CS 11 C track: lecture 5

- Last week: pointers
- This week:
  - Pointer arithmetic
  - Arrays and pointers
  - Dynamic memory allocation
  - The stack and the heap

Pointers (from last week)

- Address: location where data stored
- Pointer: variable that holds an address

```
int i = 10;

int *j = &i;

int k = 2 * (*j); /* dereference j */
```

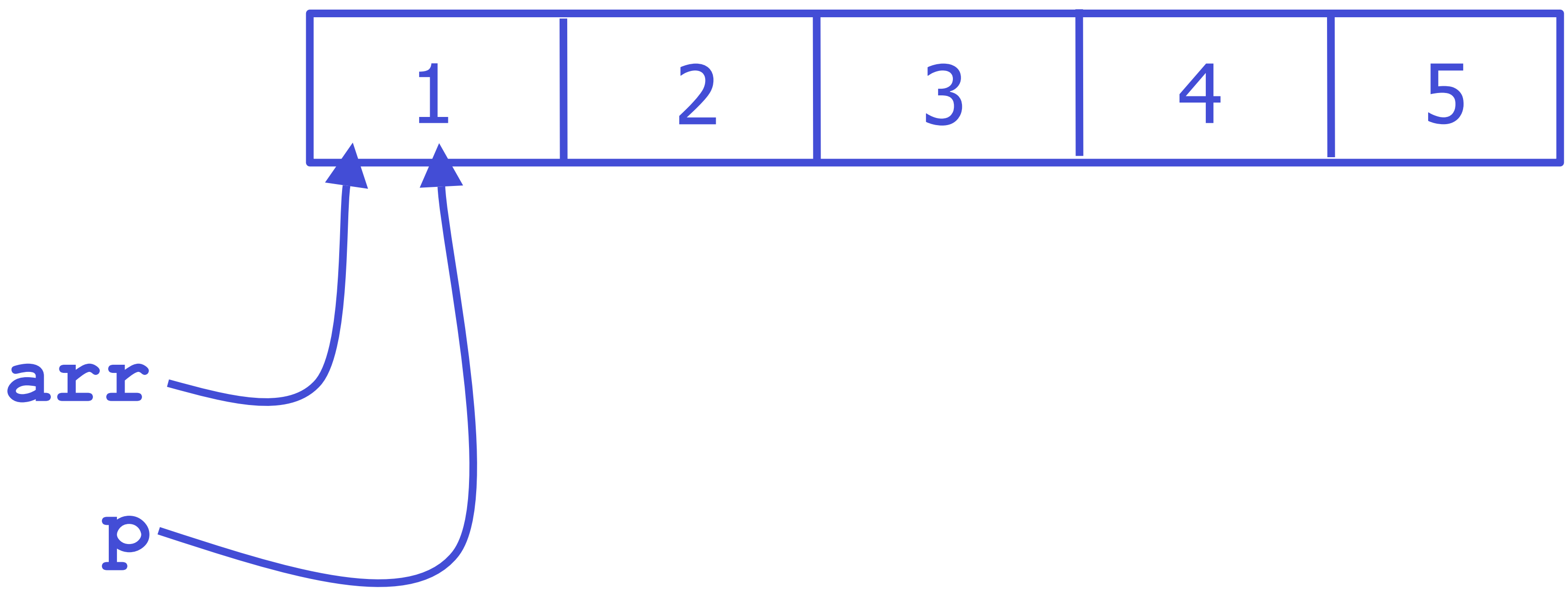
Pointer arithmetic (1)

- Can add/subtract integers to/from pointers

```
int arr[] = { 1, 2, 3, 4, 5 };
int *p = arr;   /* (*p) == ? */
p++;            /* (*p) == ? */
p += 2;         /* (*p) == ? */
p -= 3;         /* (*p) == ? */
```

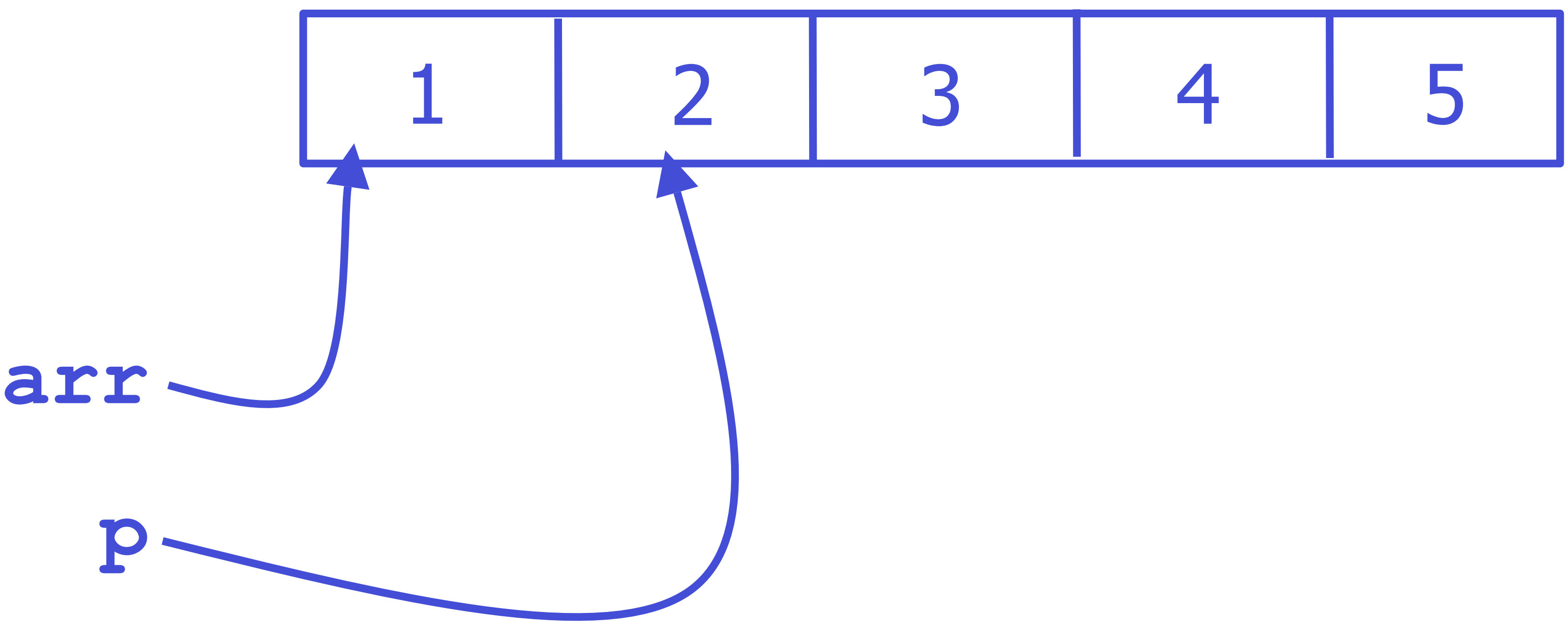
Pointer arithmetic (2)

```c
int arr[] = { 1, 2, 3, 4, 5 };
int *p = arr;  /* (*p) == ? */
```



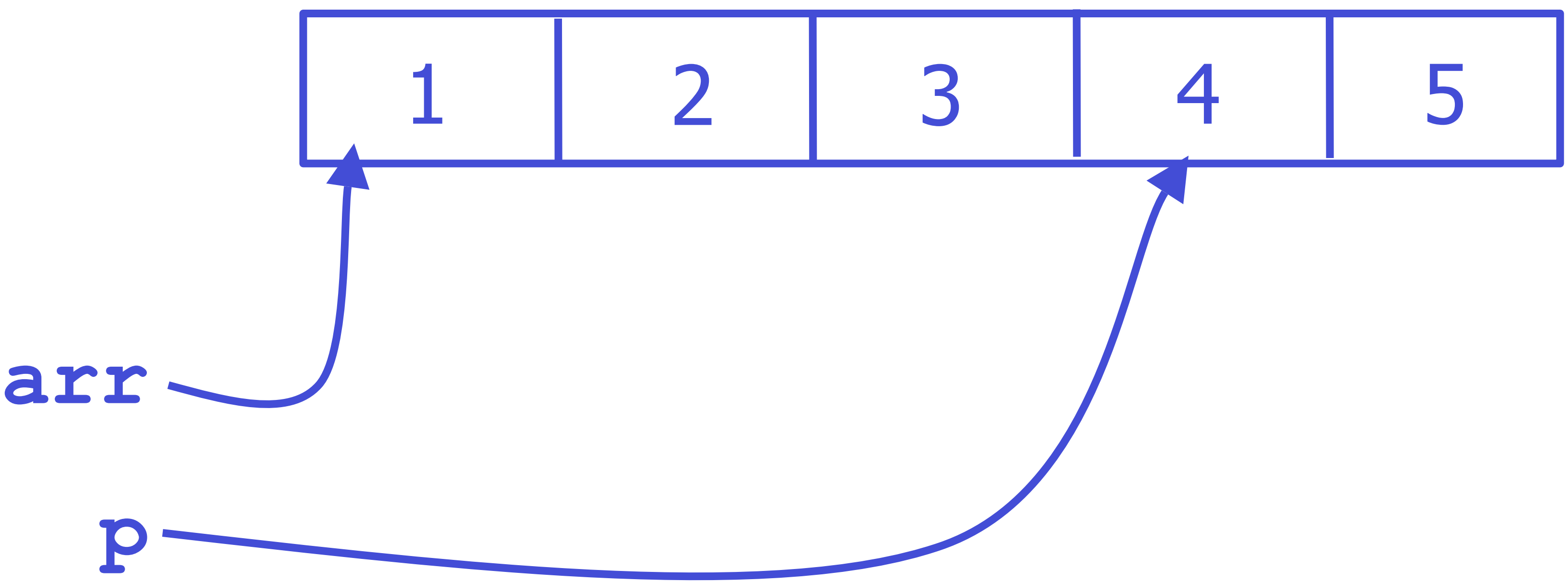
Pointer arithmetic (3)

```
p++;        /* (*p) == ? */
```



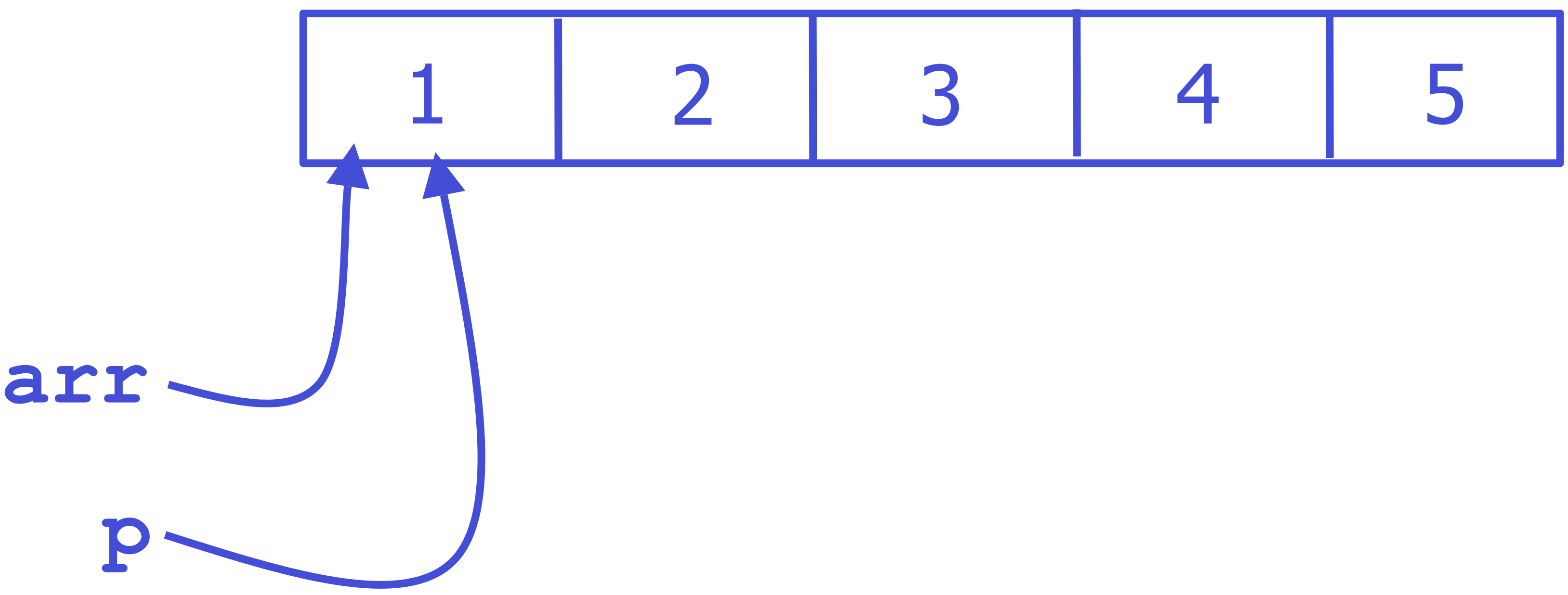
Pointer arithmetic (4)

```
p += 2;  /* (*p) == ? */
```



Pointer arithmetic (5)

```
p -= 3;    /* (*p) == ? */
```

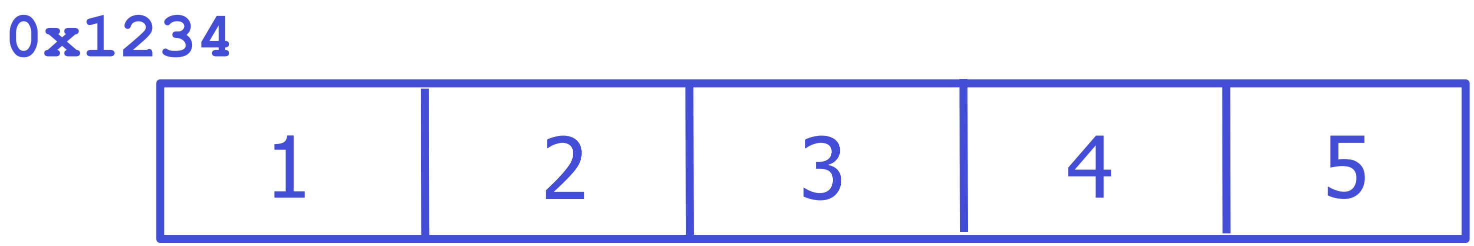


Pointer arithmetic (6)

Let's try that using addresses only...

Pointer arithmetic (7)

```c
int arr[] = { 1, 2, 3, 4, 5 };
int *p = arr;  /* (*p) == ? */
```

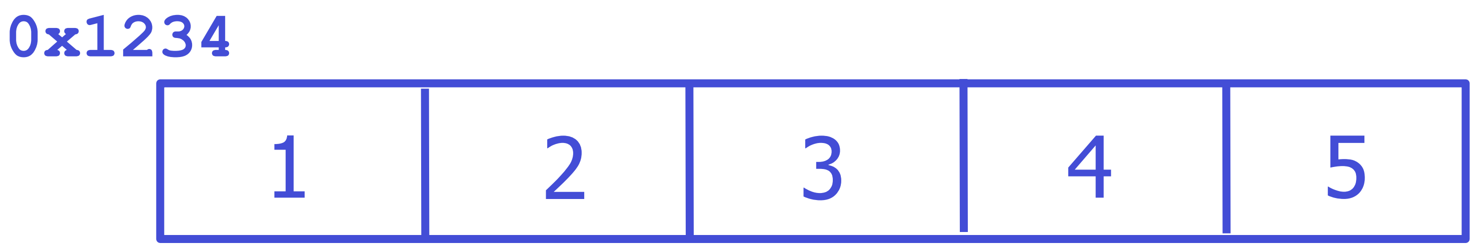



```
arr  0x1234

  p  0x1234
```

Pointer arithmetic (8)

```
p++;        /* (*p) == ? */
```




```
arr  0x1234

  p  0x1238
```

(assume 4 byte integers)

Pointer arithmetic (9)

```
p += 2;      /* (*p) == ? */
```

0x1234


```
arr  0x1234

  p  0x1240
```

(0x1240 = 0x1234 + 0x0c;

0x0c == 12 decimal or 3x4)

Pointer arithmetic (10)

```
p-= 3;        /* (*p) == ? */
```

0x1234

1	2	3	4	5

```
arr  0x1234

  p  0x1234
```

Pointer arithmetic (11)

- Get size of a type using the **sizeof** operator:

```
printf("size of integer: %d\n",
        sizeof(int));
printf("size of (int *): %d\n",
        sizeof(int *));
```

- N.B. **sizeof** is not a function
  - takes a type name as an argument!

Pointer arithmetic (12)

- N.B. pointer arithmetic doesn't add/ subtract address directly but in multiples of the size of the type in bytes

```
int arr[] = { 1, 2, 3, 4, 5 };
int *p = arr;
p++; /* means: p = p + sizeof(int);*/
```

Pointer arithmetic (13)

```
p++;        /* (*p) == ? */
```

```
0x1234
```

1	2	3	4	5

```
arr  0x1234

  p  0x1238
```

(j = 0x1234 + sizeof(int) = 0x1238,

not 0x1235)

Arrays and pointers (1)

- Arrays are pointers in disguise!
  - Arrays: "syntactic sugar" for pointers

```
int arr[] = {1, 2, 3, 4, 5};
printf("arr[3] = %d\n", arr[3]);
printf("arr[3] = %d\n", *(arr + 3));
```

- `arr[3]` and `*(arr + 3)` are identical!
- `arr` is identical to `&arr[0]`

Arrays and pointers (2)

- Can use pointer arithmetic wherever we use array operations; consider this:

```
int i;
double array[1000];
for (i = 1; i < 999; i++) {
  array[i] = (array[i-1] +
    array[i] + array[i+1]) / 3.0;
}
```

Arrays and pointers (3)

- Exactly the same as:

```
int i;
double array[1000];
for (i = 1; i < 999; i++) {
   *(array+i) = (*(array+i-1) +
      *(array+i) + *(array+i+1)) / 3.0;
}
```

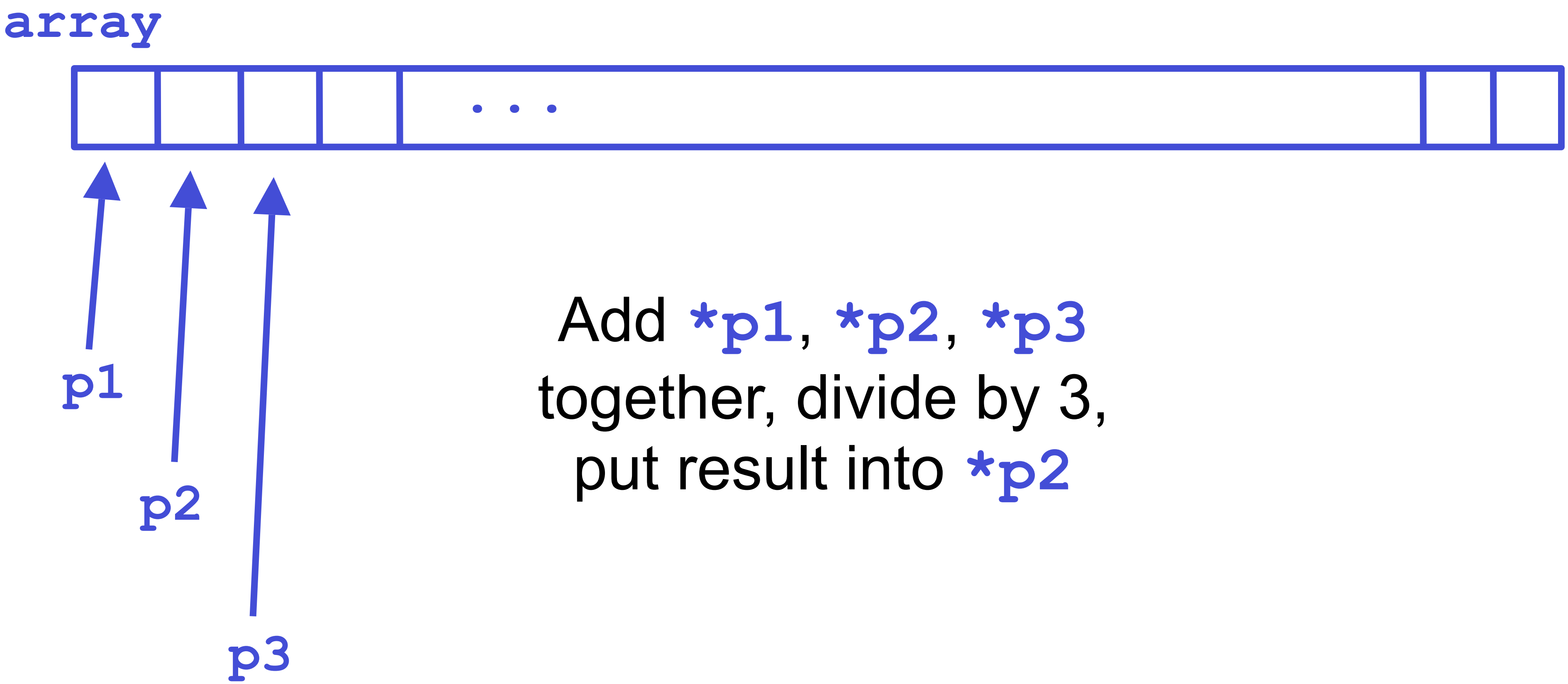
Arrays and pointers (4)

- When you say `*(array + i)`, you have to add `i` to `array` and dereference
- For large values of `i`, this is relatively slow
- Incrementing pointers by 1 is faster than adding a large number to a pointer
- Can use this fact to optimize the preceding code in an interesting way

Arrays and pointers (5)

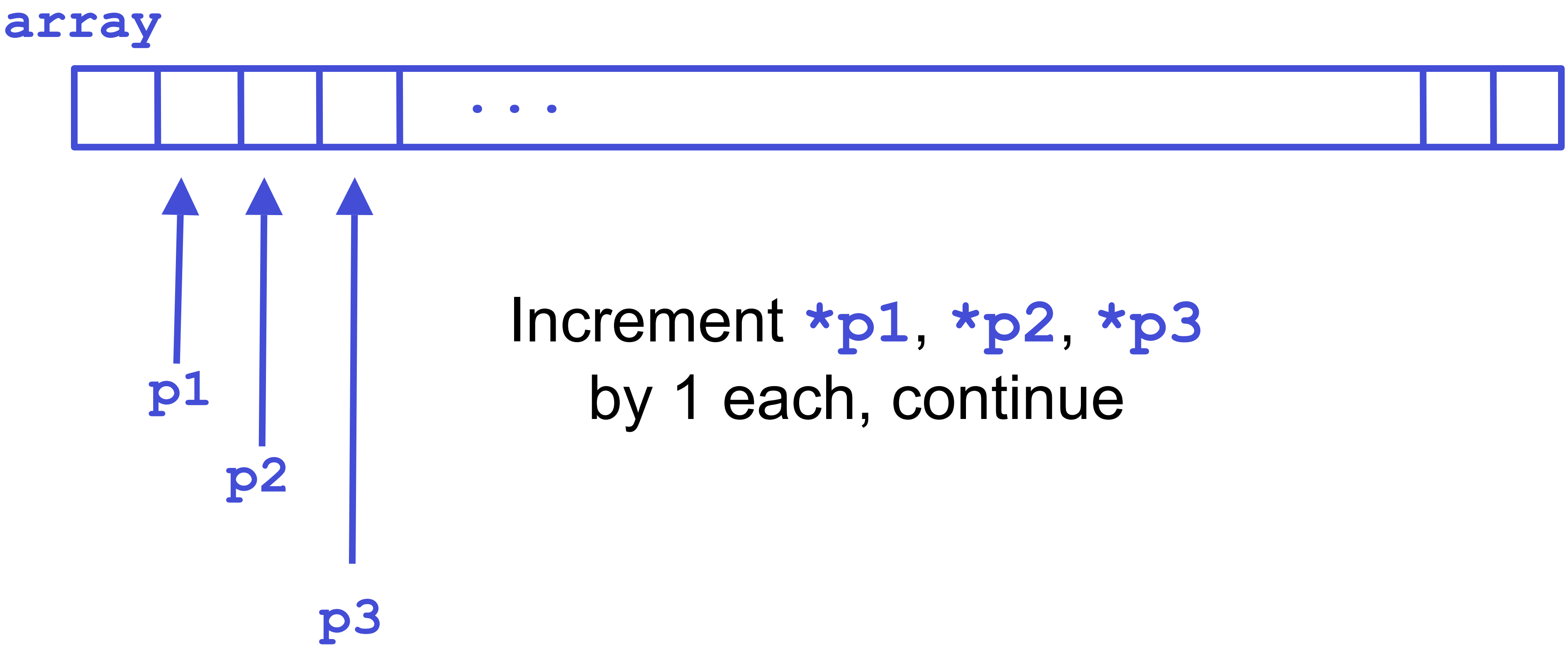
```
double array[1000];
double *p1, *p2, *p3;
p1=array; p2=array+1; p3=array+2;
for (i = 1; i < 999; i++) {
  *p2 = (*p1 + *p2 + *p3) / 3.0;
  p1++; p2++; p3++;
}
```

Arrays and pointers (6)



Add *p1, *p2, *p3 together, divide by 3, put result into *p2

Arrays and pointers (7)



Increment **`*p1`**, **`*p2`**, **`*p3`**
by 1 each, continue

Arrays and pointers (8)

- We replaced 3 pointer additions with three pointer increments, which are usually faster

- Even more significant for 2-d arrays

Dynamic memory allocation (1)

- Recall that we can't do this:

```
int n = 10;
int arr[n];  /* not legal C */
```

- However, often want to allocate an array where size of array not known in advance
- This is known as "dynamic memory allocation"
  - dynamic as opposed to "static" (size known at compile time)

Dynamic memory allocation (2)

- Let's say we want to allocate memory for e.g. arrays "on the fly"
- Later will have to deallocate memory
- Three new library functions for this:
  - `void *malloc(int size)`
  - `void *calloc(int nitems, int size)`
  - `void free(void *ptr)`
- All found in `<stdlib.h>` header file

`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`/`calloc` return `void *`
- `free` takes a `void *` argument

Using `malloc()` (1)

- `malloc()` stands for "memory allocator"
- `malloc()` takes one argument: the size of the chunk of memory to be allocated in bytes
  - recall: a byte == 8 bits
  - an `int` is 32 bits or 4 bytes
- `malloc()` returns the address of the chunk of memory that was allocated

Using `malloc()` (2)

- `malloc()` is often used to dynamically allocate arrays
- For instance, to dynamically allocate an array of 10 `ints`:

```
int *arr;
arr = (int *) malloc(10 * sizeof(int));
/* now arr has the address
   of an array of 10 ints */
```

Using `calloc()` (1)

- `calloc()` is a variant of `malloc()`
- `calloc()` takes two arguments: the number of "things" to be allocated and the size of each "thing" (in bytes)
- `calloc()` returns the address of the chunk of memory that was allocated
- `calloc()` also sets all the values in the allocated memory to zeros (`malloc()` doesn't)

Using `calloc()` (2)

- `calloc()` is also used to dynamically allocate arrays
- For instance, to dynamically allocate an array of 10 `ints`:

```
int *arr;
arr = (int *) calloc(10, sizeof(int));
/* now arr has the address
   of an array of 10 ints, all 0s */
```

`malloc/calloc` return value (1)

- `malloc` and `calloc` both return the address of the newly-allocated block of memory
- However, they are not guaranteed to succeed!
  - maybe there is no more memory available
- If they fail, they return `NULL`
- You must always check for NULL when using `malloc` or `calloc`
  - We sometimes leave it out here for brevity

`malloc/calloc` return value (2)

- bad:

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

- good:

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

- Always do this!

`malloc()` vs. `calloc()`

- `malloc`/`calloc` both allocate memory
- `calloc` has slightly different syntax
  - as we've seen
- Most importantly: `calloc()` zeros out allocated memory, `malloc()` doesn't.
- `calloc()` a tiny bit slower
- I prefer `calloc()`

Using `free()` (1)

- `malloc()` and `calloc()` return the address of the chunk of memory that was allocated
- Normally, we store this address in a pointer variable
- When we have finished working with this chunk of memory, we "get rid of it" by calling the `free()` function with the pointer variable as its argument
- This is also known as "deallocating" the memory or just "freeing" it

Using `free()` (2)

```
int *arr;
arr = (int *) calloc(10, sizeof(int));
/* now arr has the address
   of an array of 10 ints, all 0s */
/* Code that uses the array... */
/* Now we no longer need the array, so "free"
  it: */
free(arr);
/* Now we can't use arr anymore. */
```

Using `free()` (3)

- NOTE: When we `free()` some memory, the memory is not erased or destroyed
- Instead, the operating system is informed that we don't need the memory any more, so it may use it for e.g. another program
- Trying to use memory after freeing it can cause a segmentation violation (program crash)

Dynamic memory allocation (3)

```
#include <stdlib.h>
int *foo(int n) {
  int i[10];  /* memory allocated here */
  int i2[n];  /* ERROR: NOT VALID! */
  int *j;
  j = (int *)malloc(n * sizeof(int));
  /* Alternatively: */
  /* j = (int *)calloc(n, sizeof(int)); */
  return j;
} /* i's memory deallocated here; j's not */
```

Dynamic memory allocation (4)

```
void bar(void) {
  int *arr = foo(10);
  arr[0] = 10;
  arr[1] = 20;
  /* ... do something with arr ... */
  free(arr);  /* deallocate memory */
}
```

- Not calling `free()` leads to memory leaks !

Memory leaks (1)

```
void leaker(void) {
  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 memory allocated in the function → memory leak

Memory leaks (2)

```
void not_leaker(void) {
  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 leak. */
```

- Here, we explicitly `free()` the memory allocated by `malloc()` before exiting the function.

Memory leaks (3)

```
void not_leaker2(void) {
  int arr[10];
  /* Now have allocated space for 10 ints;
   * do something with it.
   */
} /* No leak. */
```

- Here, we don't have to `free()` the memory, since it was allocated locally (on the "stack").
- "What's the stack?" (you may ask...)

Memory leaks (4)

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

- Here, we `free()` memory we don't need to free!
- Anything can happen (e.g. core dump)

Memory leaks (5)

- Rules of thumb:
- 1) Any time you allocate memory using `malloc()` or `calloc()`, you must eventually call `free()` on that memory
- 2) You must `free()` the exact same pointer (address) that was returned from `malloc()` or `calloc()`
- 3) You don't have to `free()` the memory in the same function as the one where `malloc/calloc` was called

The stack and the heap (1)

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

The stack and the heap (2)

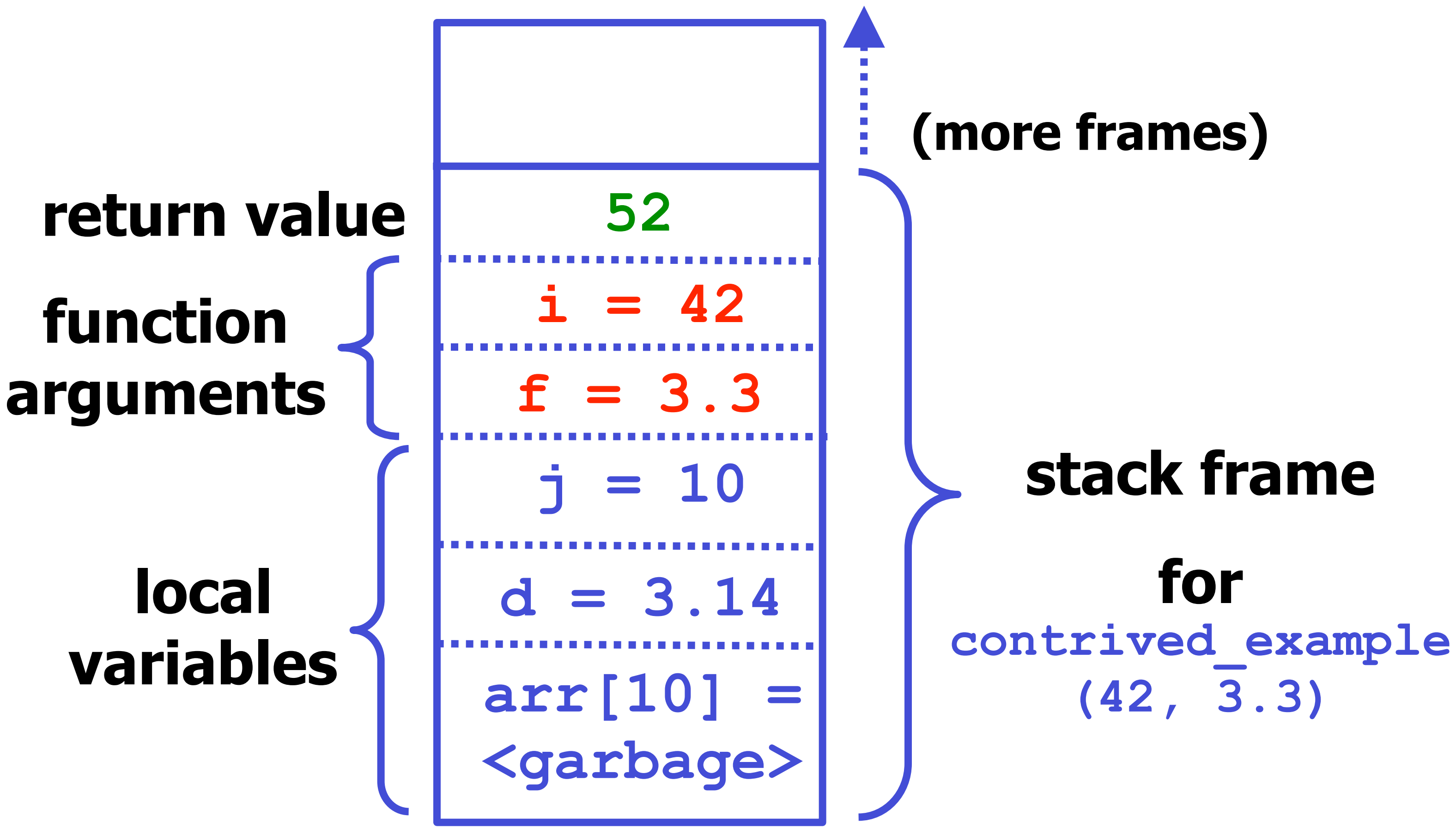
```
int contrived_example(int i, float f)
{
    int j = 10;
    double d = 3.14;
    int arr[10];
    /* do some stuff, then return */
    return (j + i);
}
```

The stack and the heap   (3)

```
/* somewhere in code */
int k = contrived_example(42, 3.3);
```

- What does this look like on the stack?

The stack and the heap (4)



The stack and the heap (5)

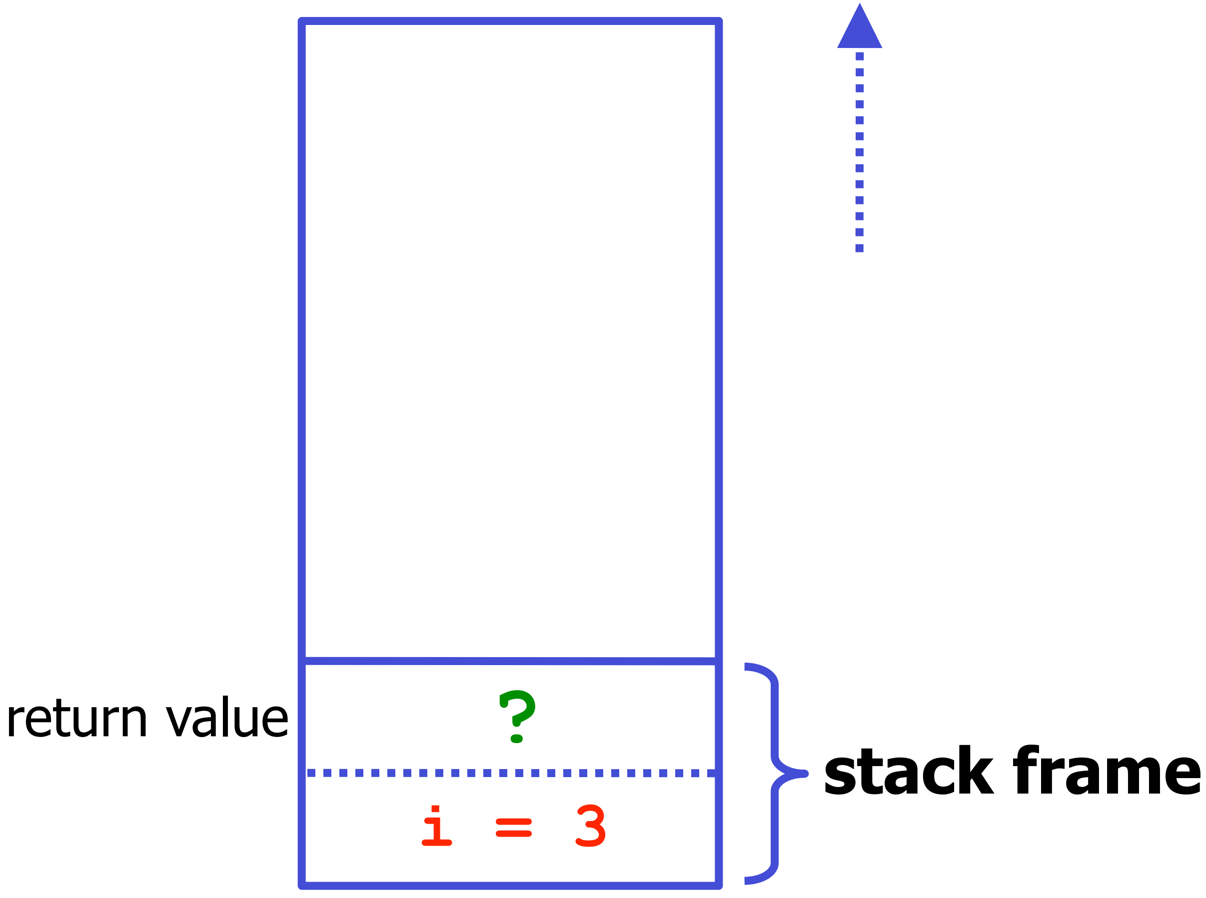
- Another example:

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

The stack and the heap (6)

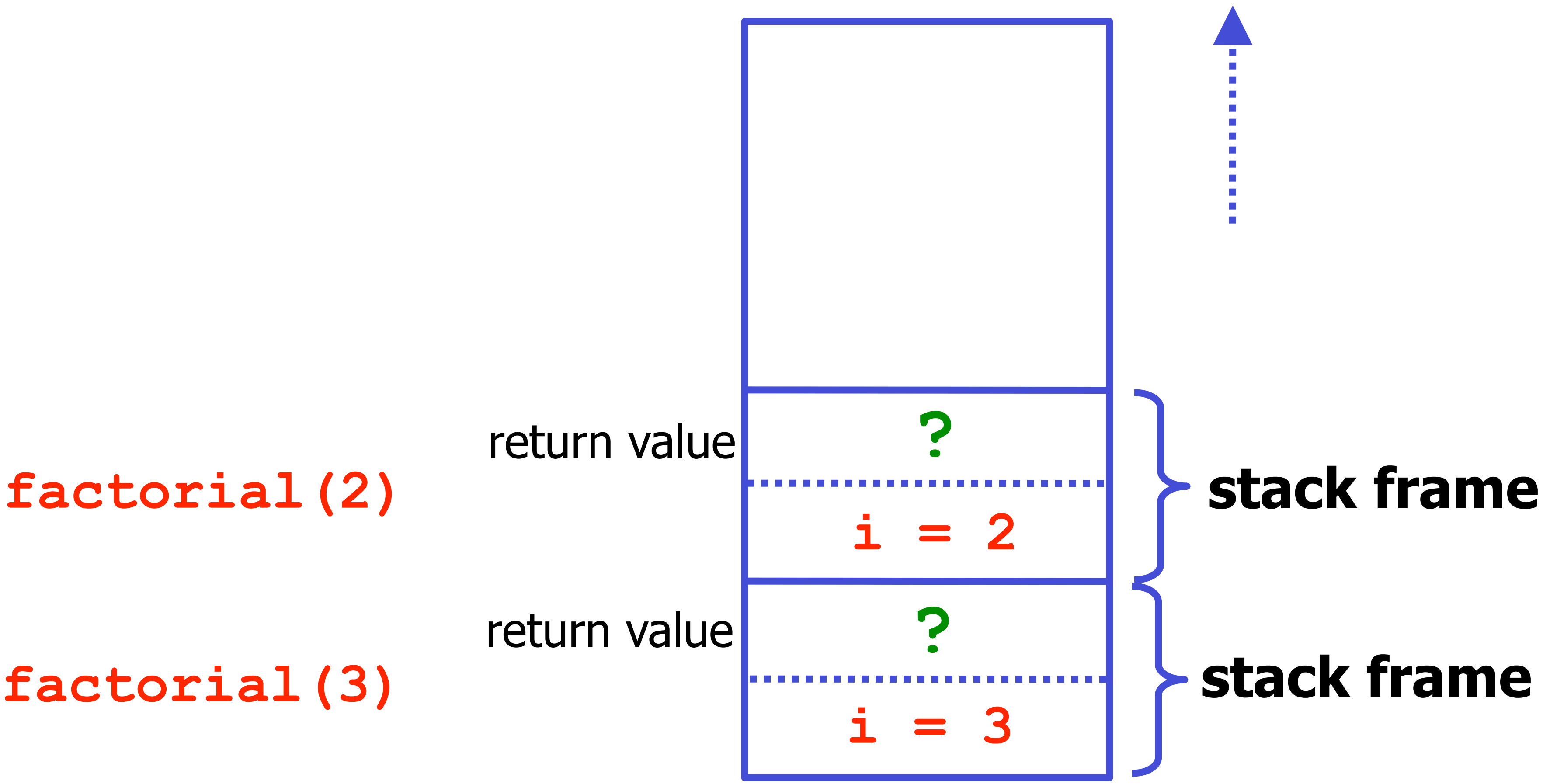
- Pop quiz: what goes on the stack for `factorial (3)`?
- For each stack frame, have...
  - no local variables
  - one argument (`i`)
  - one return value
- Each recursive call generates a new stack frame
  - which disappears after the call is complete

The stack and the heap (7)

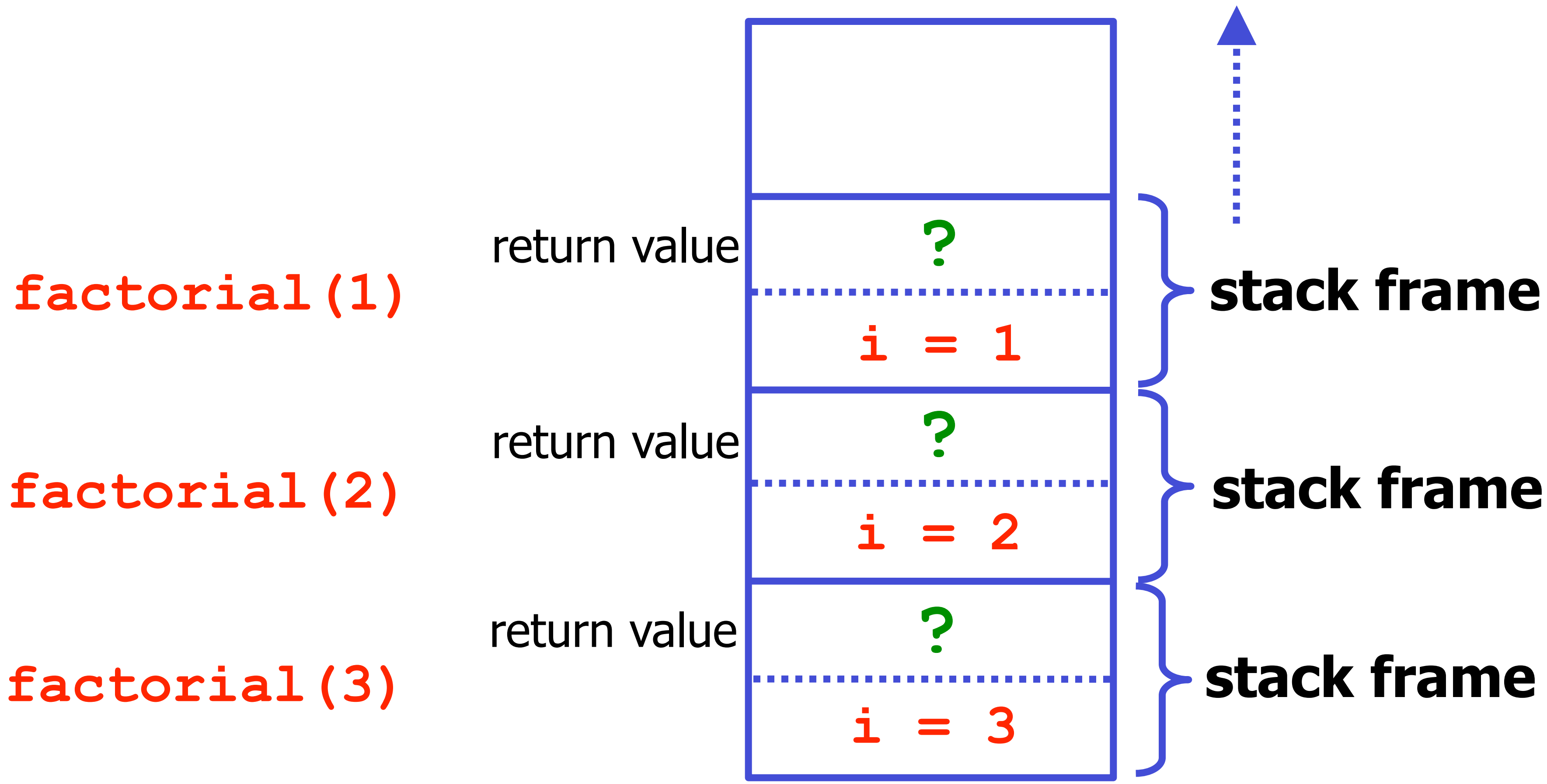


factorial(3)

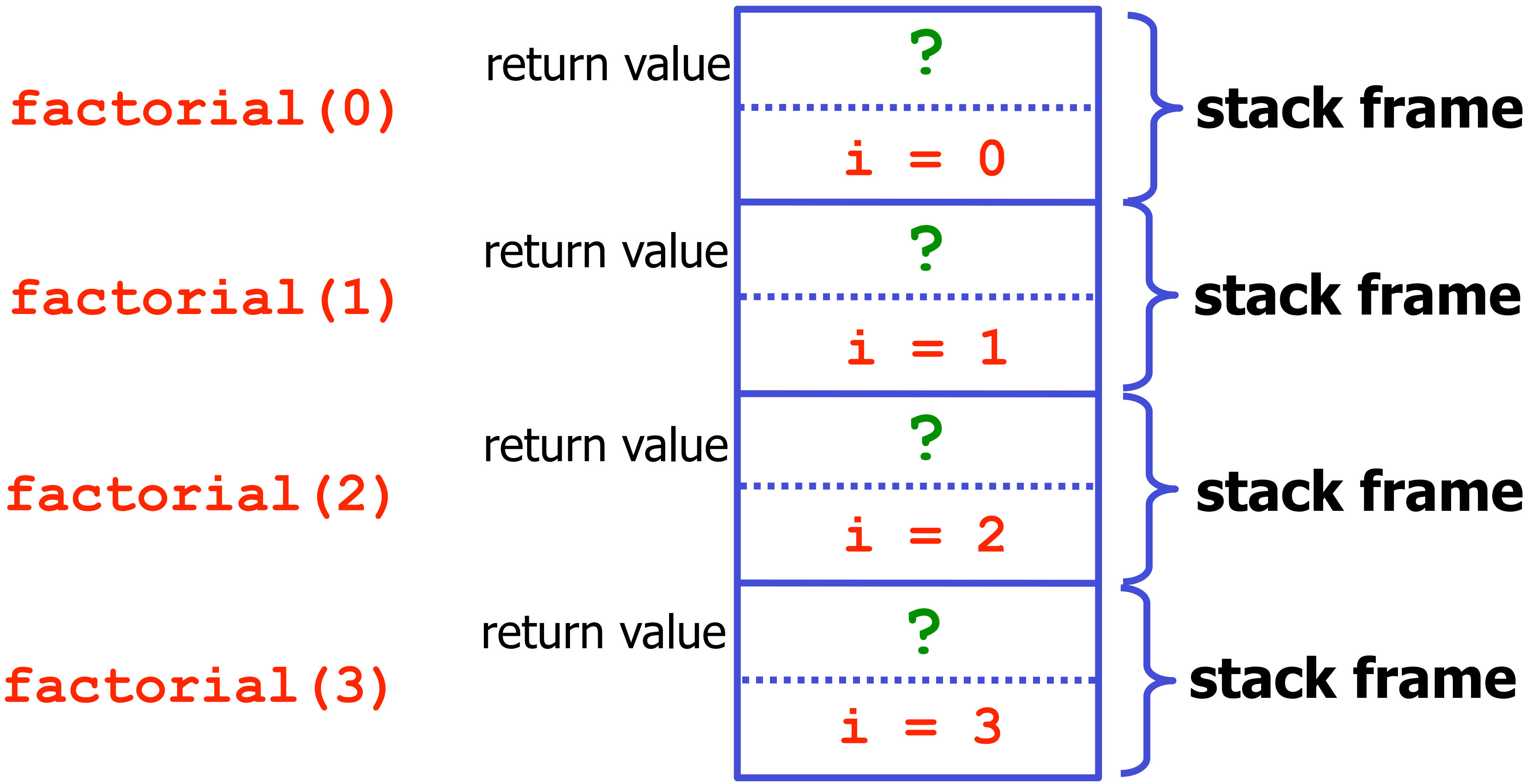
The stack and the heap (8)



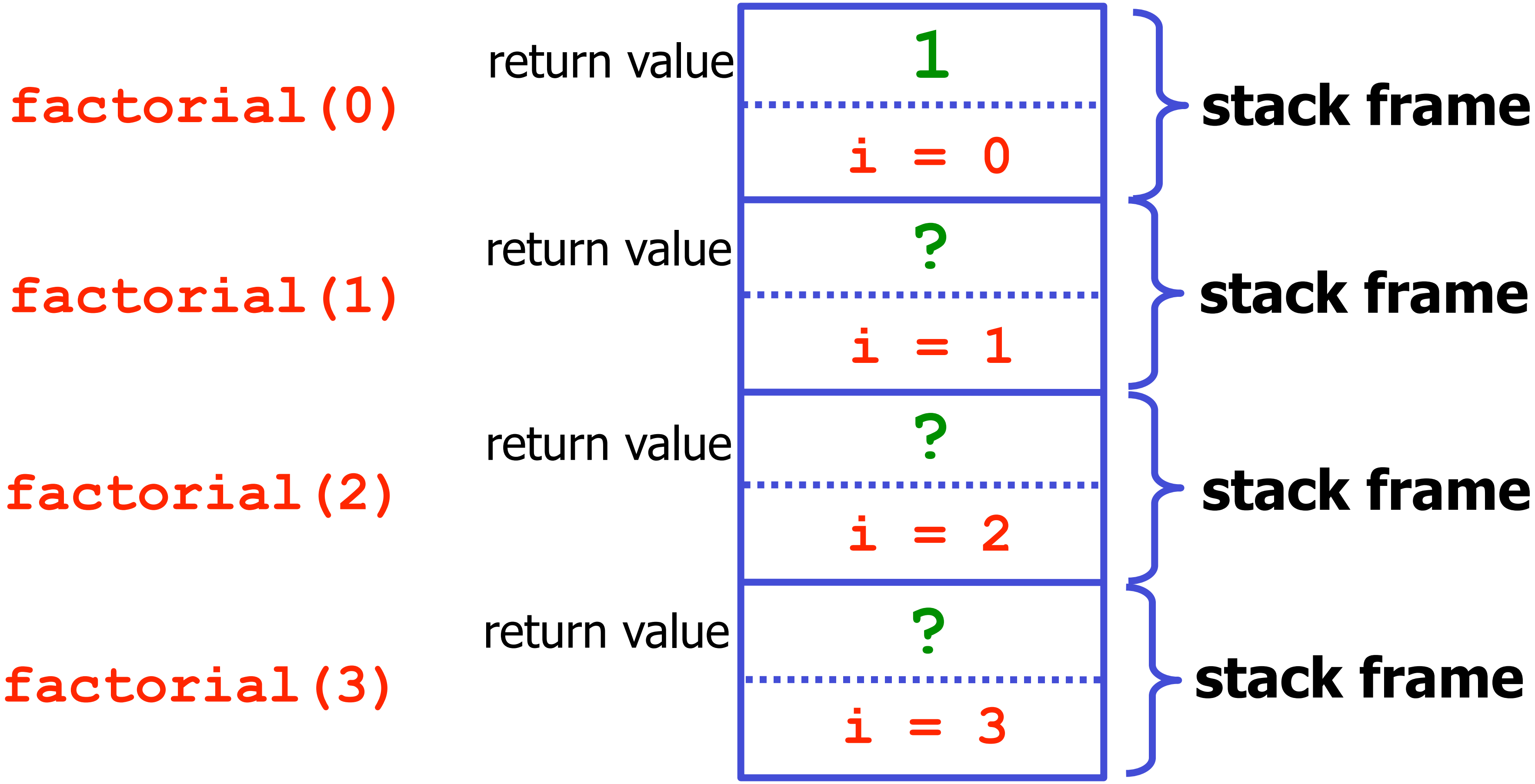
The stack and the heap (9)



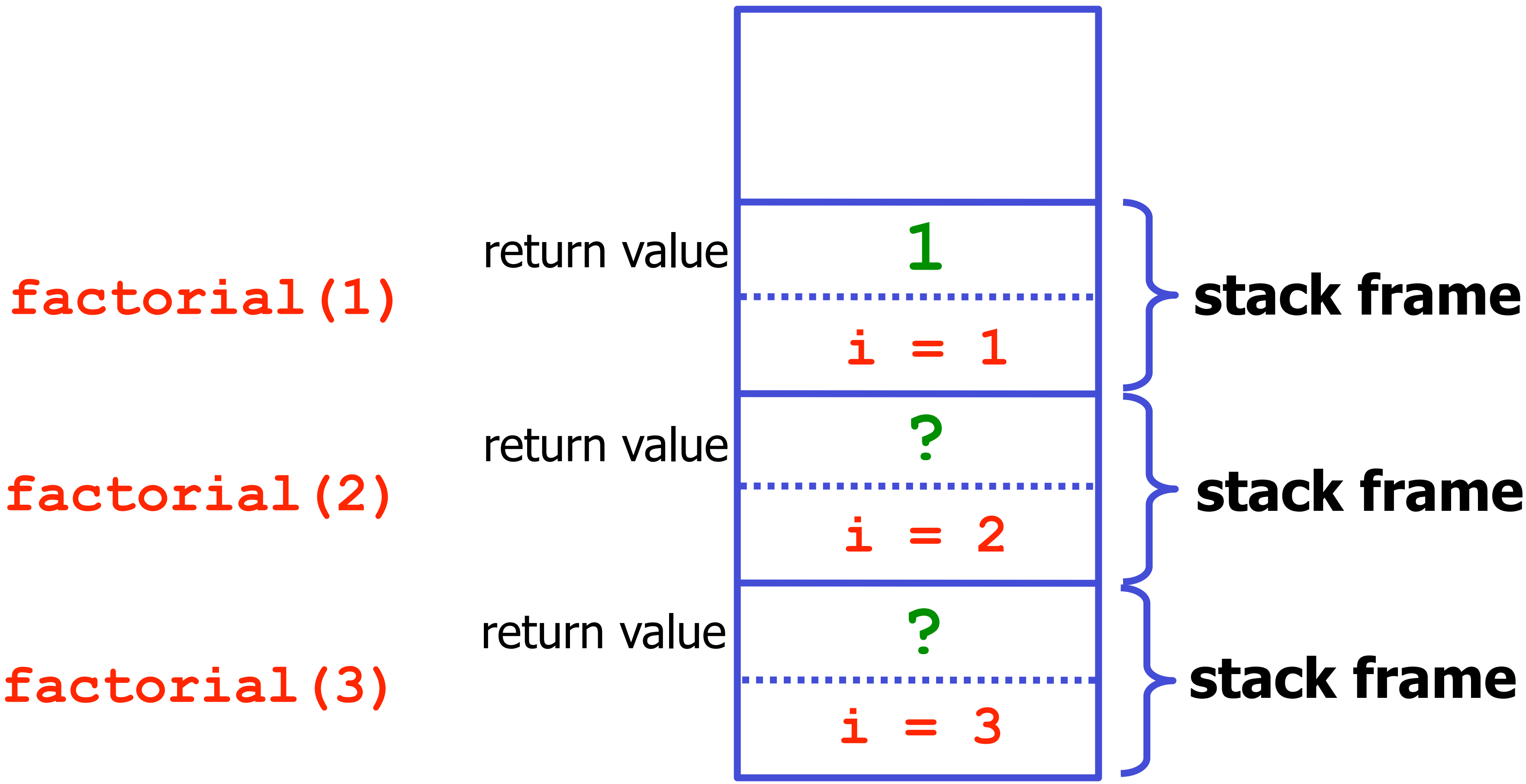
The stack and the heap (10)



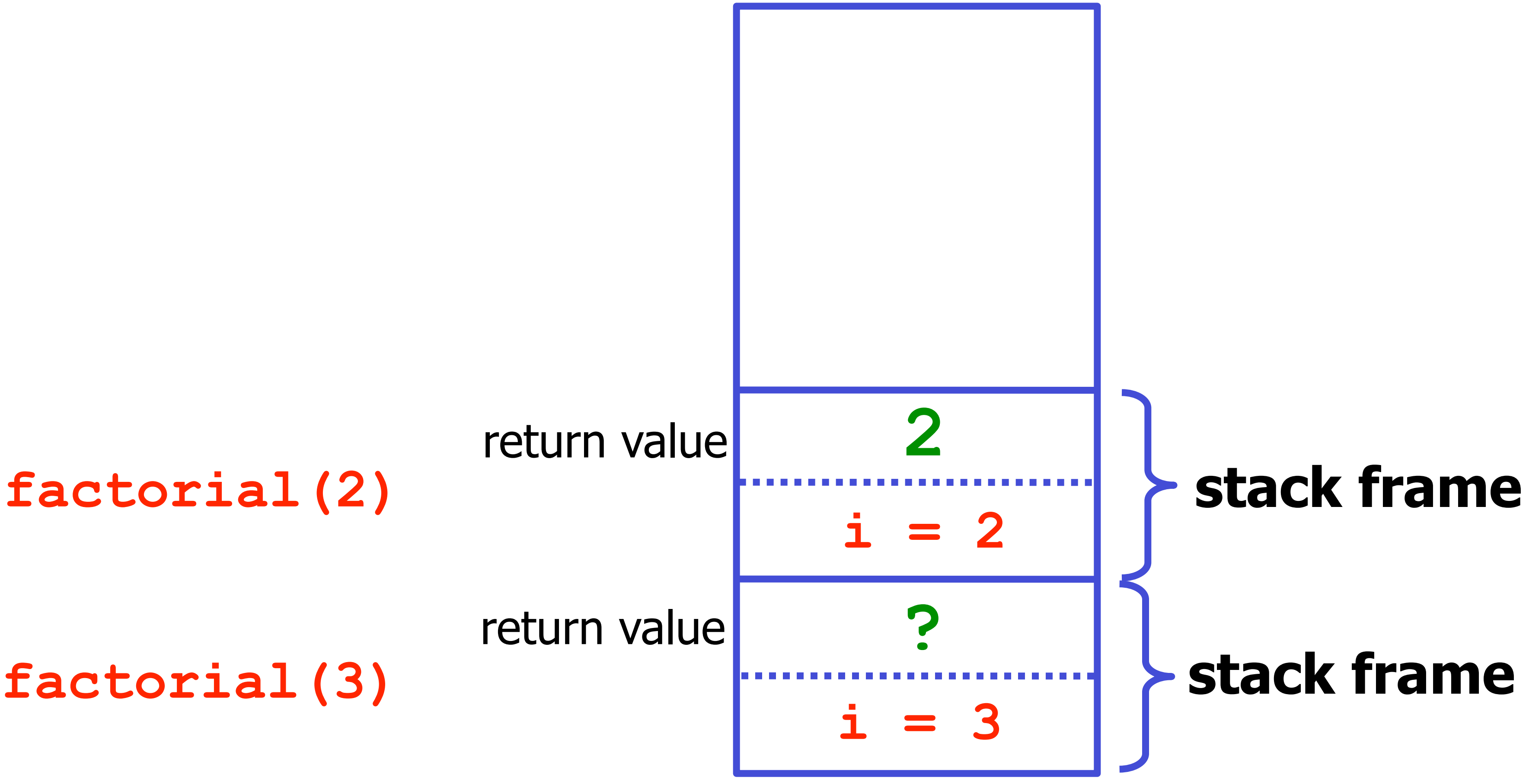
The stack and the heap (11)



The stack and the heap (12)

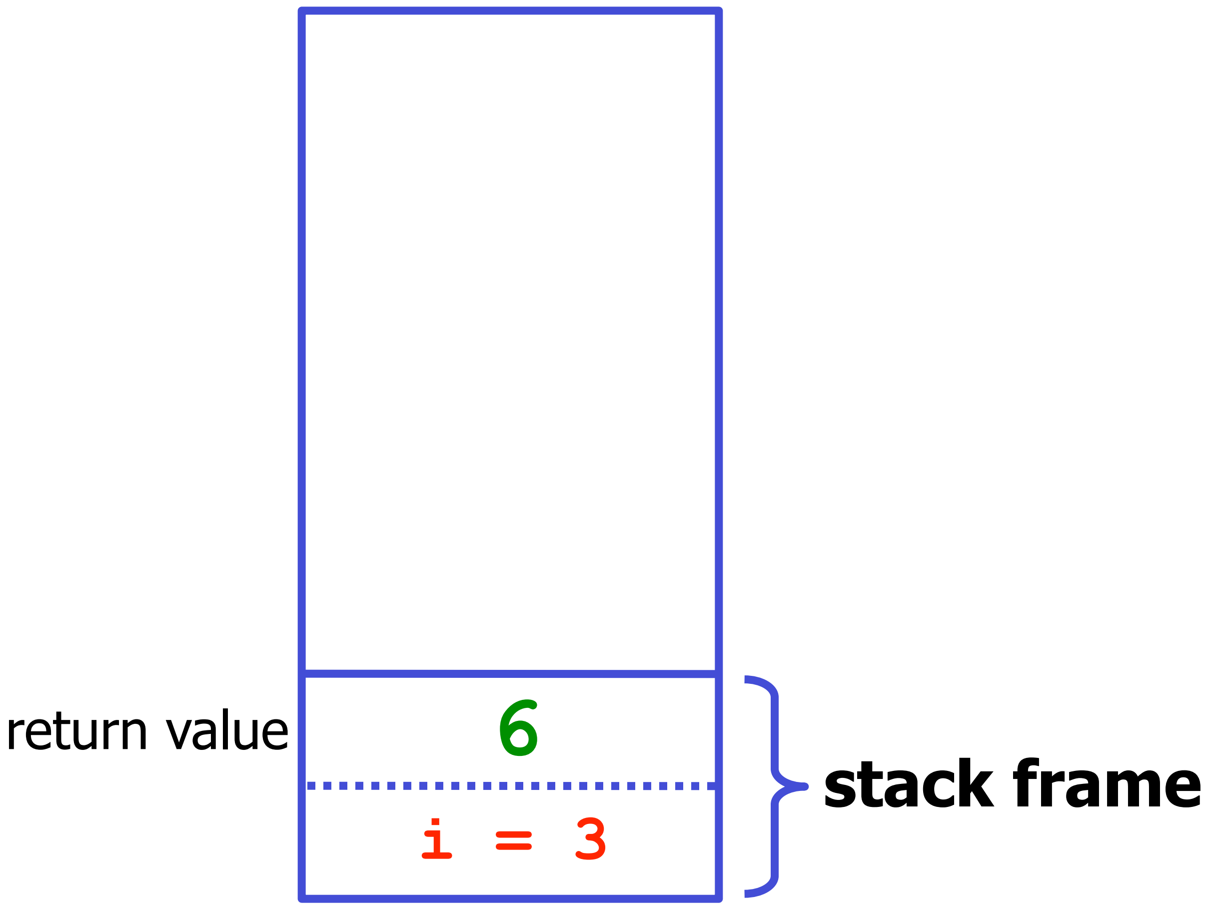


The stack and the heap (13)



The stack and the heap (14)

factorial(3)



The stack and the heap (15)

`factorial(3)`

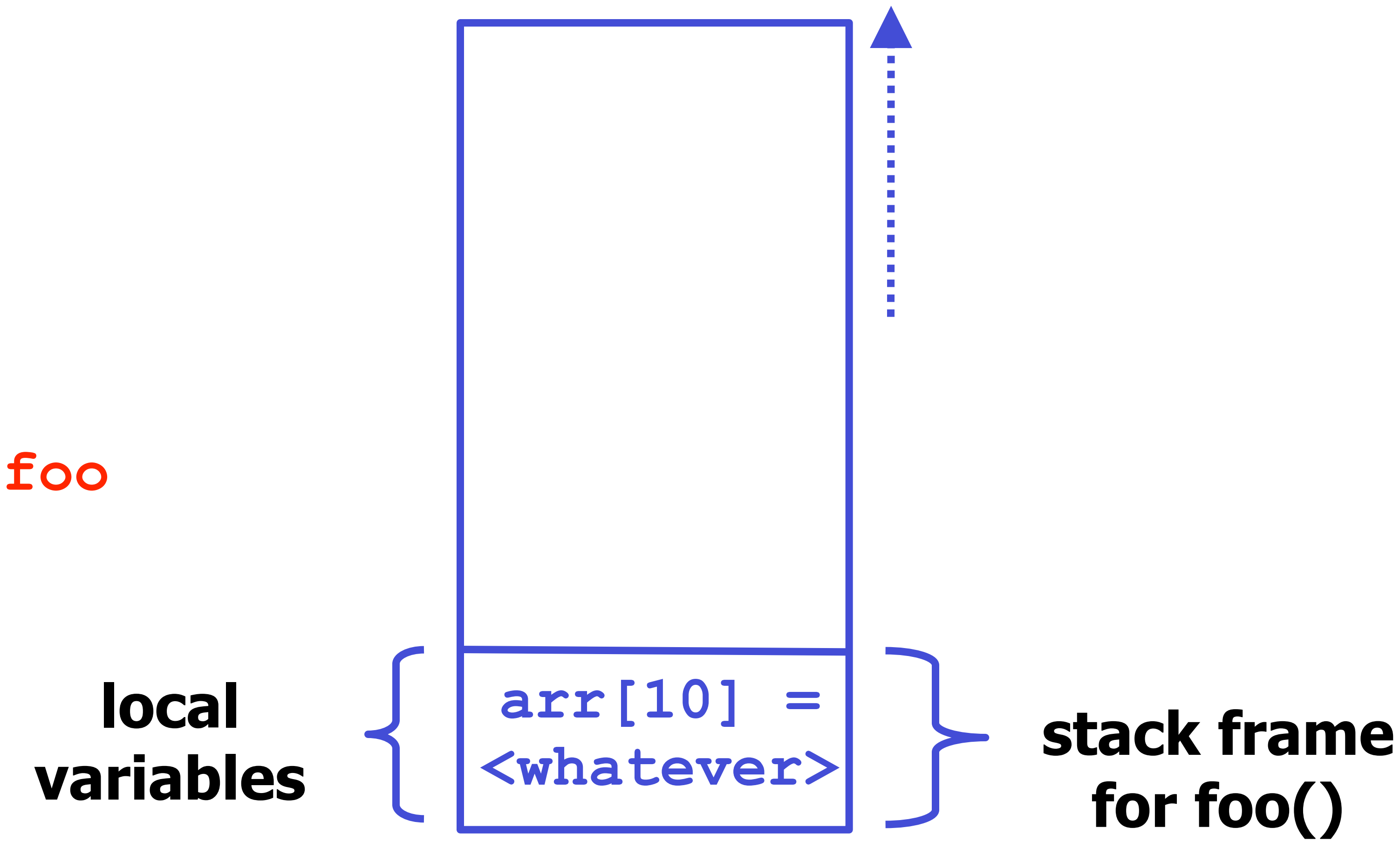
result: **6**

The stack and the heap (16)

```
void foo(void) {
    int arr[10];  /* local (on stack) */
    /* do something with arr */
}   /* arr is deallocated */
```

- Local variables sometimes called "automatic" variables; deallocation is automatic

The stack and the heap (17)



The stack and the heap (18)

- The "heap" is the general pool of computer memory
- Memory is allocated on the heap using `malloc()` or `calloc()`
- Heap memory must be explicitly freed using `free()`
- Failure to do so → memory leak!

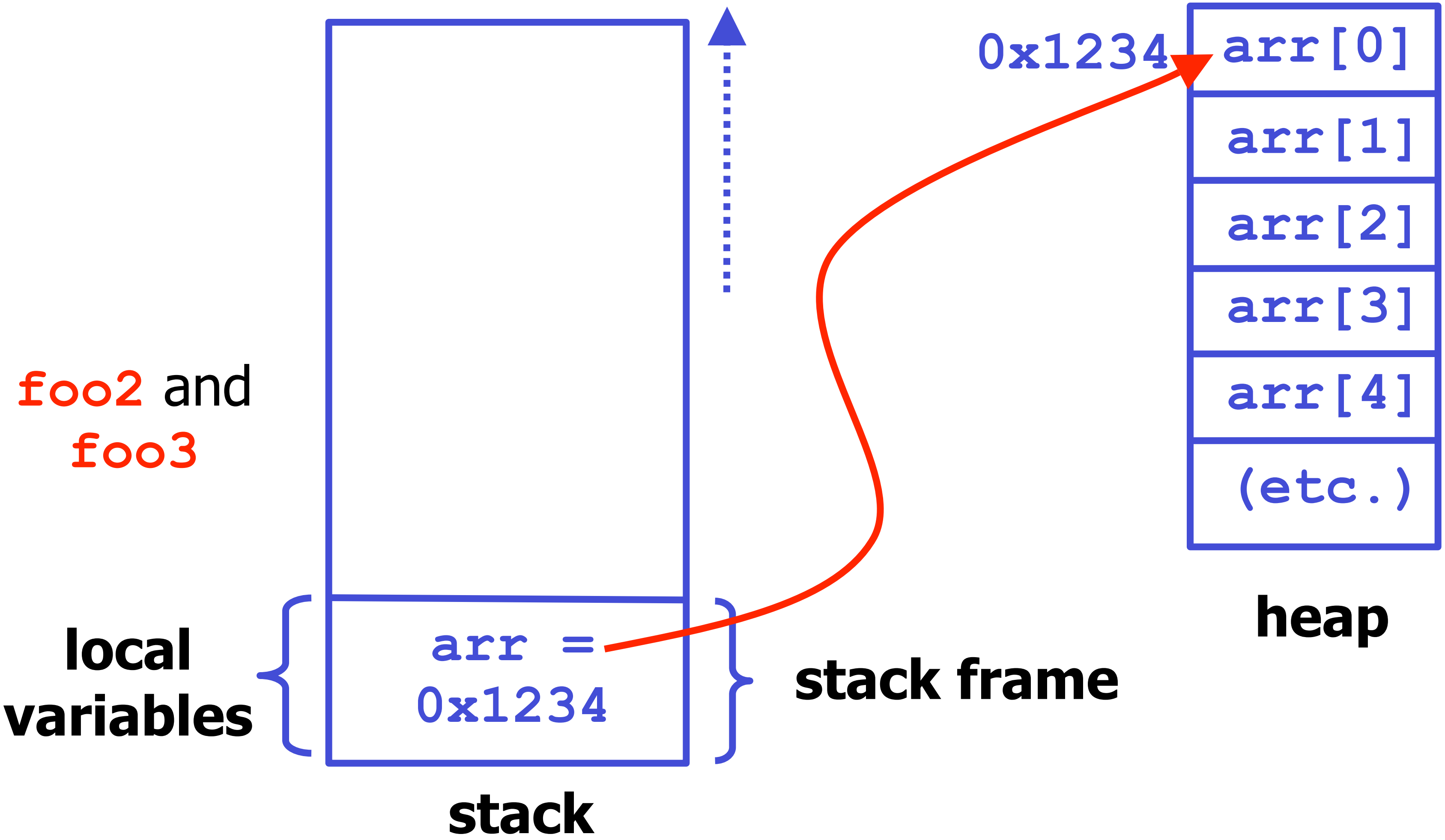
The stack and the heap (19)

```
void foo2(void) {
    int *arr;
    /* allocate memory on the heap: */
    arr = (int *)calloc(10, sizeof(int));
    /* do something with arr */
}   /* arr is NOT deallocated */
```

The stack and the heap (20)

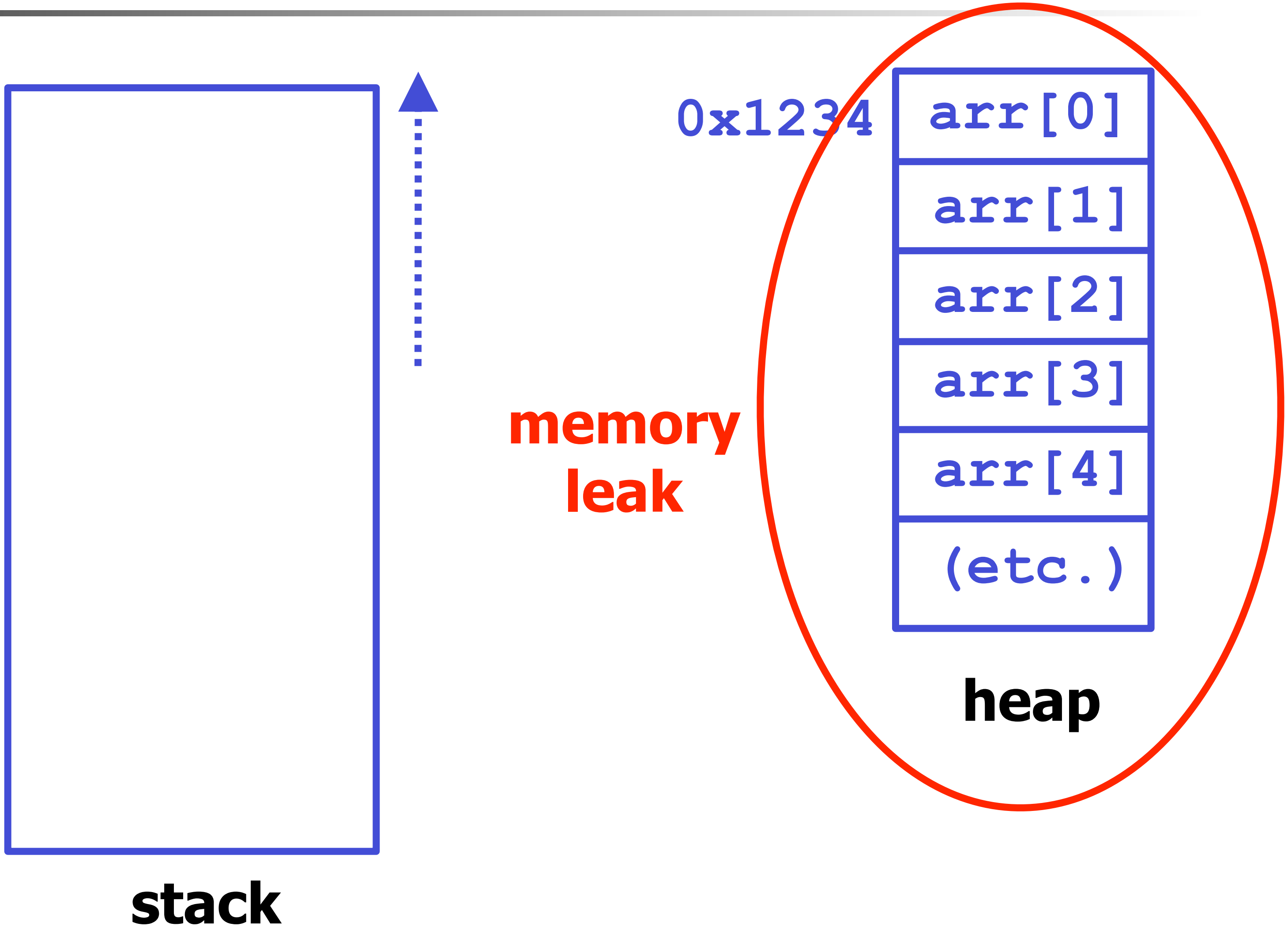
```
void foo3(void) {
    int *arr;
    /* allocate memory on the heap: */
    arr = (int *)calloc(10, sizeof(int));
    /* do something with arr */
    free(arr);
}
```

The stack and the heap (21)



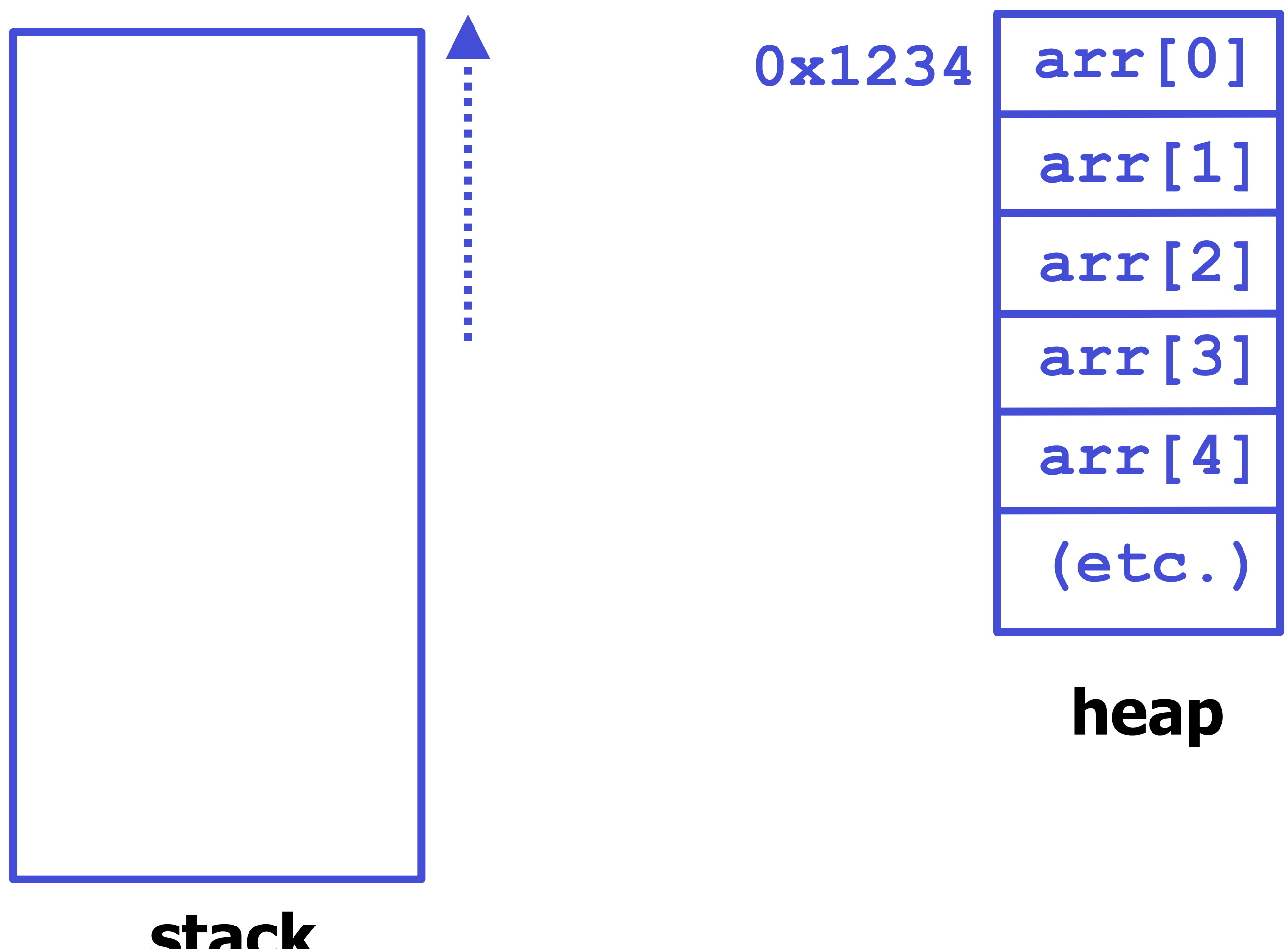
The stack and the heap (22)

(after `foo2` exits, without freeing memory)



The stack and the heap (23)

(after `foo3` exits, with freeing memory)



Memory leaks

- Memory leaks are one of the <u>worst</u> kinds of bugs
  - often, no harm done at all
  - eventually may cause long-running program to crash
    - out of memory
  - very hard to track down
- Special tools (e.g. `valgrind`) exist to debug memory leaks
- I supply you with a very simple leak checker

Next week

- `struct`
- `typedef`
- Linked lists