Pointers and Memory

9/29/16

Recall: Allocating (Heap) Memory

- The standard C library (#include <stdlib.h>) includes functions for allocating memory

`void *malloc(size_t size)`

  - Allocate `size` bytes on the heap and return a pointer to the beginning of the memory block

`void free(void *ptr)`

  - Release the `malloc()ed` block of memory starting at `ptr` back to the system

What do you expect to happen to the 100-byte chunk if we do this?

```
// What happens to these 100 bytes?
int *ptr = malloc(100);


ptr = malloc(2000);
```

A. The 100-byte chunk will be lost.

B. The 100-byte chunk will be automatically freed (garbage collected) by the OS.

C. The 100-byte chunk will be automatically freed (garbage collected) by C.

D. The 100-byte chunk will be the first 100 bytes of the 2000-byte chunk.

E. The 100-byte chunk will be added to the 2000-byte chunk (2100 bytes total).

Memory Leak

- Memory that is allocated, and not freed, for which there is no longer a pointer.

- In many languages (Java, Python, …), this memory will be cleaned up for you.
  - “Garbage collector” finds unreachable memory blocks, frees them.
  - C doesn’t does NOT do this for you!

Why doesn’t C do garbage collection?

A. It’s impossible in C.

B. It requires a lot of resources.

C. It might not be safe to do so. (break programs)

D. It hadn’t been invented at the time C was developed.

E. Some other reason.

Memory Bookkeeping

- To free a chunk, you MUST call free with the same pointer that malloc gave you. (or a copy)
- The standard C library keeps track of the chunks that have been allocated to your program.
  - This is called “metadata” – data about your data.
- Wait, where does it store that information?
  - It’s not like it can use malloc() to get memory…

Where should we store this metadata?

A. In the CPU

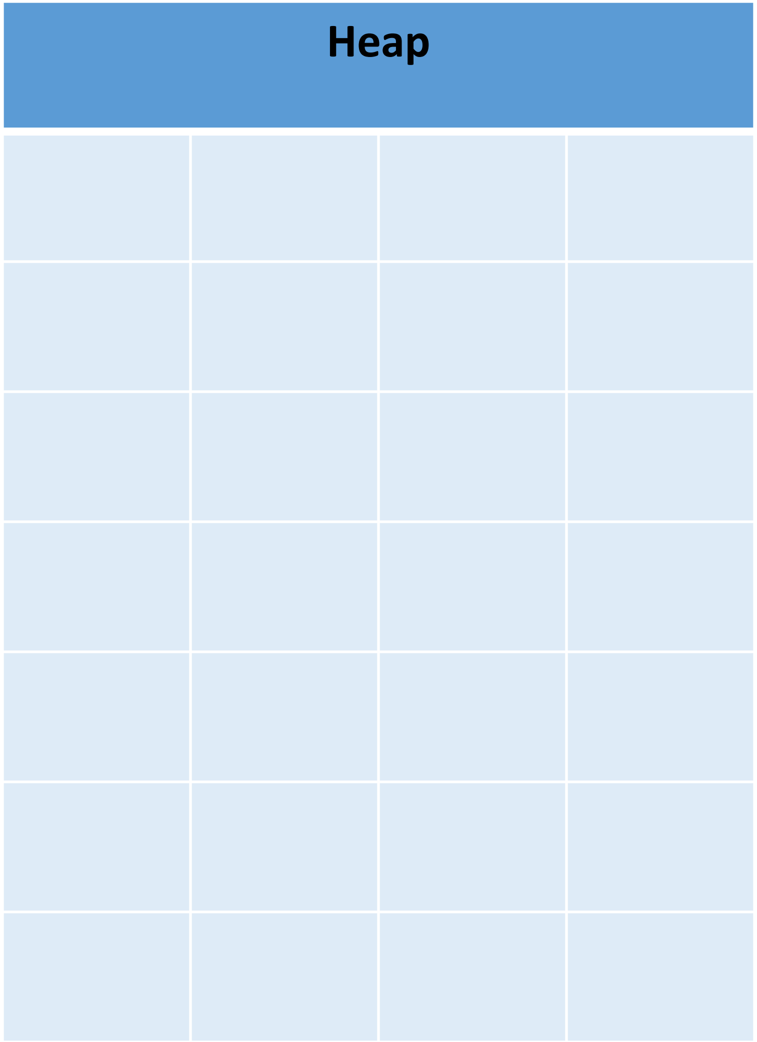
B. In main memory

C. On the hard drive

D. Somewhere else

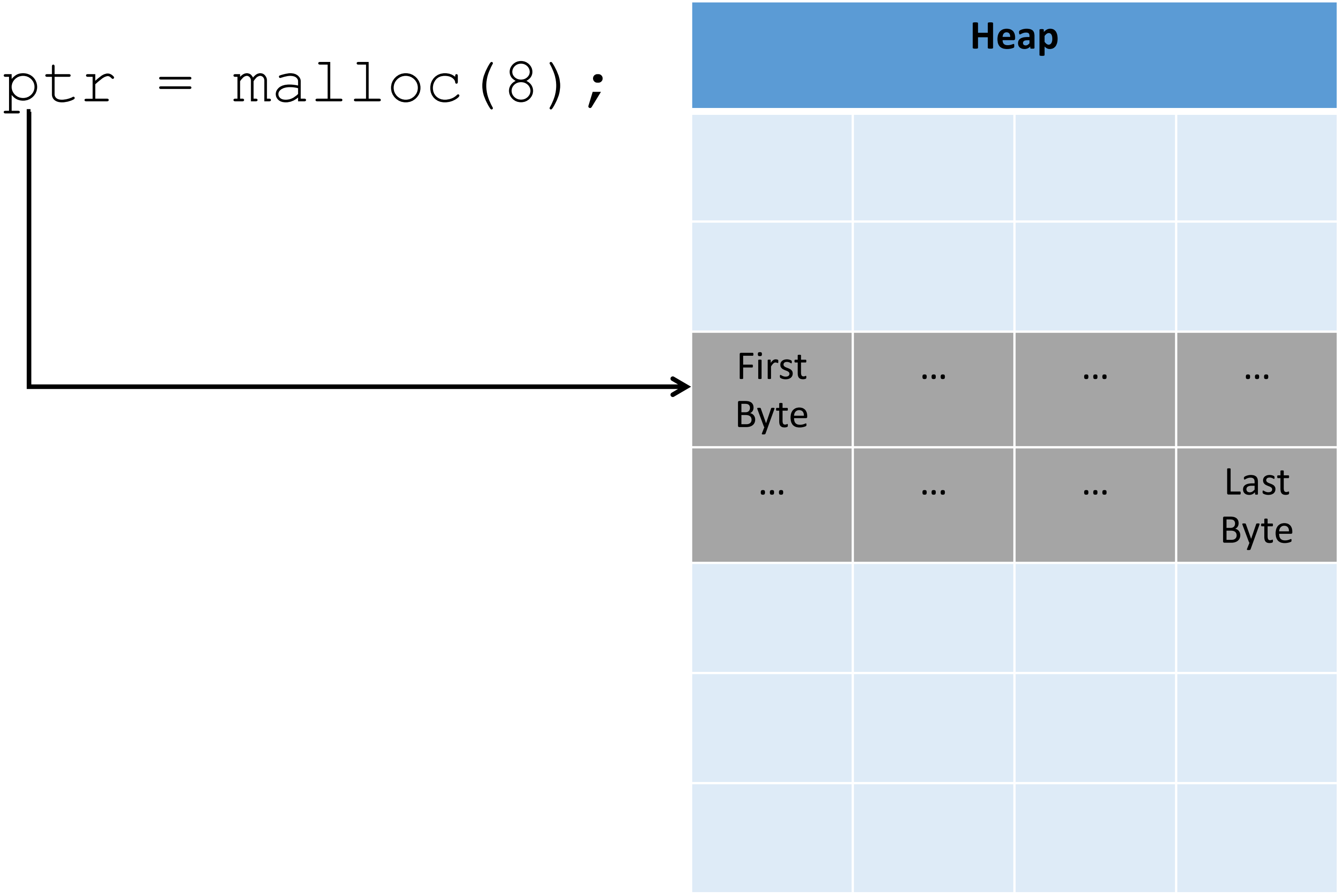
Metadata

```
int *iptr = malloc(8);
```



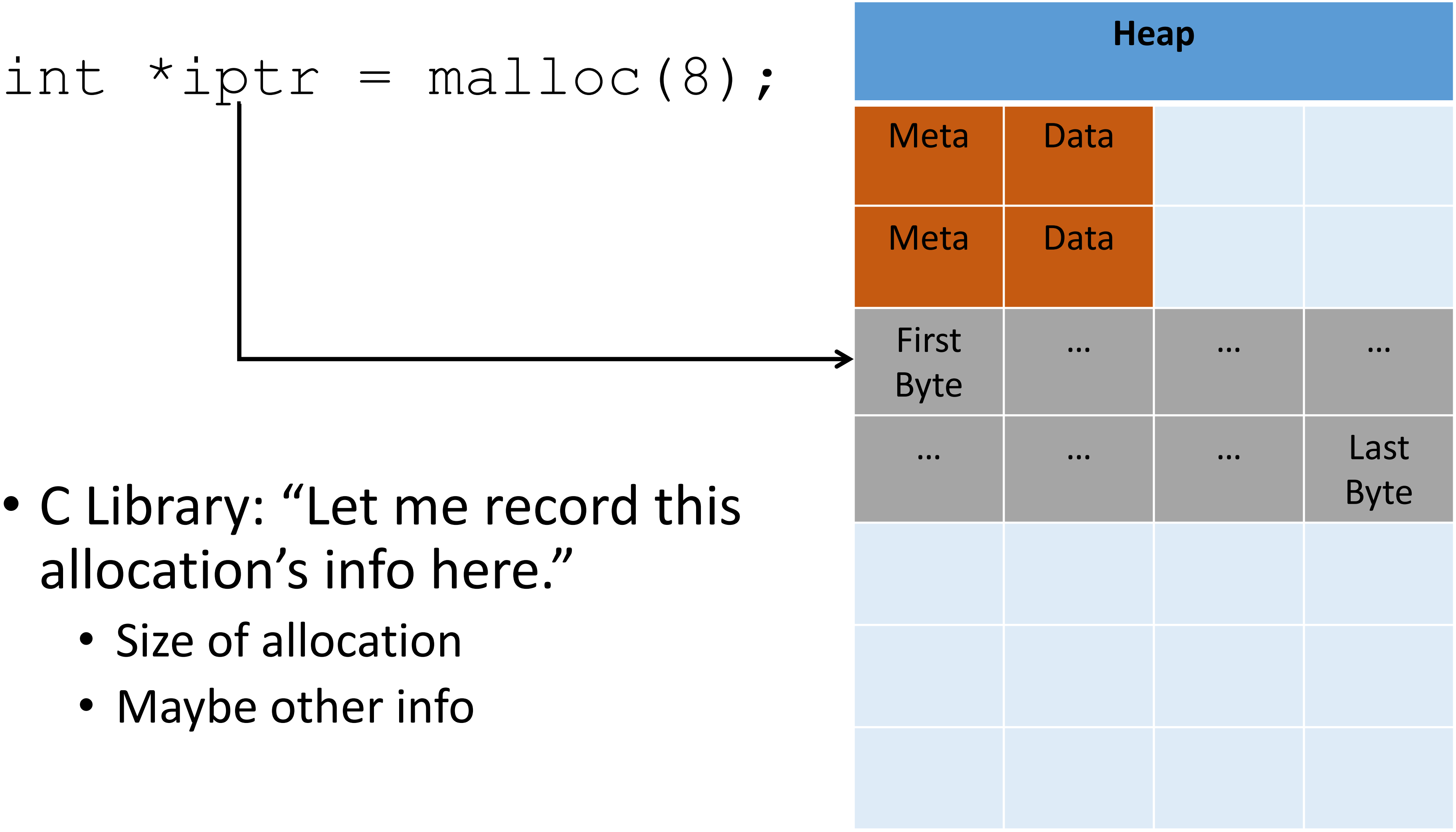
Metadata

```
int *iptr = malloc(8);
```



Metadata

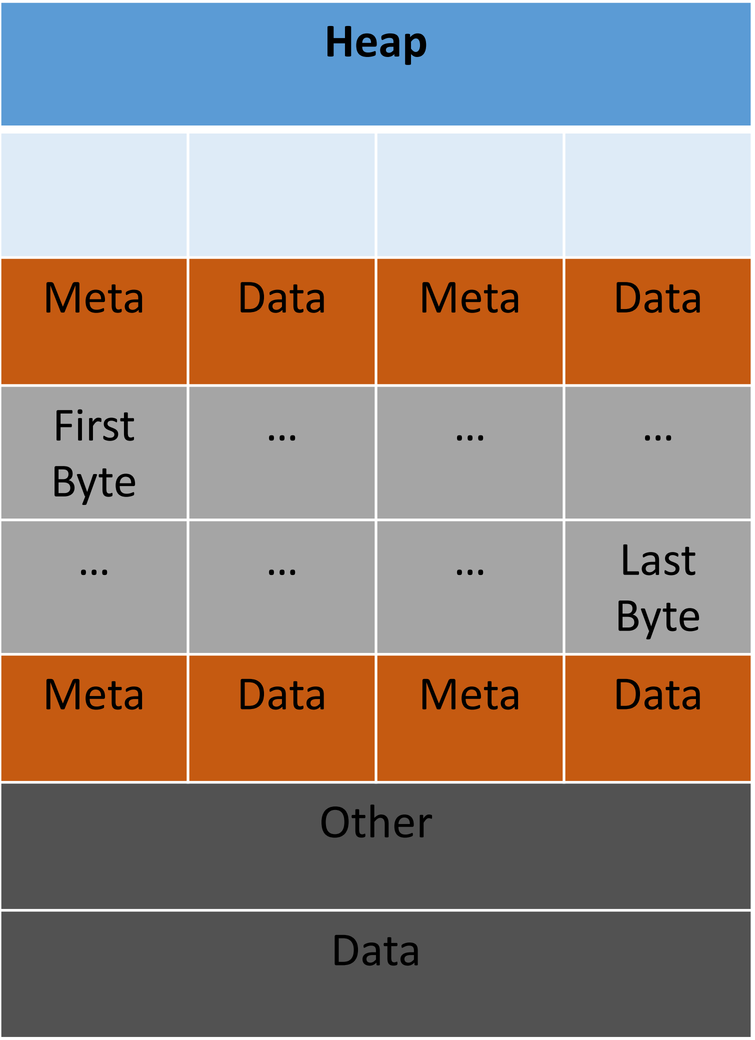
```
int *iptr = malloc(8);
```



- C Library: "Let me record this allocation's info here."
  - Size of allocation
  - Maybe other info

Metadata

```
int *iptr = malloc(8);
```

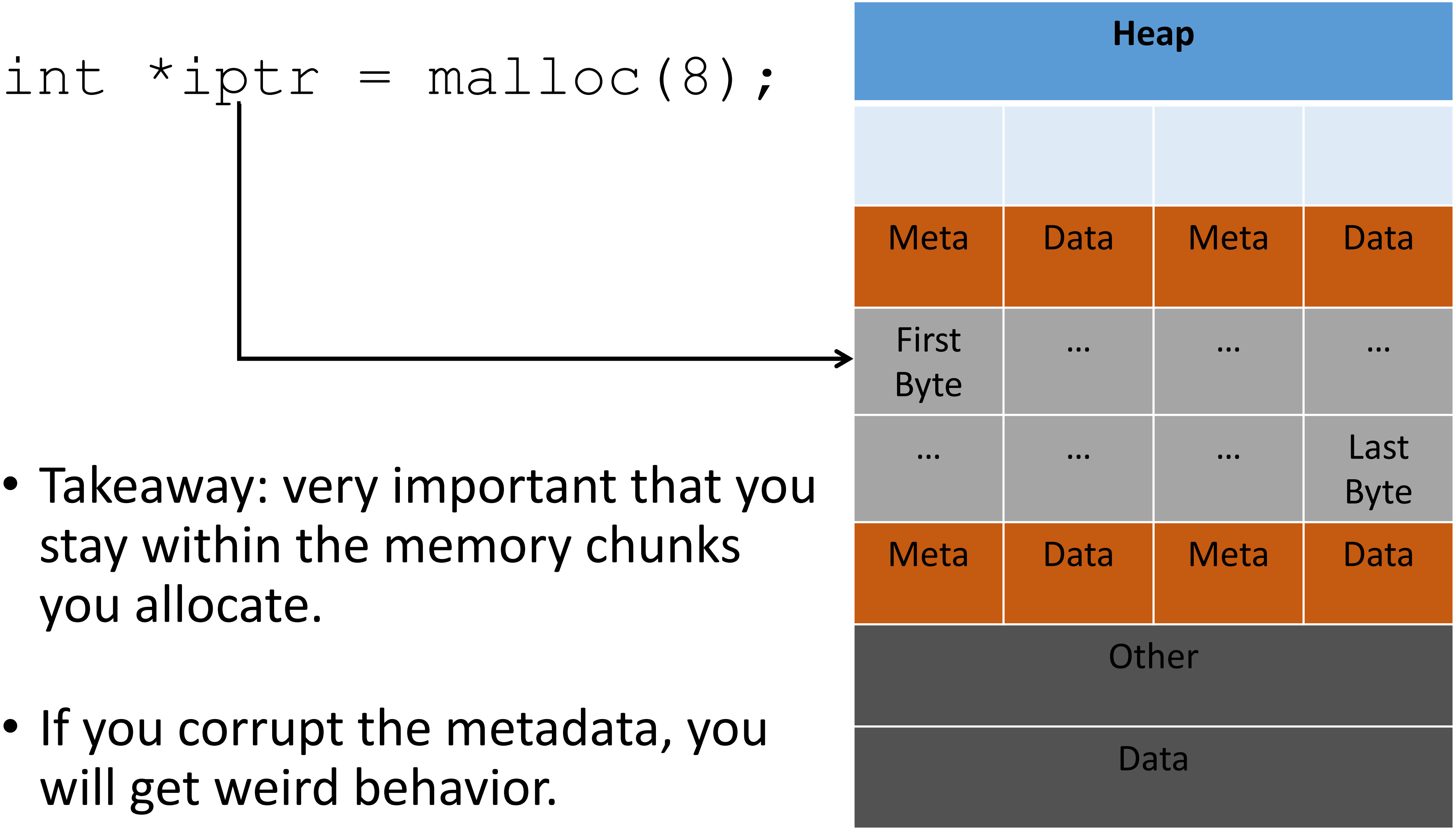


Heap			
Meta	Data	Meta	Data
First Byte	…	…	…
…	…	…	Last Byte
Meta	Data	Meta	Data
Other			
Data			

- For all you know, there could be another chunk after yours.

Metadata

`int *iptr = malloc(8);`



- Takeaway: very important that you stay within the memory chunks you allocate.
- If you corrupt the metadata, you will get weird behavior.

Valgrind is your new best friend.

Pointers as Arrays

- "Why did you allocate 8 bytes for an int pointer? Isn't an int only 4 bytes?"
  - `int *iptr = malloc(8);`
- Recall: an array variable acts like a pointer to a block of memory. The number in [] is an offset from bucket 0, the first bucket.
- We can treat pointers in the same way!

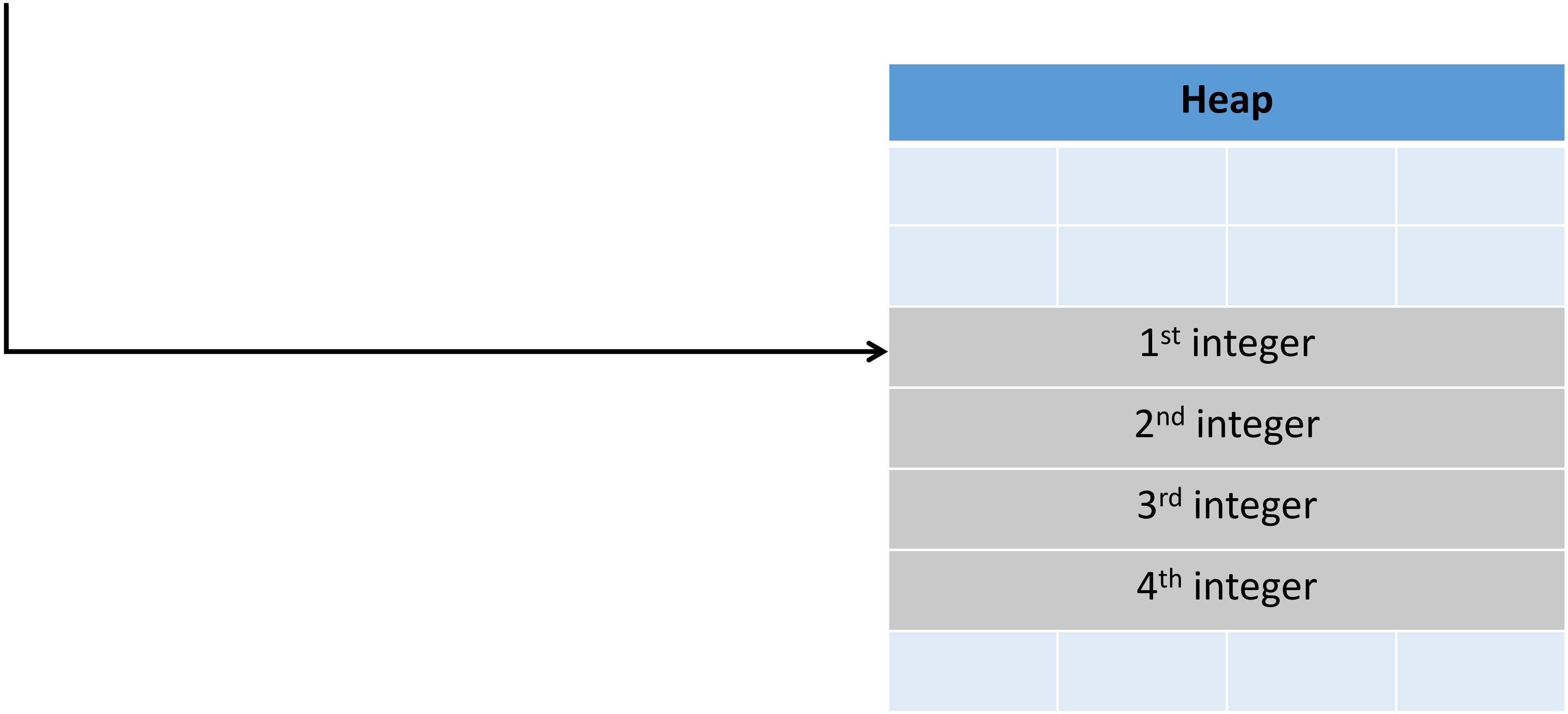
Pointers as Arrays

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```

<table>
<tr><th colspan="4">Heap</th></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>

Pointers as Arrays

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```



Heap

1st integer

2nd integer

3rd integer

4th integer

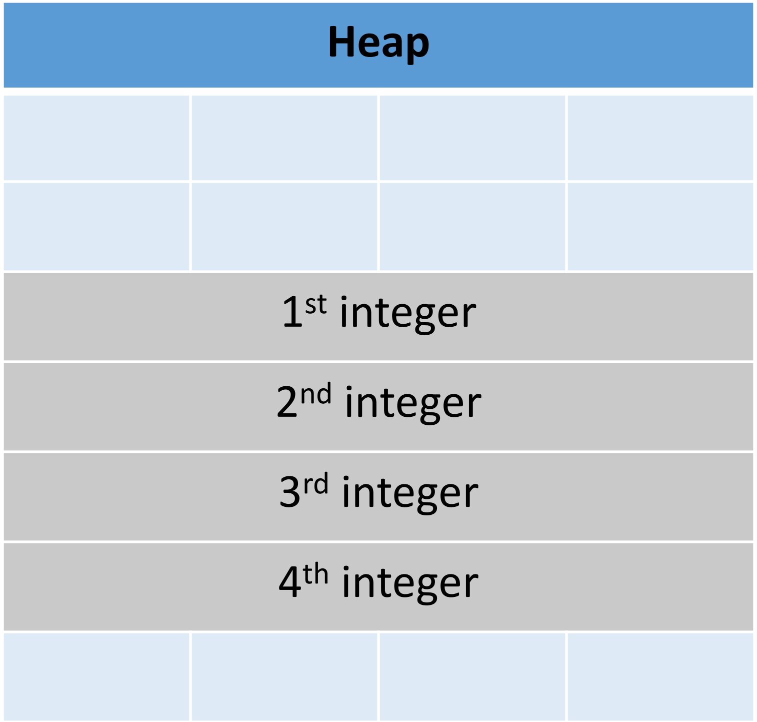
Pointers as Arrays

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```

The C compiler knows how big an integer is.

As an alternative way of dereferencing, you can use []’s like an array.

The C compiler will jump ahead the right number of bytes, based on the type.



Heap

1st integer

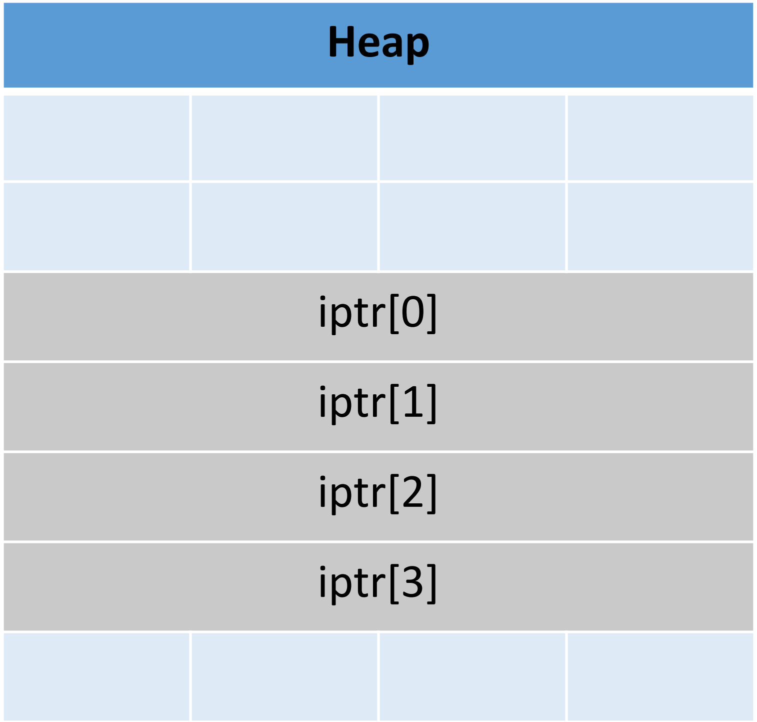
2nd integer

3rd integer

4th integer

Pointers as Arrays

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```



Heap

iptr[0]

iptr[1]

iptr[2]

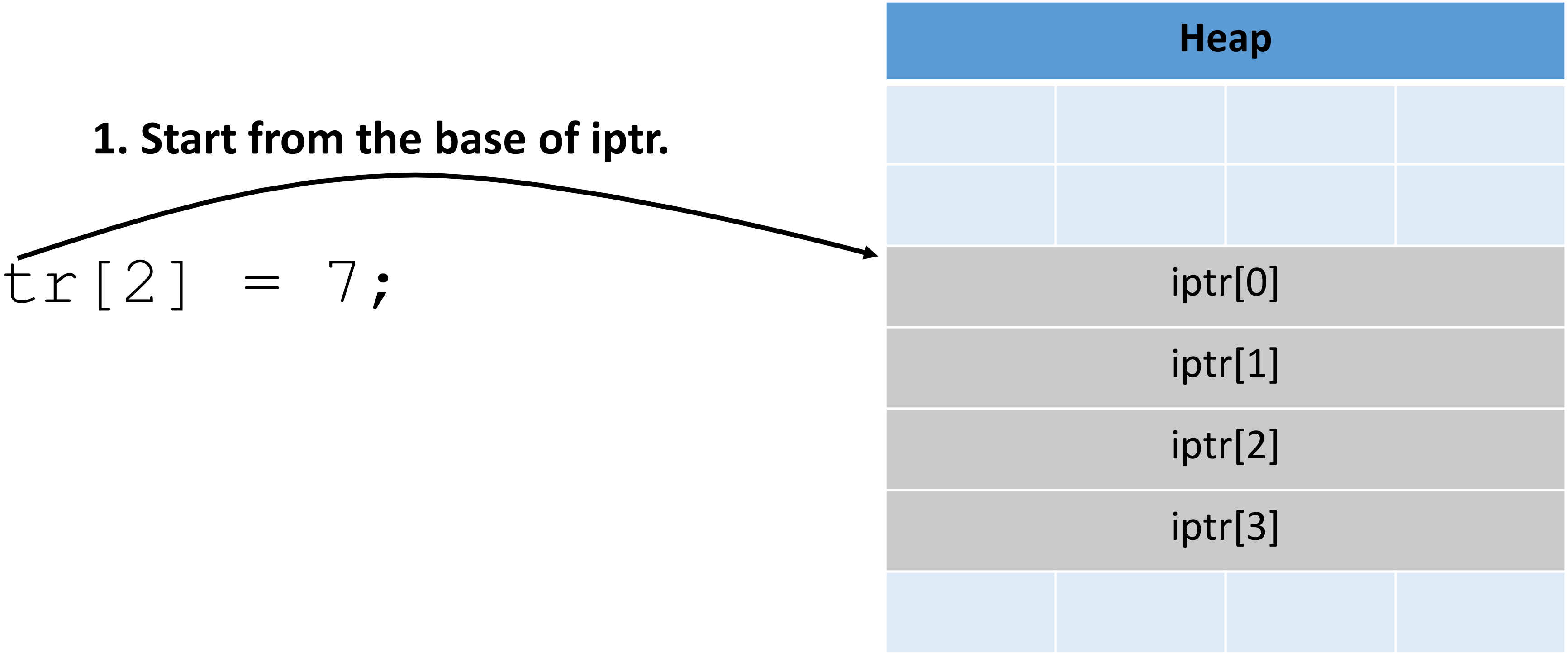
iptr[3]

Pointers as Arrays

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```

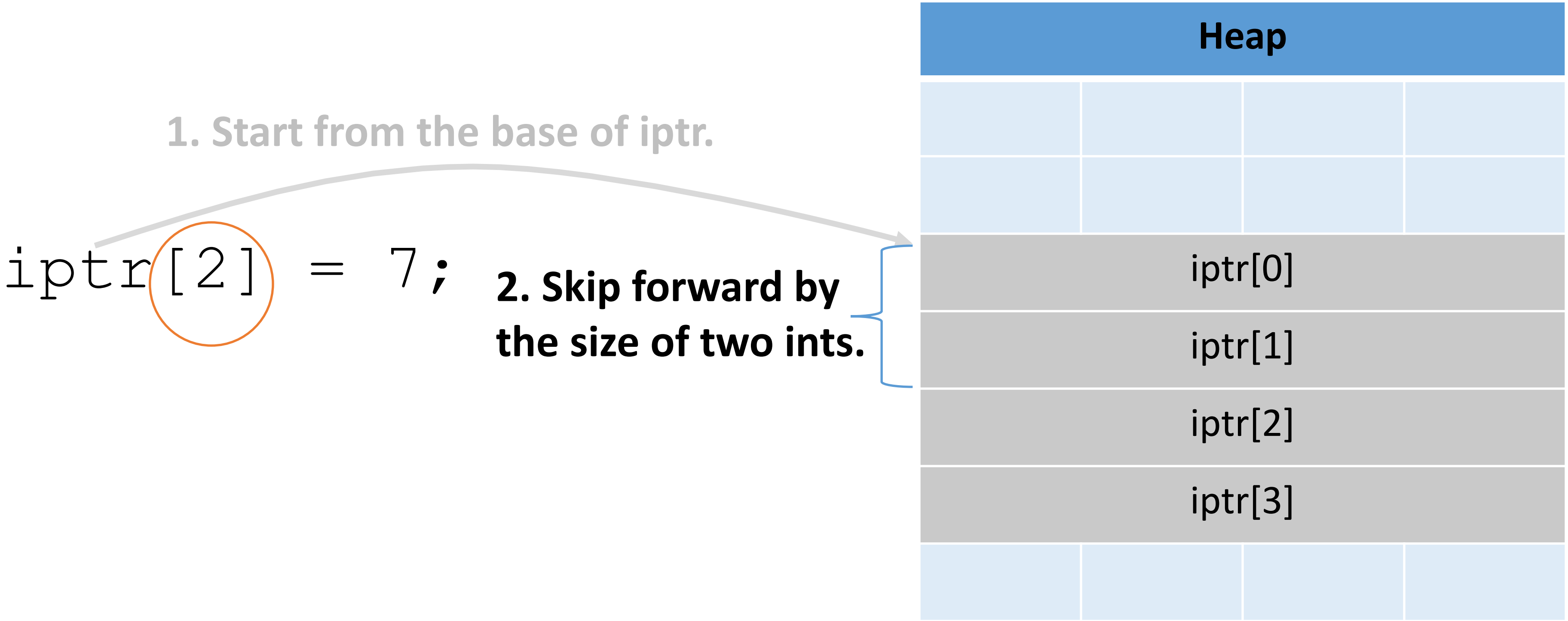
1. Start from the base of iptr.

```
iptr[2] = 7;
```



Pointers as Arrays

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```



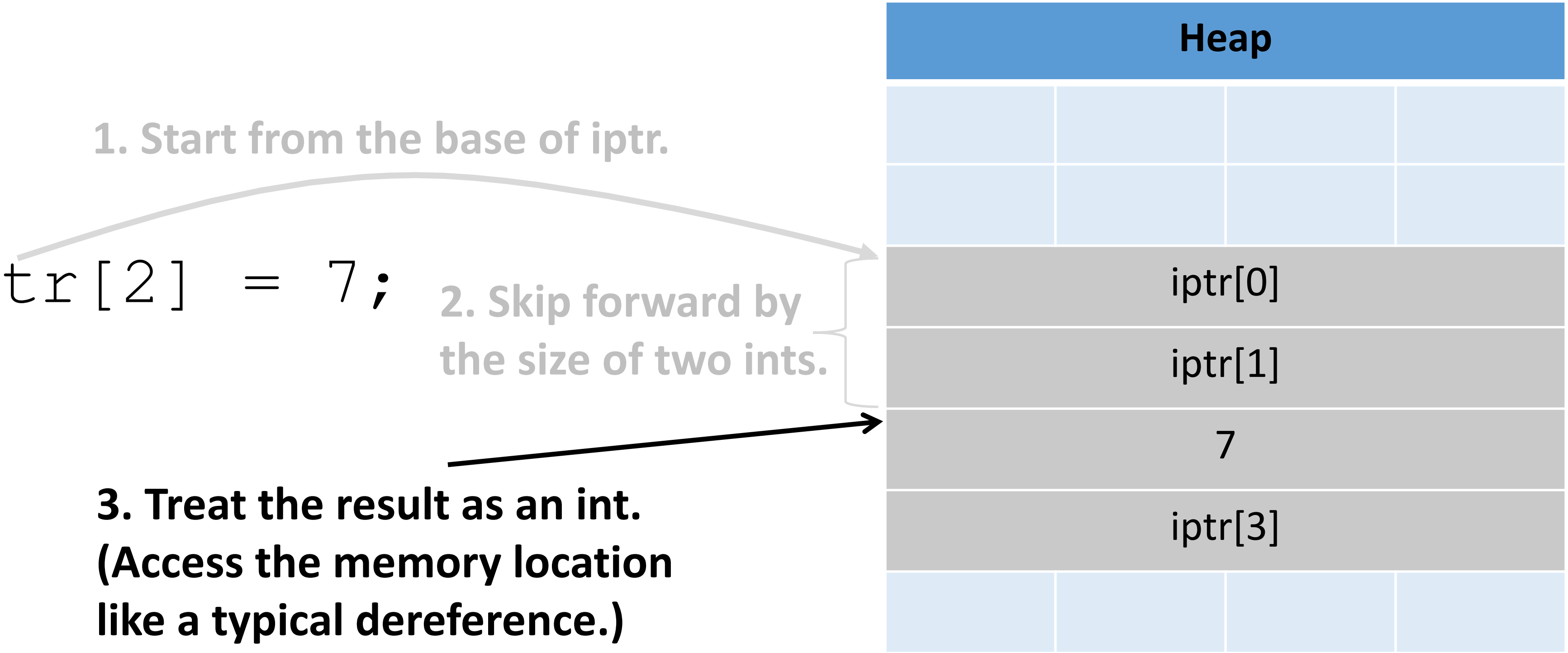
1. Start from the base of iptr.

```
iptr[2] = 7;
```

2. Skip forward by the size of two ints.

Pointers as Arrays

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```



```
iptr[2] = 7;
```

1. Start from the base of iptr.

2. Skip forward by the size of two ints.

3. Treat the result as an int. (Access the memory location like a typical dereference.)

Pointers as Arrays

- This is one of the most common ways you'll use pointers:
  - You need to dynamically allocate space for a collection of things (ints, structs, whatever).
  - You don't know how many at compile time.

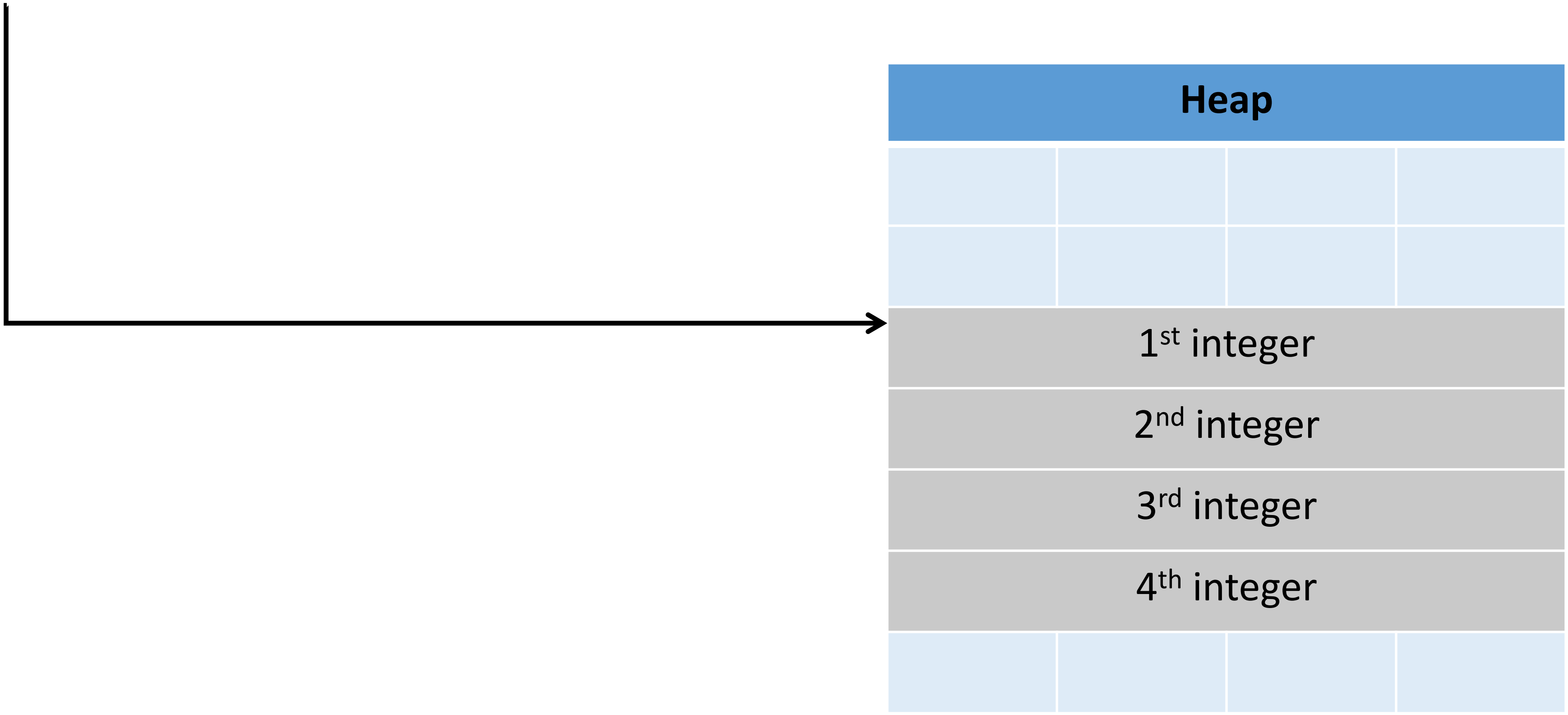
```
float *student_gpas = NULL;
student_gpas = malloc(n_students * sizeof(int));
…
student_gpas[0] = …;
student_gpas[1] = …;
```

Pointer Arithmetic

- Addition and subtraction work on pointers.
- C automatically increments by the size of the type that’s pointed to.

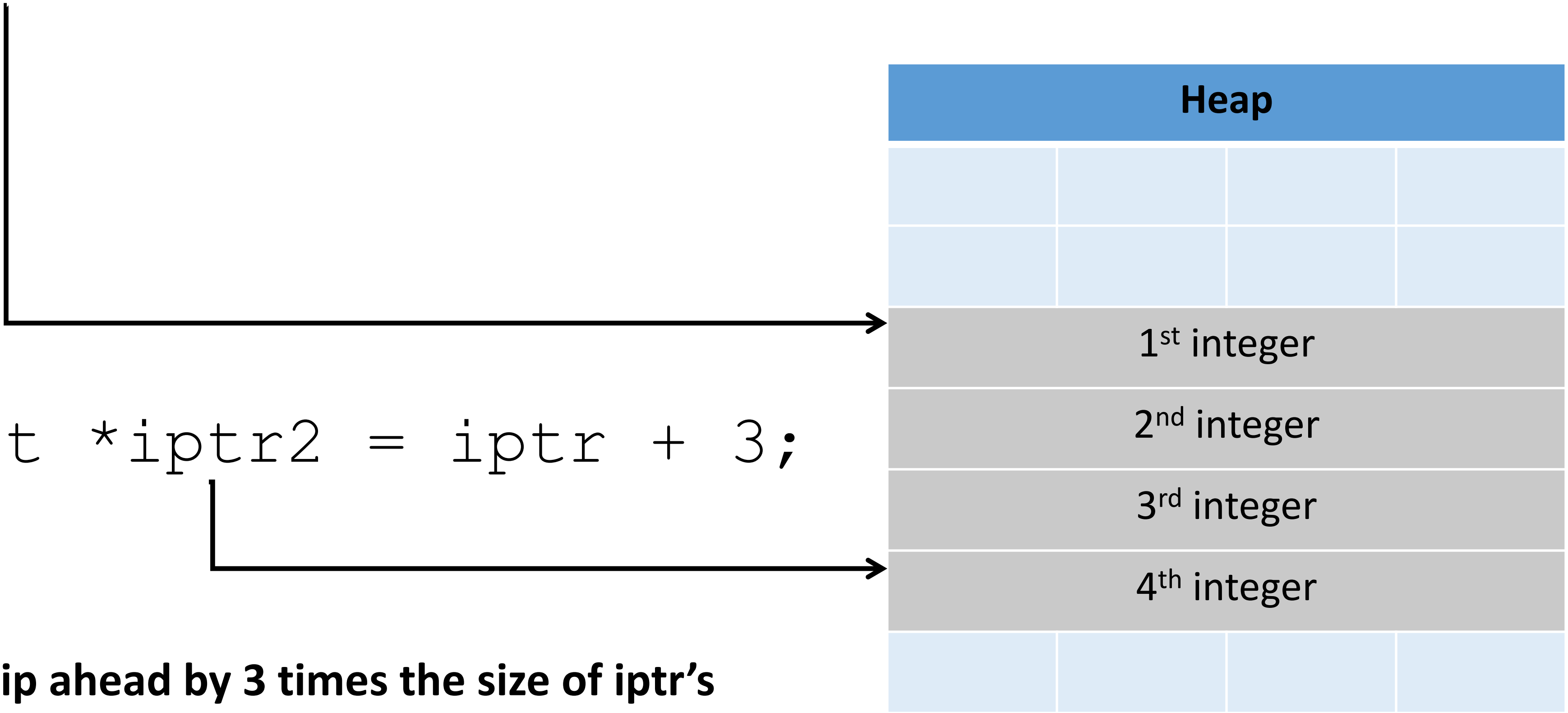
Pointer Arithmetic

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```



Pointer Arithmetic

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```



```
int *iptr2 = iptr + 3;
```

Skip ahead by 3 times the size of iptr's type (integer, size: 4 bytes).

Other uses for pointers...

1. Allowing a function to modify a variable.

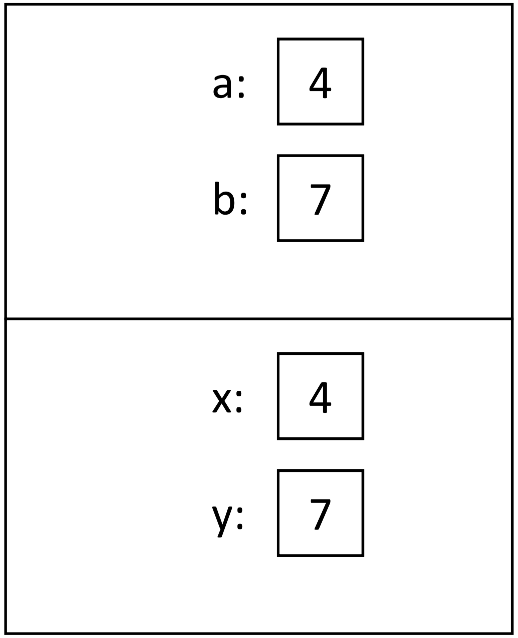
2. Allowing a function to return memory.

3. Many more…

Function Arguments

- Arguments are **passed by value**
  - The function gets a separate copy of the passed variable

```
int func(int a, int b) {
        a = a + 5;
        return a - b;
}

int main() {
        int x, y;  // declare two integers
        x = 4;
        y = 7;
        y = func(x, y);
        printf(“%d, %d”, x, y);
}
```



func:

a: 4

b: 7

main:

x: 4

y: 7

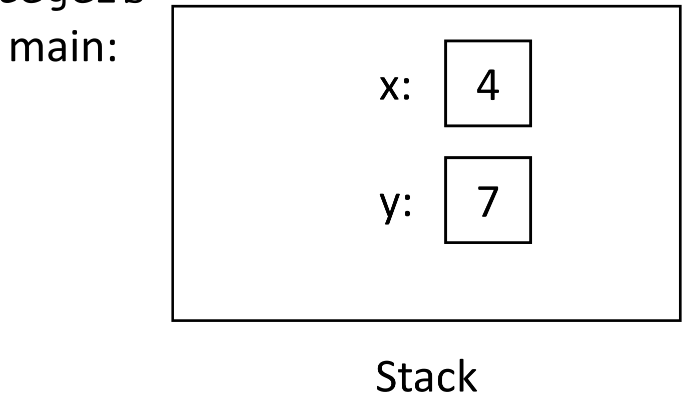
Stack

Function Arguments

- Arguments are **passed by value**
  - The function gets a separate copy of the passed variable

```
int func(int a, int b) {
        a = a + 5;
        return a - b;
}

int main() {
        int x, y;  // declare two integers
        x = 4;
        y = 7;
        y = func(x, y);
        printf(“%d, %d”, x, y);
}
```

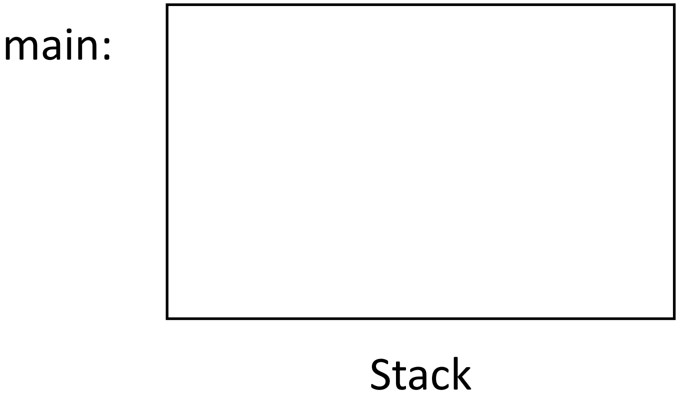
It doesn’t matter what func does with a and b. The value of x in main doesn’t change.



Function Arguments

- Arguments can be pointers!
  - The function gets the address of the passed variable!

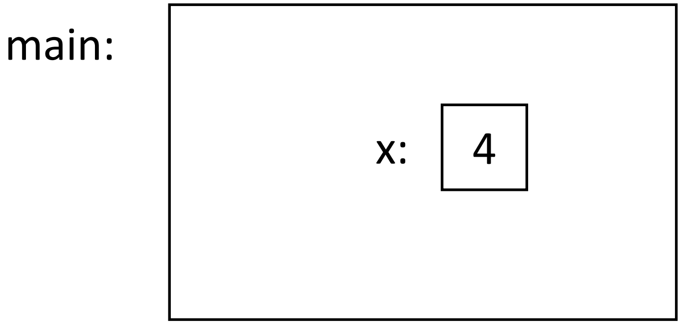
```
void func(int *a) {
      *a = *a + 5;
}

int main() {
      int x = 4;

      func(&x);
      printf(“%d”, x);
}
```



Pointer Arguments

- Arguments can be pointers!
  - The function gets the address of the passed variable!

```
void func(int *a) {
      *a = *a + 5;
}

int main() {
      int x = 4;

      func(&x);
      printf(“%d”, x);
}
```

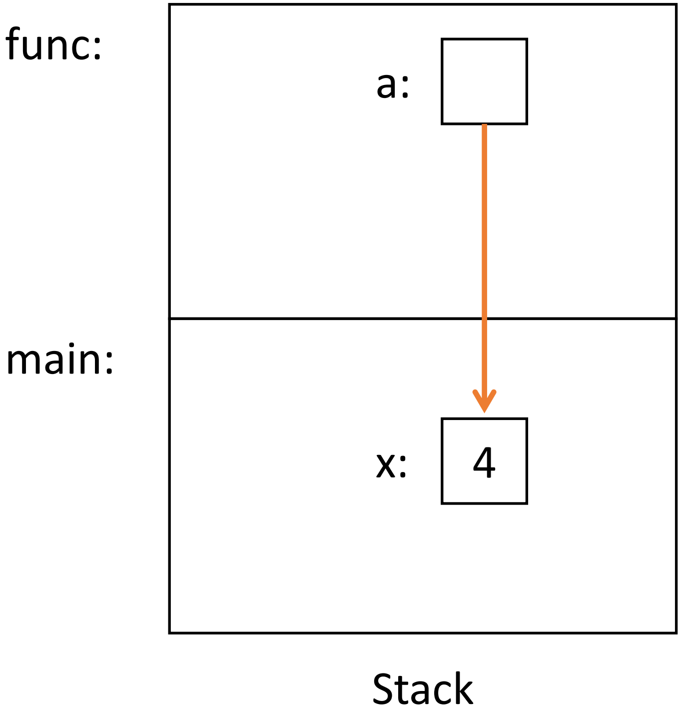


Stack

Pointer Arguments

- Arguments can be pointers!
  - The function gets the address of the passed variable!

```
void func(int *a) {
        *a = *a + 5;
}

int main() {
        int x = 4;

        func(&x);
        printf(“%d”, x);
}
```

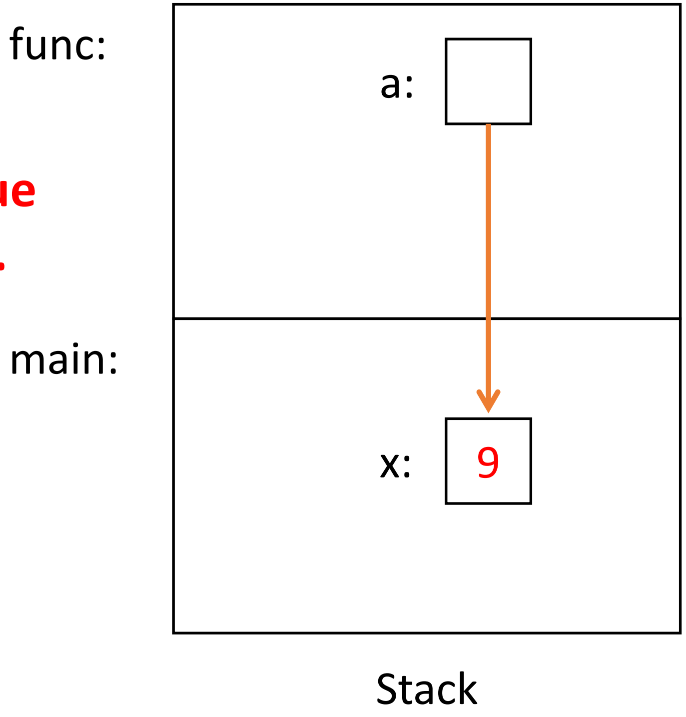


Pointer Arguments

- Arguments can be pointers!
  - The function gets the address of the passed variable!

```
void func(int *a) {
        *a = *a + 5;
}

int main() {
        int x = 4;

        func(&x);
        printf(“%d”, x);
}
```

Dereference pointer, set value that a points to.



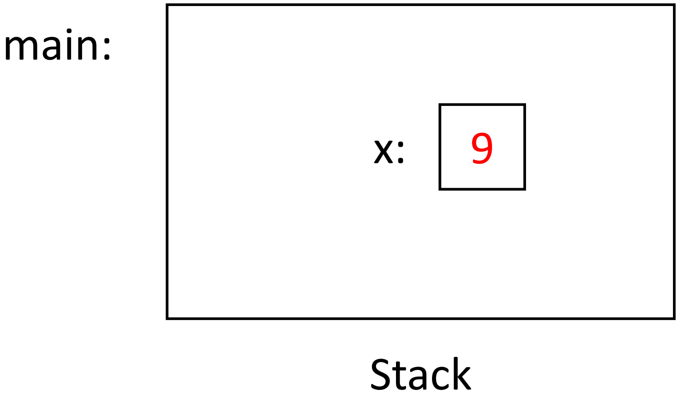
Pointer Arguments

- Arguments can be pointers!
  - The function gets the address of the passed variable!

```
void func(int *a) {
      *a = *a + 5;
}

int main() {
      int x = 4;

      func(&x);
      printf(“%d”, x);
}
```

Prints: 9

Haven’t we seen this somewhere before?


main:

x: 9

Stack

Readfile Library

- We’ve seen saw this in lab 2 and 4 with `read_int` and `read_float`.
    - This is why you needed an `&`.
    - e.g.,

```
int value;
status_code = read_int(&value);
```

- You’re asking `read_int` to modify a parameter, so you give it a pointer to that parameter.
    - `read_int` will dereference it and set it.

Other uses for pointers…

1. Allowing a function to modify a variable.

2. Allowing a function to return memory.

3. Many more…

Can you return an array?

- Suppose you wanted to write a function that copies an array of integers.

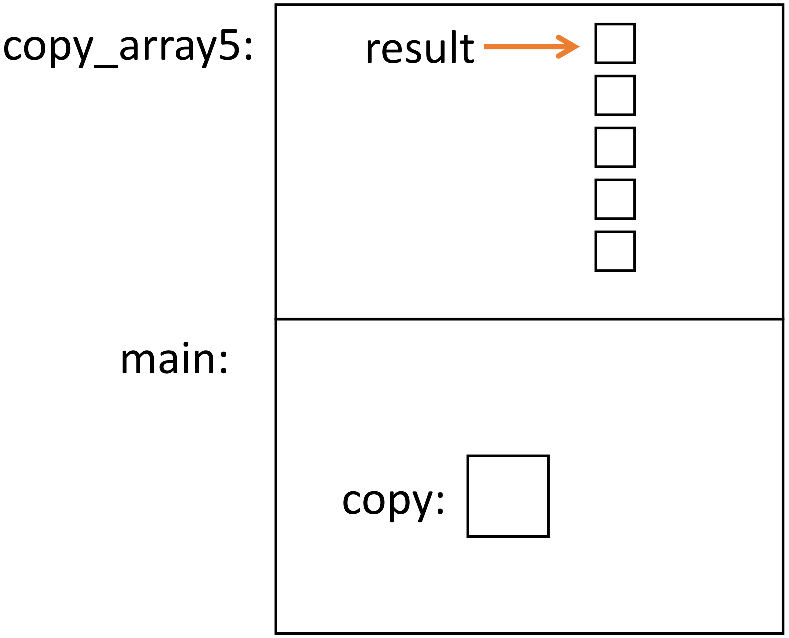
How many bugs can you find?
A=1, B=2, …

```
copy_array(int array[], int len) {
  int result[len];
  for(int i=0; i<len; i++)
    result[i] = array[i];
  return result;
}
```

This is a terrible idea.
(Don’t worry, compiler wont let you do this anyway.)

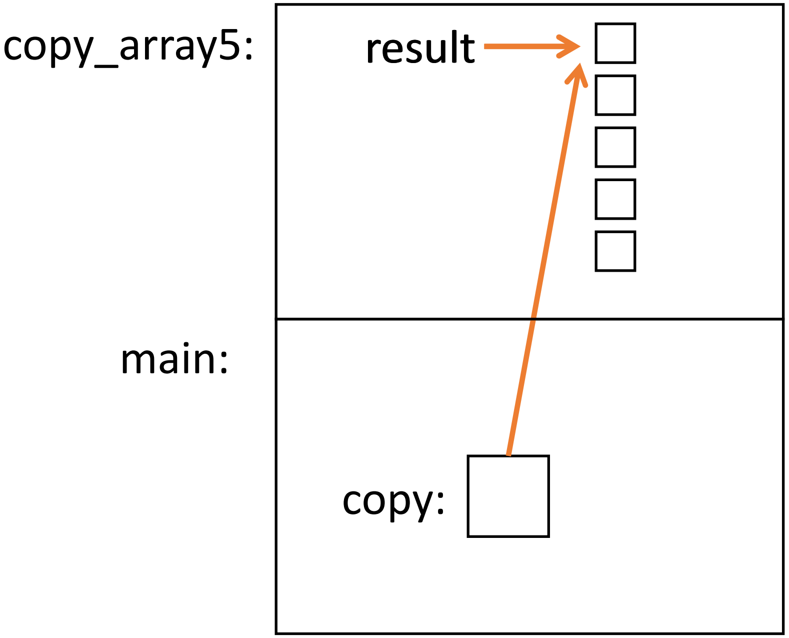
Consider the memory…

```
copy_array5(int array[]) {
  int result[5];
  for(int i=0; i<5; i++)
    result[i] = array[i];
  return result;
}

(In main):
copy = copy_array(…)
```



Consider the memory…

```
copy_array5(int array[]) {
  int result[5];
  for(int i=0; i<5; i++)
    result[i] = array[i];
  return result;
}

(In main):
copy = copy_array(…)
```

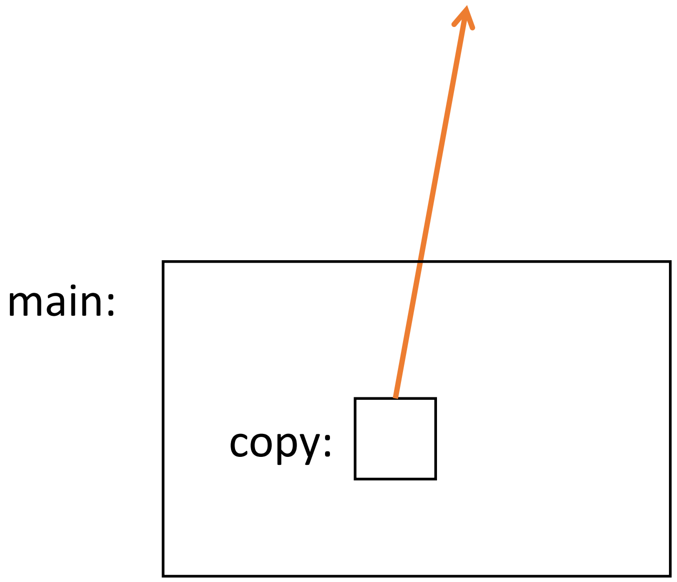


Consider the memory…

When we return from copy_array, its stack frame is gone!

Left with a pointer to nowhere.

```
(In main):
copy = copy_array(…)
```

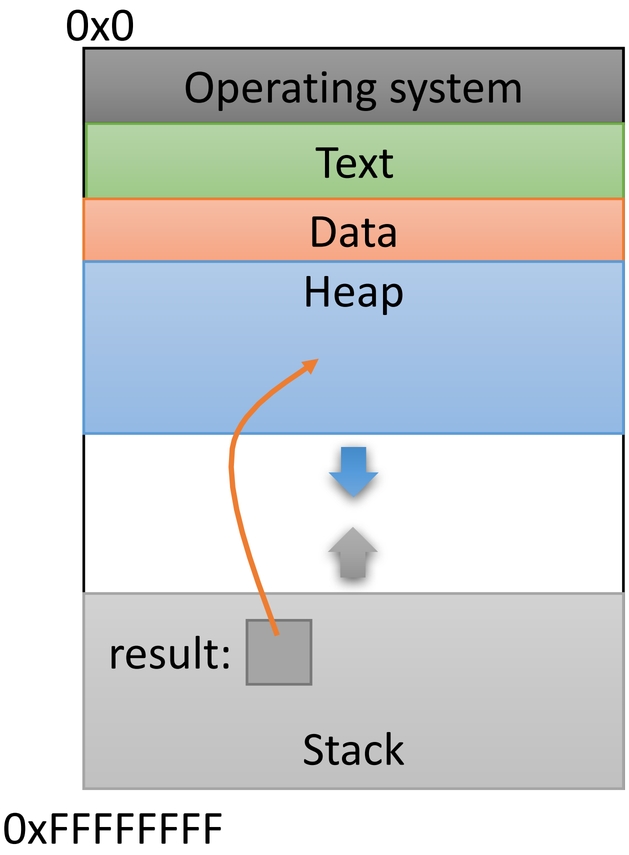


main:

copy:

Using the Heap

```
int *copy_array(int array[], int len) {
  int *result = malloc(len * sizeof(int));
  for(int i=0; i<len; i++)
    result[i] = array[i];
  return result;
}
```



malloc memory is on the heap.

Doesn’t matter what happens on the stack (function calls, returns, etc.)

Other uses for pointers…

1. Allowing a function to modify a variable.

2. Allowing a function to return memory.

   - These are both very common.
     You should be using them in lab 4.

3. Many more…
   - Avoiding copies (structs … coming up shortly)
   - Sharing between threads (end of the semester)

Pointers to Pointers

- Why stop at just one pointer?

```
int **double_iptr;
```

- "A pointer to a pointer to an int."
  - Dereference once: pointer to an int
  - Dereference twice: int

- Commonly used to:
  - Allow a function to modify a pointer (data structures)
  - Dynamically create an array of pointers.
  - (Program command line arguments use this.)

```
int main(int argv, char** argv)
```

Recall: structs on the heap

```
struct student {
  char name[40];
  int age;
  double gpa;
}

struct student *bob = NULL;
bob = malloc(sizeof(struct student));
```

Pointers to Structs -> operator

```
struct student *bob = NULL;
bob = malloc(sizeof(struct student));

(*bob).age = 20;
bob->gpa = 3.5;
```

The `->` operator is a shortcut to do a dereference (`*`) and a field access (`.`).

Arrays vs. Pointers

How are array variables different from pointer variables? Think of as many differences as you can.

- Declared differently.

```
int arr[5];
int *ptr;
ptr = malloc(5 * sizeof(int));
```

- Stored differently
  - Stack
  - Heap
- Pointers are Lvalues

Lvalues

- Anything that can be on the left-hand side of an assignment.
- Examples:
    - `int`
    - `float`
    - Structs (e.g. `struct student`) ⬅
    - pointers

- Arrays are not lvalues. You can’t move a different address into an array variable.

Struct Parameters: Pass by Value:

```
void test(struct student s1){
    s1.age = 20;
    s1.name[0] = ‘X’;
}
int main() {
  struct student jo;
  strcpy(jo.name, “Jo”);
  jo.age = 18;
  test(jo);
}
```

Q1: What is the value of the argument jo?

Q2: What value does param s1 get?

Q3: Draw the Stack

Q4: What is value of jo after the function call?