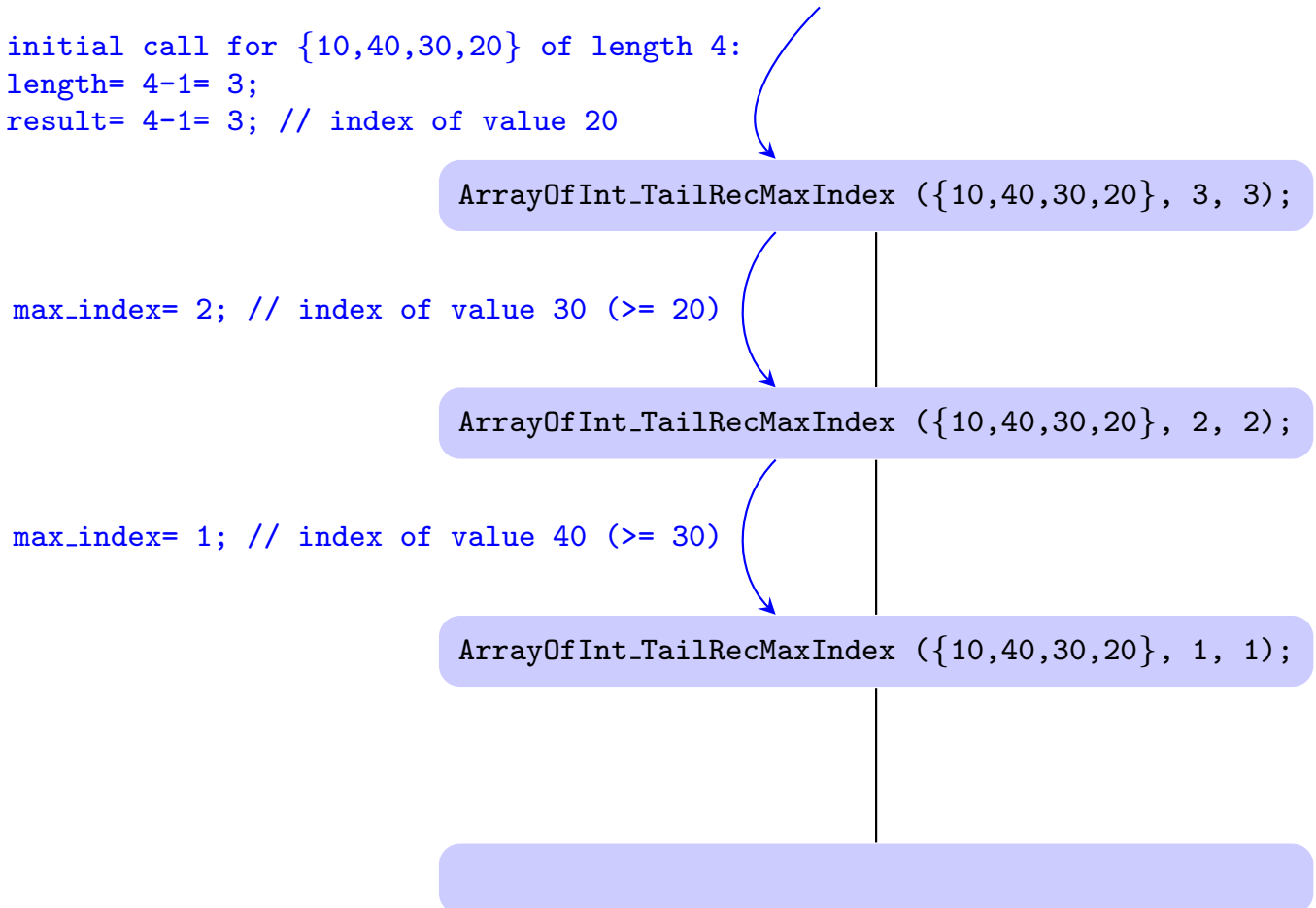
Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```



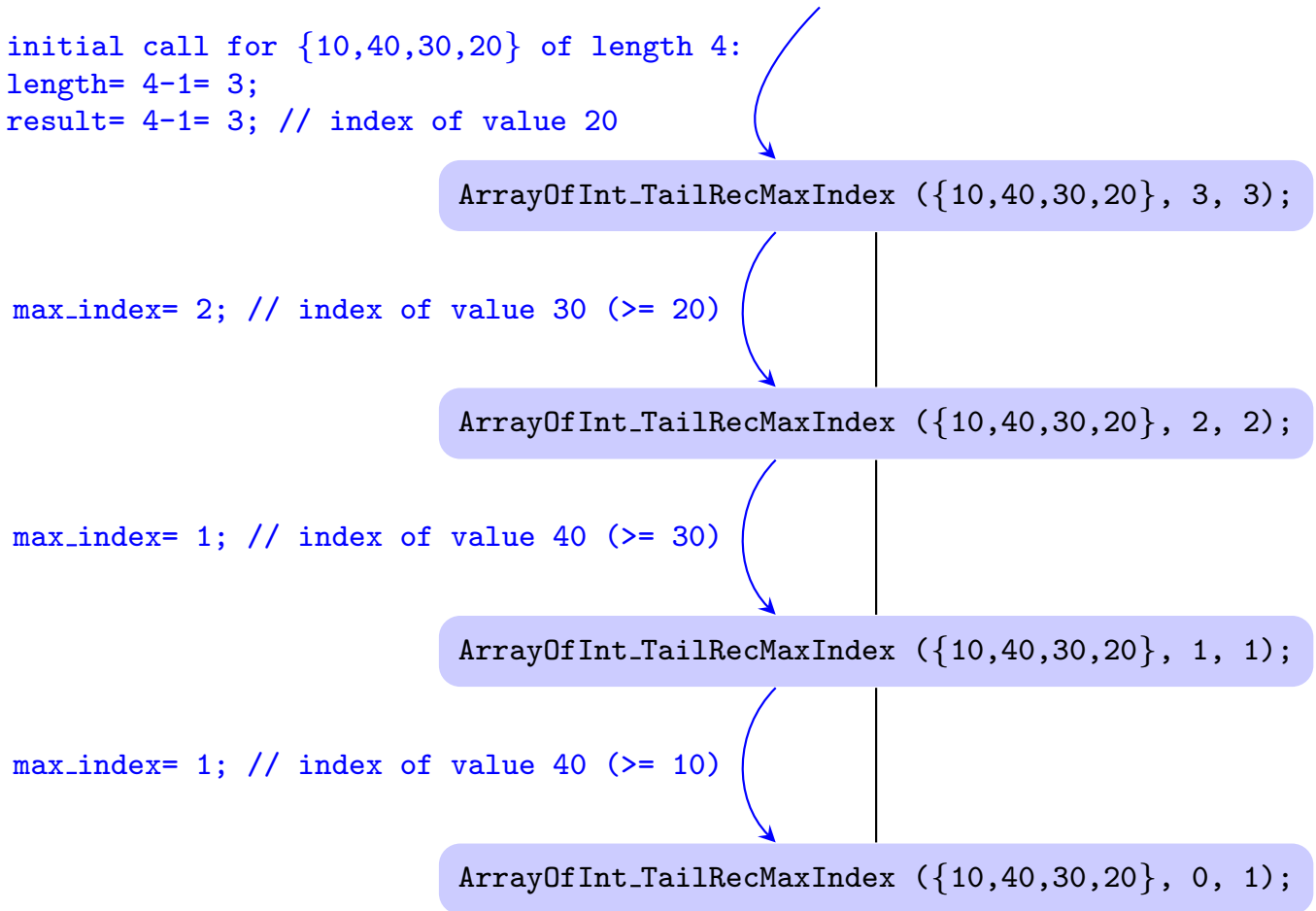
Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```



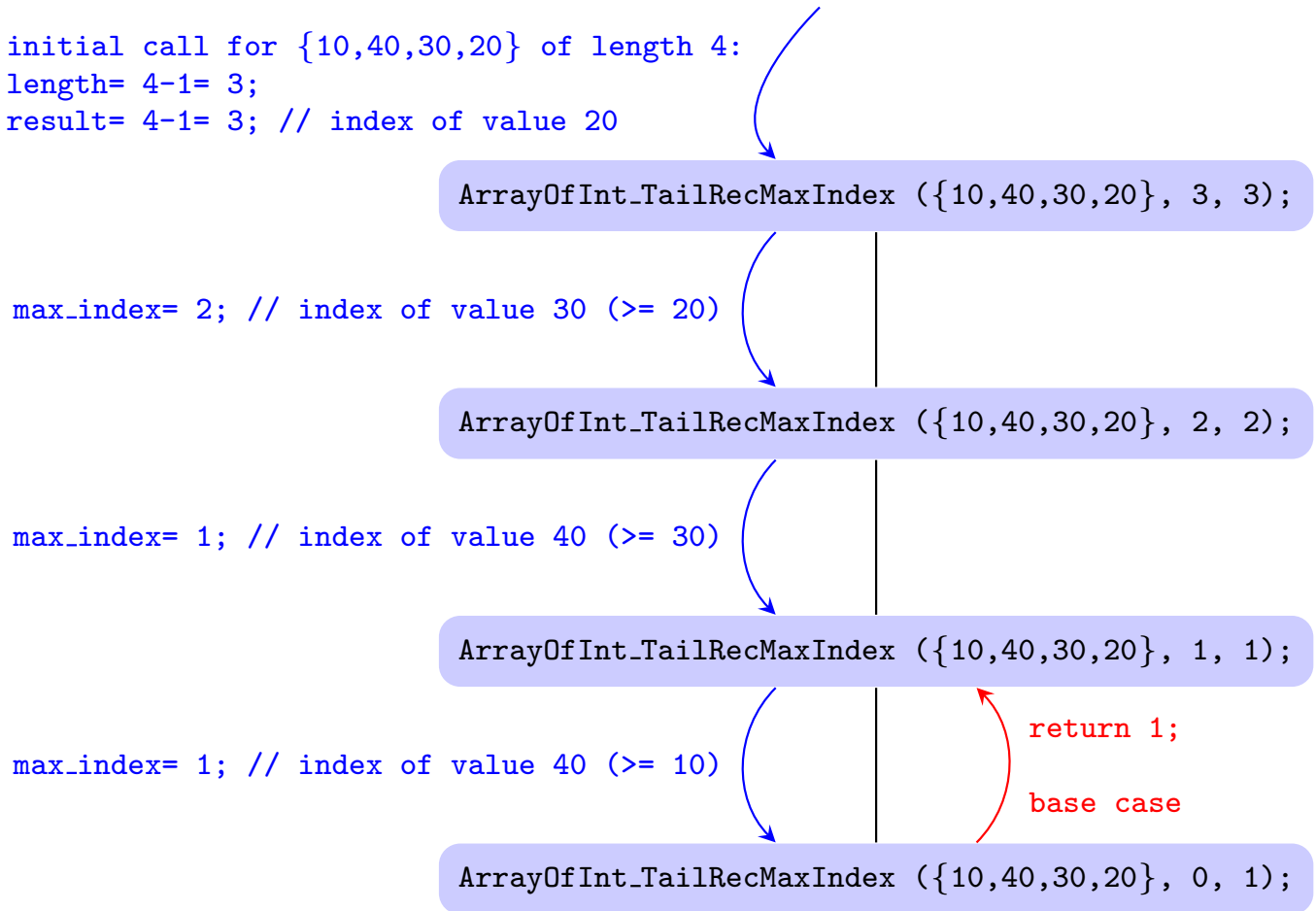
Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```



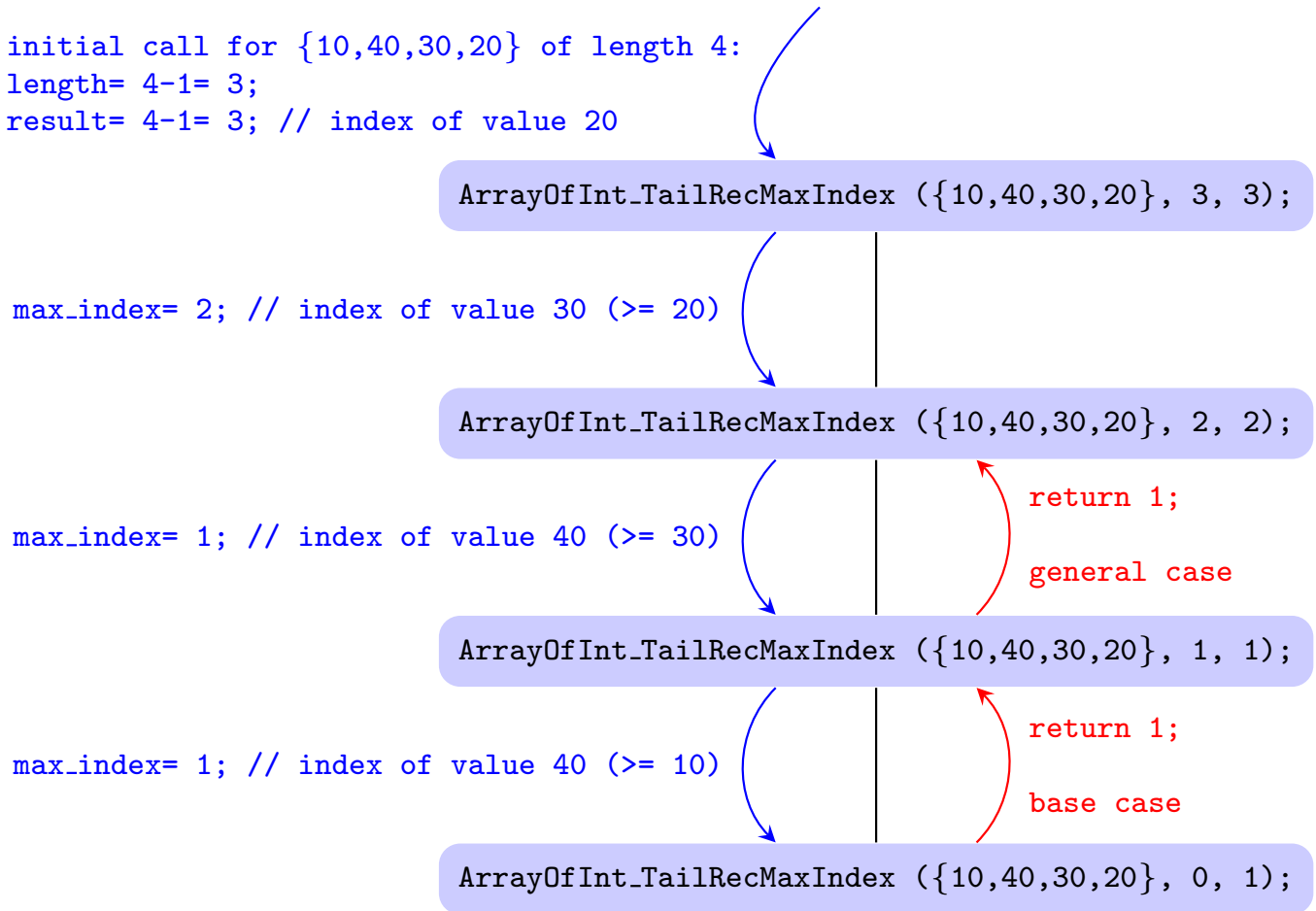
Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```



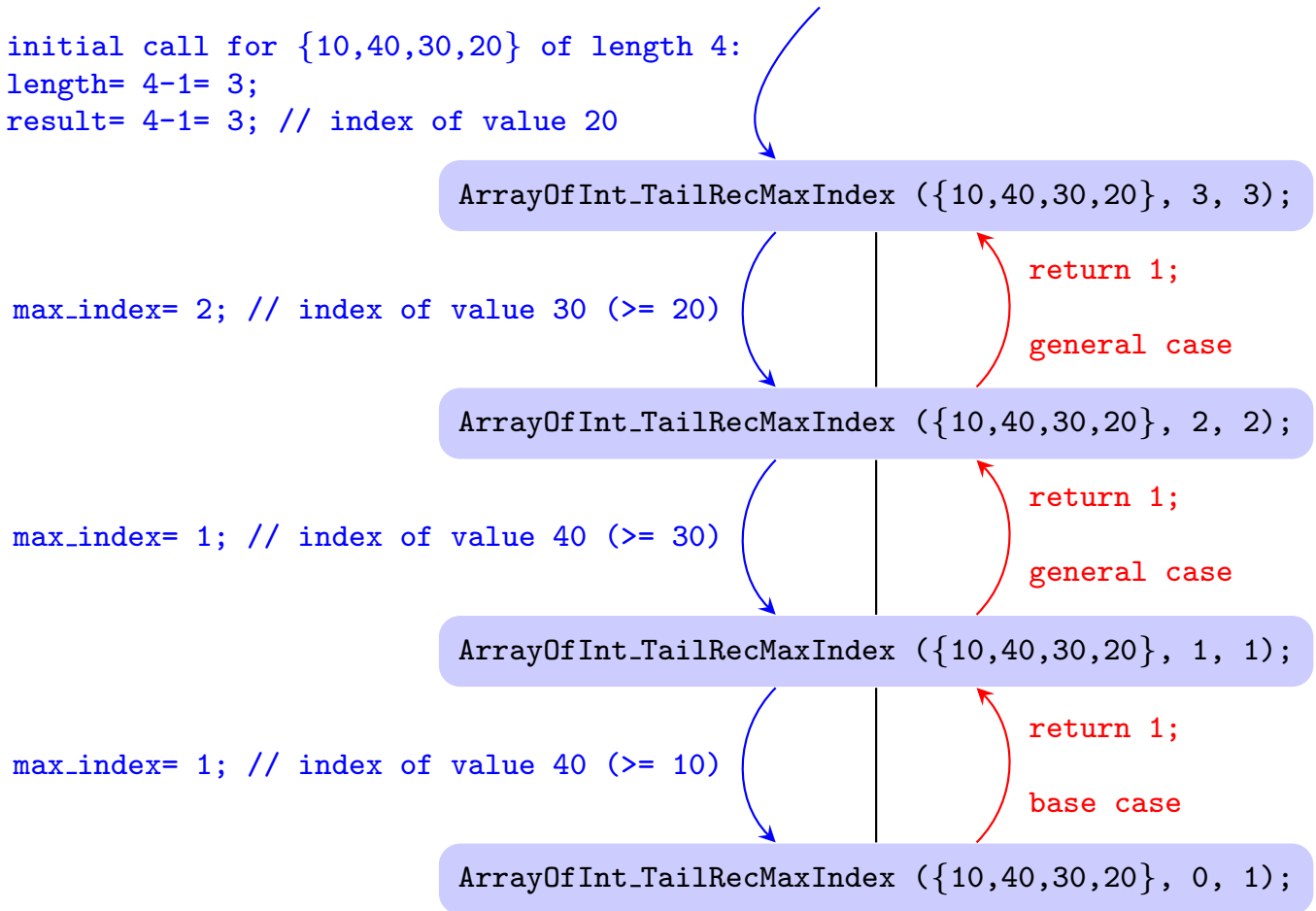
Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```



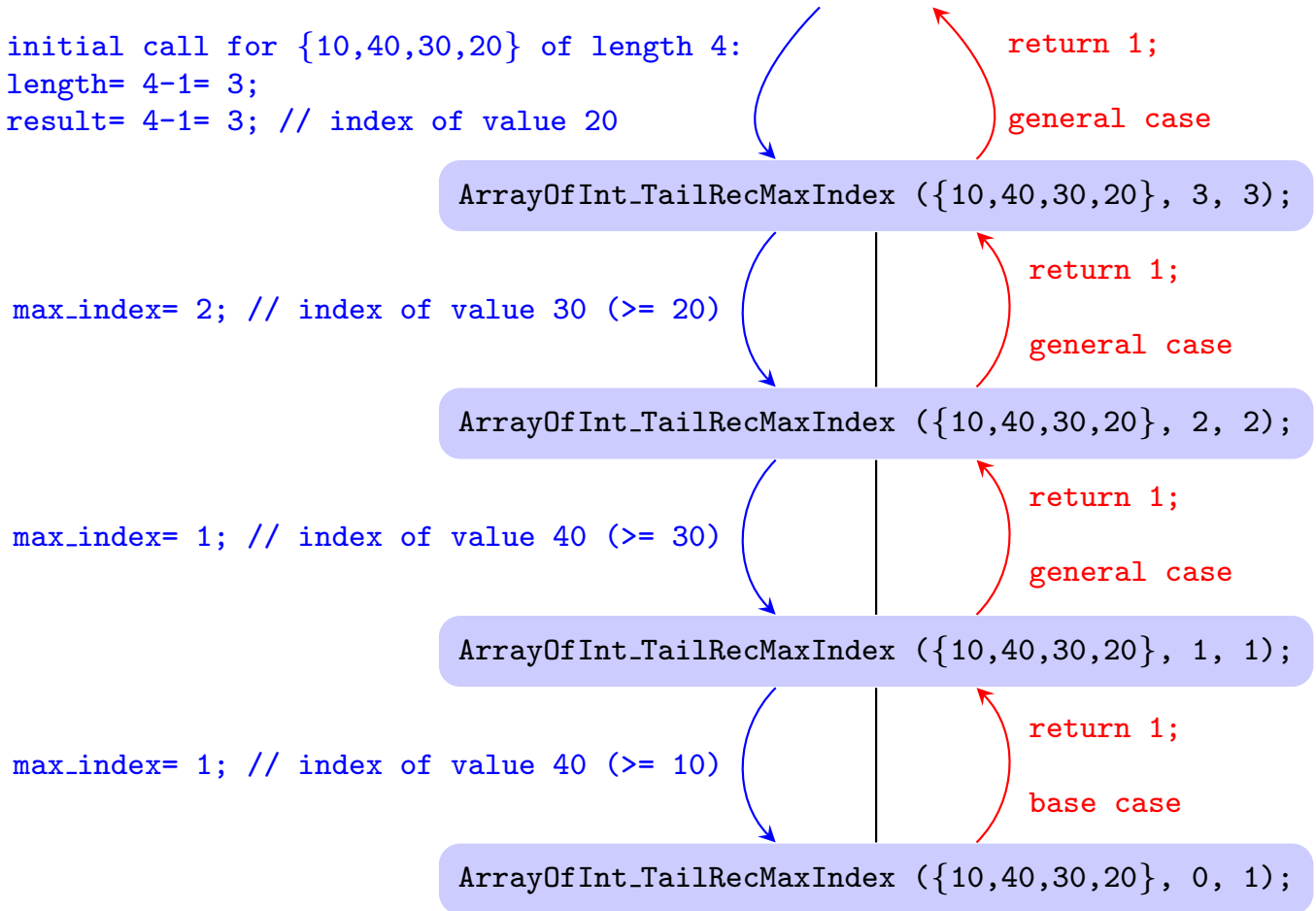
Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```



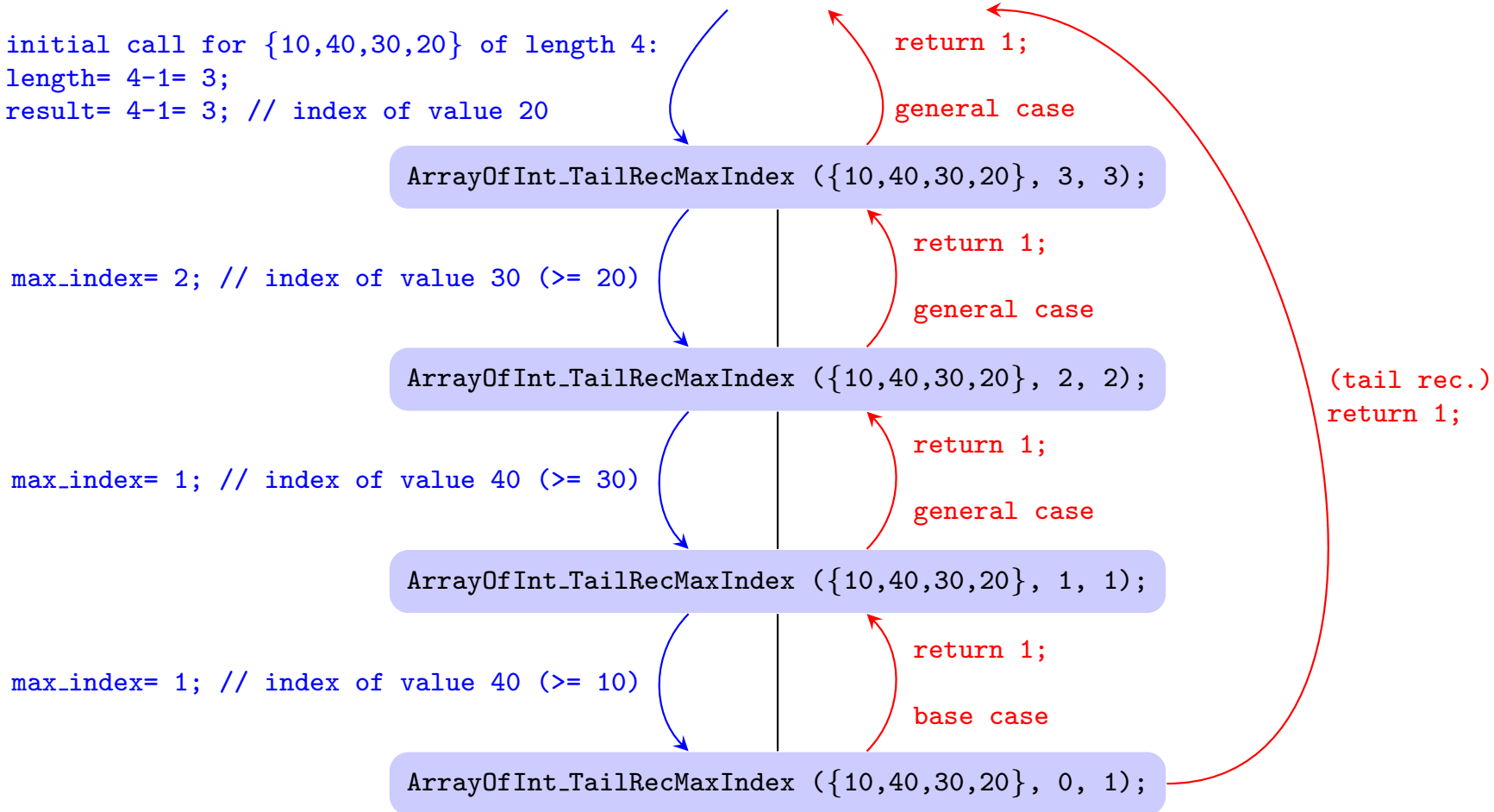
Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```



Index du max réc. terminal (1/1)

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```



Wrapper pour l'index du max récursif terminal

Pour faire fonctionner la version récursive terminale...

```
int ArrayOfInt_TailRecMaxIndex (int const array [], int length, int result) {
  if (length == 0) return result; // base case
  int max_index= (array [result] >= array [length-1]) ? result : max_index;
  return ArrayOfInt_TailRecMaxIndex (array, length-1, max_index);
}
```

il faut que l'appel initial soit de la forme suivante :

```
int ArrayOfInt_MaxIndex (int const array [], int length)
{
  return ArrayOfInt_TailRecMaxIndex (array, length-1, length-1);
}
```

Fonction récursive simple pour le sinus (1/4)

Les identités remarquables suivantes de la trigonométrie permettent de définir $sin(3x)$ en fonction de $sin(x)$:

$$\begin{array}{rcl}
\sin(a+b) & = & \sin(a)\cos(b)+\cos(a)\sin(b) \\
\cos(a+b) & = & \cos(a)\cos(b)-\sin(a)\sin(b) \\
\sin(2a) & = & 2\sin(a)\cos(a) \\
\cos(2a) & = & \cos(a)^2-\sin(a)^2 \\
1 & = & \cos(a)^2+\sin(a)^2
\end{array}$$

Fonction récursive simple pour le sinus (1/4)

Les identités remarquables suivantes de la trigonométrie permettent de définir $sin(3x)$ en fonction de $sin(x)$:

$$\begin{array}{rcl}
\sin(a+b) & = & \sin(a)\cos(b)+\cos(a)\sin(b) \\
\cos(a+b) & = & \cos(a)\cos(b)-\sin(a)\sin(b) \\
\sin(2a) & = & 2\sin(a)\cos(a) \\
\cos(2a) & = & \cos(a)^2-\sin(a)^2 \\
1 & = & \cos(a)^2+\sin(a)^2
\end{array}$$

$$\sin(3x) = \sin(x+2x)$$

Fonction récursive simple pour le sinus (1/4)

Les identités remarquables suivantes de la trigonométrie permettent de définir $sin(3x)$ en fonction de $sin(x)$:

$$\begin{array}{rcl} \sin(a+b) & = & \sin(a)\cos(b)+\cos(a)\sin(b) \\ \cos(a+b) & = & \cos(a)\cos(b)-\sin(a)\sin(b) \\ \sin(2a) & = & 2\sin(a)\cos(a) \\ \cos(2a) & = & \cos(a)^2-\sin(a)^2 \\ 1 & = & \cos(a)^2+\sin(a)^2 \end{array}$$

$$\begin{array}{rcl} \sin(3x) & = & \sin(x+2x) \\ & = & \sin(x)\cos(2x)+\cos(x)\sin(2x) \end{array}$$

Fonction récursive simple pour le sinus (1/4)

Les identités remarquables suivantes de la trigonométrie permettent de définir $sin(3x)$ en fonction de $sin(x)$:

$$\begin{array}{rcl} \sin(a+b) & = & \sin(a)\cos(b)+\cos(a)\sin(b) \\ \cos(a+b) & = & \cos(a)\cos(b)-\sin(a)\sin(b) \\ \sin(2a) & = & 2\sin(a)\cos(a) \\ \cos(2a) & = & \cos(a)^2-\sin(a)^2 \\ 1 & = & \cos(a)^2+\sin(a)^2 \end{array}$$

$$\begin{array}{rcl} \sin(3x) & = & \sin(x+2x) \\ & = & \sin(x)\cos(2x)+\cos(x)\sin(2x) \\ & = & \sin(x)[\cos(x)^2-\sin(x)^2]+\cos(x)2\sin(x)\cos(x) \end{array}$$

Fonction récursive simple pour le sinus (1/4)

Les identités remarquables suivantes de la trigonométrie permettent de définir $sin(3x)$ en fonction de $sin(x)$:

$$\begin{array}{rcl}
\sin(a+b) & = & \sin(a)\cos(b)+\cos(a)\sin(b) \\
\cos(a+b) & = & \cos(a)\cos(b)-\sin(a)\sin(b) \\
\sin(2a) & = & 2\sin(a)\cos(a) \\
\cos(2a) & = & \cos(a)^2-\sin(a)^2 \\
1 & = & \cos(a)^2+\sin(a)^2
\end{array}$$

$$\begin{array}{rcl}
\sin(3x) & = & \sin(x+2x) \\
& = & \sin(x)\cos(2x)+\cos(x)\sin(2x) \\
& = & \sin(x)[\cos(x)^2-\sin(x)^2]+\cos(x)2\sin(x)\cos(x) \\
& = & \sin(x)\ \ [\cos(x)^2-\sin(x)^2+2\cos(x)^2]
\end{array}$$

Fonction récursive simple pour le sinus (1/4)

Les identités remarquables suivantes de la trigonométrie permettent de définir $sin(3x)$ en fonction de $sin(x)$:

$$\begin{array}{rcl}
\sin(a+b) & = & \sin(a)\cos(b)+\cos(a)\sin(b) \\
\cos(a+b) & = & \cos(a)\cos(b)-\sin(a)\sin(b) \\
\sin(2a) & = & 2\sin(a)\cos(a) \\
\cos(2a) & = & \cos(a)^2-\sin(a)^2 \\
1 & = & \cos(a)^2+\sin(a)^2
\end{array}$$

$$\begin{array}{rcl}
\sin(3x) & = & \sin(x+2x) \\
& = & \sin(x)\cos(2x)+\cos(x)\sin(2x) \\
& = & \sin(x)[\cos(x)^2-\sin(x)^2]+\cos(x)2\sin(x)\cos(x) \\
& = & \sin(x)\ \ [\cos(x)^2-\sin(x)^2+2\cos(x)^2] \\
& = & \sin(x)\ \ [3\cos(x)^2-\sin(x)^2]
\end{array}$$

Fonction récursive simple pour le sinus (1/4)

Les identités remarquables suivantes de la trigonométrie permettent de définir $sin(3x)$ en fonction de $sin(x)$:

$$\begin{array}{rcl}
\sin(a+b) & = & \sin(a)\cos(b)+\cos(a)\sin(b) \\
\cos(a+b) & = & \cos(a)\cos(b)-\sin(a)\sin(b) \\
\sin(2a) & = & 2\sin(a)\cos(a) \\
\cos(2a) & = & \cos(a)^2-\sin(a)^2 \\
1 & = & \cos(a)^2+\sin(a)^2
\end{array}$$

$$\begin{array}{rcl}
\sin(3x) & = & \sin(x+2x) \\
& = & \sin(x)\cos(2x)+\cos(x)\sin(2x) \\
& = & \sin(x)[\cos(x)^2-\sin(x)^2]+\cos(x)2\sin(x)\cos(x) \\
& = & \sin(x)\ \ [\cos(x)^2-\sin(x)^2+2\cos(x)^2] \\
& = & \sin(x)\ \ [3\cos(x)^2-\sin(x)^2] \\
& = & \sin(x)\ \ [3-3\sin(x)^2-\sin(x)^2]
\end{array}$$

Fonction récursive simple pour le sinus (1/4)

Les identités remarquables suivantes de la trigonométrie permettent de définir $sin(3x)$ en fonction de $sin(x)$:

$$\begin{array}{rcl}
\sin(a+b) & = & \sin(a)\cos(b)+\cos(a)\sin(b) \\
\cos(a+b) & = & \cos(a)\cos(b)-\sin(a)\sin(b) \\
\sin(2a) & = & 2\sin(a)\cos(a) \\
\cos(2a) & = & \cos(a)^2-\sin(a)^2 \\
1 & = & \cos(a)^2+\sin(a)^2
\end{array}$$

$$\begin{array}{rcl}
\sin(3x) & = & \sin(x+2x) \\
& = & \sin(x)\cos(2x)+\cos(x)\sin(2x) \\
& = & \sin(x)[\cos(x)^2-\sin(x)^2]+\cos(x)2\sin(x)\cos(x) \\
& = & \sin(x)\ [\cos(x)^2-\sin(x)^2+2\cos(x)^2] \\
& = & \sin(x)\ [3\cos(x)^2-\sin(x)^2] \\
& = & \sin(x)\ [3-3\sin(x)^2-\sin(x)^2] \\
& = & \sin(x)\ [3-4\sin(x)^2]
\end{array}$$

Fonction récursive simple pour le sinus (2/4)

On a donc la relation simple $\sin(x) = \sin(\frac{x}{3})\ [3 - 4\sin(\frac{x}{3})^2]$.
Par ailleurs, $\sin(\varepsilon) \approx \varepsilon$ lorsque ε est proche de 0.

```
bool Double_IsSmall (double number, double epsilon)
{
  return fabs (number) <= epsilon;
}
```

On peut donc écrire `Math_Sin()` récursivement comme suit :

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}
```

Le point-clé ici est de poser la variable `s` pour $\sin(\frac{x}{3})$,
afin de n'avoir qu'un seul appel récursif dans la fonction,
alors que cette expression apparaît 3 fois dans la définition.

Fonction récursive simple pour le sinus (3/4)

Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```

Fonction récursive simple pour le sinus (3/4)

Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```

Fonction récursive simple pour le sinus (3/4)

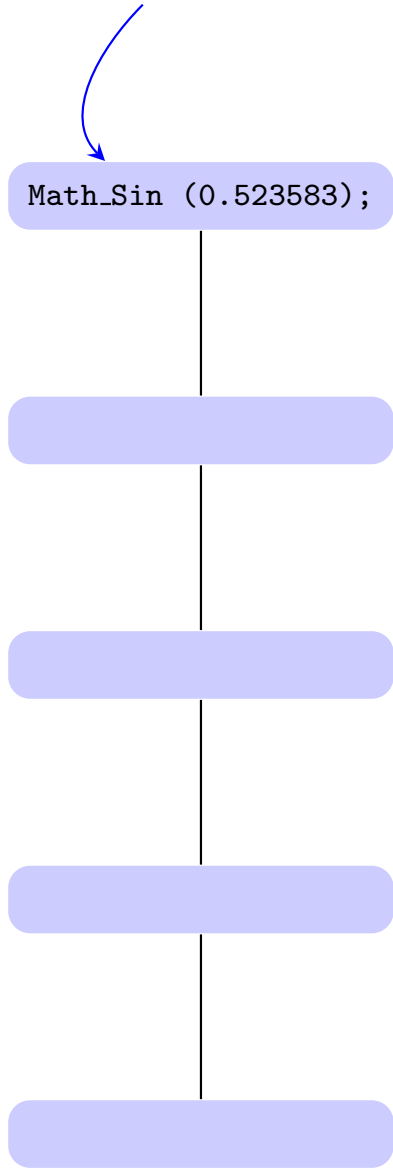
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```



Fonction récursive simple pour le sinus (3/4)

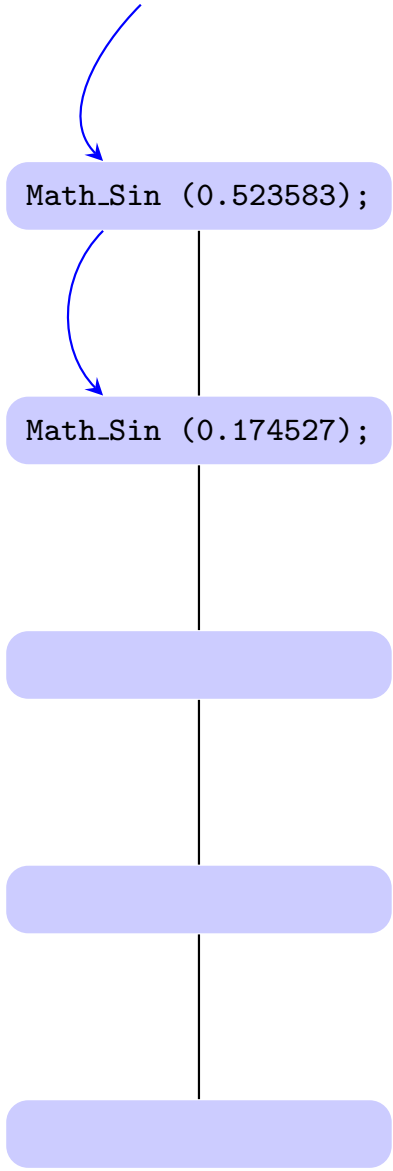
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```



Fonction récursive simple pour le sinus (3/4)

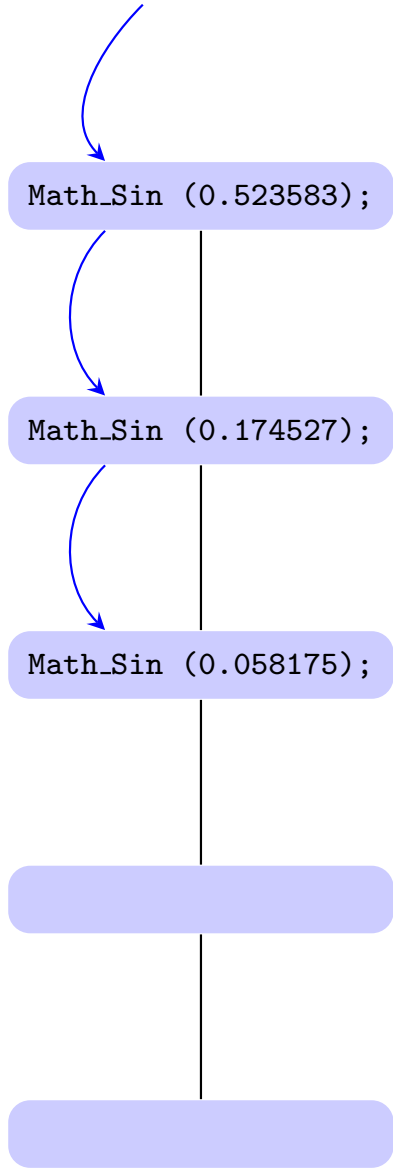
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```



Fonction récursive simple pour le sinus (3/4)

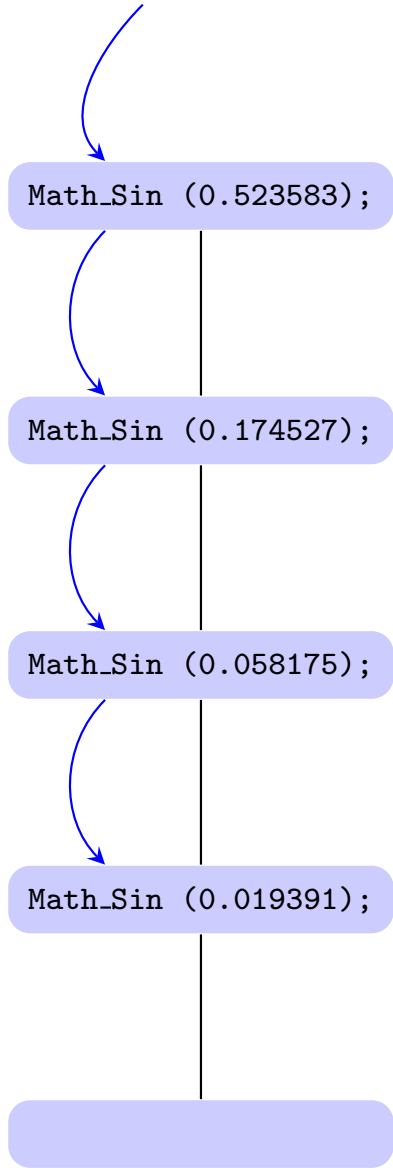
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```



Fonction récursive simple pour le sinus (3/4)

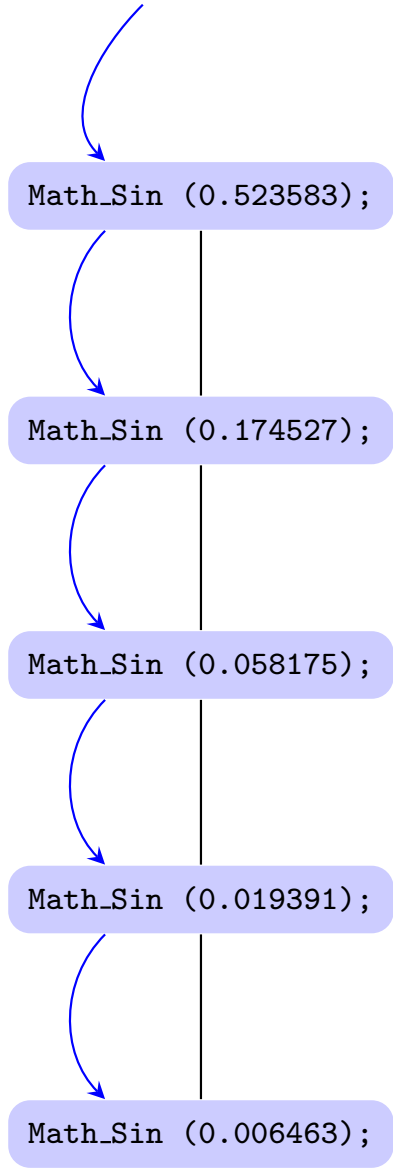
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```



Fonction récursive simple pour le sinus (3/4)

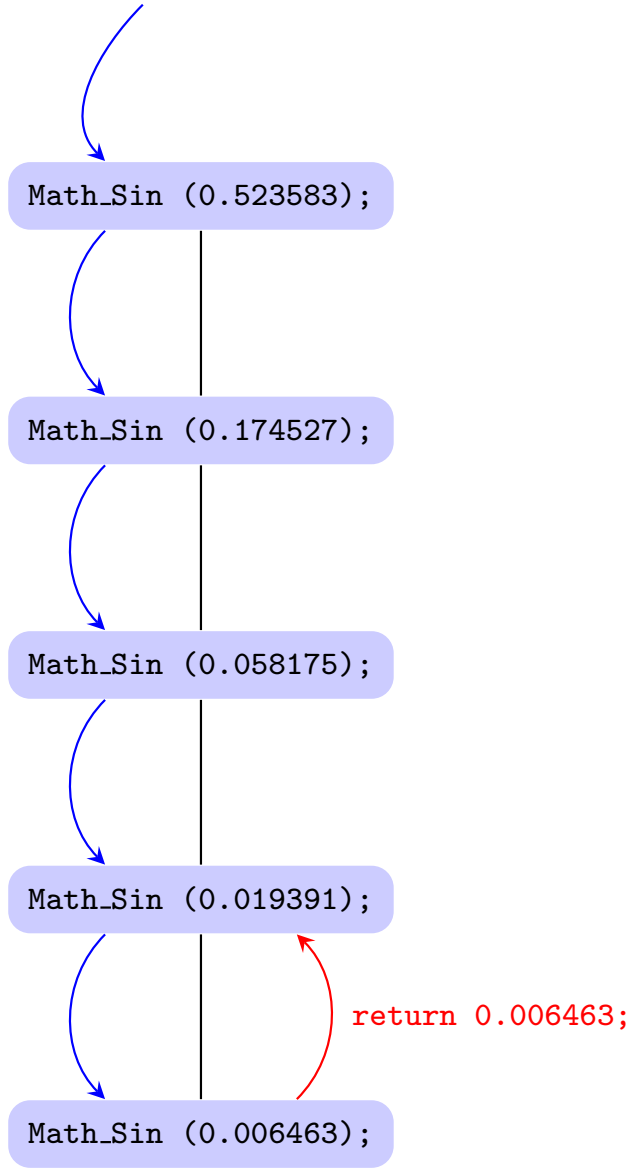
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```



Fonction récursive simple pour le sinus (3/4)

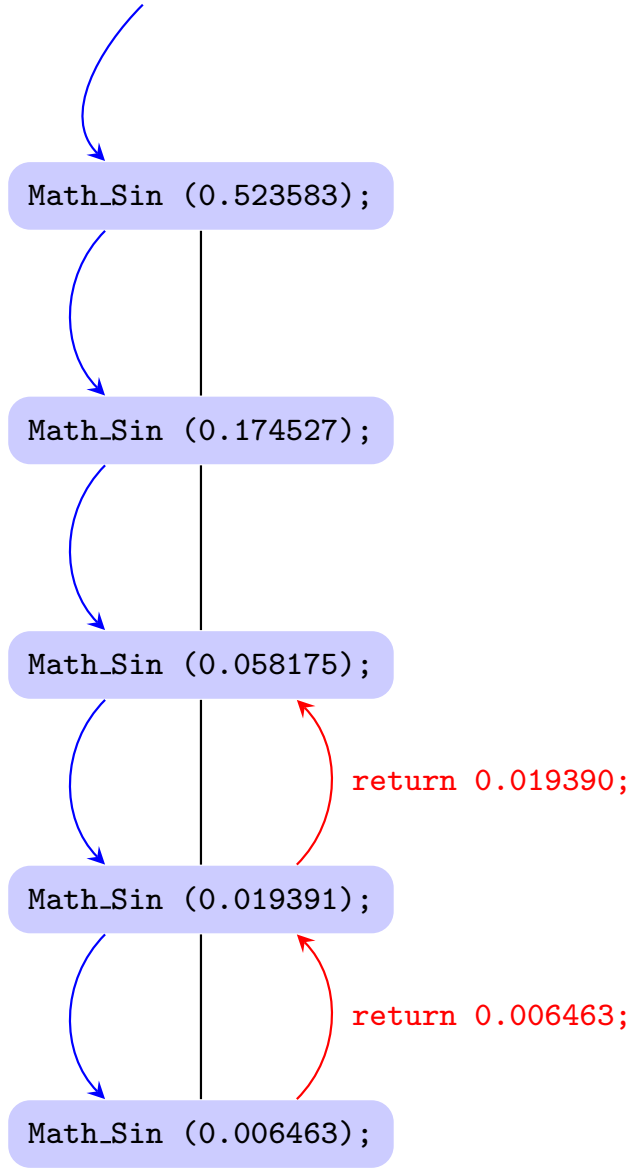
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```



Fonction récursive simple pour le sinus (3/4)

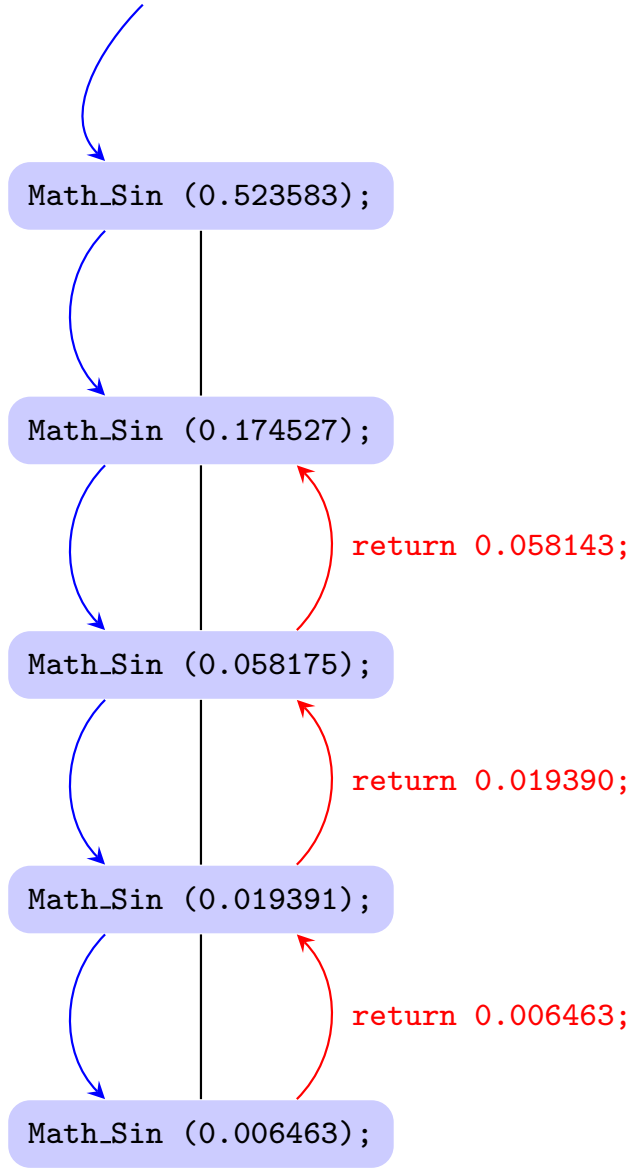
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```



Fonction récursive simple pour le sinus (3/4)

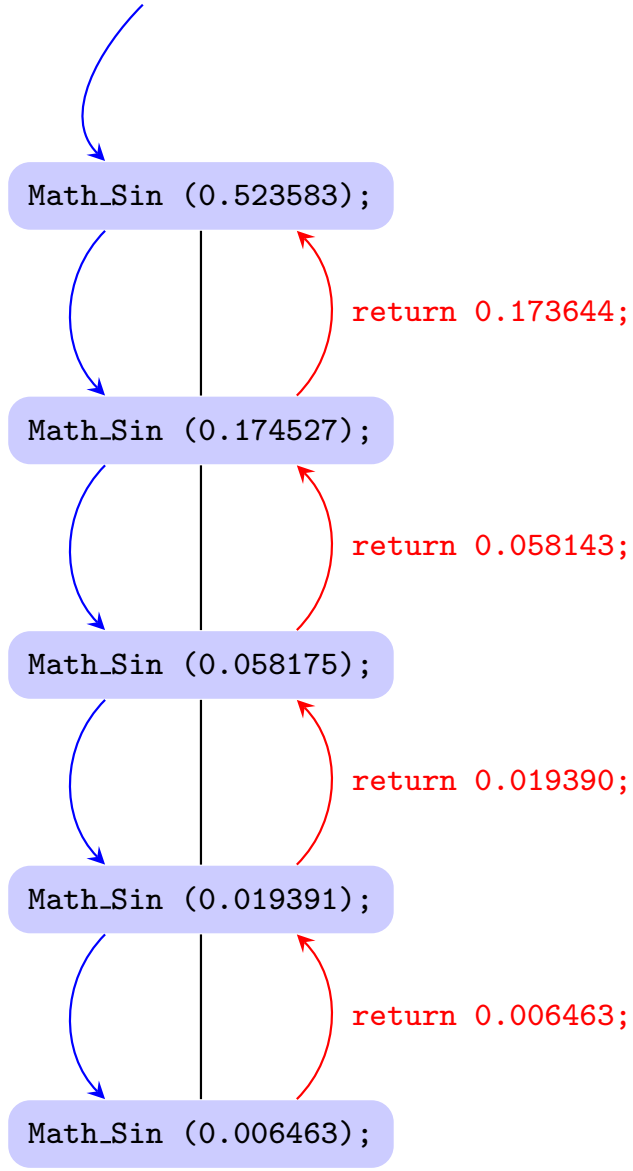
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```



Fonction récursive simple pour le sinus (3/4)

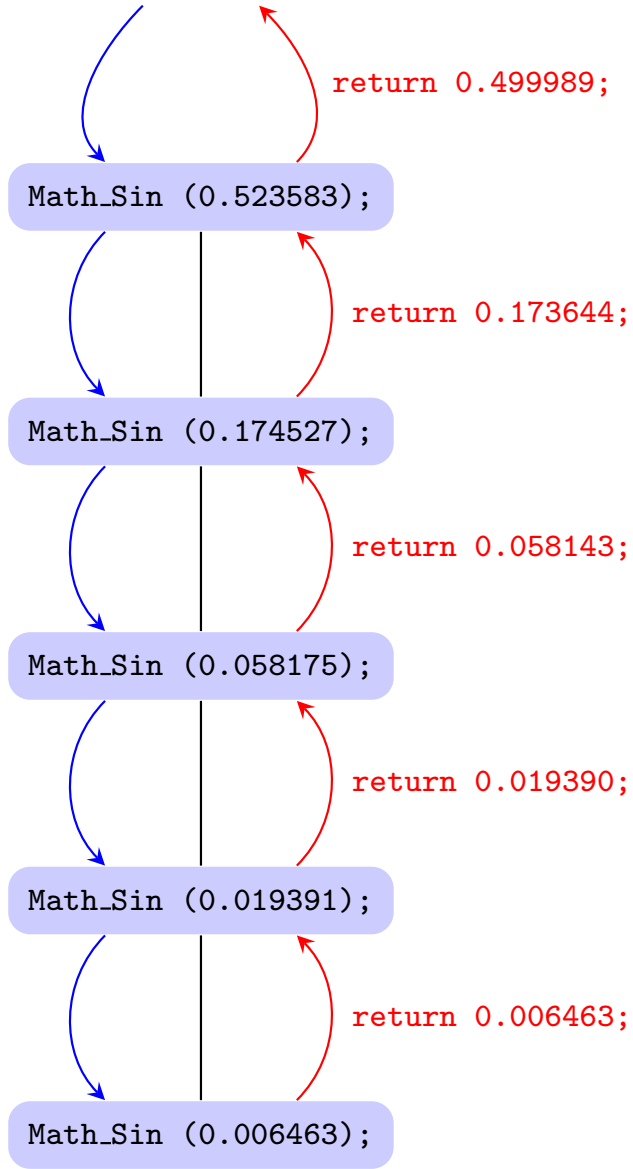
Testons la fonction avec $\frac{\pi}{6}$,
pour lequel on sait que $\sin(\frac{\pi}{6}) = \frac{1}{2}$.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double s= Math_Sin (angle / 3.0);
  return s*(3 - 4*s*s);
}

int main (void){
  double x= 3.1415 / 6.0;
  printf ("Math_Sin (x)= %.6f\n", Math_Sin (x));
  printf ("     sin (x)= %.65f\n", sin (x));
  return 0;
}
```

On obtient la sortie :

```
Math_Sin (x)= 0.499989
     sin (x)= 0.499986
```

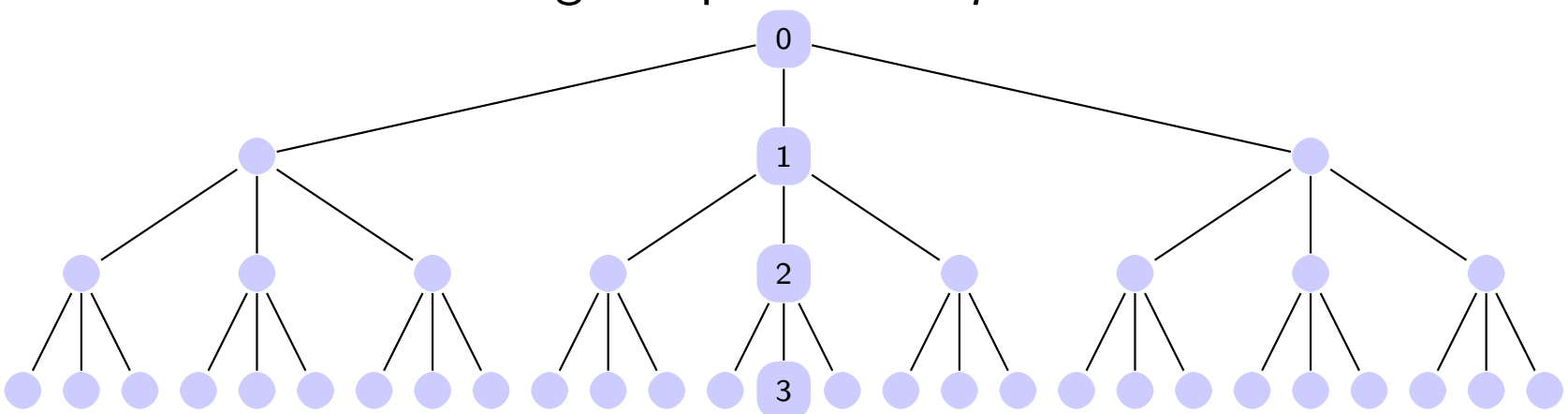


Fonction récursive simple pour le sinus (4/4)

Si l'on n'avait pas posé la variable `s`,
il y aurait 3 appels récursifs dans la fonction,
et il y aurait alors une explosion exponentielle d'appels.

```
double Math_Sin (double angle)
{
  if (Double_IsSmall (angle, 0.01)) return angle;
  double a= angle / 3.0;
  return Math_Sin(a) * (3 - 4 * Math_Sin(a) * Math_Sin(a));
}
```

Dans l'arbre ci-dessous, chaque noeud est un appel à fonction codée ci-dessus. Un étage de profondeur p contient 3^p noeuds.



Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un `'.'` sont visitables.
Les cellules marquées d'un `'X'` sont des murs.

```
.X...
.X...
XX.X.
.....
```

Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un `'.'` sont visitables.
Les cellules marquées d'un `'X'` sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un `'*'`.

```
.X...
.X...
XX.X.
.....
```

```
.X...
.X...
XX.X.
..*..
```

Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un `'.'` sont visitables.
Les cellules marquées d'un `'X'` sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un `'*'`.

```
.X...  .X...  .X...
.X...  .X...  .X...
XX.X.  XX.X.  XX*X.
.....  ..*..  ..*..
```

Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un `'.'` sont visitables.
Les cellules marquées d'un `'X'` sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un `'*'`.

```
.X...  .X...  .X...  .X...
.X...  .X...  .X...  .X*..
XX.X.  XX.X.  XX*X.  XX*X.
.....  ..*..  ..*..  ..*..
```

Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un `'.'` sont visitables.
Les cellules marquées d'un `'X'` sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un `'*'`.

```
.X...  .X...  .X...  .X...  .X*..
.X...  .X...  .X...  .X*..  .X*..
XX.X.  XX.X.  XX*X.  XX*X.  XX*X.
.....  ..*..  ..*..  ..*..  ..*..
```

Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un `'.'` sont visitables.
Les cellules marquées d'un `'X'` sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un `'*'`.

```
.X...  .X...  .X...  .X...  .X*..  .X**.
.X...  .X...  .X...  .X*..  .X*..  .X*..
XX.X.  XX.X.  XX*X.  XX*X.  XX*X.  XX*X.
.....  ..*..  ..*..  ..*..  ..*..  ..*..
```

Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un `'.'` sont visitables.
Les cellules marquées d'un `'X'` sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un `'*'`.

```
.X...  .X...  .X...  .X...  .X*..  .X**.  .X***
.X...  .X...  .X...  .X*..  .X*..  .X*..  .X*..
XX.X.  XX.X.  XX*X.  XX*X.  XX*X.  XX*X.  XX*X.
.....  ..*..  ..*..  ..*..  ..*..  ..*..  ..*..
```

Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un '.' sont visitables.
Les cellules marquées d'un 'X' sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un '*'.

```
.X...
.X...
XX.X.
.....
```

```
.X...
.X...
XX.X.
..*..
```

```
.X...
.X...
XX*X.
..*..
```

```
.X...
.X*..
XX*X.
..*..
```

```
.X*..
.X*..
XX*X.
..*..
```

```
.X**.
.X*..
XX*X.
..*..
```

```
.X***
.X*..
XX*X.
..*..
```

```
.X***
.X*.*
XX*X.
..*..
```

Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un '.' sont visitables.
Les cellules marquées d'un 'X' sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un '*'.

```
.X...
.X...
XX.X.
.....
```

```
.X...
.X...
XX.X.
..*..
```

```
.X...
.X...
XX*X.
..*..
```

```
.X...
.X*..
XX*X.
..*..
```

```
.X*..
.X*..
XX*X.
..*..
```

```
.X**.
.X*..
XX*X.
..*..
```

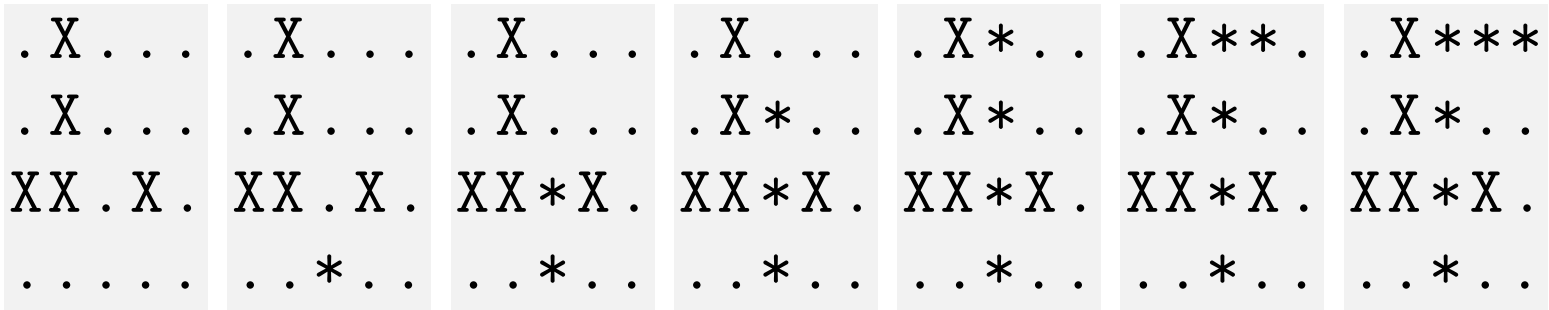
```
.X***
.X*..
XX*X.
..*..
```

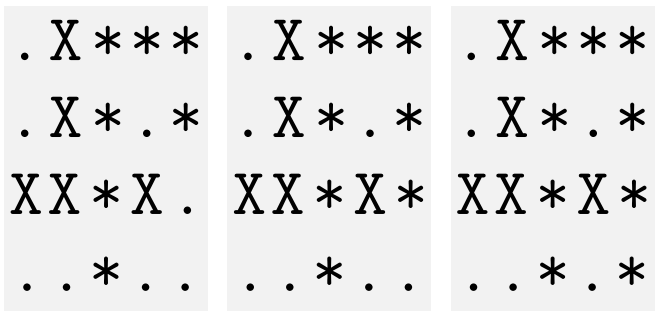
```
.X***
.X*.*
XX*X.
..*..
```

```
.X***
.X*.*
XX*X*
..*..
```

Visite de cellules dans une grille (1/5)

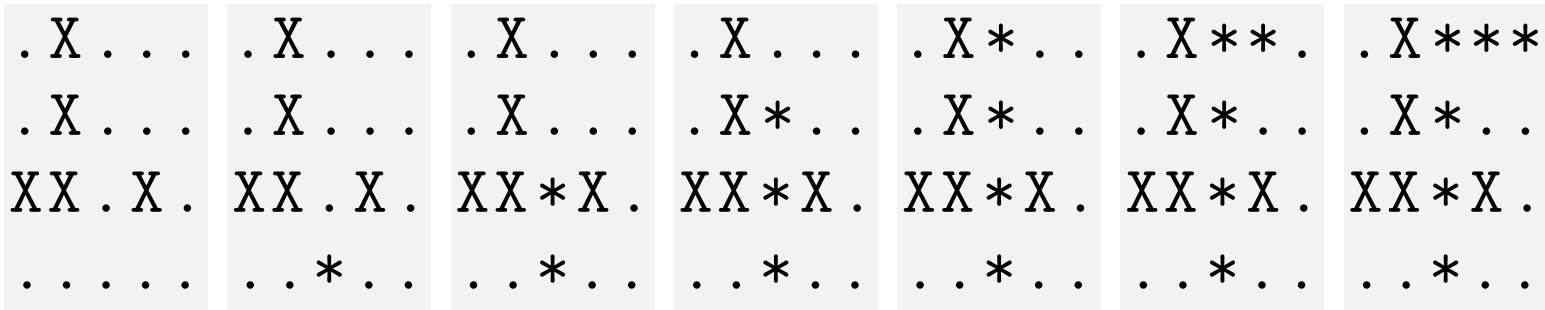
Dans cette grille, les cellules marquées d'un '.' sont visitables.
Les cellules marquées d'un 'X' sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un '*'.

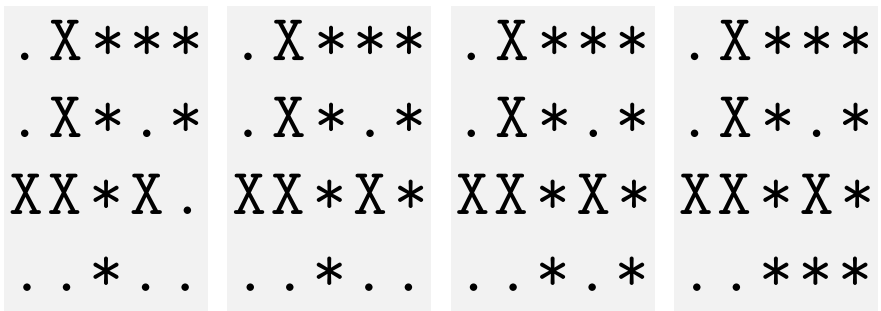




Visite de cellules dans une grille (1/5)

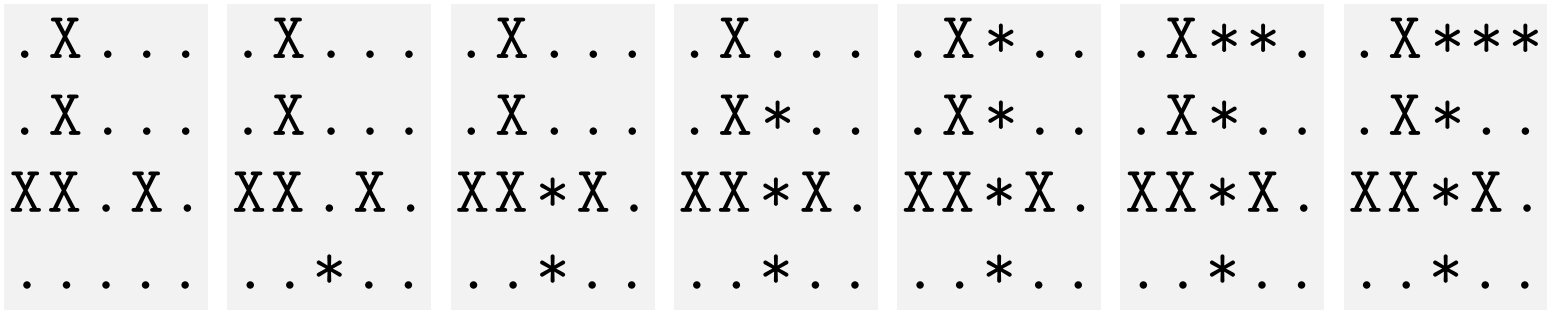
Dans cette grille, les cellules marquées d'un '.' sont visitables.
Les cellules marquées d'un 'X' sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un '*'.

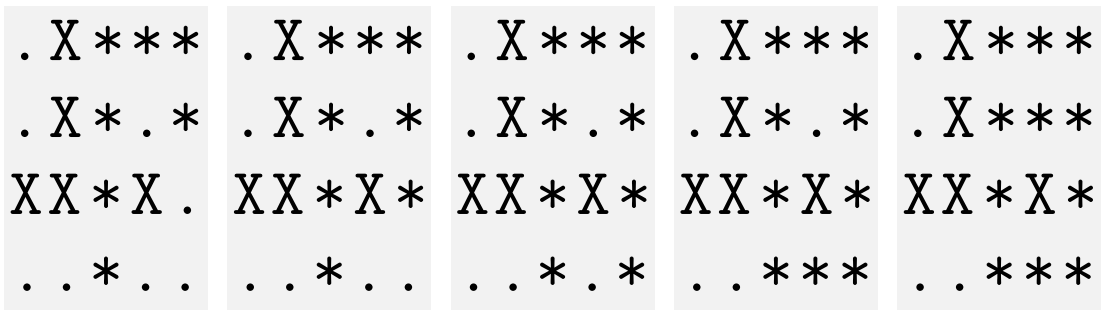




Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un '.' sont visitables.
Les cellules marquées d'un 'X' sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un '*'.





Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un '.' sont visitables.
Les cellules marquées d'un 'X' sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un '*'.

```
.X...
.X...
XX.X.
.....
```

```
.X...
.X...
XX.X.
..*..
```

```
.X...
.X...
XX*X.
..*..
```

```
.X...
.X*..
XX*X.
..*..
```

```
.X*..
.X*..
XX*X.
..*..
```

```
.X**.
.X*..
XX*X.
..*..
```

```
.X***
.X*..
XX*X.
..*..
```

```
.X***
.X*.*
XX*X.
..*..
```

```
.X***
.X*.*
XX*X*
..*..
```

```
.X***
.X*.*
XX*X*
..*.*
```

```
.X***
.X*.*
XX*X*
..***
```

```
.X***
.X***
XX*X*
..***
```

```
.X***
.X***
XX*X*
.****
```

Visite de cellules dans une grille (1/5)

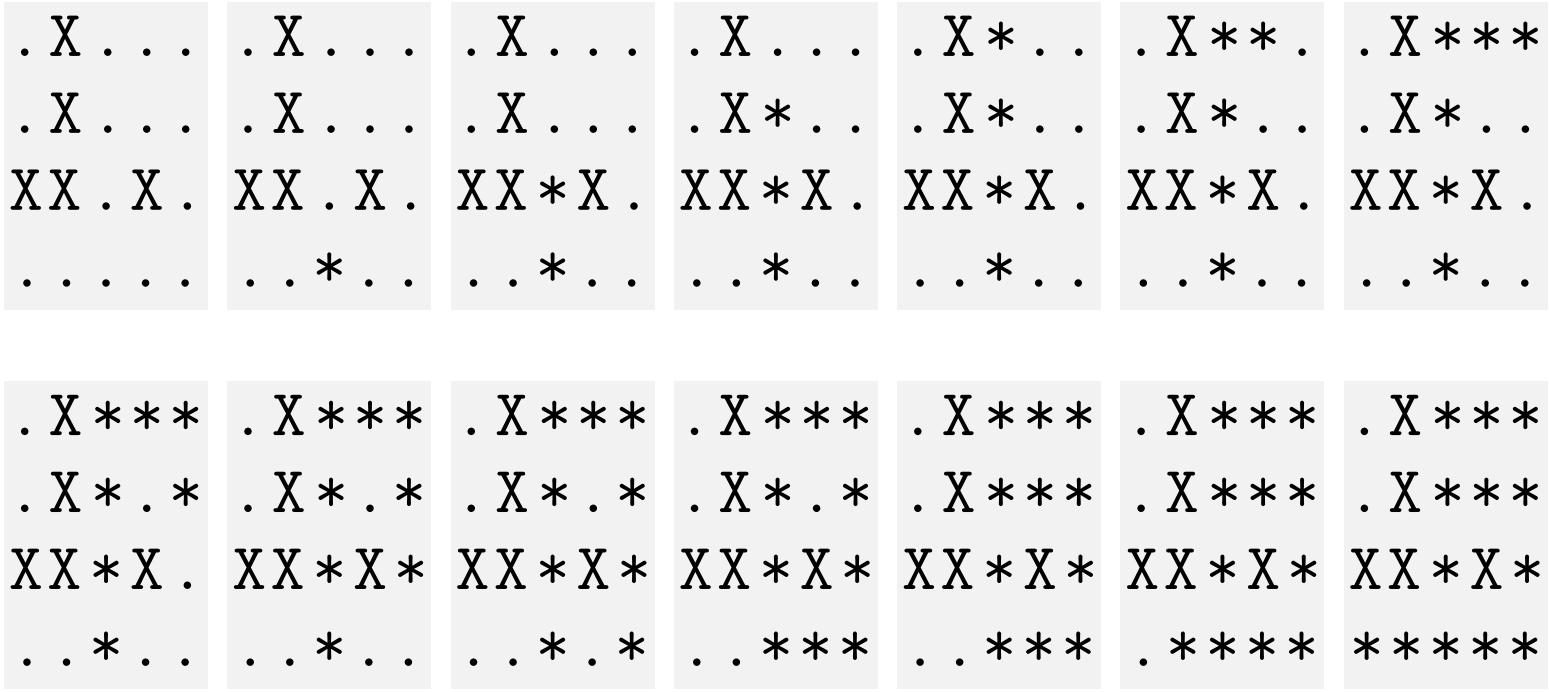
Dans cette grille, les cellules marquées d'un '.' sont visitables.
Les cellules marquées d'un 'X' sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un '*'.

```
.X...  .X...  .X...  .X...  .X*..  .X**.  .X***
.X...  .X...  .X...  .X*..  .X*..  .X*..  .X*..
XX.X.  XX.X.  XX*X.  XX*X.  XX*X.  XX*X.  XX*X.
.....  ..*..  ..*..  ..*..  ..*..  ..*..  ..*..
```

```
.X***  .X***  .X***  .X***  .X***  .X***  .X***
.X*.*  .X*.*  .X*.*  .X*.*  .X***  .X***  .X***
XX*X.  XX*X*  XX*X*  XX*X*  XX*X*  XX*X*  XX*X*
..*..  ..*..  ..*.*  ..***  ..***  .****  *****
```

Visite de cellules dans une grille (1/5)

Dans cette grille, les cellules marquées d'un '.' sont visitables.
Les cellules marquées d'un 'X' sont des murs.
Un voyageur part d'une cellule donnée et tente de visiter toutes les cellules accessibles les marquant d'un '*'.



13 cellules ont été visitées.
(2 cellules visitables sont restées inaccessibles).

Visite de cellules dans une grille (2/5)

On se donne une structure `Grid` pour la grille de cellule :

```
#define MAX_ROWS 15
#define MAX_COLS 20

typedef struct Grid {
  char cells [MAX_ROWS][MAX_COLS];
  int nb_rows, nb_cols;
} Grid;
```

Visite de cellules dans une grille (2/5)

On se donne une structure `Grid` pour la grille de cellule :

```
#define MAX_ROWS 15
#define MAX_COLS 20

typedef struct Grid {
  char cells [MAX_ROWS][MAX_COLS];
  int nb_rows, nb_cols;
} Grid;
```

On cherche a écrire la fonction de visite `Grid_VisitArea()` et la fonction d'affichage `Grid_Print()` utilisées ci-dessous :

```
int main (void) {
  Grid grid= (Grid) {
    .nb_rows= 4,
    .nb_cols= 5,
    .cells= {
      { '.', 'X', '.', '.', '.' },
      { '.', 'X', '.', '.', '.' },
      { 'X', 'X', '.', 'X', '.' },
      { '.', '.', '.', '.', '.' },
    },
  };
  Grid_Print (& grid, stdout);
  int nb_visited= Grid_VisitArea (& grid, 3, 2);
  printf ("%d cells were visited\n", nb_visited);
  return 0;
}
```

Visite de cellules dans une grille (3/5)

La fonction d'affichage `Grid_Print()` est triviale :

```
void Grid_Print (Grid const * g, FILE * file) {
  for (int row= 0; row < g->nb_rows; row++) {
    for (int col= 0; col < g->nb_cols; col++) {
      fprintf (file, "%c", g->cells[row][col]);
    }
    fprintf (file, "\n");
  }
  fprintf (file, "\n");
}
```

Visite de cellules dans une grille (4/5)

Pour visiter une zone, il faut déjà pouvoir visiter une position seule. `Grid_VisitPos()` fait ce travail. À l'instar de `Grid_VisitArea()`, elle renvoie le nombre de positions visitées, c'est-à-dire soit 1 soit 0 (lorsque la position n'est pas visitable).

```
int Grid_VisitPos (Grid * g, int row, int col) {
  if ( ! Grid_PosIsValid (g, row, col)) return 0;
  if (g->cells [row][col] != '.') return 0;
  g->cells [row][col]= '*';
  Grid_Print (g, stdout);
  return 1;
}
```

Les positions hors-grille ne sont pas visitables :

```
bool Grid_RowIsValid (Grid const * g, int row) {
  return 0 <= row && row < g->nb_rows;
}

bool Grid_ColIsValid (Grid const * g, int col) {
  return 0 <= col && col < g->nb_cols;
}

bool Grid_PosIsValid (Grid const * g, int row, int col) {
  return Grid_RowIsValid (g, row)
    &&   Grid_ColIsValid (g, col);
}
```

Visite de cellules dans une grille (5/5)

Visiter une zone depuis une position donnée consiste à tenter visiter la position seule, puis en cas de succès, à visiter la zone depuis les positions voisines. Ceci se fait en appelant `Grid_VisiArea()` récursivement 4 fois (1 fois sur chacune positions voisines) :

```
int Grid_VisitArea (Grid * g, int row, int col) {
  int nb_visited= Grid_VisitPos (g, row, col);
  if (nb_visited == 0) return 0;

  nb_visited += Grid_VisitArea (g, row-1, col  );
  nb_visited += Grid_VisitArea (g, row  , col+1);
  nb_visited += Grid_VisitArea (g, row+1, col  );
  nb_visited += Grid_VisitArea (g, row ,  col-1);
  return nb_visited;
}
```

Ce genre d'algorithme s'appelle un **parcours en profondeur** (ou **DFS** pour **Depth First Search**).