last update: 28th February 2024

Session: 2023-24

COMP281 Lecture 10

Principles of C and Memory Management

Phil Jimmieson

UNIVERSITY OF LIVERPOOL
Department of Computer Science

Last Lecture

- struct, union, typedef, string

Last Lecture

- struct, union, typedef, string

Last Lecture

- struct, union, typedef, string

Last Lecture

- struct, union, typedef, string

Last Lecture

- struct, union, typedef, string

Today's learning outcomes

- Storage Classes
- Dynamic Memory Allocation
- Stack and Heap

Storage classes

Storage classes

Each variable has a storage class which decides the following things:

- **Scope**: where the value of the variable would be available inside a program.
- **Default initial value**: default initial value if NOT explicitly initialised.
- **Lifetime**: for how long will that variable exist.

The following storage classes are mostly used:

- **Automatic variables**
- **External variables**
- **Static variables**
- **Register variables**

Automatic variables

Each variable has a storage class which decides the following things:

- **Scope**: local to the function block inside which they are defined.
- **Default initial value**: any random value, i.e. garbage value.
- **Lifetime**: till the end of the function/method block where the variable is defined.

```
{
    int detail;        /* by default it’s an 'auto' */
    /* or */
    auto int details;     /* Both are the same */
    return 0;
}
```

External (Global) variables

Each variable has a storage class which decides the following things:

- **Scope**: not bound by any function; available everywhere.
- **Default initial value**: 0(zero).
- **Lifetime**: Until the program finishes executing.

Global values can be changed by any function in the program!

Example 13

```c
#include <stdio.h>
int number; /* global variable */

void fun1() {
   number = 20;
   printf("I am in function fun1. My value is %d.\n", number);
}

void fun2() {
   printf("I am in function fun2. My value is %d.\n", number);
}

int main() {
   number = 10;
   printf("I am in main function. My value is %d\n", number);
   fun1(); /* function calling, discussed in next topic */
   fun2(); /* function calling, discussed in next topic */
   return 0;
}
```

Example 13

```c
#include <stdio.h>
int number; /* global variable */

void fun1() {
   number = 20;
   printf("I am in function fun1. My value is %d.\n", number);
}

void fun2() {
   printf("I am in function fun2. My value is %d.\n", number);
}

int main() {
   number = 10;
   printf("I am in main function. My value is %d\n", number);
   fun1(); /* function calling,
   fun2(); /* function calling,
   return 0;
}
```

Output

```
I am in main function. My value is 10
I am in function fun1. My value is 20.
I am in function fun2. My value is 20.
```

Keyword extern

The `extern` keyword is used with a variable to inform the compiler that this variable is declared somewhere else. The extern declaration does not allocate storage for variables.

file1.c

```
#include <stdio.h>
/* global variable */
int a = 7;

void func() {
   a++;
   printf("%d", a);
   … …
}
```

file2.c

```
#include "file.c";

int main() {
   extern int a;
   func();
   … …
   return 0;
}
```

Problem when extern is not used

```
int main() {
    a = 10; /* Error: cannot find definition of variable 'a' */
    printf("%d", a);
    return 0;
}
```

Example using extern in same file

```
int main() {
    extern int x; /* informs the compiler that it is defined
                     somewhere else */
    x = 10;
    printf("%d", x);
}
int x; //Global variable x
```

Static variables

Tells the compiler to persist/save the variable until the end of program

- **Scope**: local to the block in which the variable is defined.
- **Default initial value**: 0(zero).
- **Lifetime**: Until the program finishes executing.

Instead of creating and destroying a variable every time when it comes into and goes out of scope, a static variable is initialised only once and remains in existence until the end of the program.

Example 14

```c
#include <stdio.h>

void test();

int main() {
   test();
   test();
   test();
   return 0;
}
void test() {
   static int a = 0; /* a static variable */
   a = a + 1;
   printf("%d\t",a);
}
```

Output

```
1       2       3
```

Register variables

Tells the compiler to store the variable in a CPU register

- **Scope**: local to the function in which it is declared.
- **Default initial value**: Any random value, i.e. garbage value.
- **Lifetime**: Until the end of function/method block, in which the variable is defined.

Syntax

```
register int number;
```

- Register variables inform the compiler to store the variable in **CPU** register instead of **memory**.
- Register variables have faster accessibility than a normal variable.
- We cannot get the address of such variables.

Which storage class should be used and when?

To improve the speed of execution of the program and to carefully use the memory space occupied by the variables:

- Use **static** storage class only when we want the value of the variable to remain same every time we call it using different function calls.
- Use **register** storage class only for those variables that are used in our program very often. CPU registers are limited and thus should be used carefully.
- Use **external (global)** storage class only for those variables that are being used by almost all the functions in the program.
- If we do not have the purpose of any of the above mentioned storage classes, then we should use the **automatic** storage class.

Memory Management in C

C manages memory

- Statically
- Automatically
- Dynamically

Static-duration Variables

- Allocated in the **main memory**
- Usually along with the executable code of the program
- Persist for the lifetime of the program

Automatic-duration Variables

- Allocated on the **stack**
- Come and go as functions are called and return

Static-duration Variables

&

Automatic-duration Variables

The size of the allocation must be

compile-time **constant**.

Static-duration Variables

&

Automatic-duration Variables

The size of the allocation must be **compile-time constant**.

Flexibility Concerns

Static-duration Variables

persist for the life of the program whether needed or not

Automatic-duration Variables

cannot persist across multiple function calls

Static-duration Variables

persist for the life of the program whether needed or not

Automatic-duration Variables

cannot persist across multiple function calls

Lifetime Concerns

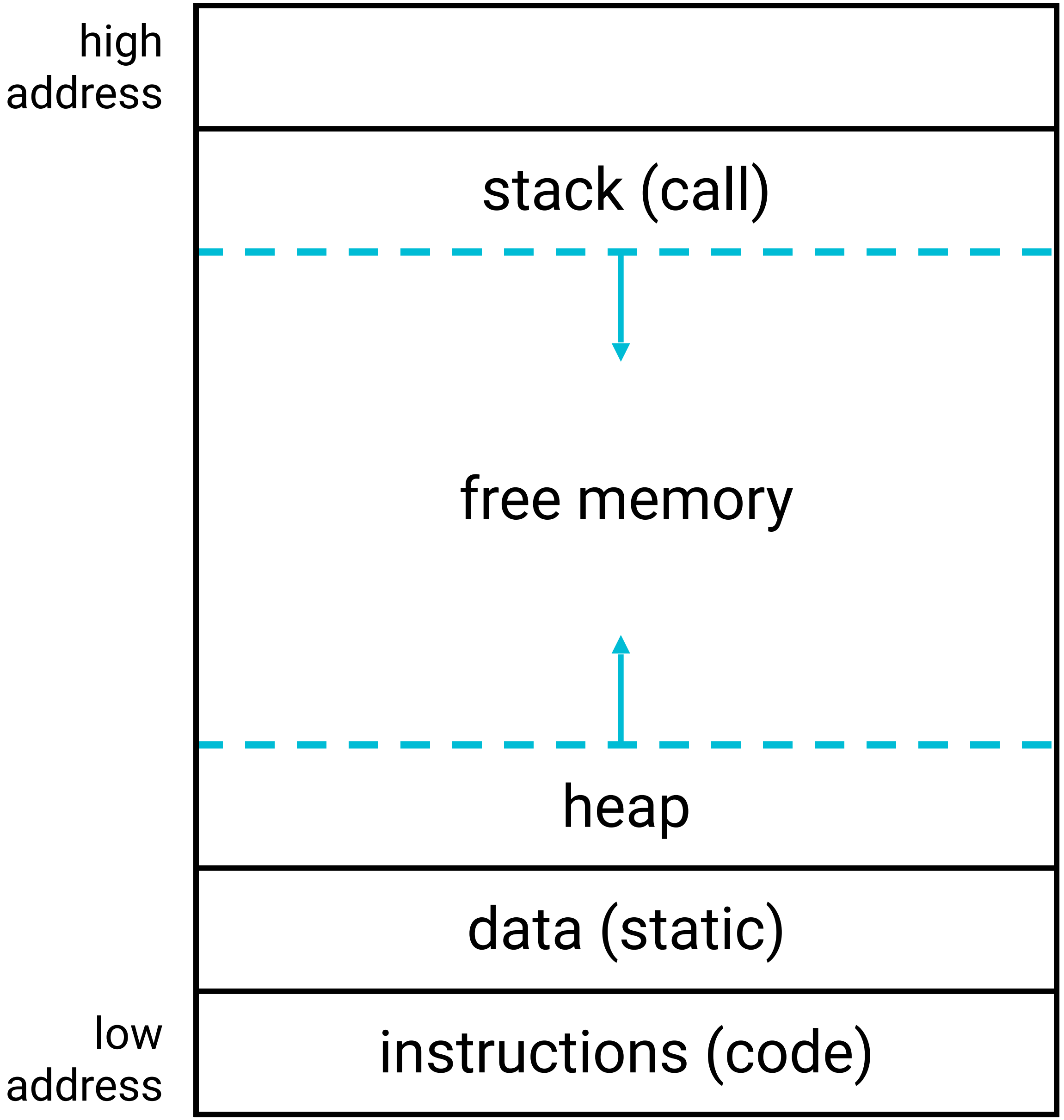
How to avoid these limitations?

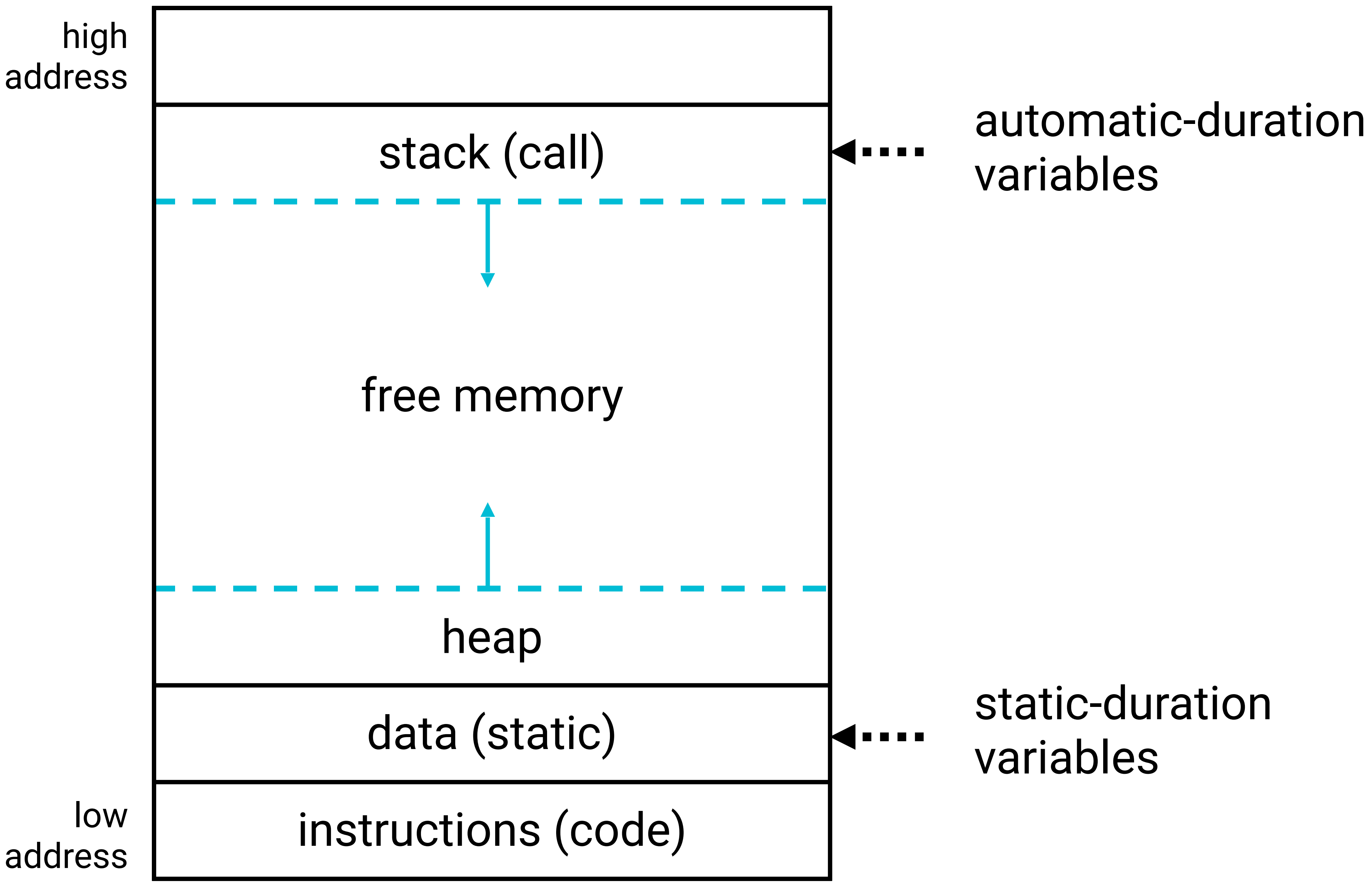
Dynamic Memory Allocation

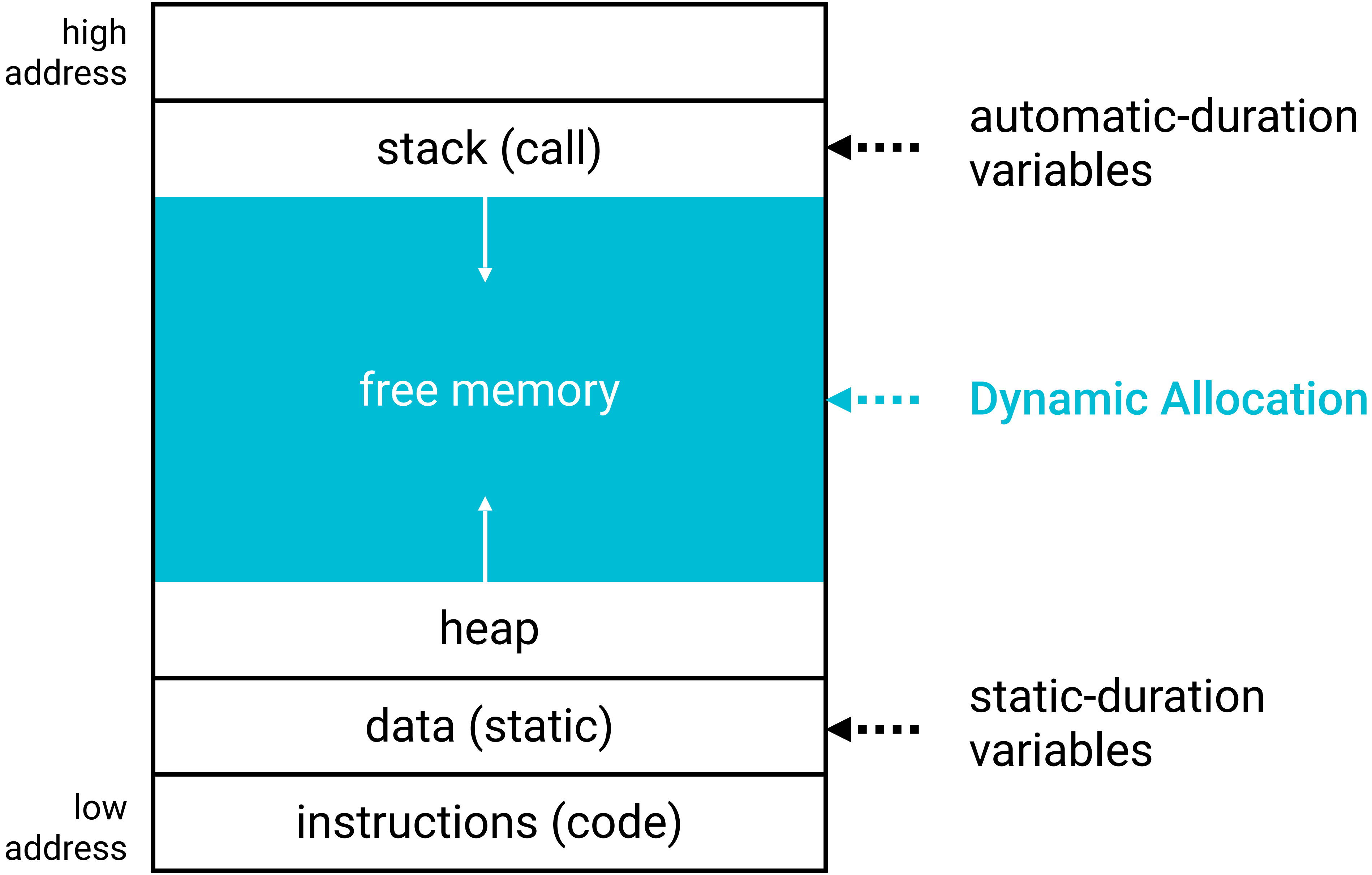
(size known at **run-time**)

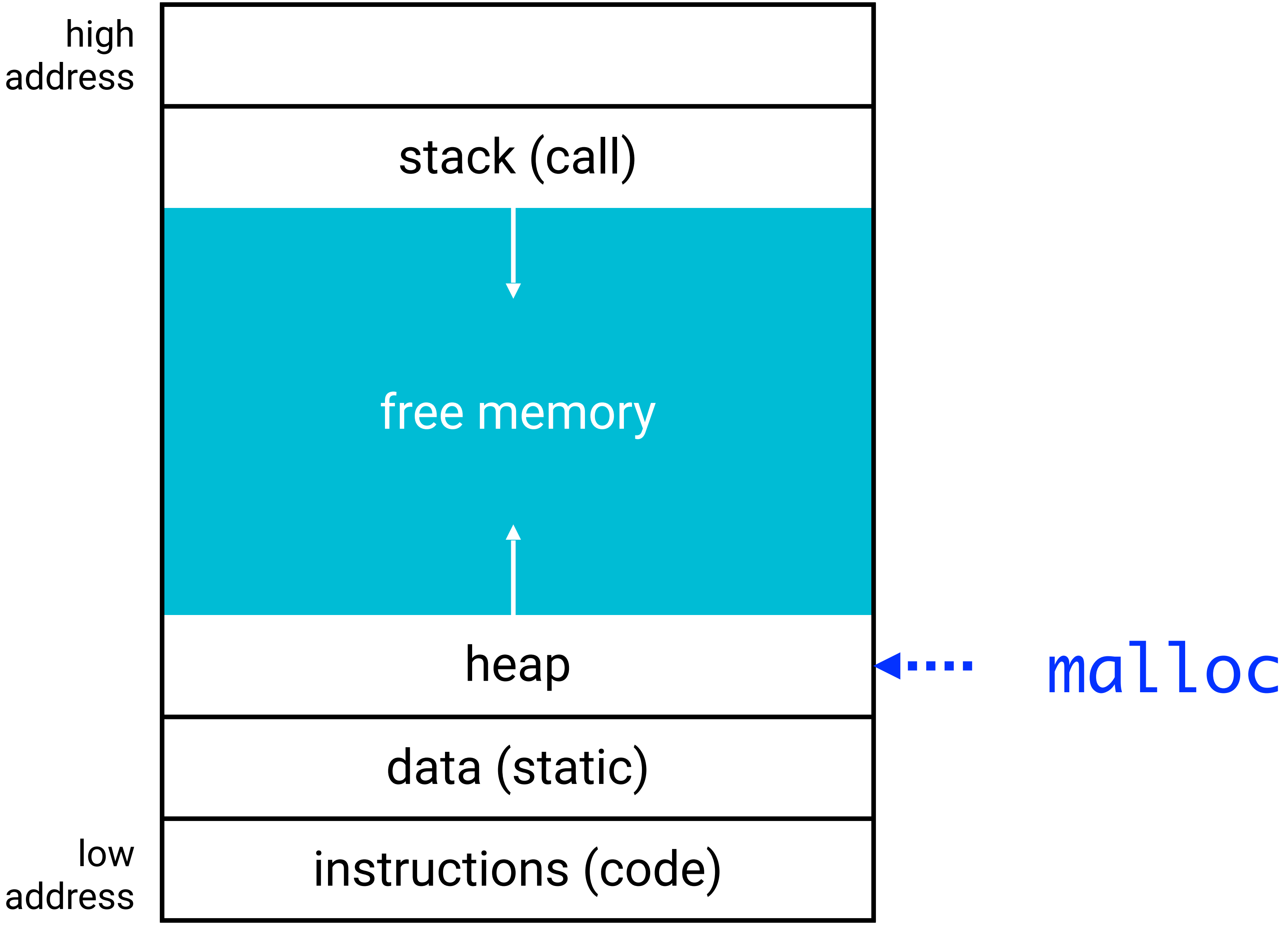
as opposed to

(size known at **compile-time**)


high
address
stack (call)
free memory
heap
data (static)
low
address
instructions (code)


high
address
stack (call)
free memory
heap
data (static)
instructions (code)
low
address
automatic-duration
variables
static-duration
variables


high
address
stack (call)
free memory
heap
data (static)
instructions (code)
low
address
automatic-duration
variables
Dynamic Allocation
static-duration
variables


high
address
stack (call)
free memory
heap
malloc
data (static)
instructions (code)
low
address

malloc

```
void *malloc(size_t size);
```

- `malloc` stands for "memory allocator".
- `malloc` is used to allocate a block of memory on the heap.
- `malloc` takes 1 argument - the size, in bytes, of the chunk of memory to be allocated

Return a `void *`? what does `void *` mean?

- It’s a "pointer to anything"
- Actual type either doesn’t matter or will be given later by a type cast

Return a `void *`? what does `void *` mean?

- It’s a "pointer to anything"
- Actual type either doesn’t matter or will be given later by a **type cast**

`malloc`

`malloc` is often used for dynamically allocated arrays

e.g., to dynamically allocate an array of 10 `int`s:

```
int *arr;
arr = (int *)malloc(10 * sizeof(int));
```

malloc

`malloc` is often used for dynamically allocated arrays

e.g., to dynamically allocate an array of 10 `int`s:

```
int *arr;
arr = (int *)malloc(10 * sizeof(int));
```

```
void *malloc(size_t size);
```

`malloc`

`malloc` is often used for dynamically allocated arrays

e.g., to dynamically allocate an array of 10 `int`s:

```
int *arr;
arr = (int *)malloc(10 * sizeof(int));
```

This computes the number of bytes that 10 `int`s occupy in memory, then requests that many bytes from `malloc` and assigns the result to an int pointer named `arr`

calloc

```
void *calloc(size_t nmemb, size_t size);
```

- `calloc` is a variant of `malloc`
- `calloc` takes **2 arguments** – the number of "things" to be allocated and the size of each "thing" (in bytes).
- `calloc` returns a pointer pointing to the chunk of memory that was allocated.

`calloc`

```
void *calloc(size_t nmemb, size_t size);
```

- `calloc` is a variant of `malloc`
- `calloc` takes **2 arguments** – the number of "things" to be allocated and the size of each "thing" (in bytes).
- `calloc` returns a pointer pointing to the chunk of memory that was allocated.
- `calloc` also sets each of the values in the allocated memory to zero (`malloc` doesn't).

calloc

calloc is also used for dynamically allocated arrays

e.g., to dynamically allocate an array of 10 ints:

```
int *arr;
arr = (int *)calloc(10, sizeof(int));
```

calloc

`calloc` is also used to dynamically allocated arrays

e.g., to dynamically allocate an array of 10 `int`s:

```
int *arr;
arr = (int *)calloc(10, sizeof(int));
```

```
void *calloc(size_t nmemb, size_t size);
```

calloc

calloc is also used to dynamically allocated arrays

e.g., to dynamically allocate an array of 10 ints:

```
int *arr;
arr = (int *)calloc(10, sizeof(int));
```

vs

```
arr = (int *)malloc(10 * sizeof(int));
```

```
void *malloc(size_t size);
```

vs

```
void *calloc(size_t nmemb, size_t size);
```

	`malloc`	`calloc`
argument	1 argument - the mount of memory to allocate in bytes	2 arguments - the number of variables to allocate; and the size in bytes of a single variable.
initialisation	No	0 (zero)

`malloc`/`calloc` return value

`malloc` and `calloc` both return **a pointer** pointing to the newly-allocated chunk of memory (**the address**).

`malloc/calloc` return value

`malloc` and `calloc` both return a pointer pointing to the newly-allocated chunk of memory (the address).

BUT!

Not guaranteed to succeed.

Maybe there is not enough memory available ☹

What to do if they fail?

`malloc/calloc` return value

If they fail, they return `NULL`

We must always check for `NULL`

when using `malloc` or `calloc`

`malloc/calloc` return value

```
int *arr = (int *)malloc(10 * sizeof(int));
/* code that uses arr… */
```

`malloc/calloc` return value

☹

```
int *arr = (int *)malloc(10 * sizeof(int));
/* code that uses arr… */
```

☺

```
int *arr = (int *)malloc(10 * sizeof(int));
if (arr == NULL) {
    fprintf(stderr, "out of memory!\n");
    exit(1);
}
/* code that uses arr… */
```

What if the program no longer needs the dynamic array?

`free`

When the program no longer needs the dynamic array, we should call `free` to return the memory it occupies to the system.

(this is also known as "deallocating" the memory)

```
free( arr );
```

When we free `arr`, we can no longer use the array (of course...)

free

- When we free some memory, the memory is NOT erased or destroyed, instead, the OS is informed that we no longer need it, so it may use it for e.g. another program or some other variables in our program.
- Trying to use memory after freeing it can cause a segmentation violation (when a program is trying to read or write an illegal memory location –> program crash).

free

```
void *foo(int n) {
    int i = 7;
    int *j;
    j = (int *)malloc(n * sizeof(int));
    /* j = (int *)calloc(n, sizeof(int)); */
    return j;
} /* memory storing the variables i and j deallocated
here; block of memory pointed to by j is not */
```

free

```
void bar() {
    int *arr = foo(99);// foo creates the array
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 16;
    …
    free(arr); /* deallocate memory */
}
```

free

```
void bar() {
    int *arr = foo(99);
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 16;
    …                                      ?
    free(arr); /* deallocate memory */
}
```

Memory Leaks

- Failure to deallocate memory using `free` leads to build-up of non-reusable memory, which is no longer used by the program.
- This wastes memory resources and can lead to allocation failures when these resources are exhausted.

Memory Leaks

```
void leaker() {
    int *arr = (int *)malloc (10 * sizeof(int));
    /* Now have allocated space for 10 ints;
     * do something with it and return without
     * calling free() */
} /* arr memory is leaked here. */
```

After `leaker()` returns, nothing points to the memory allocated in the function -> memory leak ☹

Memory Leaks

```
void not_leaker() {
   int *arr = (int *)malloc (10 * sizeof(int));
   /* Now have allocated space for 10 ints;
    * do something with it */
   free(arr); /* free arr's memory */
} /* no memory leaked here. */
```

Here, we explicitly `free()` the memory allocated by `malloc()` before exiting the function -> no memory leak ☺

Memory Leaks

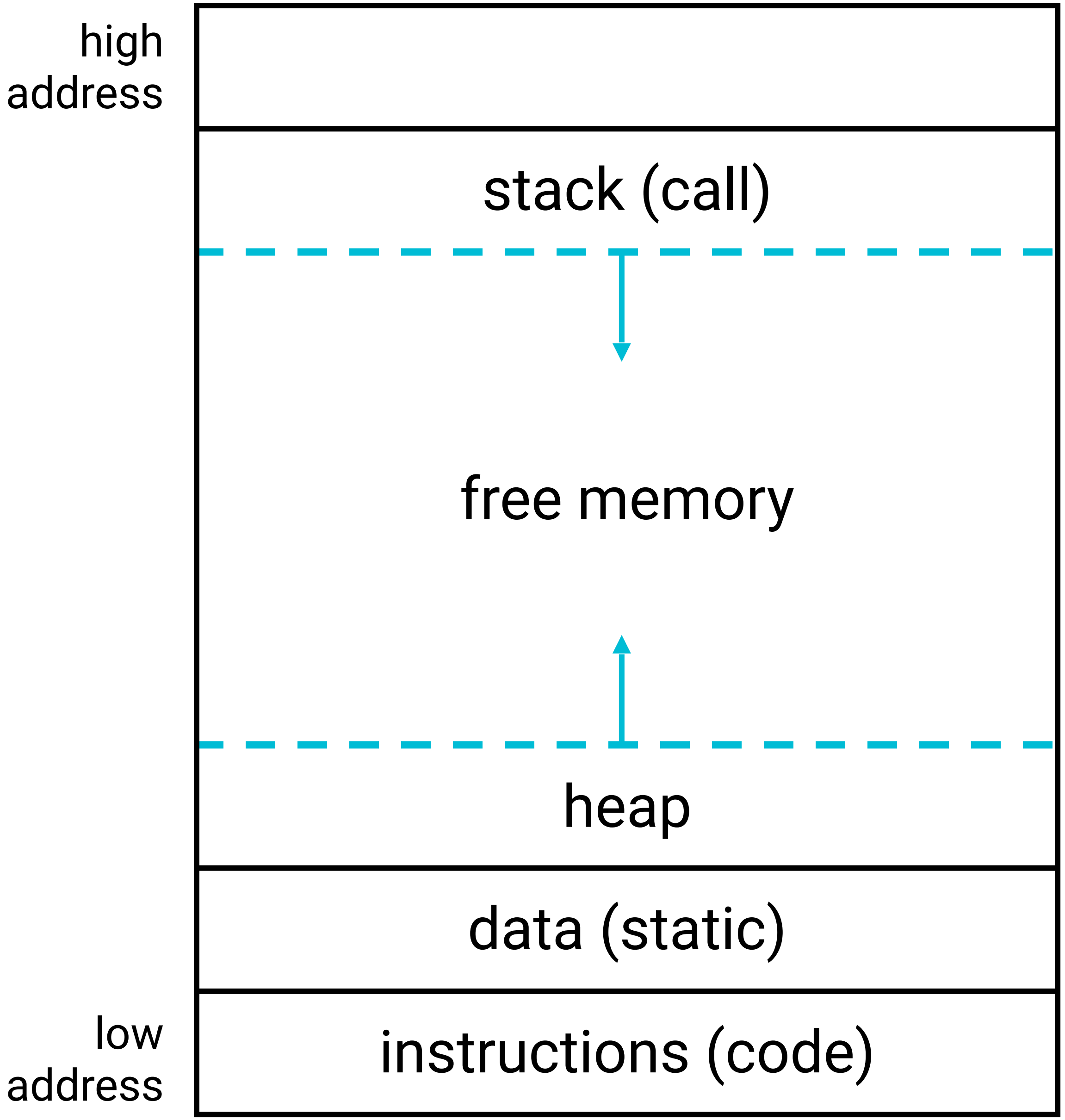
```
void not_leaker_2() {
    int arr[10];
    /* Now have allocated space for 10 ints;
     * do something with it */
} /* no memory leaked here. */
```

Here, we don’t need to `free()` the memory, because it was allocated locally (on the "stack") -> no memory leak ☺

Memory Leaks

```
void not_leaker_2() {
    int arr[10];
    /* Now have allocated s
     * do something with it
} /* no memory leaked here
```

Here, we don't need to `free()`
allocated locally (on the "stack"



Memory Leaks

```
void crasher() {
    int arr[10];
    /* Now have allocated space for 10 ints;
     * do something with it */
    free(arr); /* shocking! */)
}
```

Here, we `free()` the memory we don’t need to free

-> anything can happen 💣

Example 1

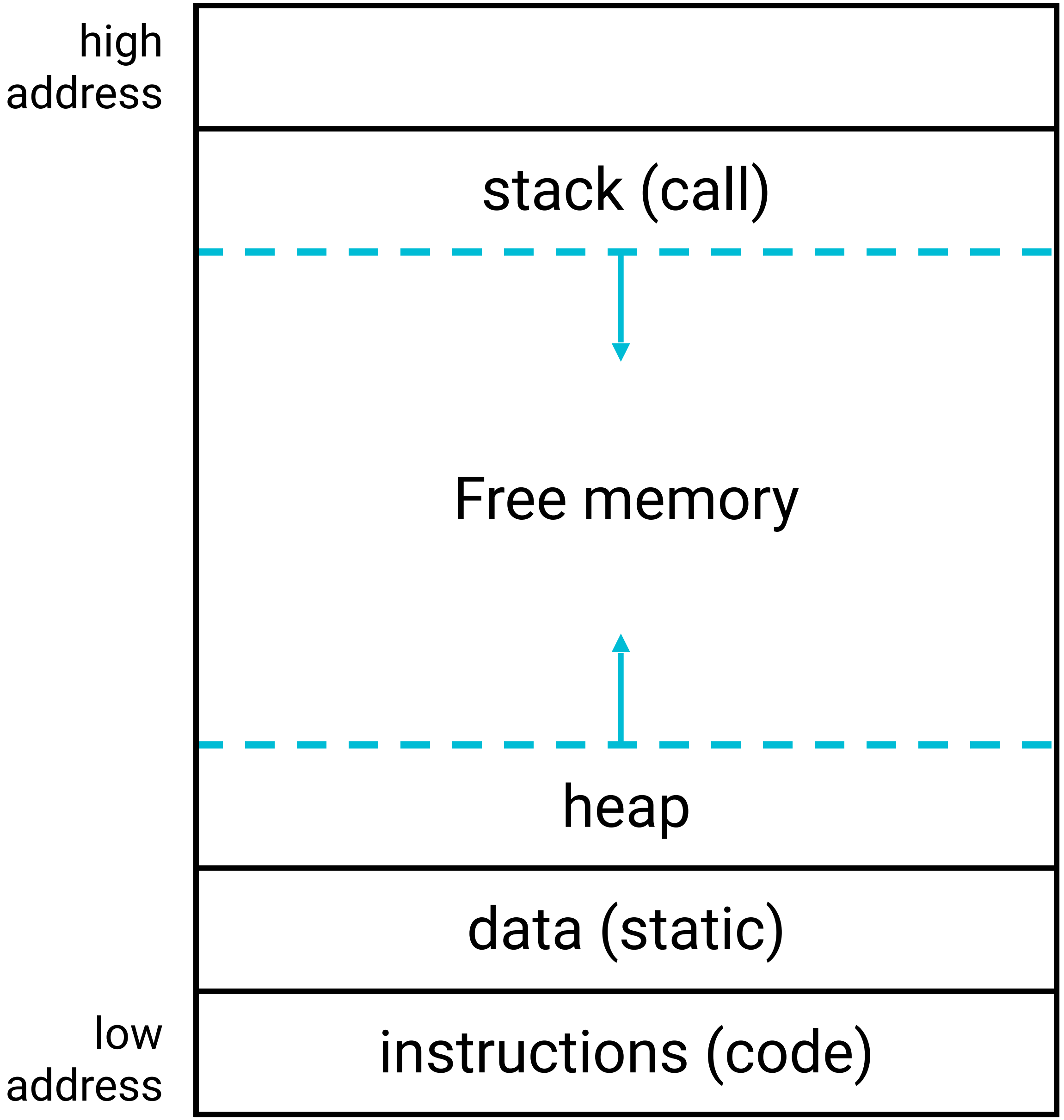
```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i, n;
    int *item;
    printf("Enter total number of items: ");
    scanf("%d", &n);
    item = (int*)calloc(n, sizeof(int));
    if (item == NULL) {
        printf("Error. Not enough space available");
        exit(1);
    }
    for (i = 0; i < n; i++) {
       scanf("%d", item+i);
    }
    for (i = 1; i < n; i++) {
        if (*item > *(item+i)) {
            *item = *(item+i);
        }
    }
    printf("Smallest item is %d\n", *item);
    free(item);
    return 0;
}
```

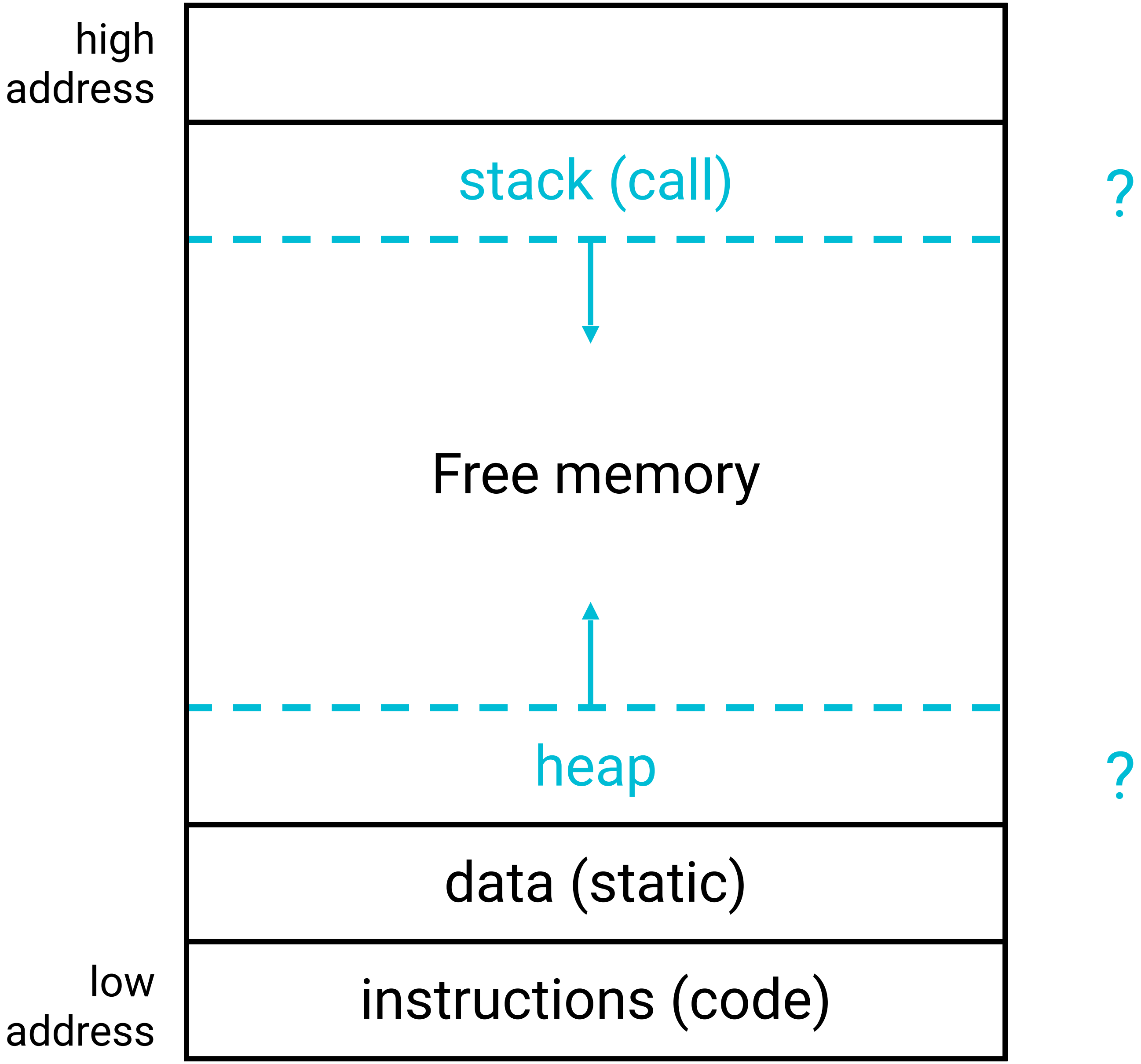
Example 1

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i, n;
    int *item;
    printf("Enter total number of items: ");
    scanf("%d", &n);
    item = (int*)calloc(n, sizeof(int));
    if (item == NULL) {
        printf("Error. Not enough space available");
        exit(1);
    }
    for (i = 0; i < n; i++) {
       scanf("%d", item+i);
    }
    for (i = 1; i < n; i++) {
        if (*item > *(item+i)) {
            *item = *(item+i);
        }
    }
    printf("Smallest item is %d\n", *item);
    free(item);
    return 0;
}
```

Output

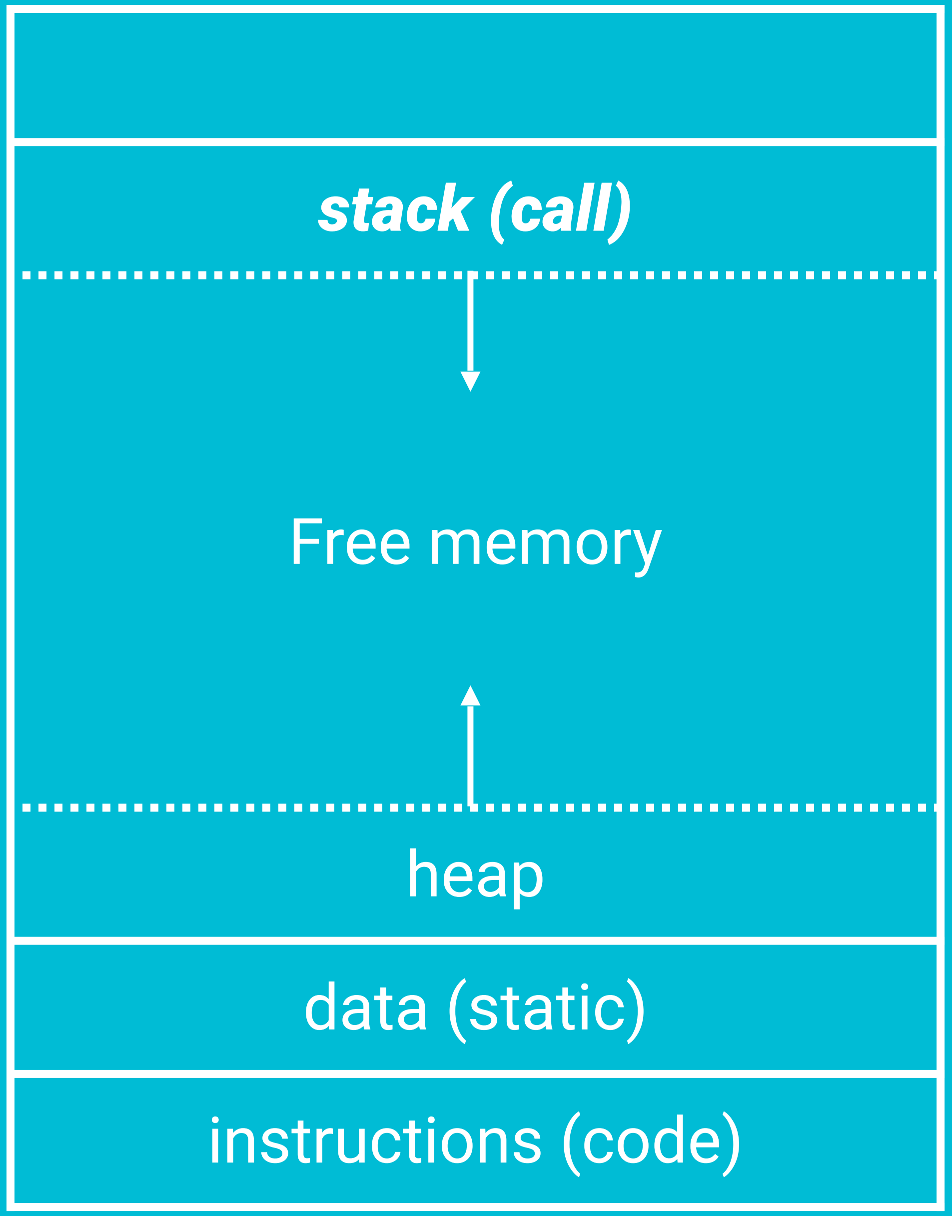
```
Enter total number of items: 5
5 2 7 3 9
Smallest item is 2
```


high
address
stack (call)
Free memory
heap
data (static)
low
address
instructions (code)


high
address
stack (call)
?
Free memory
heap
?
data (static)
low
address
instructions (code)

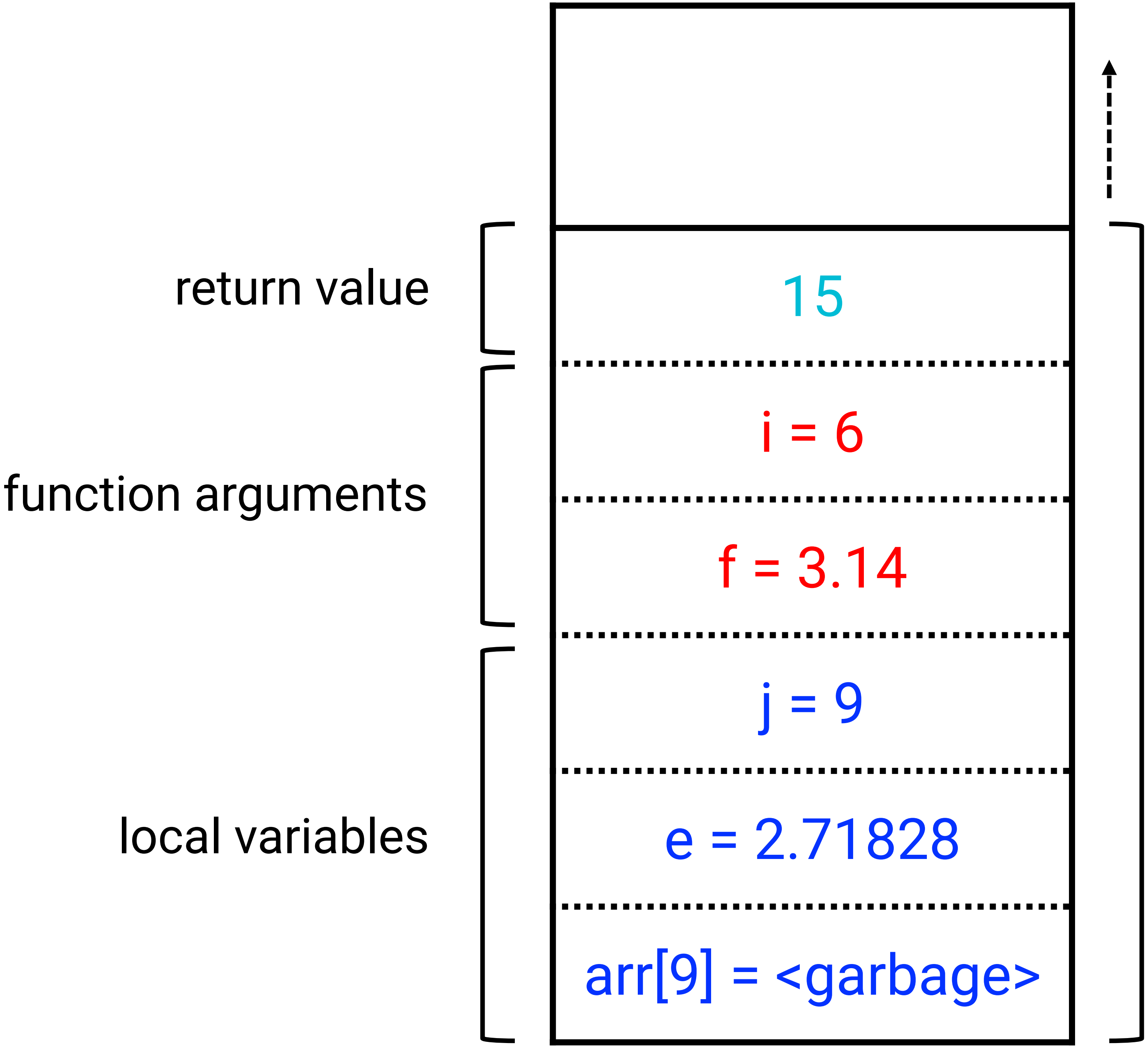
The stack

- Local variables, function arguments, return values are all stored in a ***stack***.
- Each function call generates a new "stack frame".
- After a function returns, the stack frame disappears, along with all local variables and function arguments for that invocation.



The Stack

```
int contrived_example(int i, float f) {
    int j = 9;
    double e = 2.71828;
    int arr[9];
    /* do something there, then return */
    return (i + j);
}
/* somewhere in the code */
int k = contrived_example(6, 3.14);
```



(More frames)

Stack frame

for

`contrived_example(6, 3.14)`

The Stack

```
int factorial(int i) {
    if (i == 0) {
        return 1;
    } else {
        return i * factorial(i - 1);
    }
}
```

The Stack

```c
int factorial(int i) {
    if (i == 0) {
        return 1;
    } else {
        return i * factorial(i - 1);
    }
}
```

The Stack

- What goes on the stack for `factorial(3)`?

- For each stack frame…

  - no local variables

  - one argument `i`

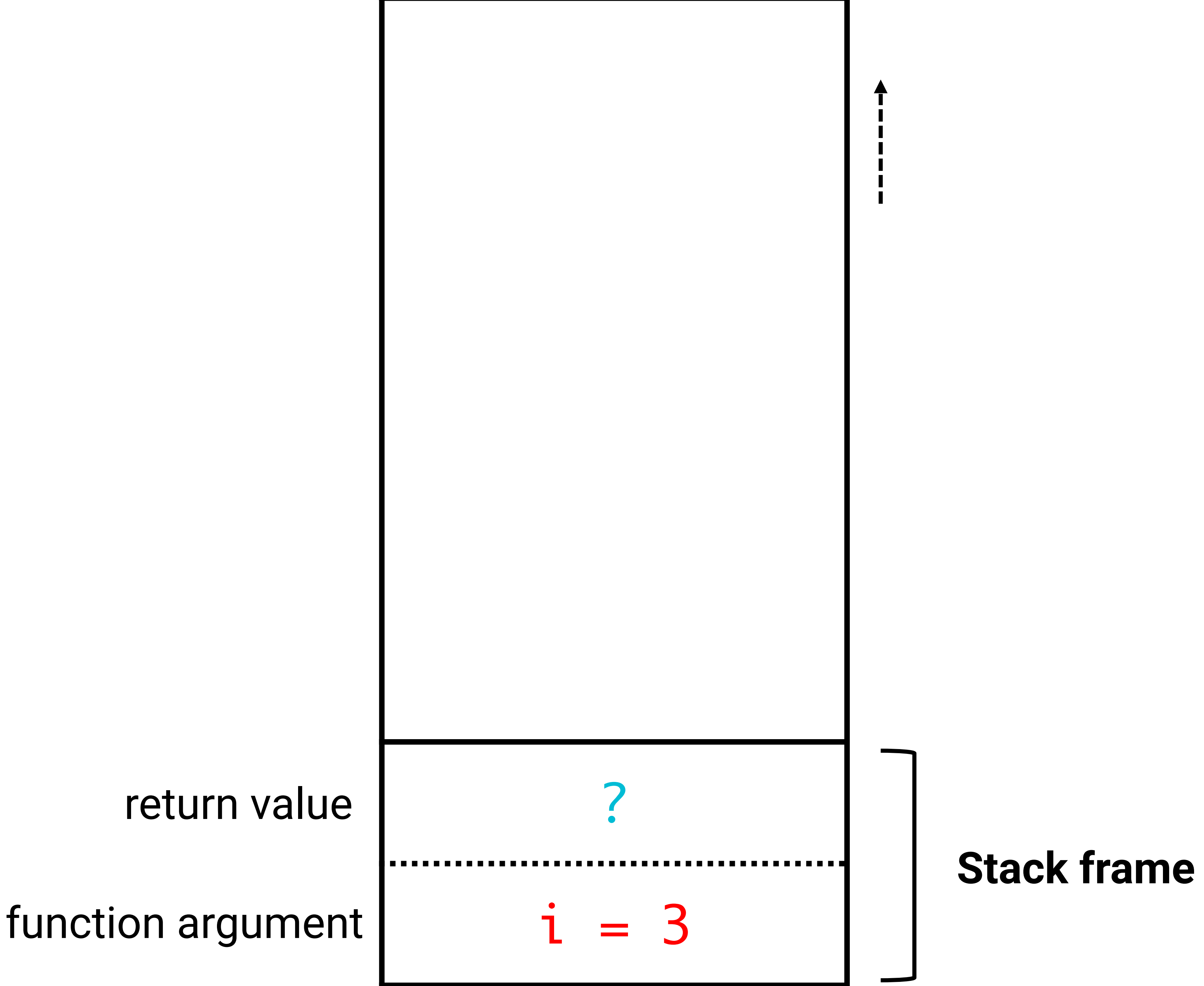
  - one return value

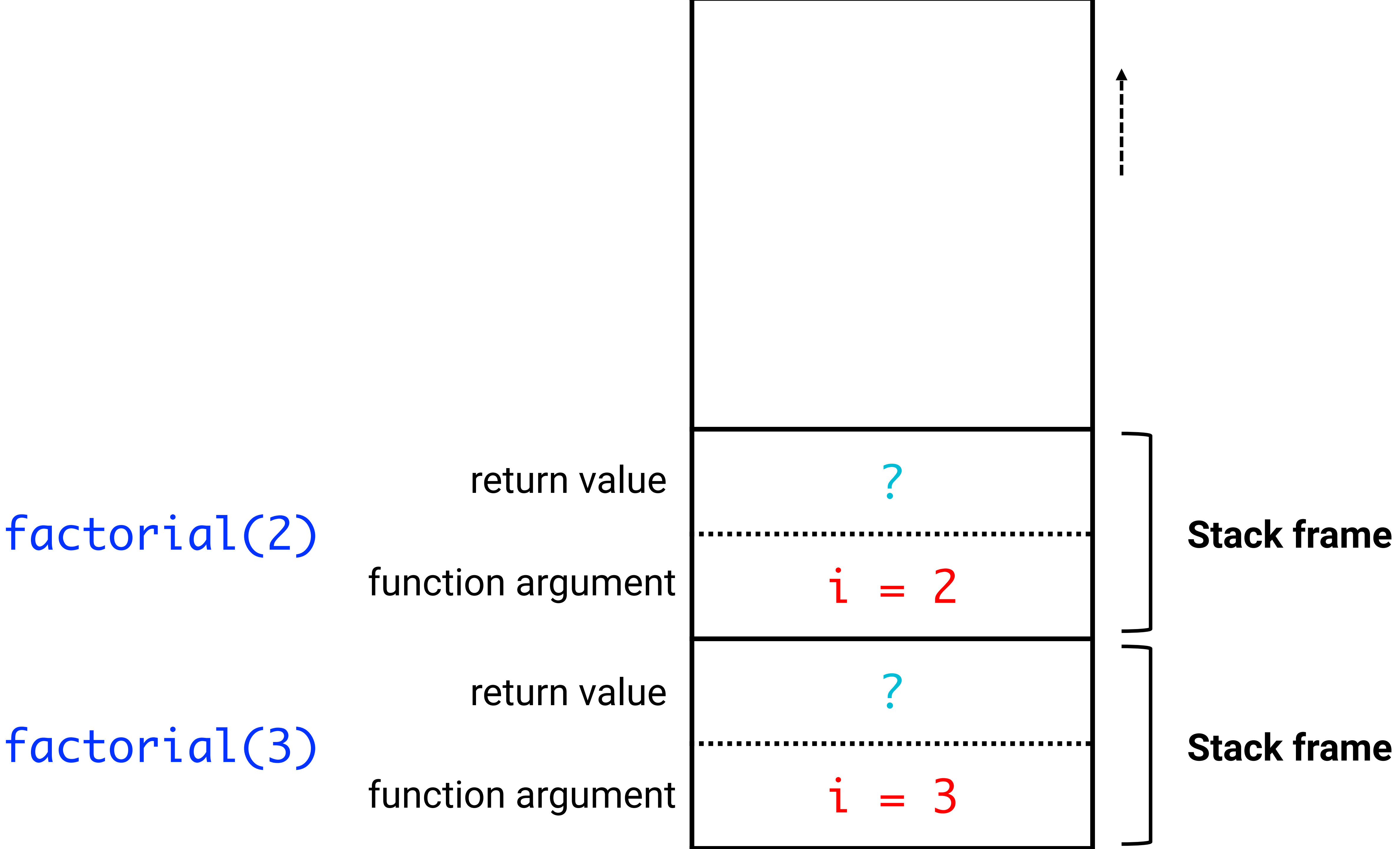
```
int factorial(int i) {
  if (i == 0) {
    return 1;
  } else {
    return i * factorial(i - 1);
  }
}
```

- Each recursive call generates a new stack frame

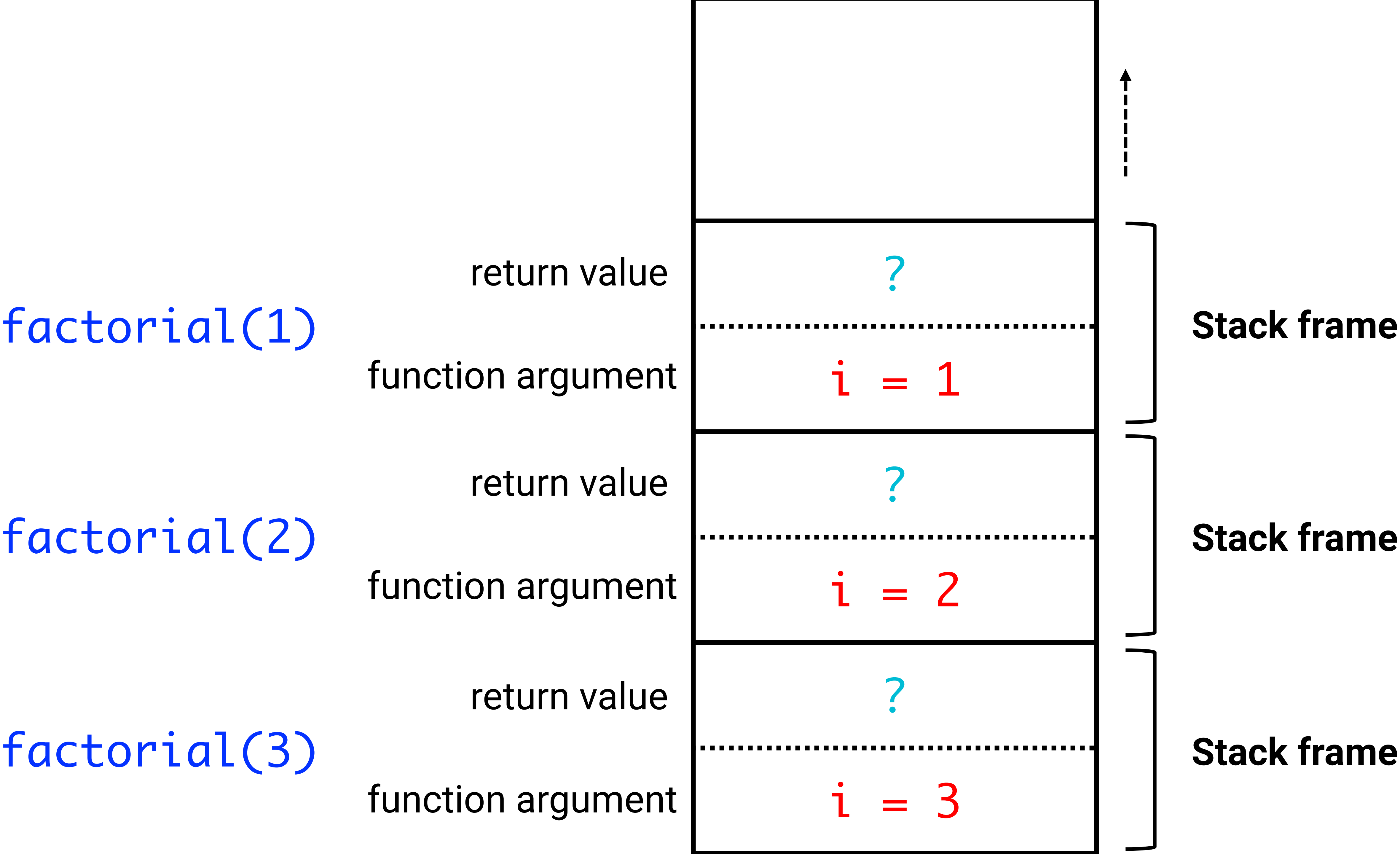
  - which disappears after the call is complete and the recursion unwinds.

`factorial(3)`

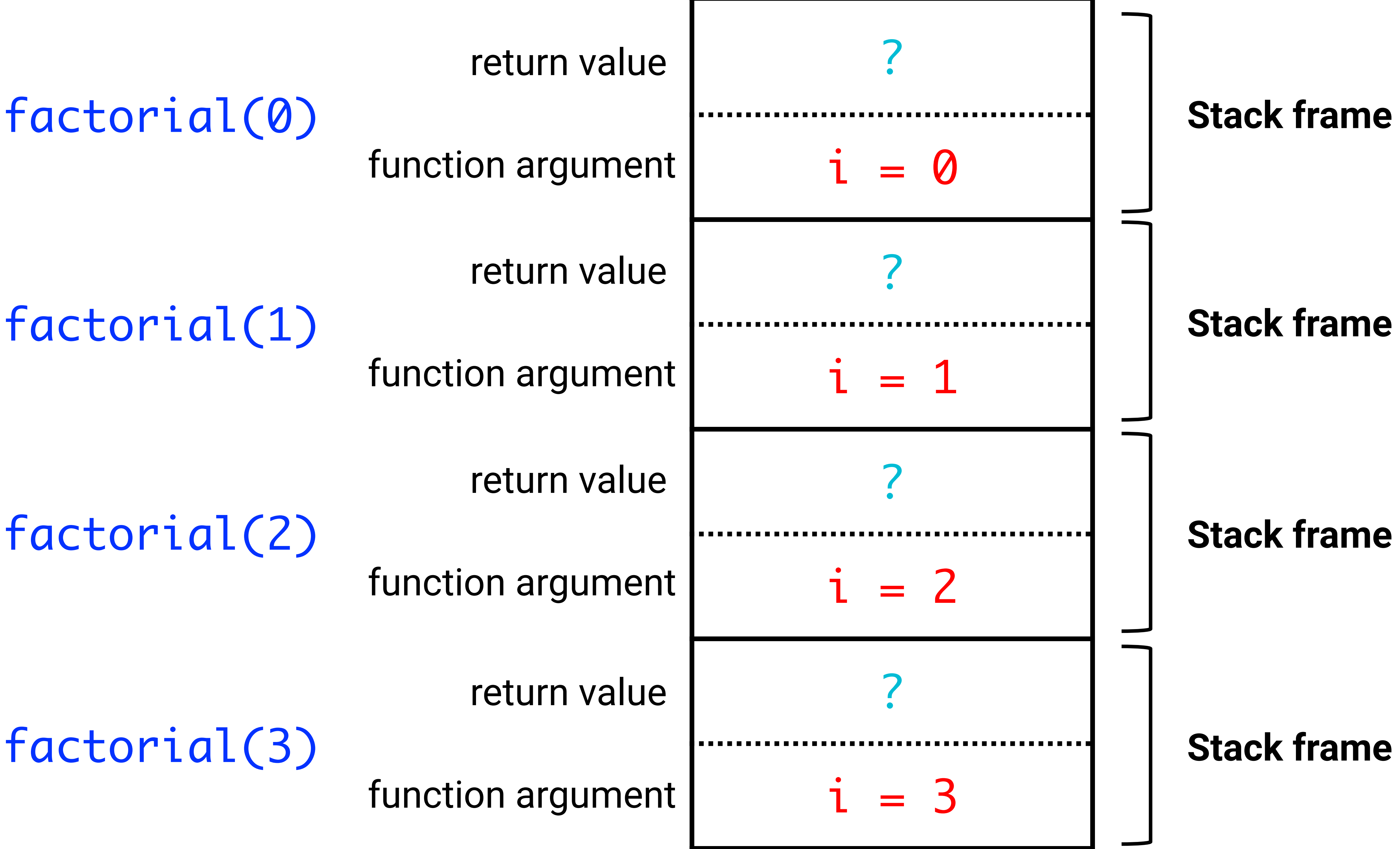


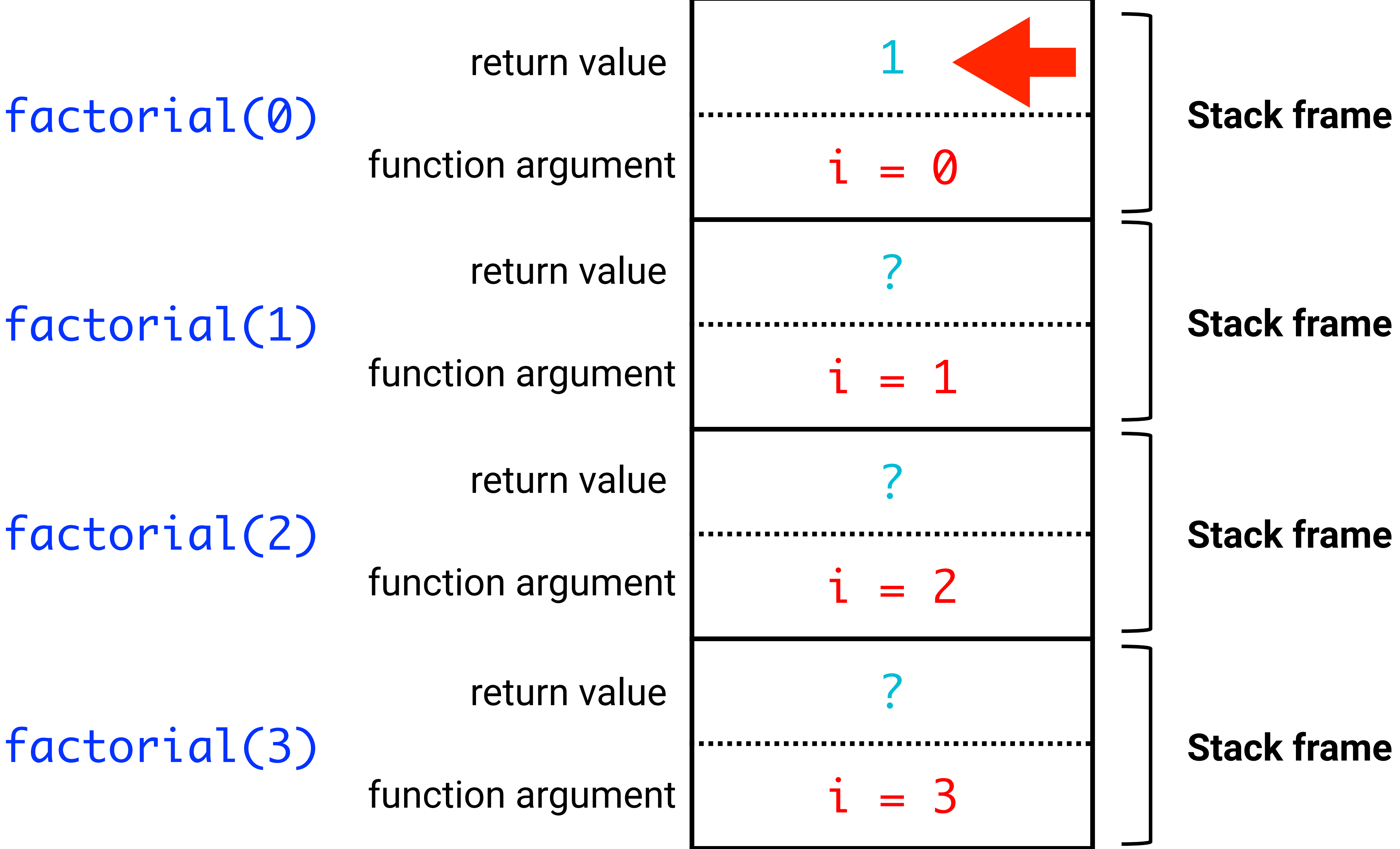


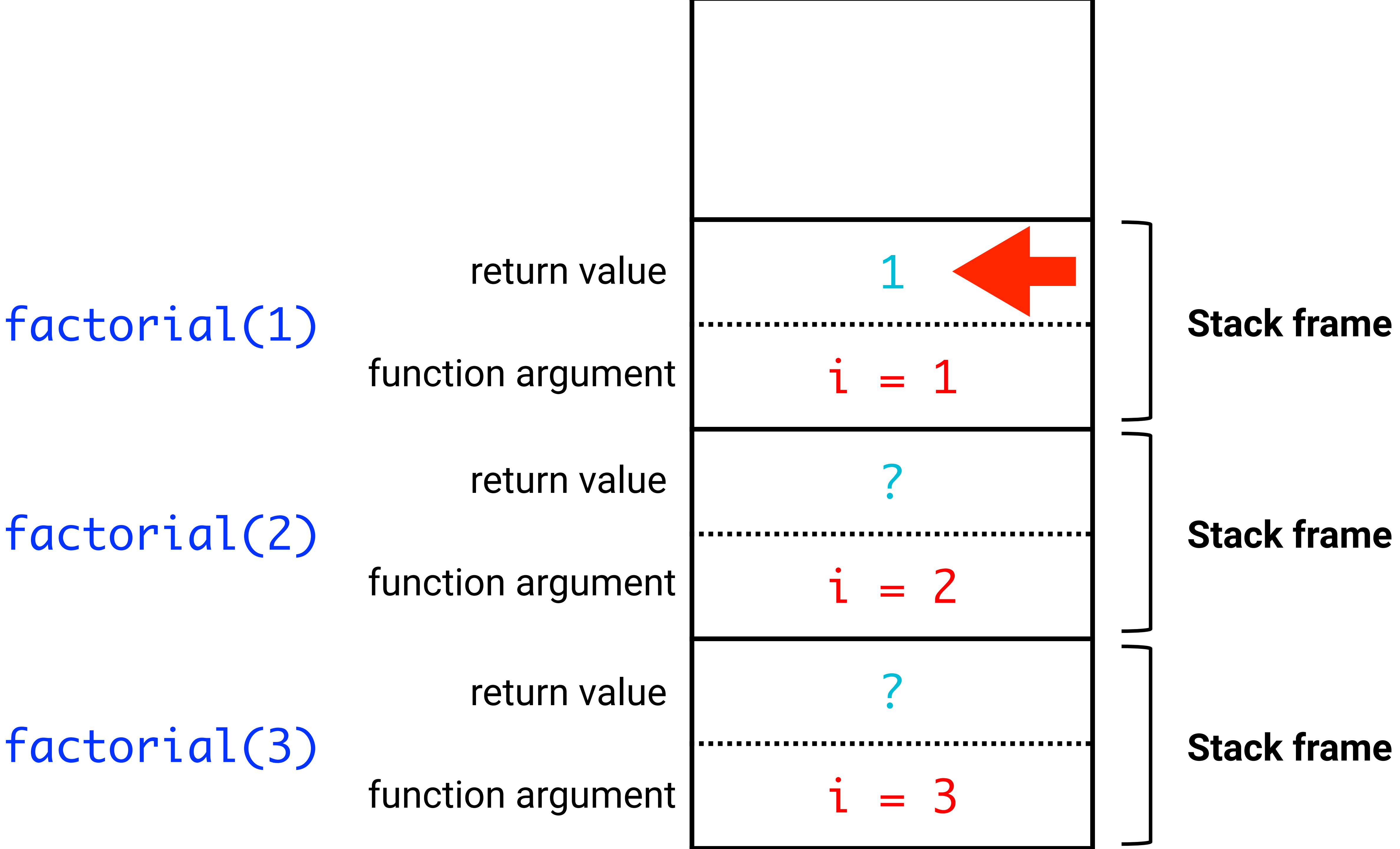
return value
factorial(2)
function argument
?
i = 2
Stack frame
return value
factorial(3)
function argument
?
i = 3
Stack frame



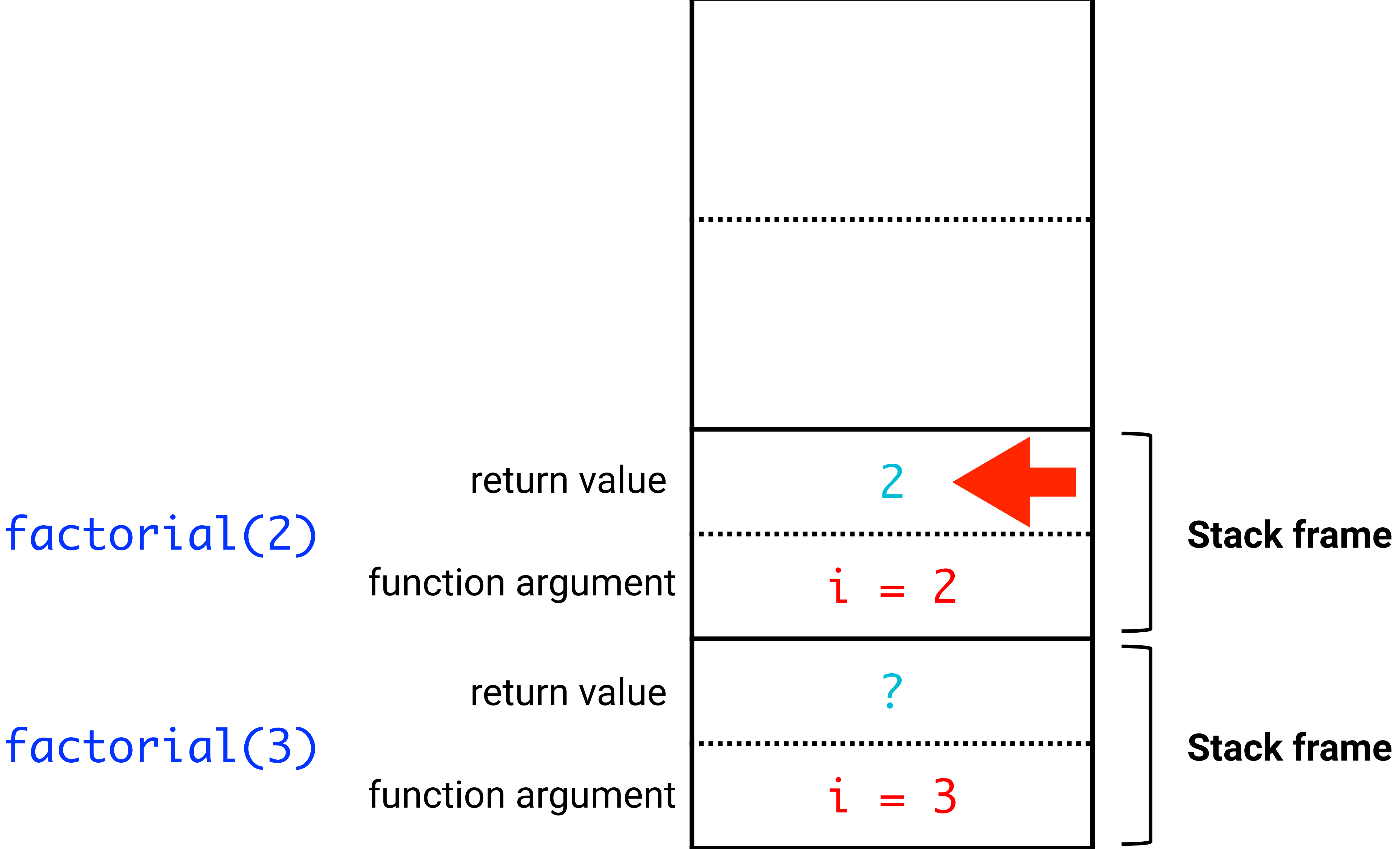
return value
?
factorial(1)
Stack frame
function argument
i = 1
return value
?
factorial(2)
Stack frame
function argument
i = 2
return value
?
factorial(3)
Stack frame
function argument
i = 3


factorial(0)
return value
?
function argument
i = 0
Stack frame
factorial(1)
return value
?
function argument
i = 1
Stack frame
factorial(2)
return value
?
function argument
i = 2
Stack frame
factorial(3)
return value
?
function argument
i = 3
Stack frame

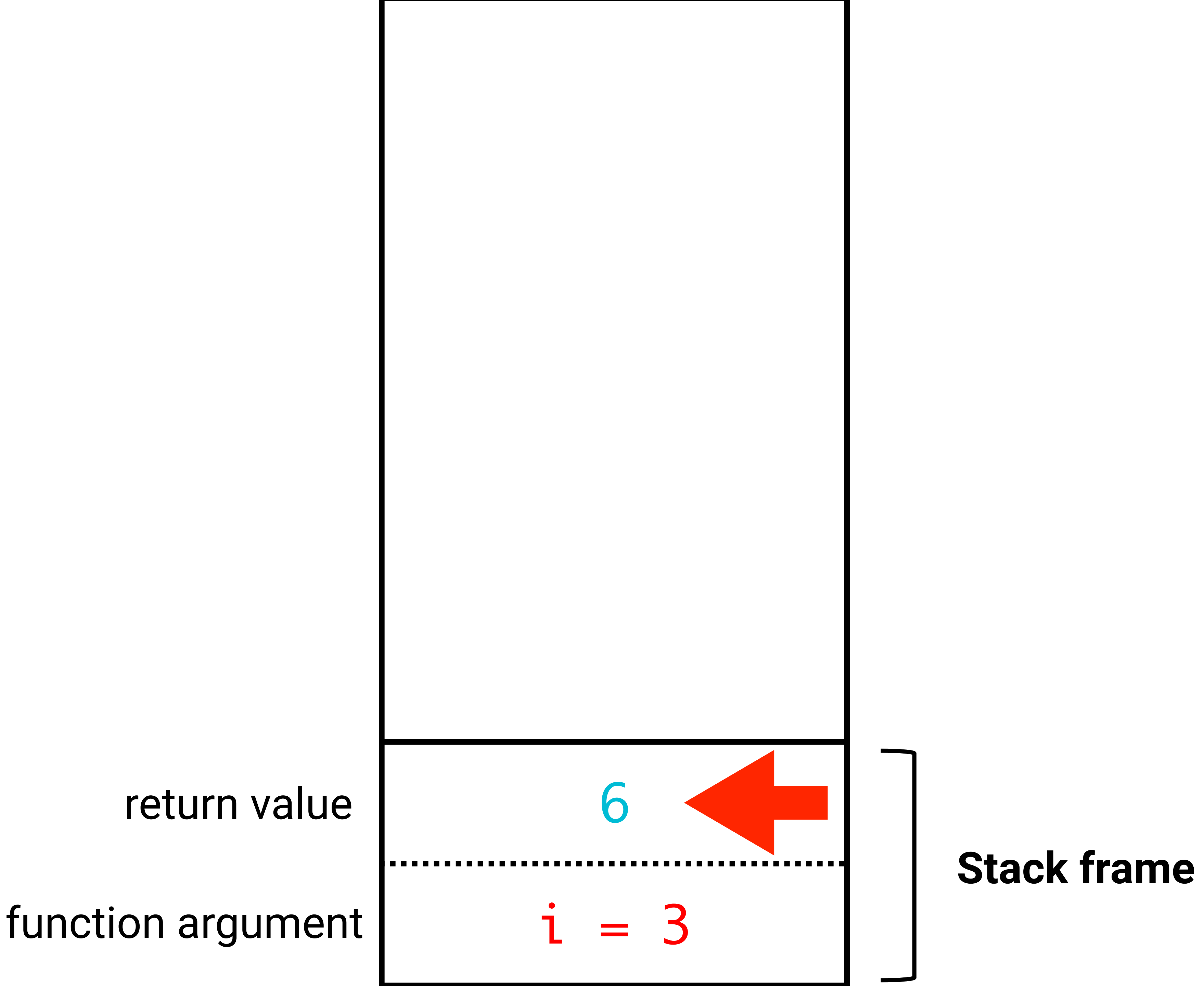

factorial(0)
return value
1
function argument
i = 0
Stack frame
factorial(1)
return value
?
function argument
i = 1
Stack frame
factorial(2)
return value
?
function argument
i = 2
Stack frame
factorial(3)
return value
?
function argument
i = 3
Stack frame



factorial(1)
return value
1
function argument
i = 1
Stack frame
factorial(2)
return value
?
function argument
i = 2
Stack frame
factorial(3)
return value
?
function argument
i = 3
Stack frame


return value
2
factorial(2)
function argument
i = 2
Stack frame
return value
?
factorial(3)
function argument
i = 3
Stack frame

`factorial(3)`

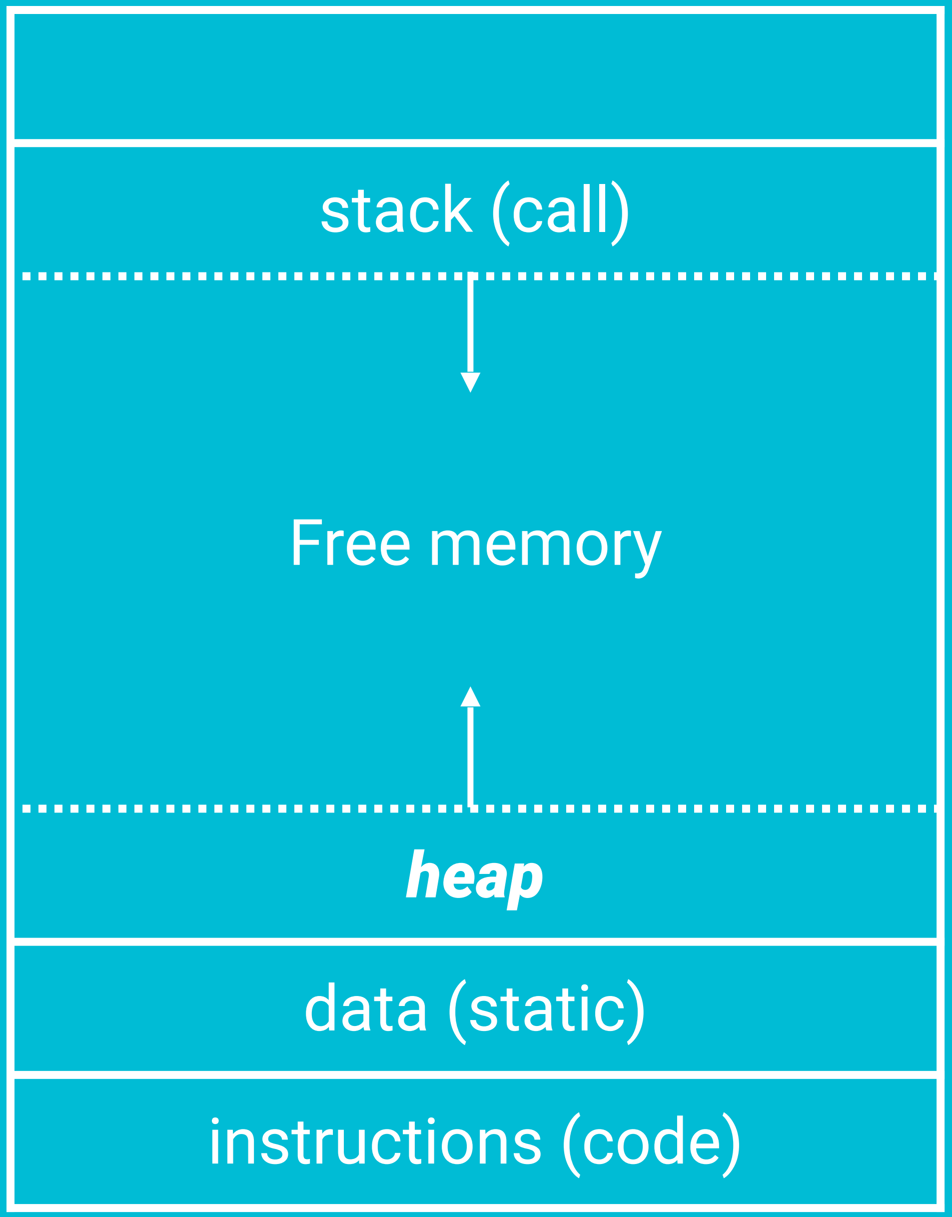


```
factorial(3)
```

result: 6

The heap

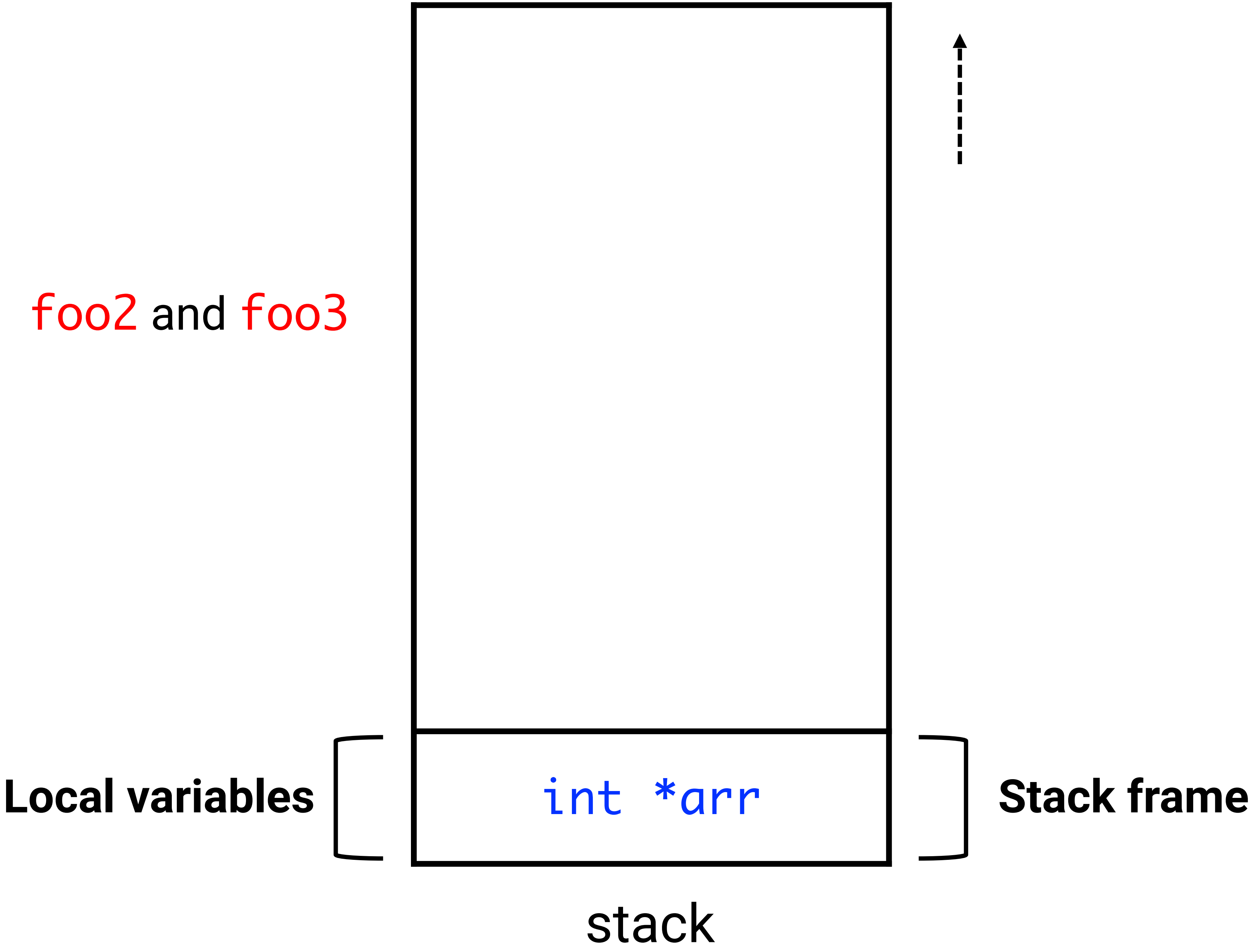
- The ***heap*** is the general pool of computer memory.
- Memory is allocated on the heap using `malloc` / `calloc`
- The ***heap*** memory must be **explicitly** freed using `free`.
- Failure to do so -> memory leak

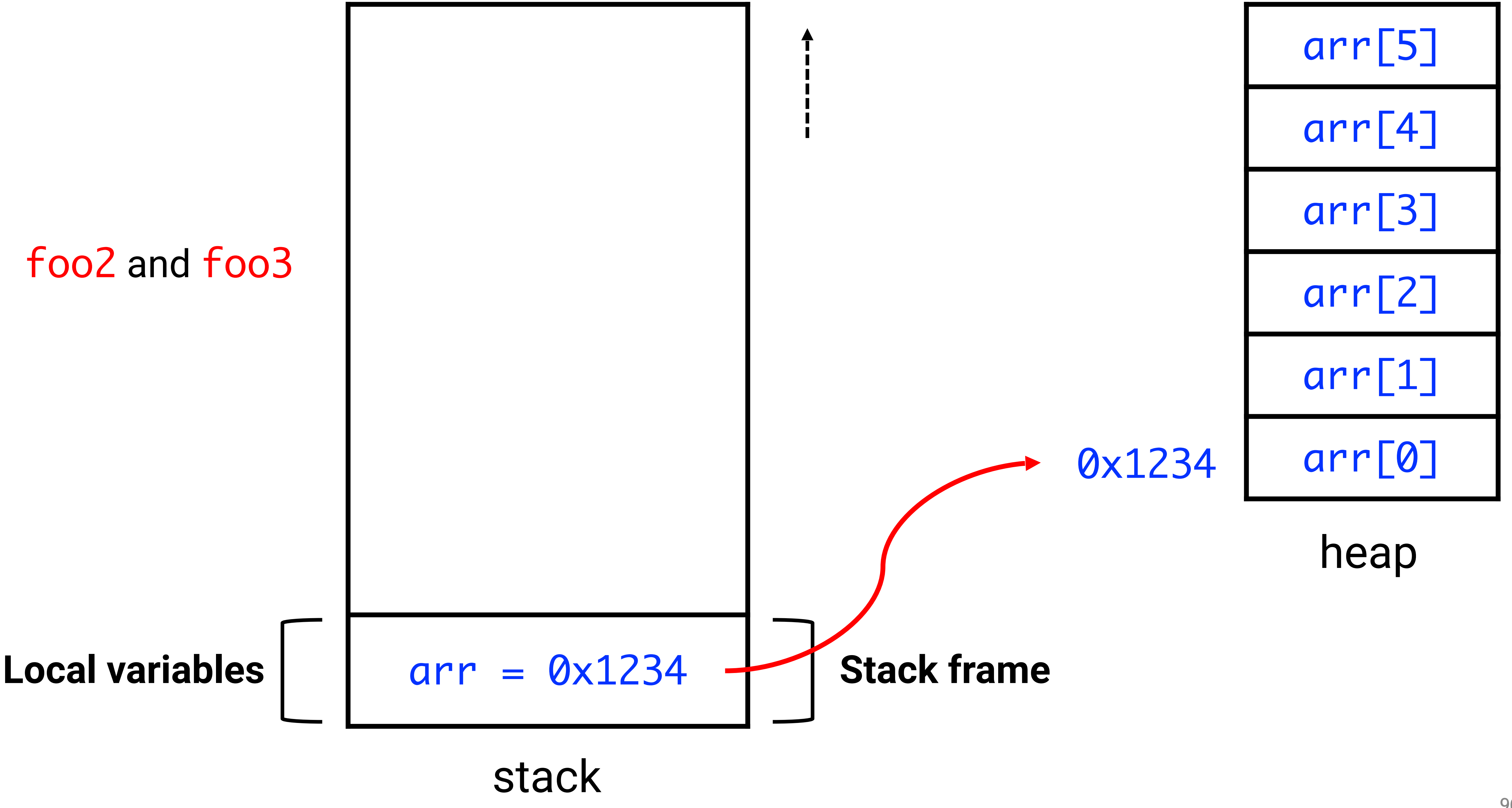


```c
void foo2() {
    int *arr;
    /* allocate memory on the heap */
    arr = (int *)calloc(6, sizeof(int));
    /* do something with arr */
}  /* arr is NOT deallocated */

void foo3() {
    int *arr;
    /* allocate memory on the heap */
    arr = (int *)calloc(6, sizeof(int));
    /* do something with arr */
    free(arr);
}
```

foo2 and foo3

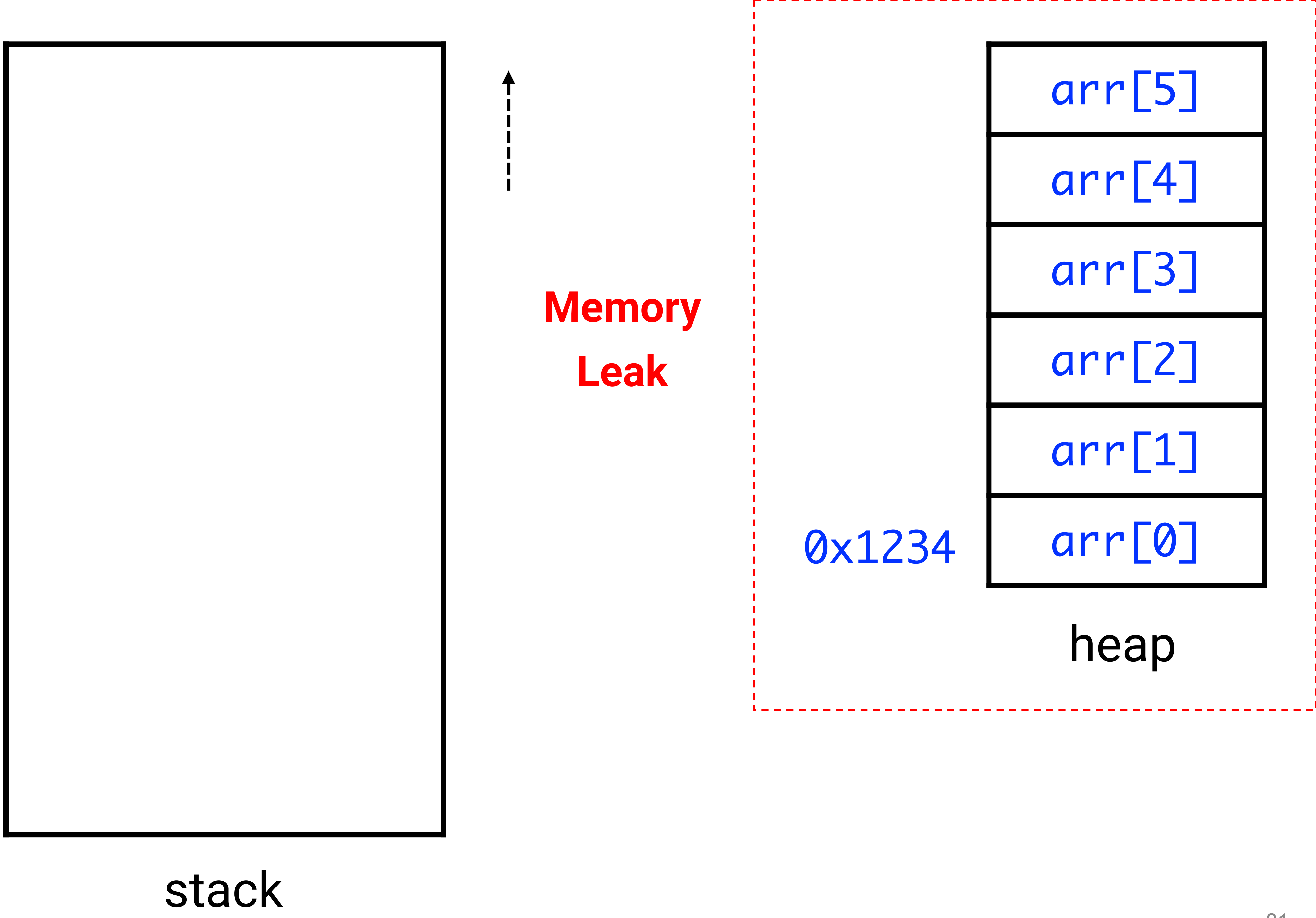



foo2 and foo3
arr[5]
arr[4]
arr[3]
arr[2]
arr[1]
0x1234
arr[0]
heap
Local variables
arr = 0x1234
Stack frame
stack

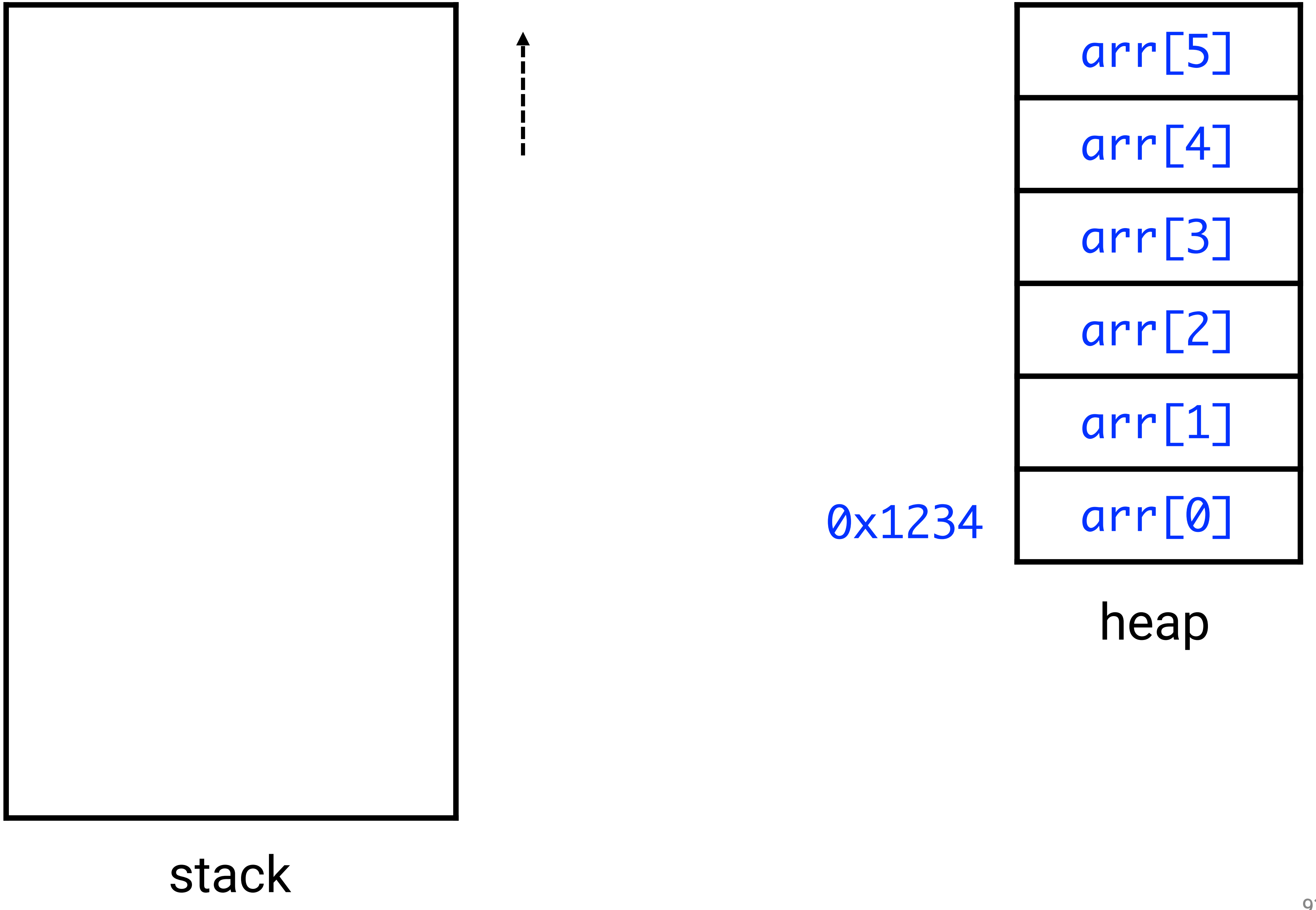
After foo2 exits,

without freeing

memory



After foo3 exits, **with** freeing memory



- Memory leaks are one of the **worst** kinds of bugs
  - Often, no harm done at all
  - Eventually may cause a long-running program to crash
    - Out of memory
  - Very hard to track down
- Special tools (e.g. `valgrind`) exist to debug memory leaks

Summary

Today

- Storage Classes

Today

- Storage Classes
  - auto, external, static, register variables

Today

- Storage Classes
- Dynamic Memory Allocation

Today

- Storage Classes
- Dynamic Memory Allocation
  - `void *malloc(size_t size);`
  - `void *calloc(size_t nmemb, size_t size);`
  - `void free(pointer);`

Today

- Storage Classes
- Dynamic Memory Allocation
  - `void *malloc(size_t size);`
  - `void *calloc(size_t nmemb, size_t size);`
  - `void free(pointer);`
  - `void *realloc(pointer, int new_size);`

    increases or decreases the size of the specified block of memory. Reallocates it if needed.

Today

- Storage Classes
- Dynamic Memory Allocation
- Stack and Heap
  - How Stack and Heap work
  - How to prevent memory leaks

Today

- Storage Classes
- Dynamic Memory Allocation
- Stack and Heap
  - How Stack and Heap work
  - How to prevent memory leaks
  - `valgrind` -> to debug memory leaks

Finally

- This is the last lecture for COMP281 this year, though the practical classes will run next week.
- Thank you very much for your hard work.
- I hope you have enjoyed COMP281

Finally

- This is the last lecture for COMP281 this year, though the practical classes will run next week.
- Thank you very much for your hard work.
- I hope you have enjoyed COMP281

Bye, for now...