CS 31: Introduction to Computer Systems

09: C Pointers and Assembly

02-18-2025

SWARTHMORE COLLEGE

Announcements

- Midterm-1 in-class next Tuesday

Reading Quiz

- Note the red border!
- 1 minute per question
- No talking, no laptops, phones during the quiz

<u>Check your frequency:</u>

- Iclicker2: frequency AA
- Iclicker+: green light next to selection

For new devices this should be okay,
For used you may need to reset frequency

Reset:

1. hold down power button until blue light flashes (2secs)
2. Press the frequency code: AA vote status light will indicate success

What we will learn this week

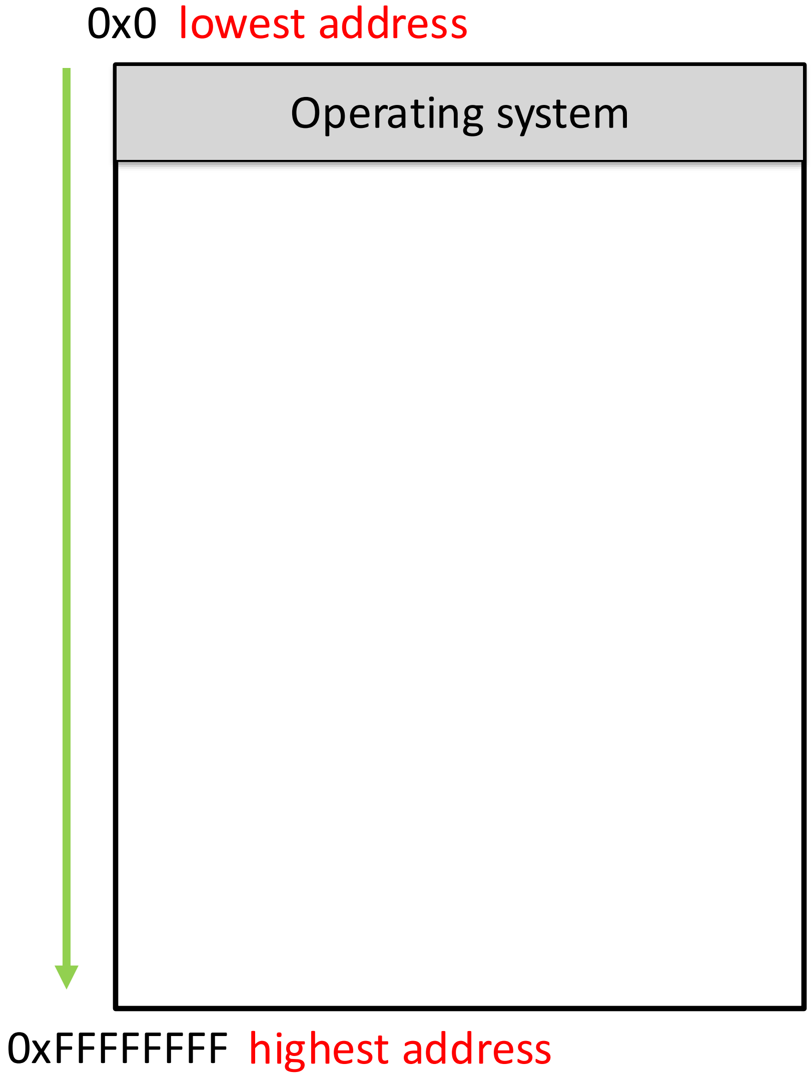
1. C Pointers, and Main Memory
   - Parts of Program Memory
   - C's support for dynamic memory allocation
   - C pointer variables that refer to memory locations
   - Where are instructions, stack, etc., in program's memory space?

2. Instruction set architecture (ISA)
   - Interface between programmer and CPU
   - Established instruction format (assembly lang)
   - Assembly programming (x86_64)

Overview

- How to reference the location of a variable in memory
- Where variables are placed in memory
- How to make this information useful
  - Allocating memory
  - Calling functions with pointer arguments

Memory

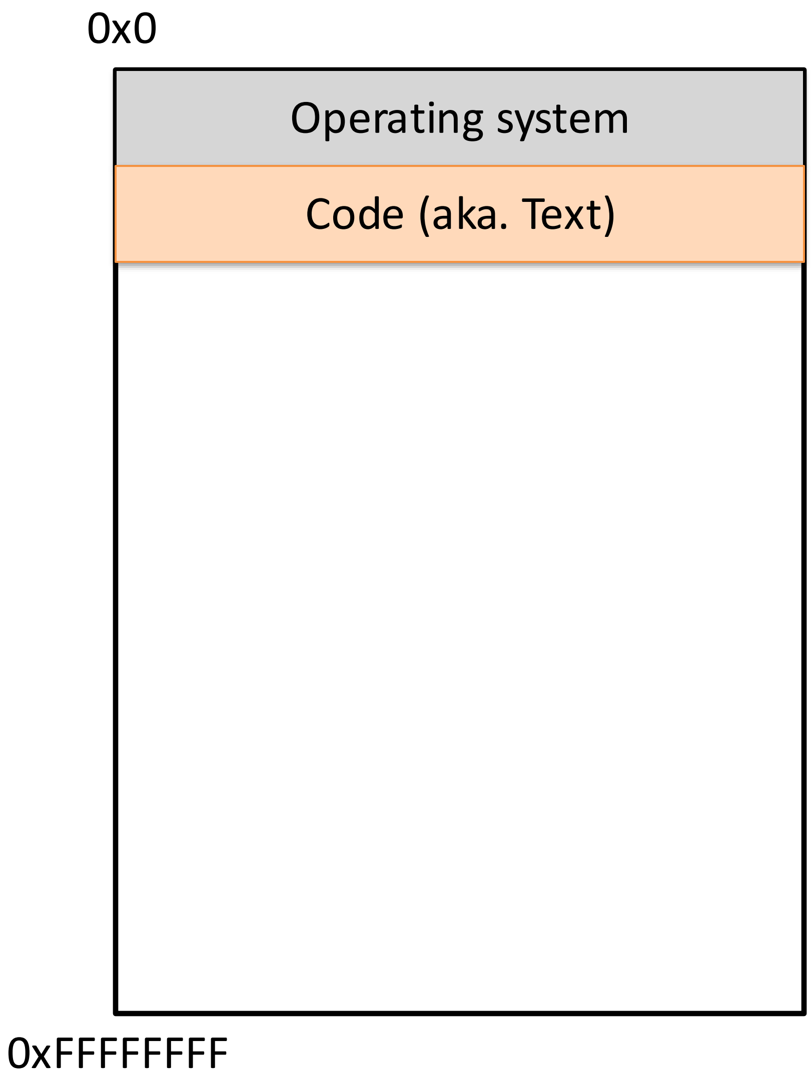


- Behaves like a big array of bytes, each with an address (bucket #)
- By convention, we divide it into regions, ordered from **lowest** to **highest**
- The region at the lowest addresses is usually reserved for the OS

Memory - Text

- After the OS, we store the program's code

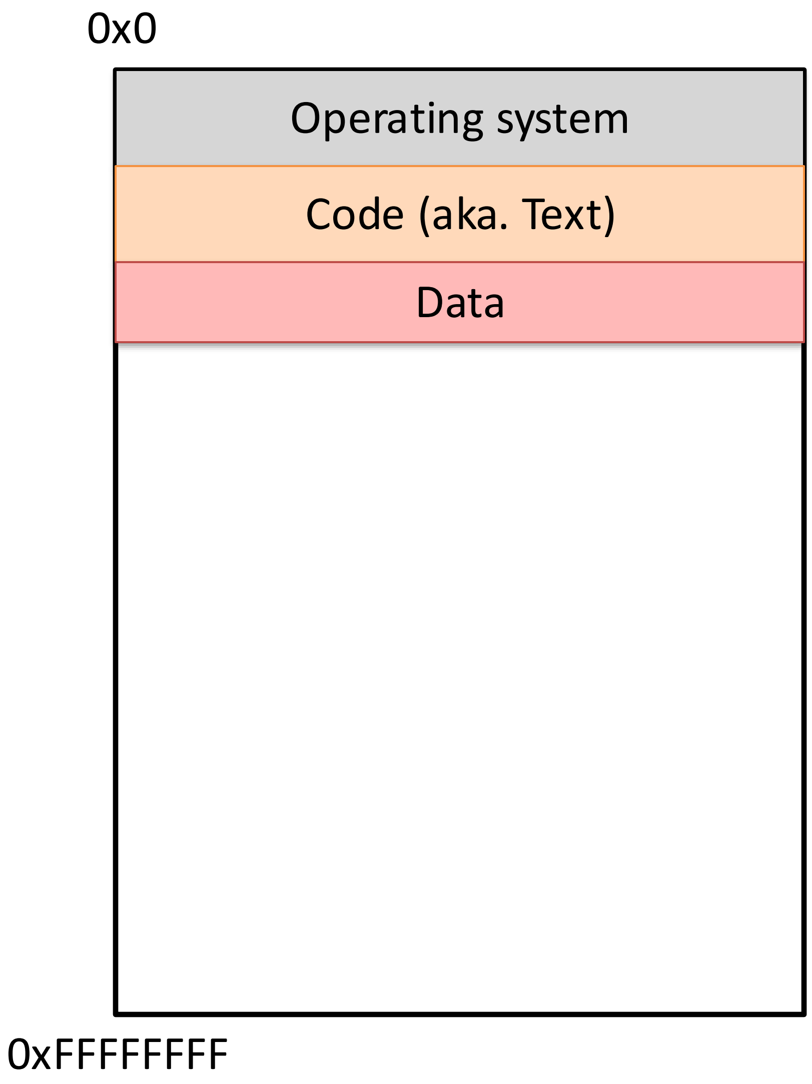
- Instructions generated by the compiler



Memory – (Static) Data

- Next, there's a fixed-size region for static data

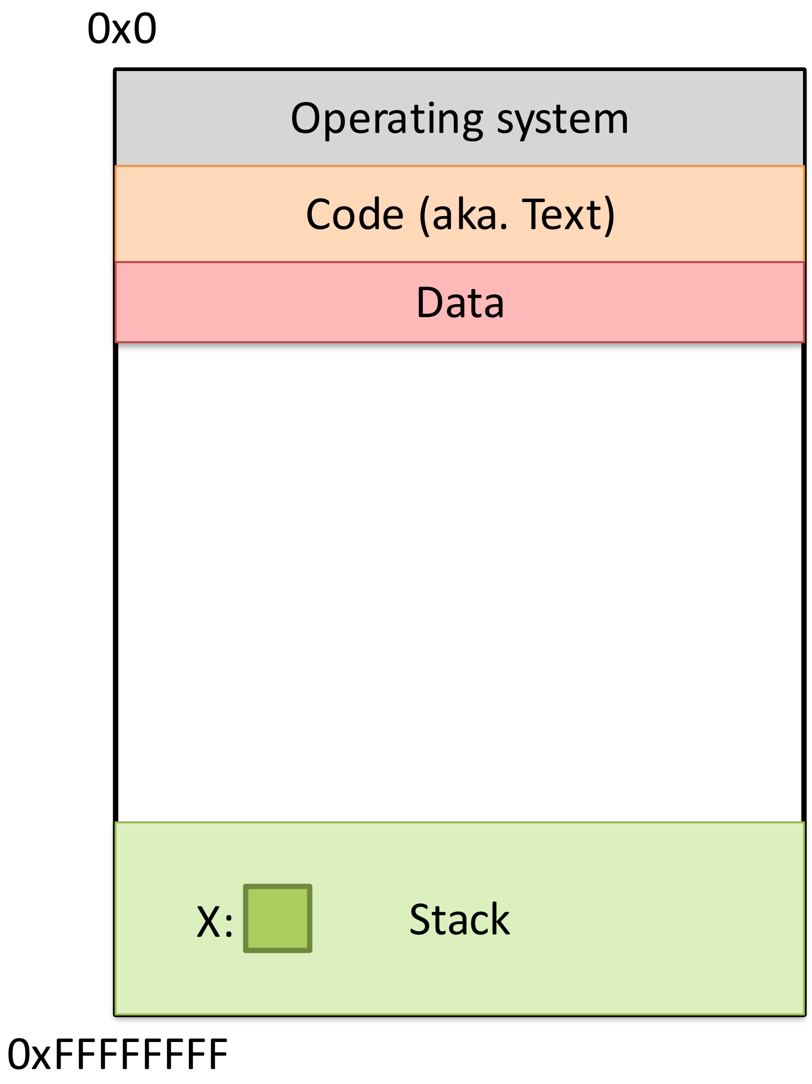
- This stores static variables that are known at compile time
  - Global variables
    - Note: Avoid using global variables!
  - Static (hard-coded) strings



Memory - Stack

- At high addresses, we keep the stack

- This stores local variables
  - The kind we've been using in C so far
  - e.g., `int x;`

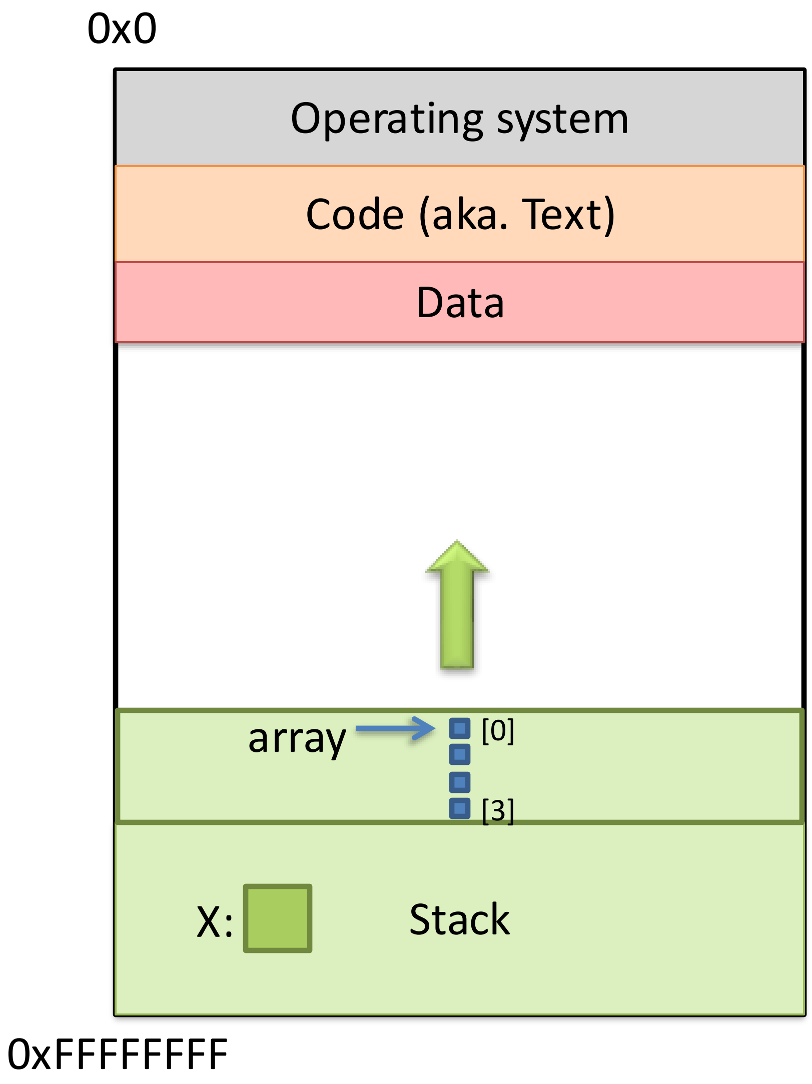


Memory - Stack

- The stack grows upwards towards lower addresses
- Example: Allocating array

```
int array[4];
```

- (Note: this differs from Python)



C Pointers Introduction

What is a pointer?

A pointer is like a mailing address,
it tells you where something is **located**.

Every object (including simple data types) reside in the **memory** of the machine.

A **pointer** is an "address" telling you where that variable is **located** in **memory**.

Pointers

- Pointer: A variable that stores a reference (index) to a memory location.

- Pointer: sequence of bits that should be interpreted as an index into memory.

- Where have we seen this before?

Putting a * in front of a variable…

- When you ***declare*** the variable:
  - Declares the variable to be a pointer
  - It stores a memory address

- When you ***get the value*** at mem. location in the pointer (dereference):
  - Like putting () around a register name
  - We follows the pointer out to memory, get the value
  - Data we access will be of the specified type
    - e.g., pointer (mem. address) to an int, pointer (mem. address) to a float .., etc.

Three Rules for Using Pointer Variables

1. Declare pointer variable: `<type> * <name>;`

it doesn't matter where the * is (note the spaces)

- This is a *promise* to the compiler:
  "This variable holds a memory address and if you follow what it points to in memory (dereference it), you'll find an integer"

```
int * ptr, x, y;
char * chptr, s;
```

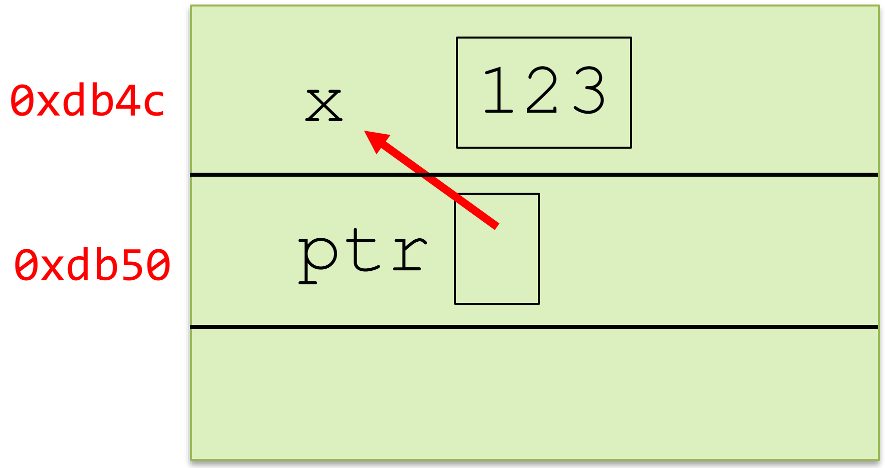
- `x` and `y` are of type **`int`**;
- `ptr` is a pointer to an int (**`int *`**),
- `chptr` is a pointer to a char (**`char *`**)
- `ptr` and `chptr` are both pointer variables but are of *different* types

Three Rules for Using Pointer Variables

2a. Initialize it (make it point to something):

```
int x;
int * ptr = &x; // ptr stores address of x  or ptr points to x
```

- & is called the "address of" operator and returns the address of that variable
- The address is a number/binary data
- We depict this relationship with the arrow pointing to the memory at that address



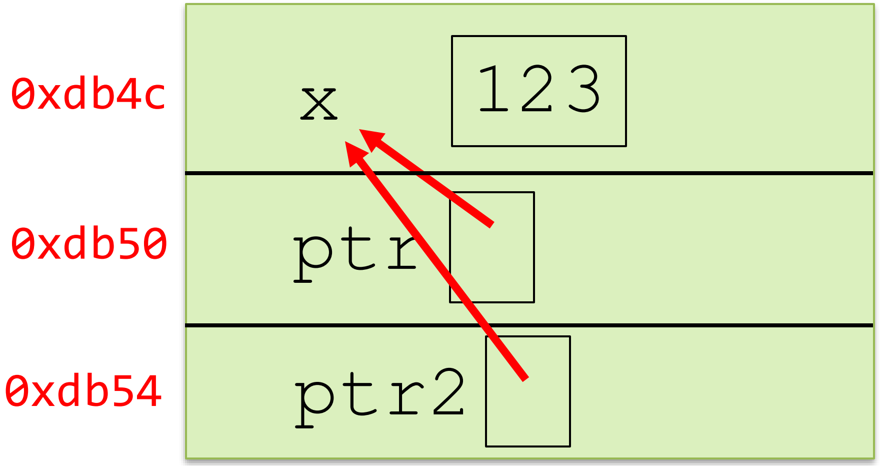
Stack

Address	Type	Name	Value
0xdb4c	int	x	123
0xdb50	int*	ptr	0xdb4c

Three Rules for Using Pointer Variables

2a. Initialize it (make it point to something):

```
int x;
int * ptr = &x; // ptr stores address of x  or ptr points to x

int * ptr2;
ptr2 = ptr; // ptr2 gets value of ptr
```



Stack

Address	Type	Name	Value
0xdb4c	int	x	123
0xdb50	int*	ptr	0xdb4c
0xdb54	int *	ptr2	0xdb4c

Suppose we set up a pointer like the one below. Which expression gives us 5, and which gives us a memory address?

* in front of a pointer, gets the value at that memory location

```
int *iptr = (the location of that memory);
```

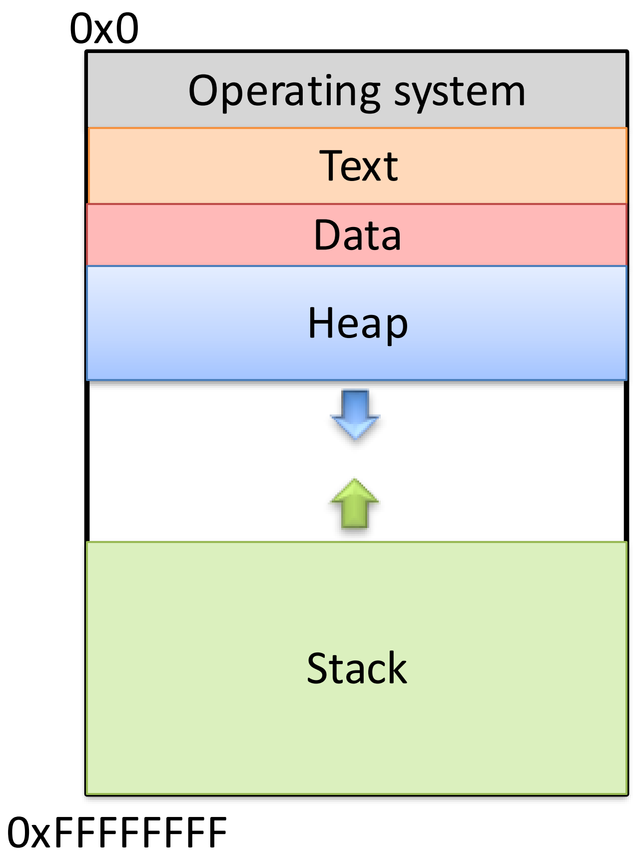
A. Memory address: `*iptr`, Value 5: `iptr`

B. Memory address: `iptr`, Value 5: `*iptr`

What will this do?

```
int main(void) {
   int *ptr;
   printf(“%d”, *ptr);
}
```

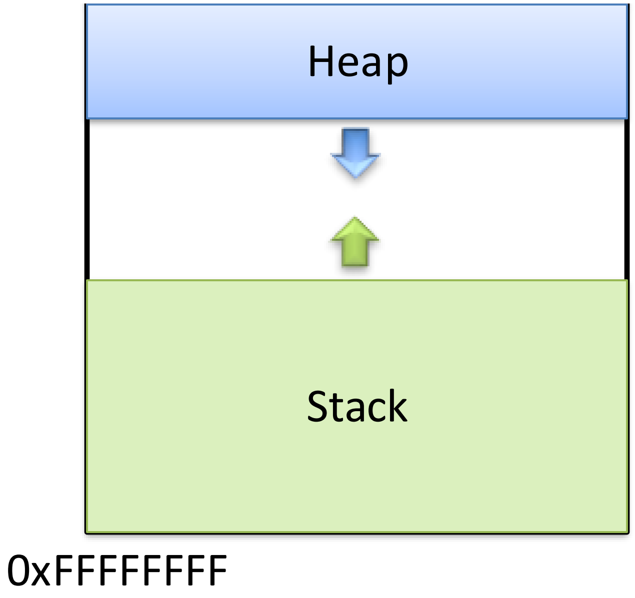
A. Print 0
B. Print a garbage value
C. Segmentation fault
D. Something else



What will this do?

```
int main(void) {
  int *ptr;
  printf(“%d”, *ptr);
}
```

A. Print 0
B. Print a garbage value
C. Segmentation fault
D. Something else



Takeaway: If you’re not immediately assigning your pointers when you declare it, initialize them to NULL

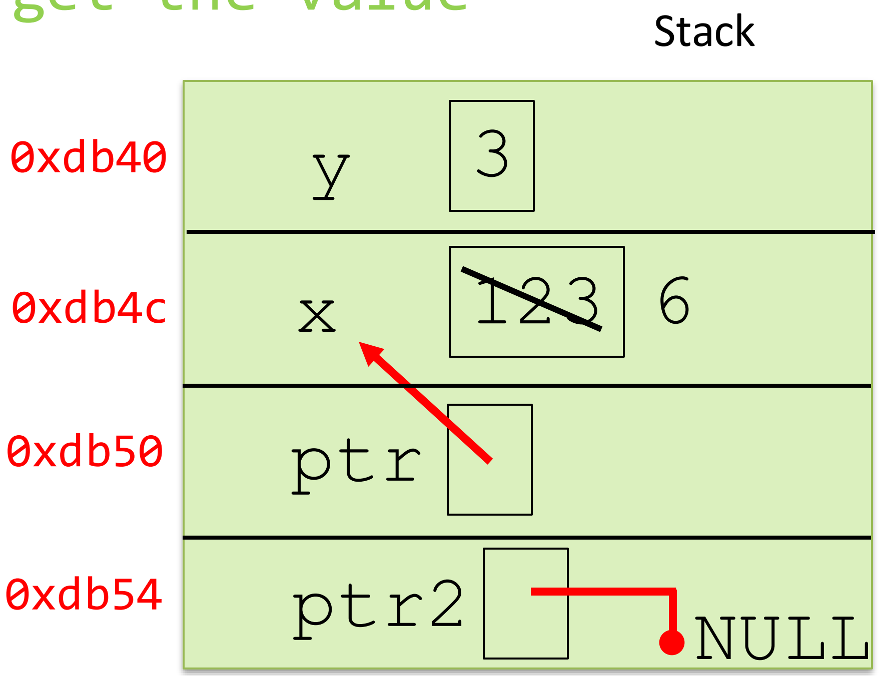
Three Rules for Using Pointer Variables

3a. Use it: dereference (*) it to set a value

```
* ptr = 6; //follow ptr out to memory, and get the value
           //at the memory address
```

3b. Use it: dereference (*) it to get a value

```
printf("%d", *ptr); // prints 6
int y = * ptr / 2; // y is set to 3
```



Can you dereference a NULL pointer?

```
ptr = NULL;
*ptr = 6; // ptr doesn't point to valid storage location
```

CRASH with segfault!

Why Pointers?

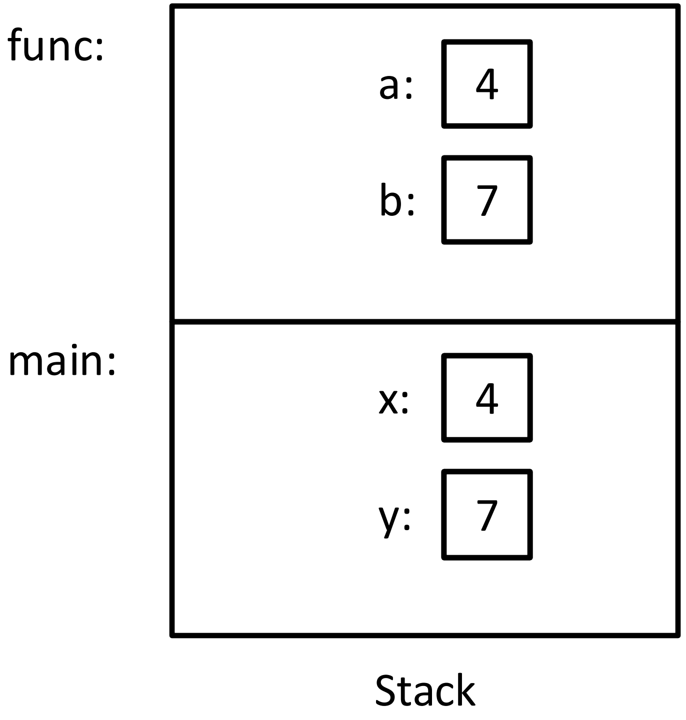
- Using pointers seems like a lot of work, and if used incorrectly, things can go wrong.
- Pointers also add a level of "indirection" to retrieve / store a value

- Two main benefits:
  1. "Pass by pointer" function parameters
     - By passing a pointer into a function, the function can dereference it so that the changes persist to the caller.
  2. Dynamic memory allocation
     - A program can allocate memory on demand, as it needs it during execution

Function Arguments

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

```
int func(int a, int b) {
    a = a + 5;
    return a - b; //DRAW STACK BEFORE RETURN
}

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

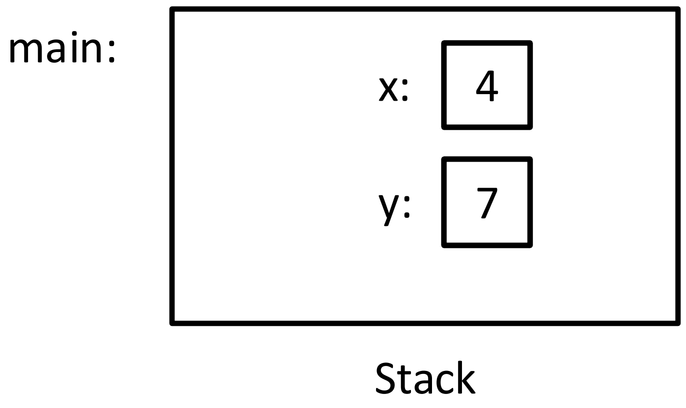


Function Arguments

- 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(void) {
    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.



Pass by Pointer

- Want a function to modify a value on the caller's stack? Pass a pointer!

- The called function can modify the memory location it points to.
  - passing the address of an argument to function:
  - pointer parameter *holds the address of* its argument
  - *dereference* parameter to modify argument's value

- You've already used functions like this:
  - `readfile` library functions and `scanf`
  - pass address of (&) argument to these functions

Function Arguments

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

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

int main(void) {
    int x = 4;

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

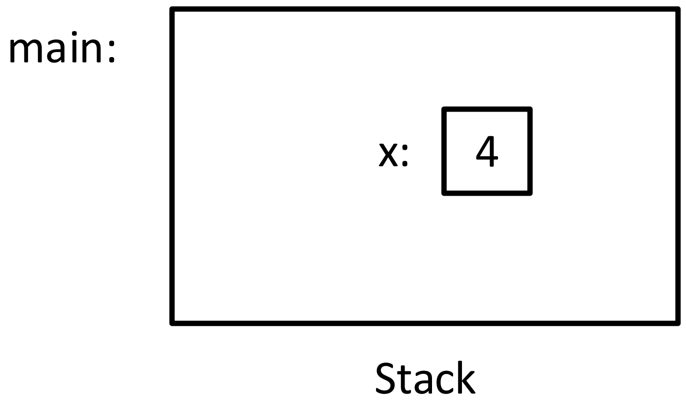
main:

Stack

Pointer Arguments

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

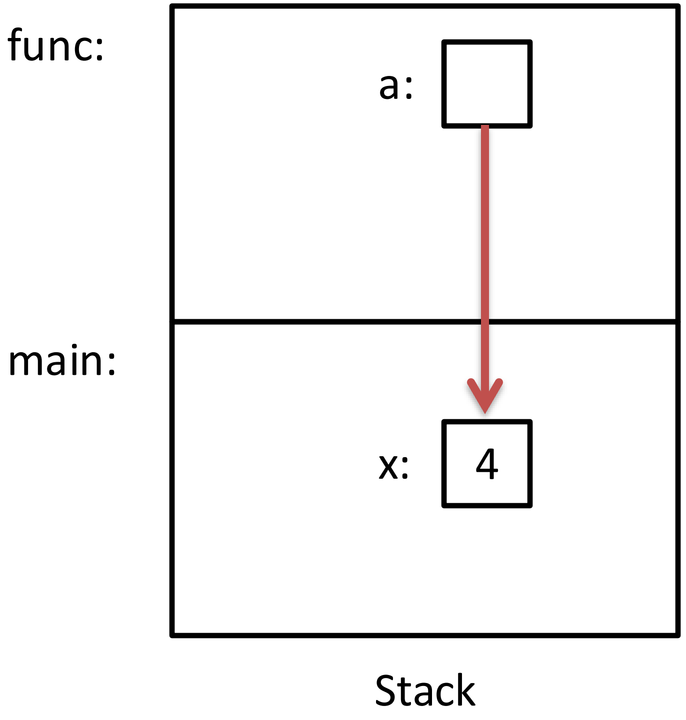
```c
void func(int *a) {
        *a = *a + 5;
}

int main(void) {
        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(void) {
        int x = 4;

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

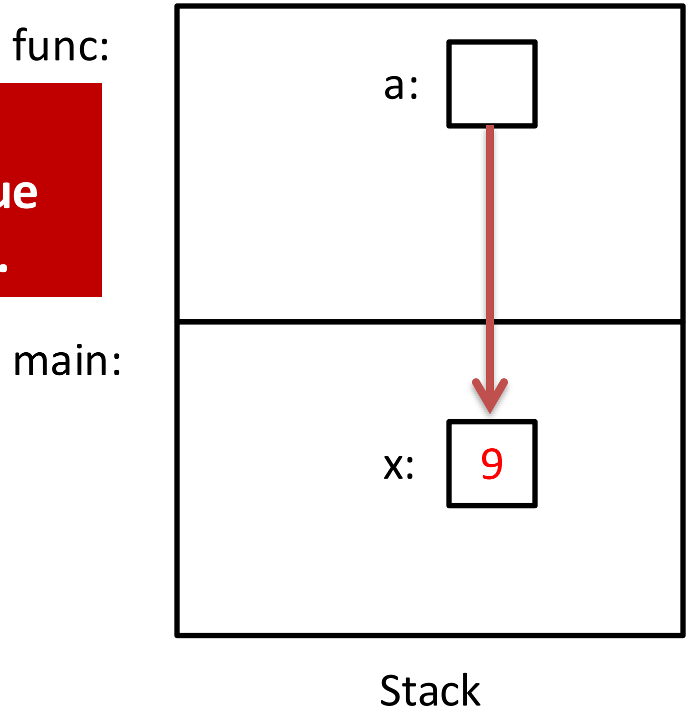


Pointer Arguments

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

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

int main(void) {
        int x = 4;

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

Dereference pointer, set value that a points to.



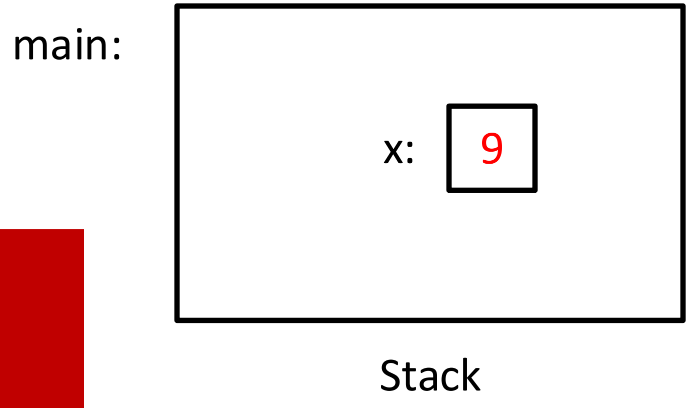
Pointer Arguments

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

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

int main(void) {
        int x = 4;

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

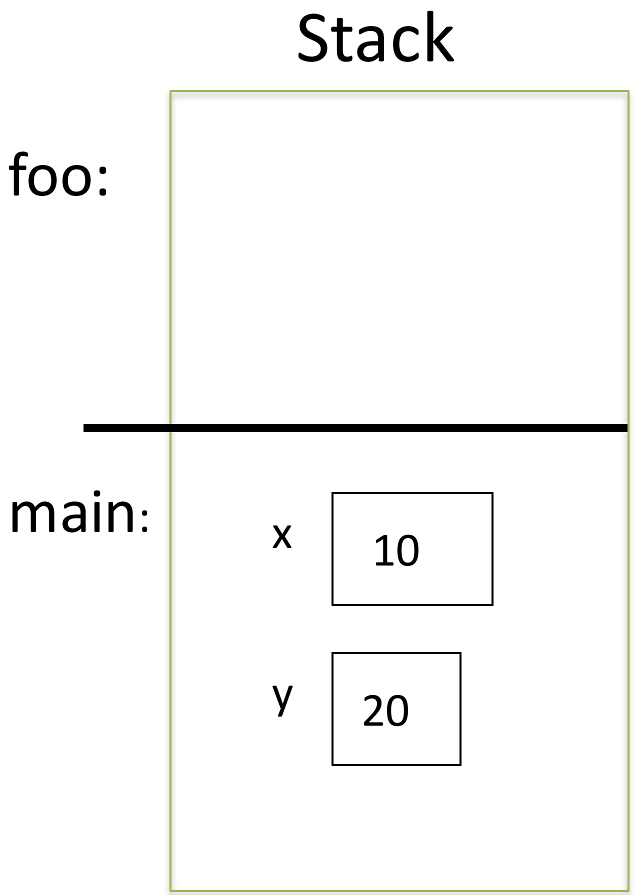
Prints: 9

Haven’t we seen this somewhere before?



Pass by Pointer - Example

```
int main(void){
   int x, y;
   x = 10; y = 20;
   foo(&x, y);
   …
}

void foo(int *b, int c){
   c = 99
  *b = 8;  // Stack drawn here
}
```

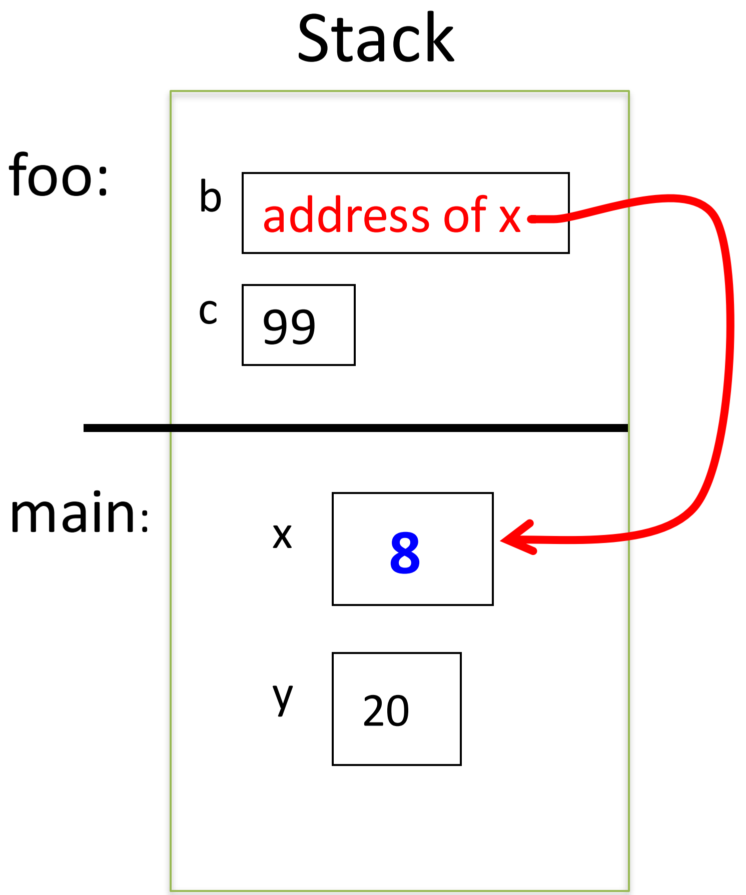


Pass by Pointer - Example

```
int main(void){
   int x, y;
   x = 10; y = 20;
   foo(&x, y);
   …
}

void foo(int *b, int c){
   c = 99
  *b = 8;  // Stack drawn here
}
```

pass the value of &x

dereference parameter b to set argument x's value



Passing Arrays

- An array argument’s value is its base address
- Array parameter “points to” its array argument

Passing Arrays

- An array argument's value is its base address
- Array parameter "points to" its array argument

```
int main(void){
   int array[10];
   foo(array, 10);
}
void foo(int arr[], int n){
   arr[2] = 6;
}
```

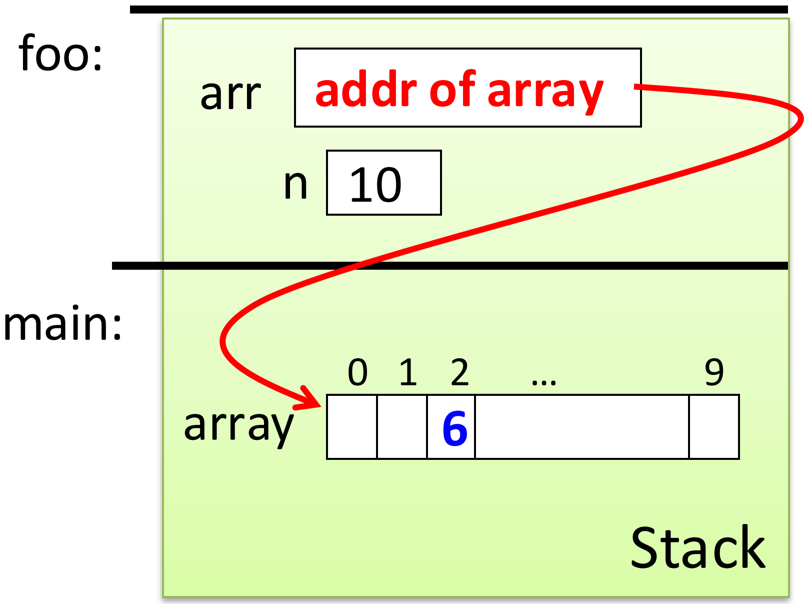
array base address

Passing Arrays

- An array argument's value is its base address
- Array parameter "points to" its array argument

```
int main(void){
   int array[10];
   foo(array, 10);
}
void foo(int arr[], int n){
   arr[2] = 6;
}
```

array base address

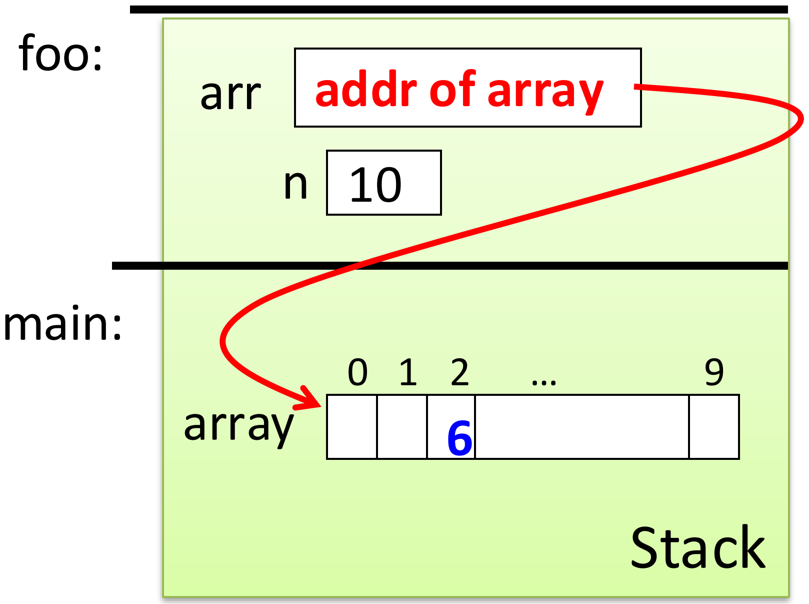


Passing Arrays

- An array argument's value is its base address
- Array parameter "points to" its array argument

```
int main(void){
   int array[10];
   foo(array, 10);
}
void foo( ________ , int n){
   arr[2] = 6;
}
```

alternative declaration?

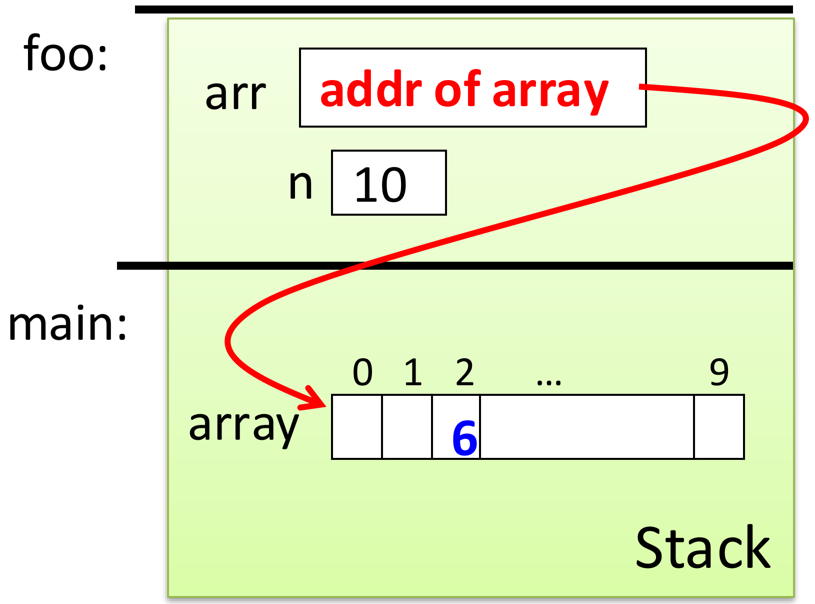


Passing Arrays

- An array argument's value is its base address
- Array parameter "points to" its array argument

```
int main(void){
   int array[10];
   foo(array, 10);
}
void foo(int *arr, int n){
   arr[2] = 6;
}
```

pass a pointer instead!



Why Pointers?

- Using pointers seems like a lot of work, and if used incorrectly, things can go wrong.
- Pointers also add a level of “indirection” to retrieve / store a value

- Two main benefits:
  1. “Pass by pointer” function parameters
     - By passing a pointer into a function, the function can dereference it so that the changes persist to the caller.
  2. **Dynamic memory allocation**
     - A program can allocate memory on demand, as needed during execution

Static vs. Dynamic Memory Allocation

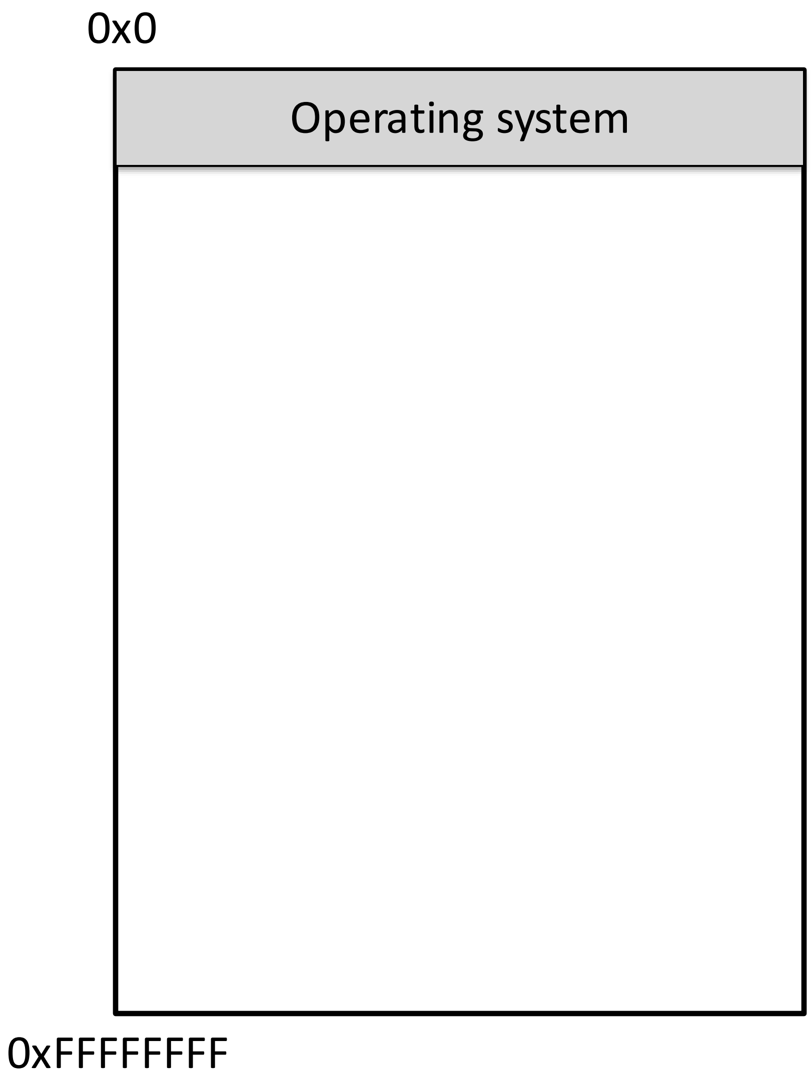
Static

- The compiler can know in advance
- The size of a C variable (based on its type)
- E.g., hard-coded constants where the size is known ahead of time

Dynamic

- The compiler cannot know — must be determined at run time
- User input (or things that depend on it, e.g., a file)
- E.g., create an array where the size is typed in by user (or file)

How is dynamically allocated memory **stored**?

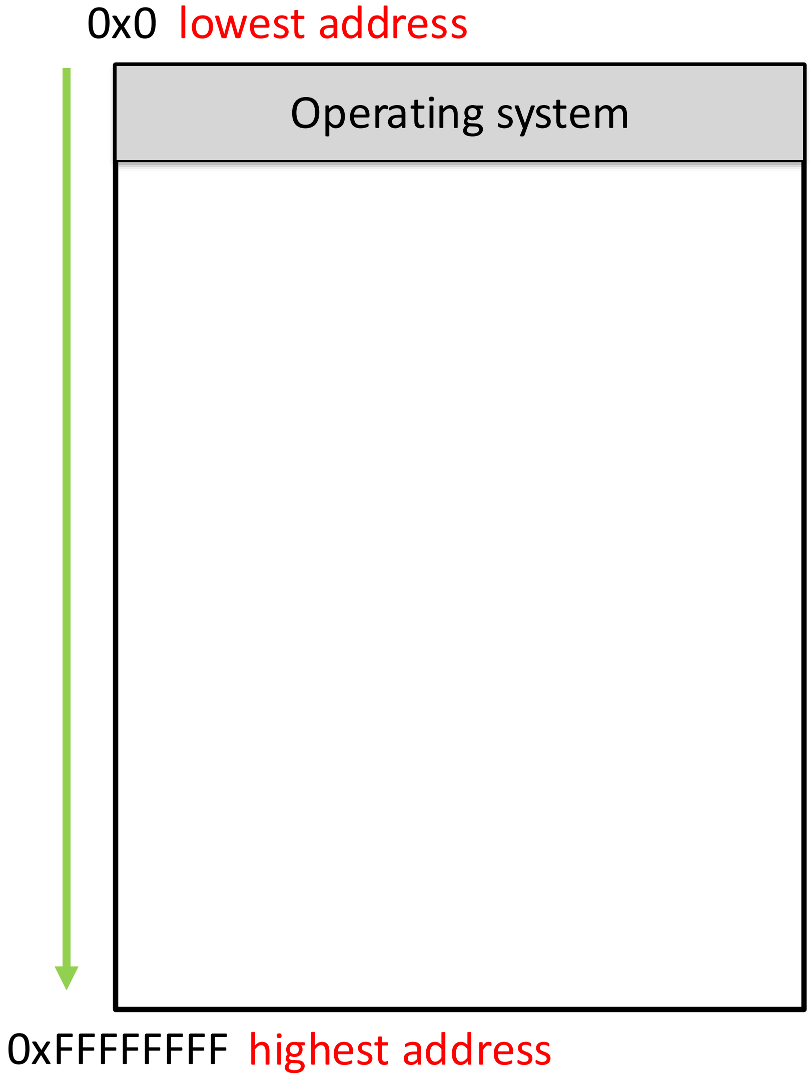


Memory

- Behaves like a big array of bytes, each with an address (bucket #)

- By convention, we divide it into regions, ordered from lowest to highest

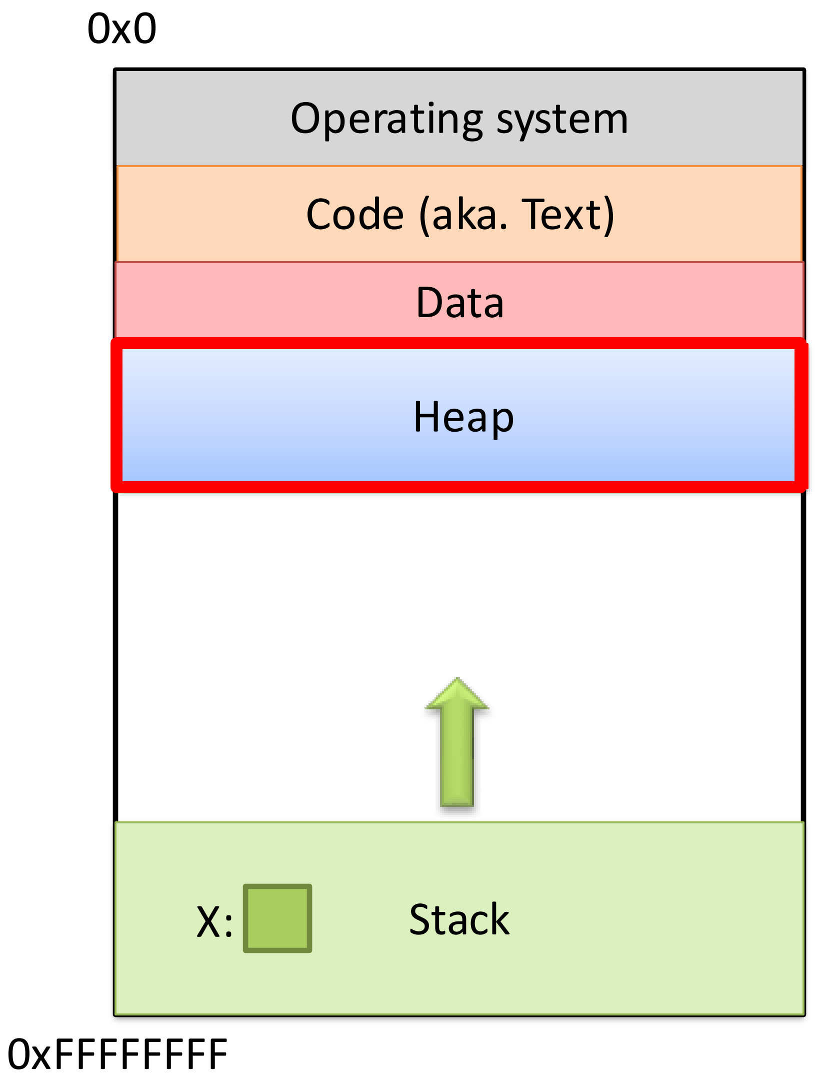
- The region at the lowest addresses is usually reserved for the OS



Memory - Heap

- The heap stores dynamically allocated variables
  - Variables are not allocated on the Heap, but variables can *point to* Heap memory

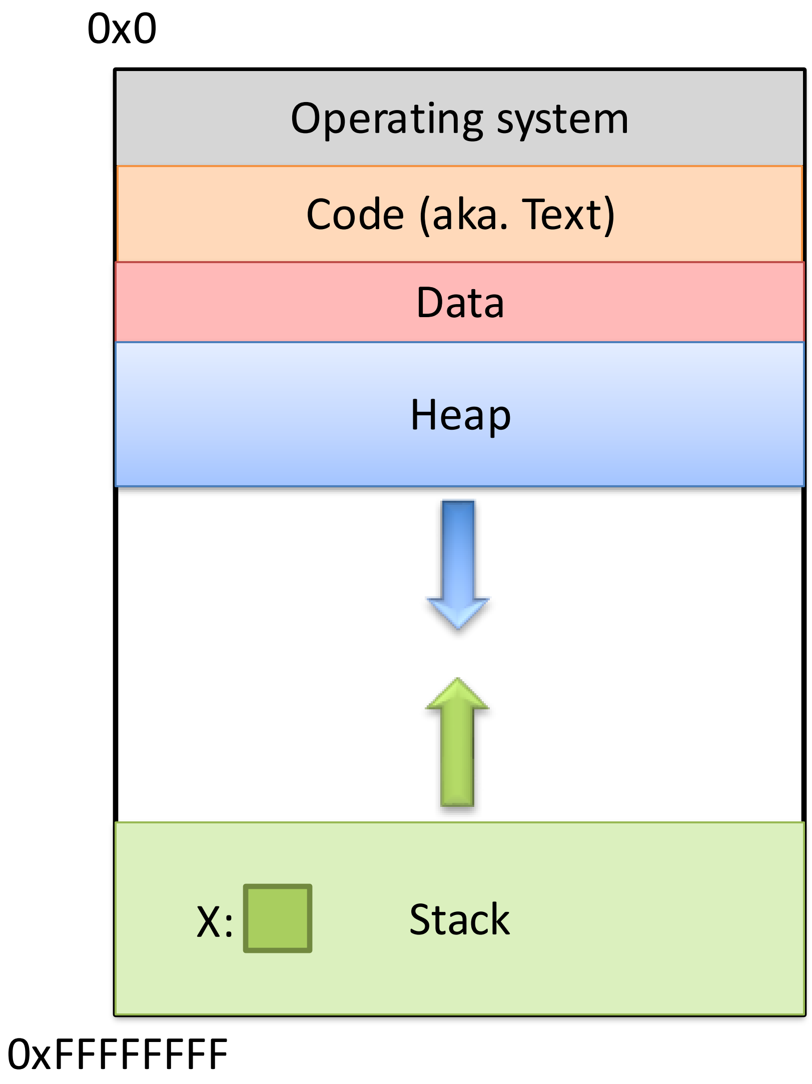
- When programs explicitly ask the OS for memory during runtime, it comes from the heap
  - malloc() function



Memory - Heap

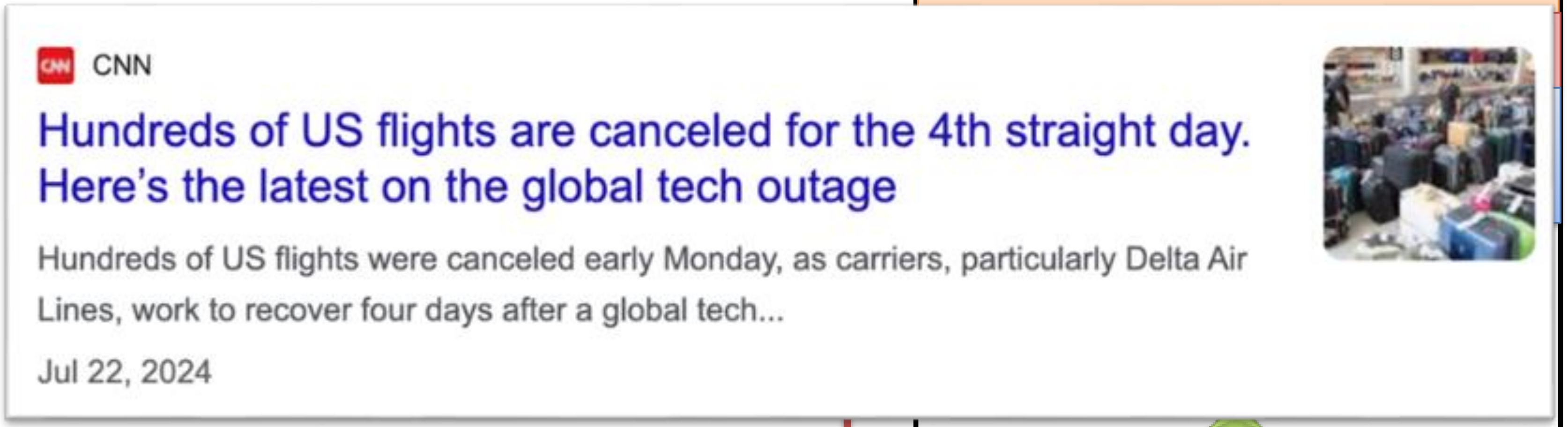
- The heap **grows downwards**, towards higher addresses

- I know you want to ask a question…



Memory - Heap

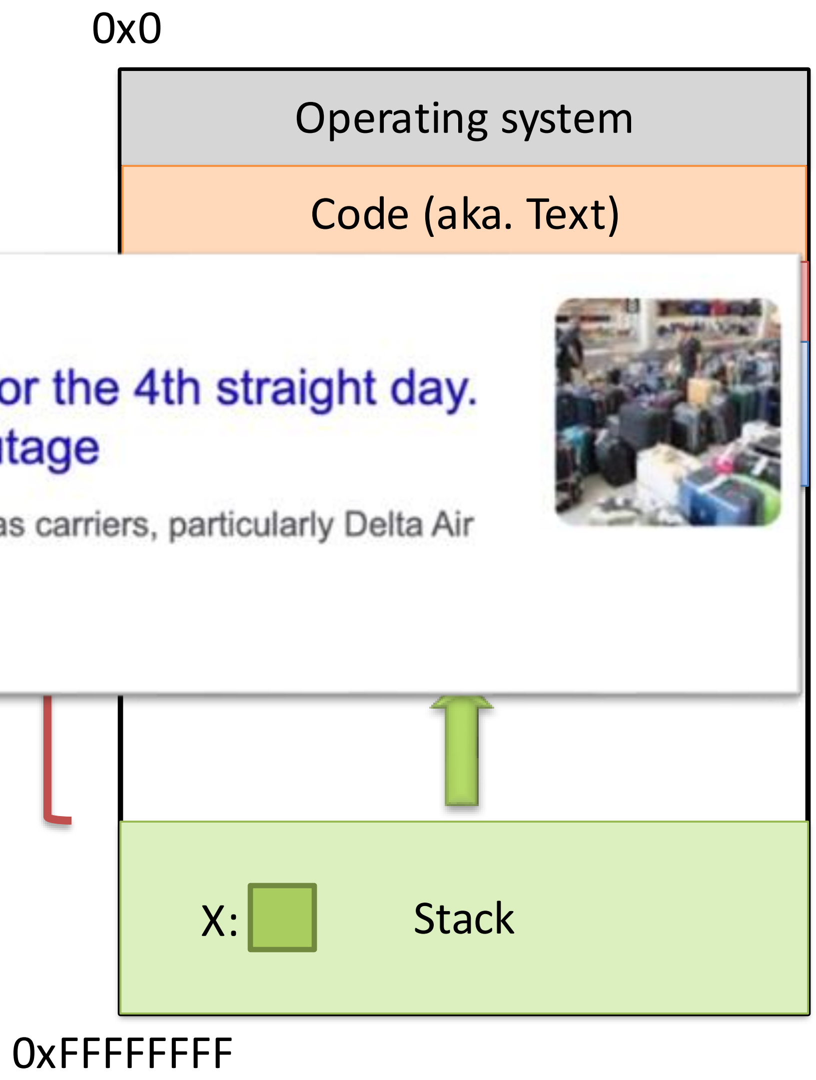
- “What happens if the heap and stack collide?”


CNN

Hundreds of US flights are canceled for the 4th straight day. Here’s the latest on the global tech outage

Hundreds of US flights were canceled early Monday, as carriers, particularly Delta Air Lines, work to recover four days after a global tech...

Jul 22, 2024

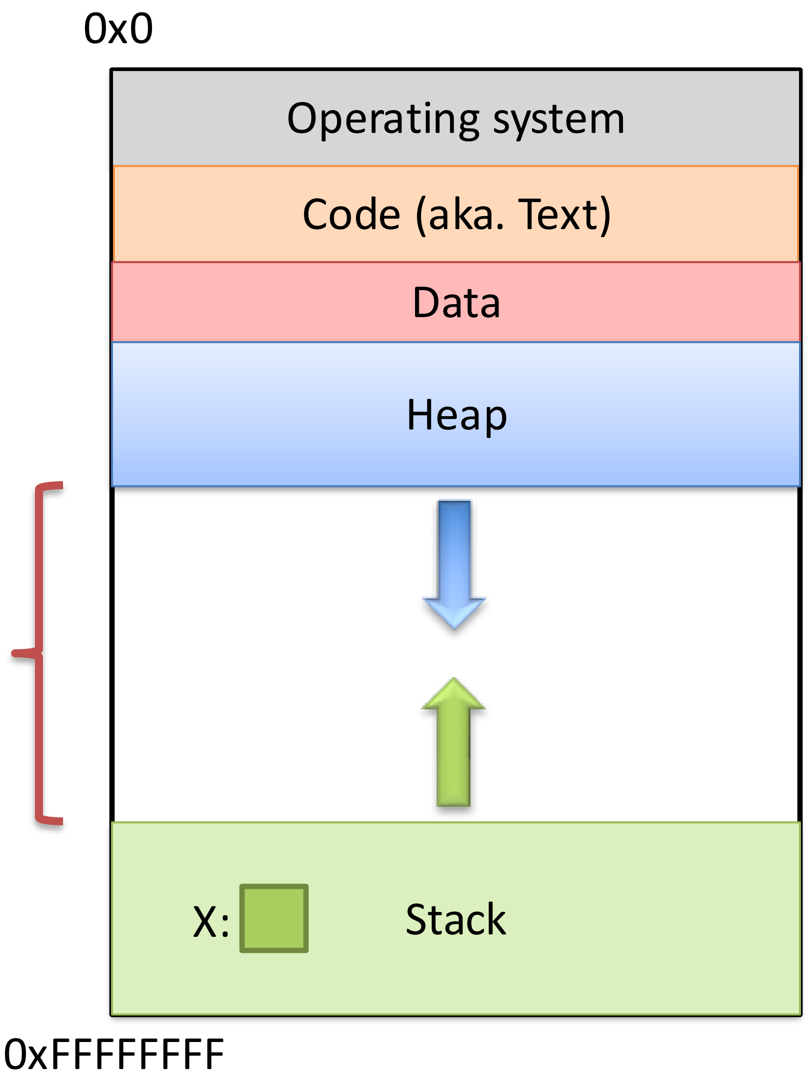


Memory - Heap

- "What happens if the heap and stack collide?"

- This picture is not to scale – the gap is huge

- The OS works really hard to prevent this
  - Would likely kill your program before it could happen



What should happen if we try to access an address that's NOT in one of these regions?

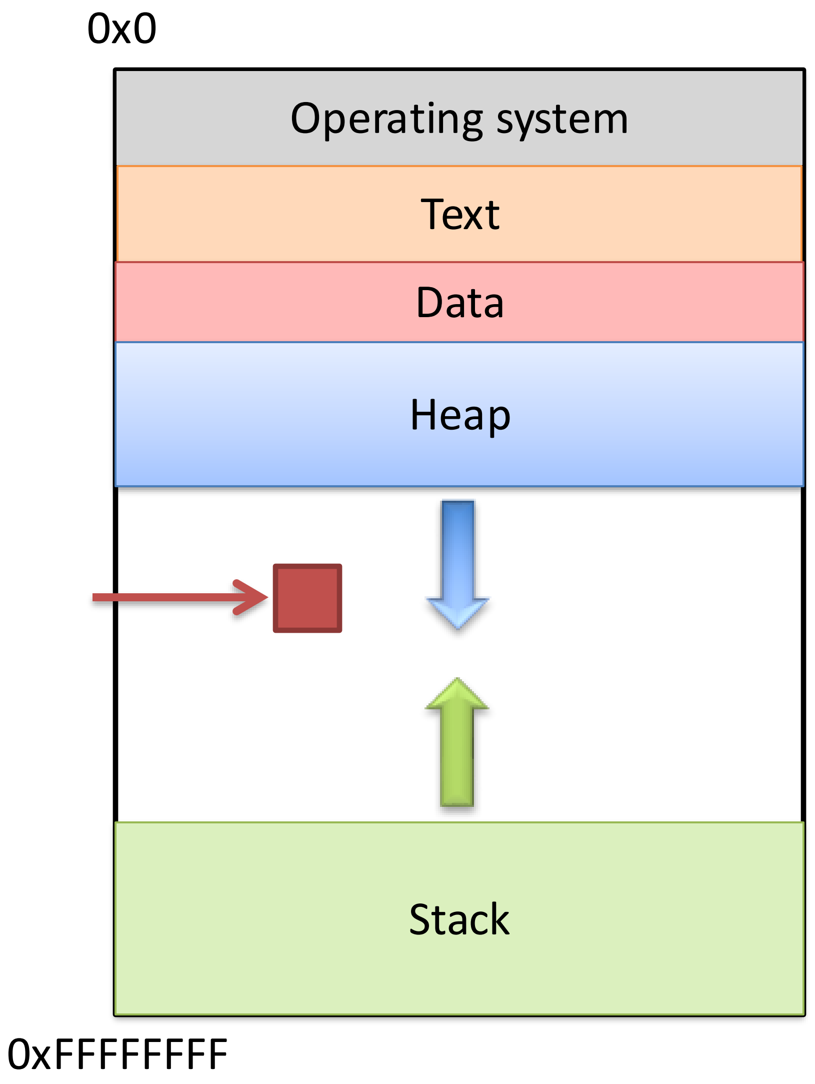
A. The address is allocated to your program.

B. The OS warns your program.

C. The OS kills your program.

D. The access fails, try the next instruction.

E. Something else



What should happen if we try to access an address that's NOT in one of these regions?

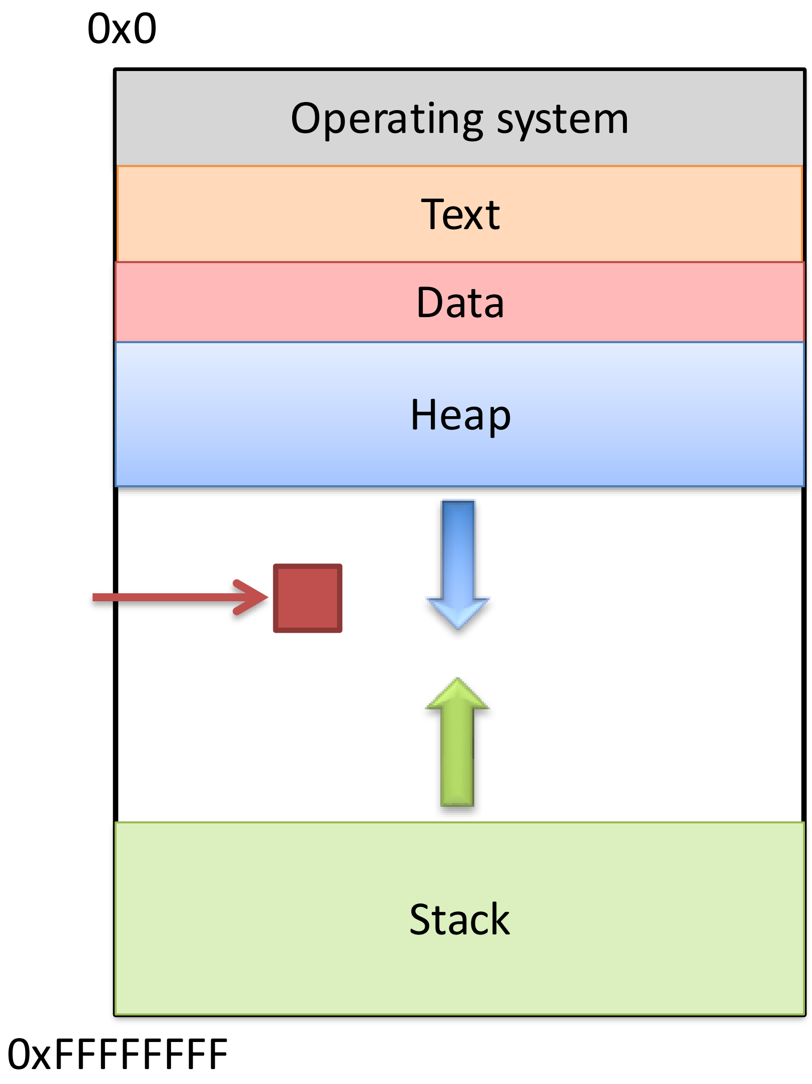
A. The address is allocated to your program.

B. The OS warns your program.

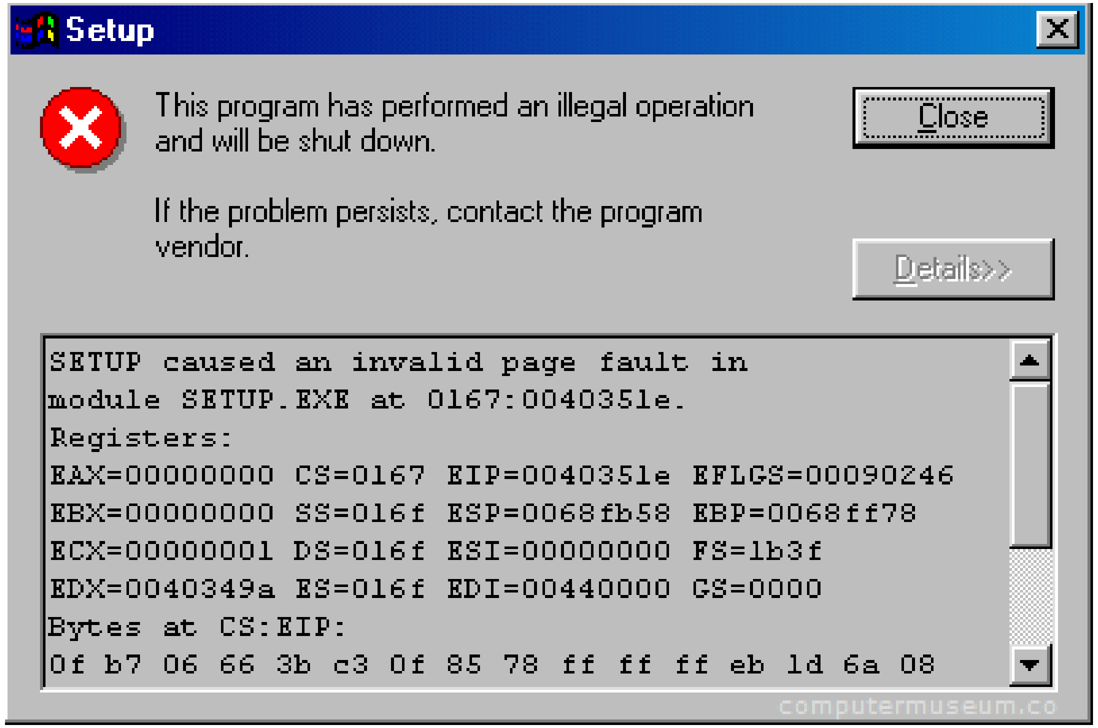
C. The OS kills your program.

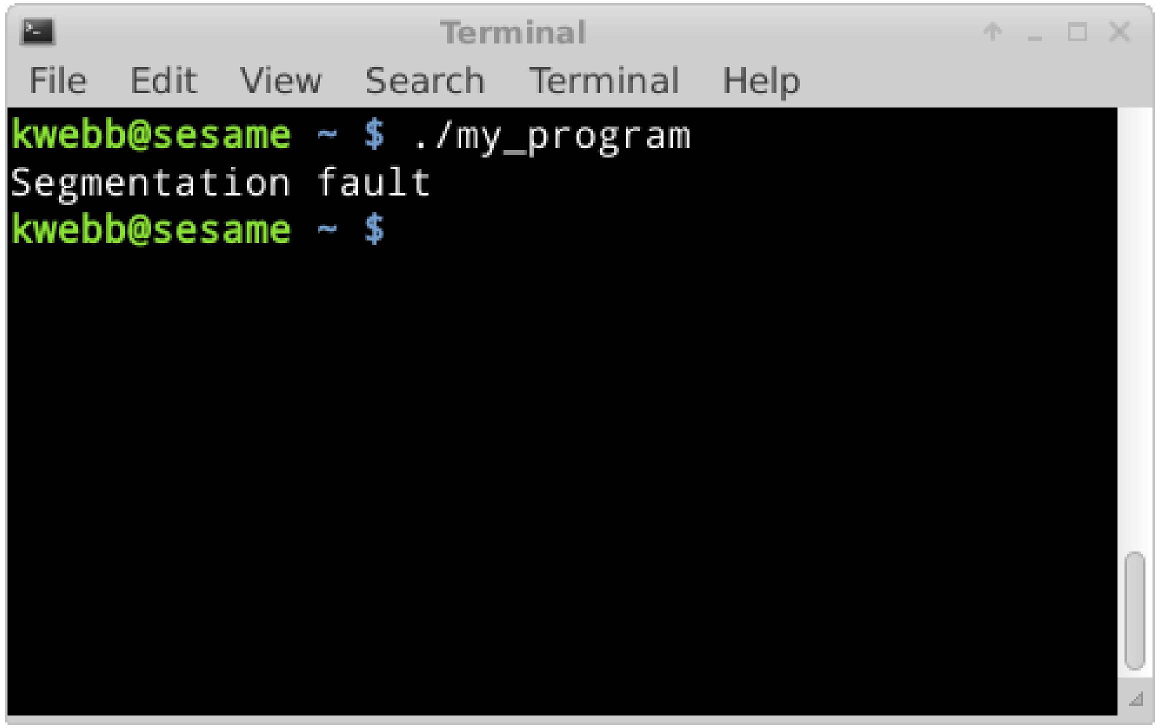
D. The access fails, try the next instruction.

E. Something else



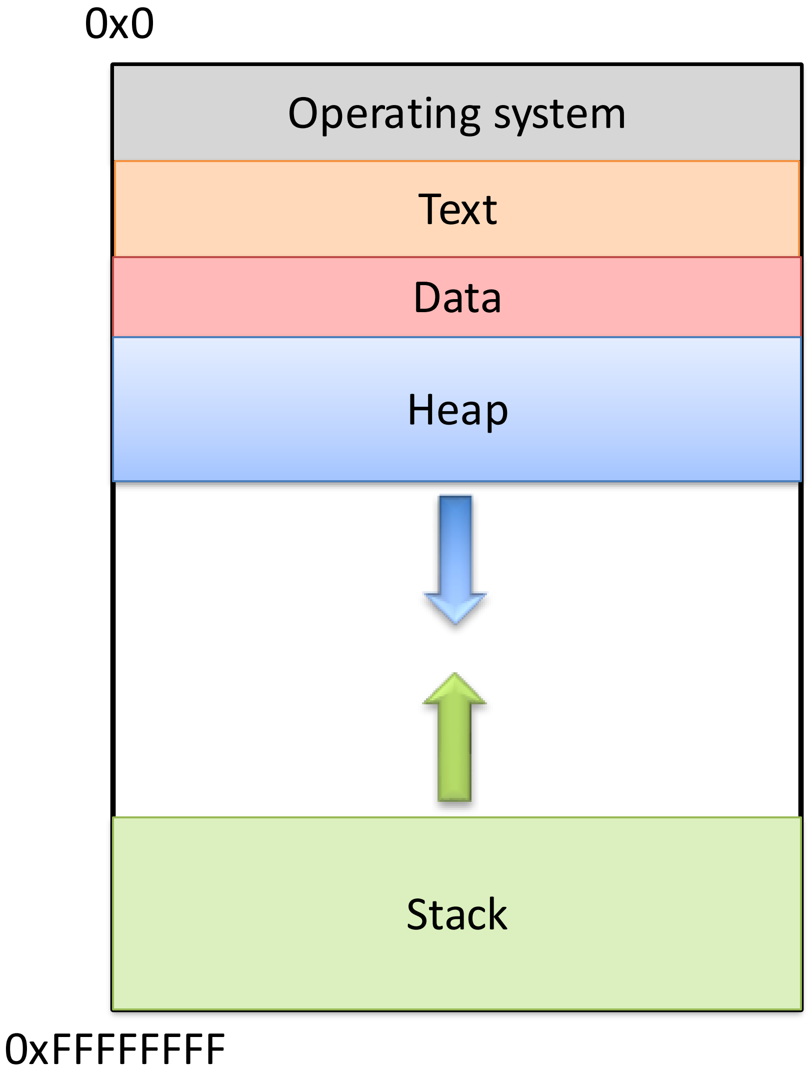
Segmentation Violation





Segmentation Violation

- Each region also known as a memory segment.
- Accessing memory outside a segment is not allowed.
- Can also happen if you try to access a segment in an invalid way.
  - OS not accessible to users
  - Text is usually read-only



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. (`size_t` is an unsigned int of 8 bytes on x86_64)

```
void free(void *ptr)
```

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

Recall: void *

- `void*` is a special type that represents "generic pointer".

- This is useful for cases when:
  1. You want to create a generic "safe value" that you can assign to any pointer variable.
  2. *You want to pass a pointer to / return a pointer from a function, but you don't know its type.*
  3. You know better than the compiler that what you're doing is safe, and you want to eliminate the warning.

- When `malloc()` gives you bytes, it doesn't know or care what you use them for…

Allocation Size

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

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

- How much memory should we ask for?

- Use C's `sizeof()` operator:

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

`sizeof()`

- Despite the ()’s, *it’s an operator, not a function*
  - Other operators:
    - addition / subtraction (+ / -)
    - address of (&)
    - indirection (*) (dereference a pointer)

- Works on any type to tell you how much memory it needs.

- Size value is determined at compile time (static).

Why `sizeof()` is important

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

How many bytes is this?
Who cares…
Let the compiler figure that out.

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

I don't want to see a number hard-coded in here!

If you call malloc(N) and N bytes are not available…

A. malloc returns NULL

B. your program is terminated by the OS

C. your program is paused until memory is available

D. your PC catches on fire

If you call malloc(N) and N bytes are not available…

A. **malloc returns NULL**

B. your program is terminated by the OS

C. your program is paused until memory is available

D. your PC catches on fire

NULL: A special pointer value.

- You can assign NULL to any pointer, regardless of what type it points to (it’s a void *).
  - `int *iptr = NULL;`
  - `float *fptr = NULL;`
- NULL is equivalent to pointing at memory address 0x0. This address is NEVER in a valid segment of your program’s memory.
  - *This guarantees a segfault if you try to dereference it*.

Generally a good ideal to initialize pointers to NULL.

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)

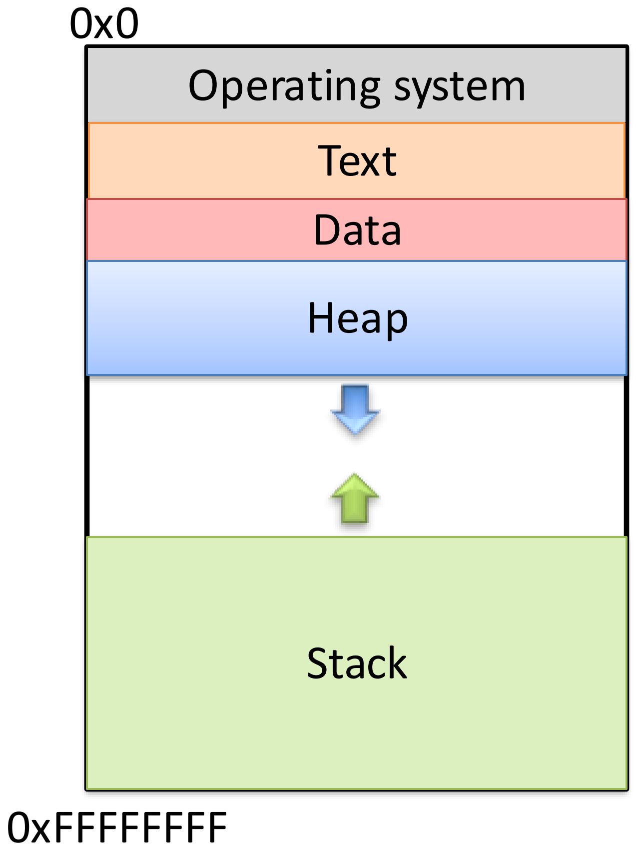
What do you expect to happen to the 100-byte chunk if we do this?

```
// 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)

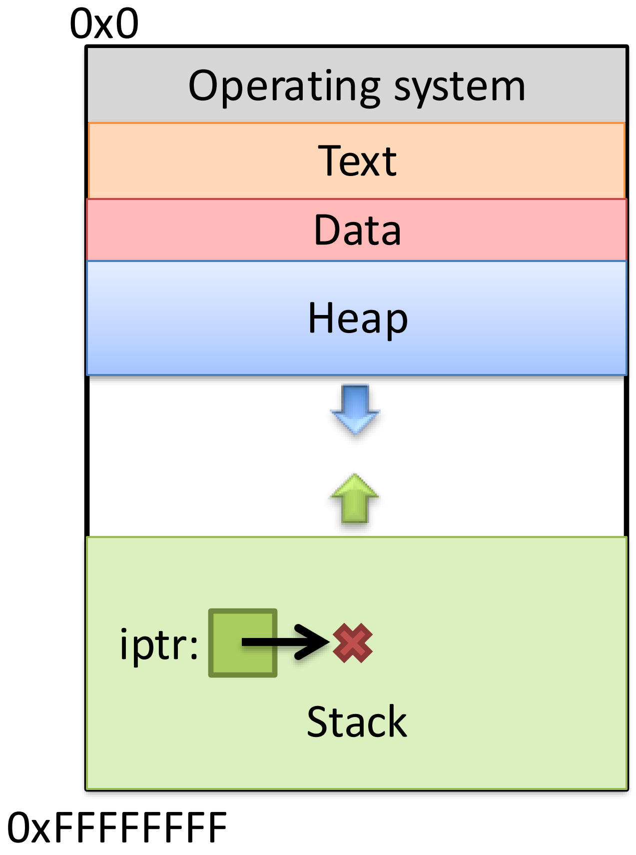
Draw the Stack Diagram

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```



Draw the Stack Diagram

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```



Create an integer pointer, named iptr, on the stack.

Assign it NULL.

Draw the Stack Diagram

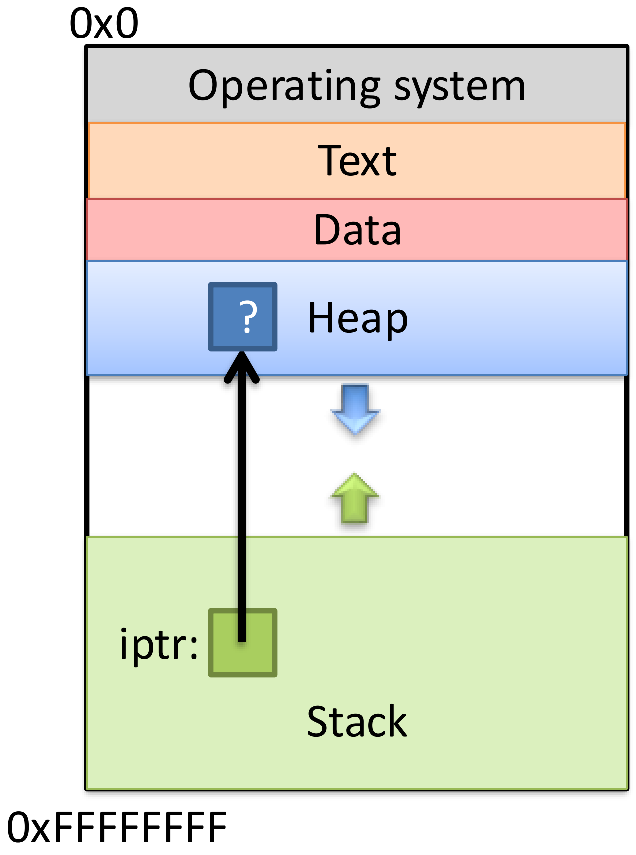
What value is stored in that area right now?

Who knows… Garbage 🤷‍♀️

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```

Allocate space for an integer on the heap (4 bytes), and return a pointer to that space.

Assign that pointer to iptr.



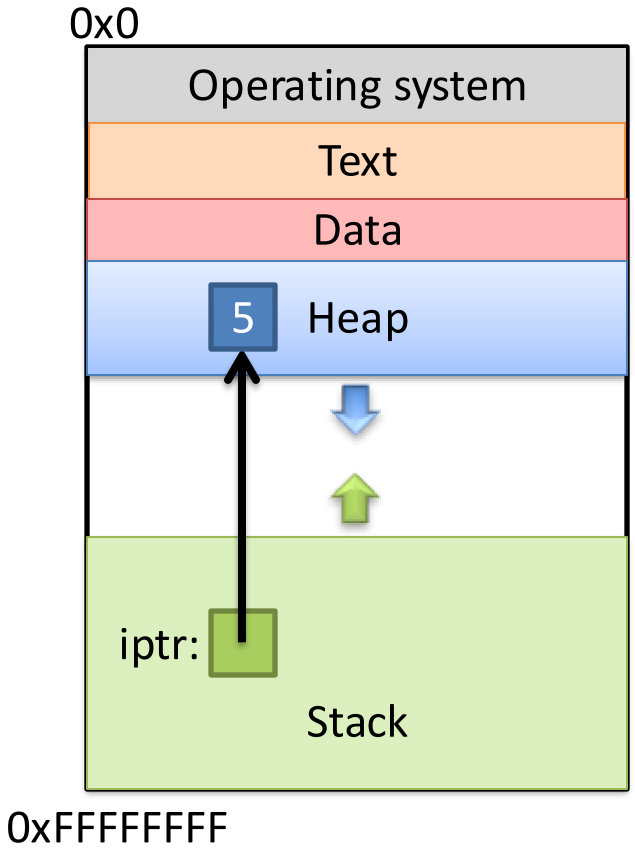
What value is stored in that area right now?

Who knows… Garbage.

Draw the Stack Diagram

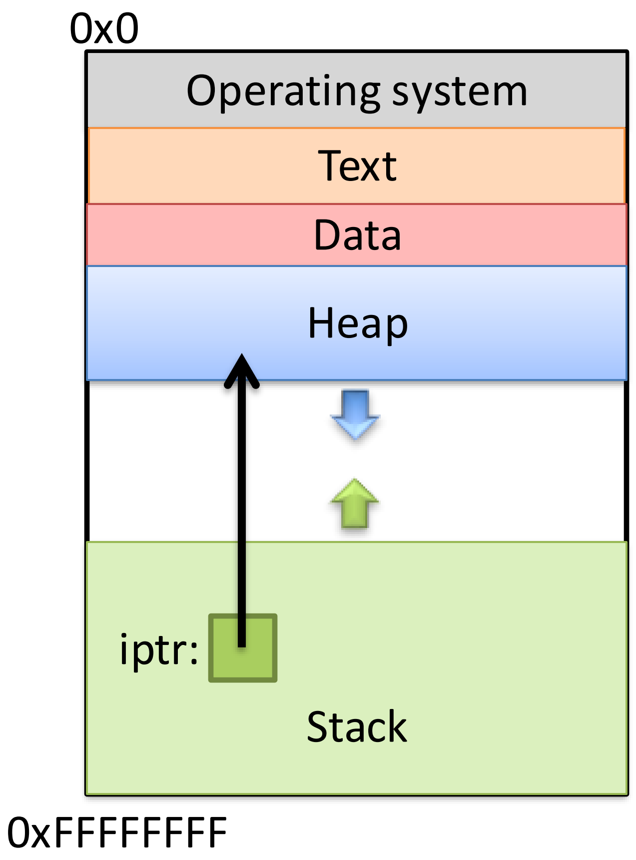
```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```

Use the allocated heap space by dereferencing the pointer.



Don't forget to free()!

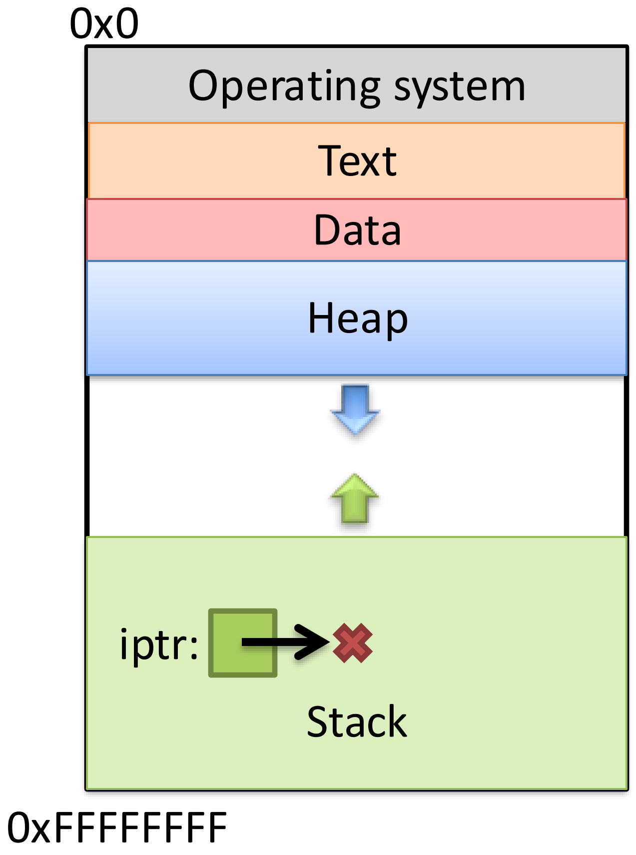
```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;

free(iptr);
```



Free up the heap memory we used.

Don’t forget to free() and set to NULL!

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;

free(iptr);
iptr = NULL;
```



Clean up this pointer, since it’s no longer valid.

Can you return an array?

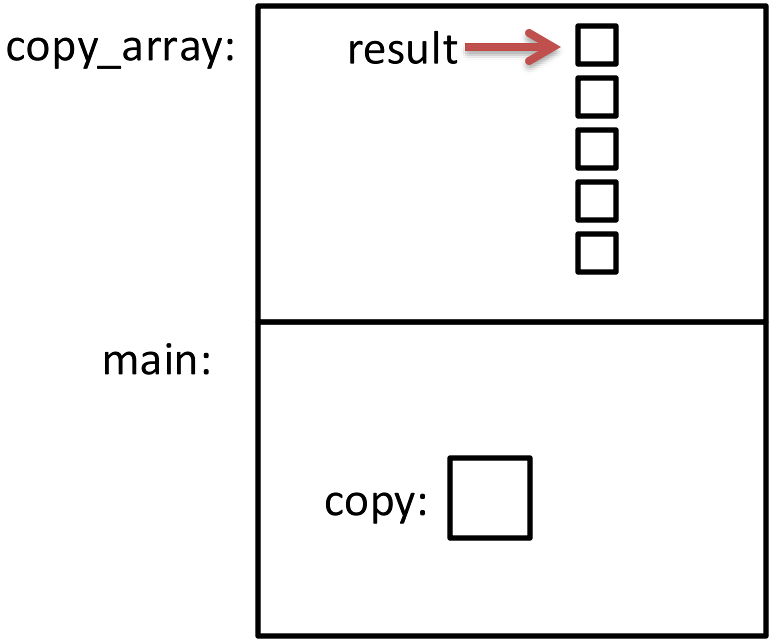
- Suppose you wanted to write a function that copies an array (of 5 integers).
  - Given: array to copy

```
copy_array(int array[]) {
  int result[5];
  result[0] = array[0];
  …
  result[4] = array[4];
  return result;
}
```

As written above, this would be a terrible way of implementing this.
(Don’t worry, compiler won’t let you do this anyway.)

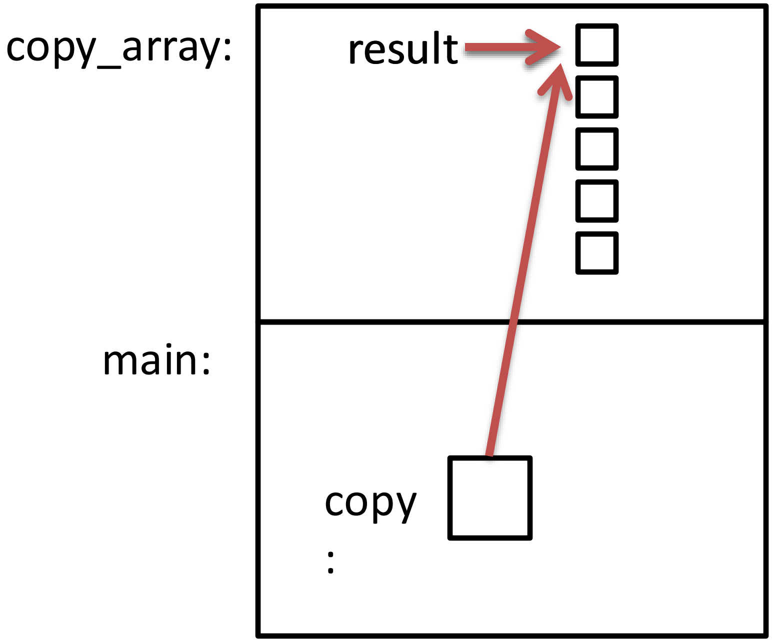
Consider the memory…

```
copy_array(int array[]) {
  int result[5];
  result[0] = array[0];
  …
  result[4] = array[4];
  return result;
}

(In main):
copy = copy_array(…)
```



Consider the memory…

```
copy_array(int array[]) {
  int result[5];
  result[0] = array[0];
  …
  result[4] = array[4];
  return result;
}

(In main):
copy = copy_array(…)
```



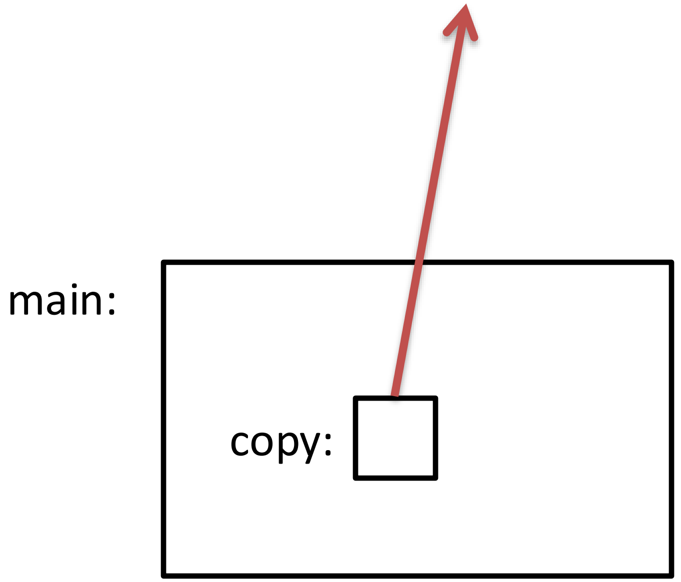
Consider the memory…

```
copy_array(int array[]) {
  int result[5];
  result[0] = array[0];
  …
  result[4] = array[4];
  return result;
}
```

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

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

Left with a pointer to nowhere.

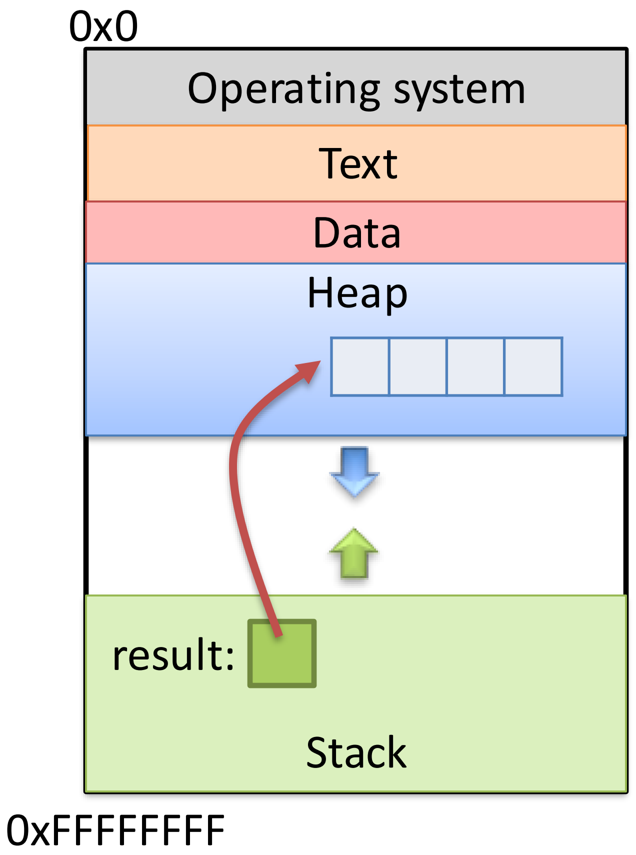


Using the Heap

```
int *copy_array(int num, int array[]) {
  int *result = malloc(num * sizeof(int));

  result[0] = array[0];
  …

  return result;
}
```

malloc memory is on the heap.

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



“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
  - This can be a time consuming feature
  - C 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. Global warming wasn't a problem when C was invented

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. Global warming wasn't a problem when C was invented

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

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…

Administration Priorities The Reco

FEBRUARY 26, 2024

PRESS RELEASE: Future Software Should Be Memory Safe

ONCD › BRIEFING ROOM › PRESS RELEASE

Leaders in Industry Support White House Call to Address Root Cause of Many of the Worst Cyber Attacks

Summary

- Three rules for using pointer variables
  - Declare a pointer
  - Initialize it (make it point to a memory address or NULL)
  - If you allocate dynamic memory → remember to free!
  - reset your pointer to NULL at the end of your function
- Pass by pointer
- Layout of program memory
  - Stack vs. Heap
  - Segmentation violation
- Dynamic memory allocation: malloc(), free(), and sizeof()
- Memory leaks and bookkeeping

Pointers as Arrays

"Why did you allocate 8 bytes for an int pointer?

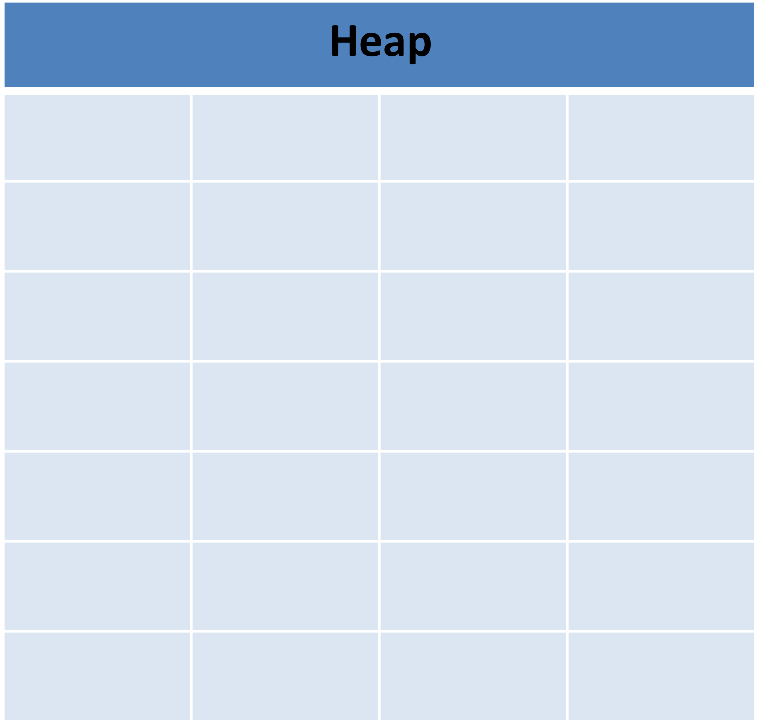
- `int *iptr = malloc(8);`

- 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!

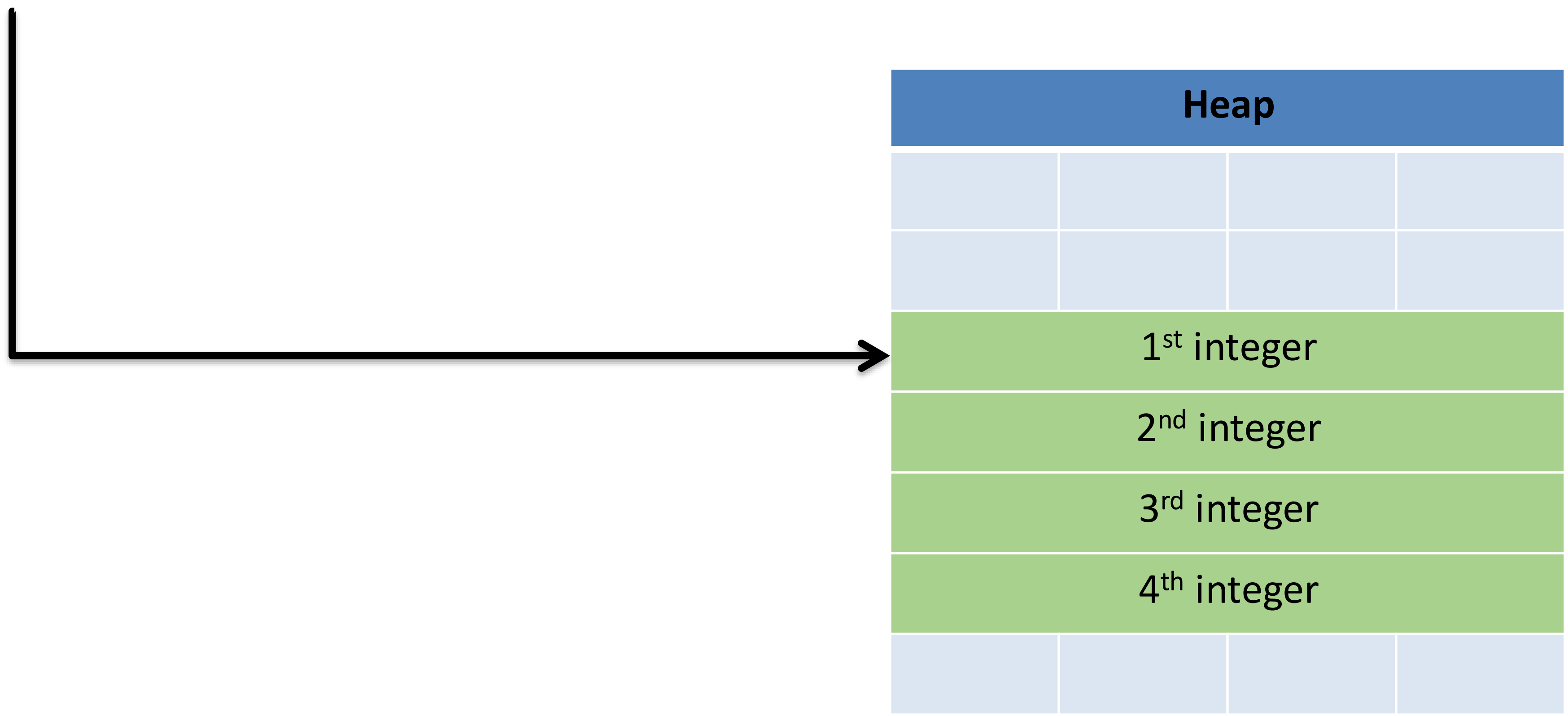
Pointers as Arrays

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



Pointers as Arrays

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



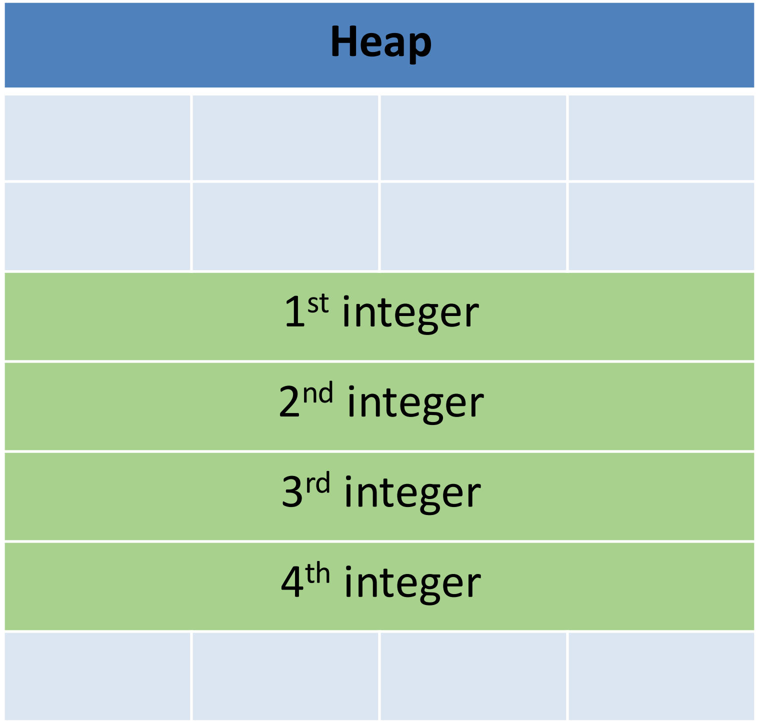
Pointers as Arrays

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

The C compiler knows how big an integer is.

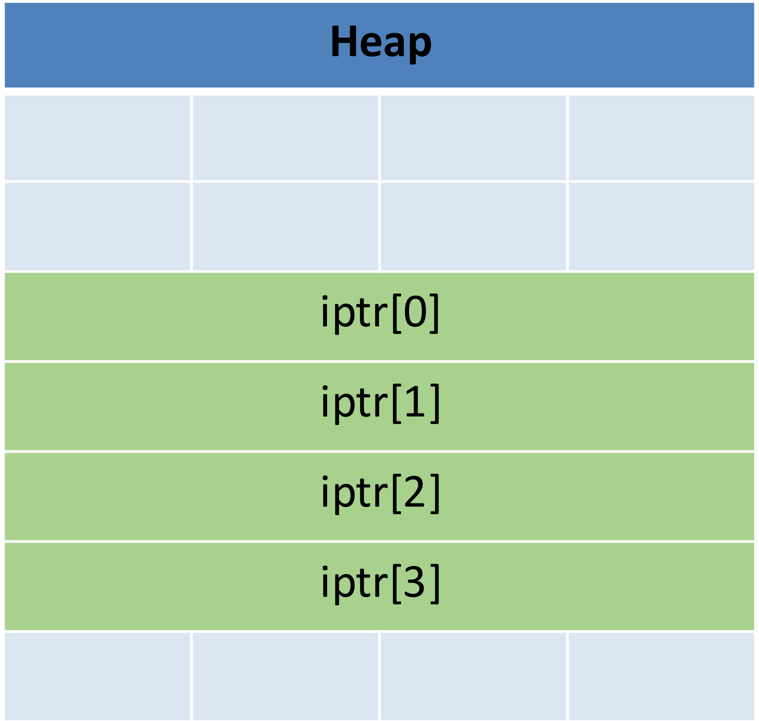
As an alternative way of dereferencing, you can use []'s like an array.

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



Pointers as Arrays

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

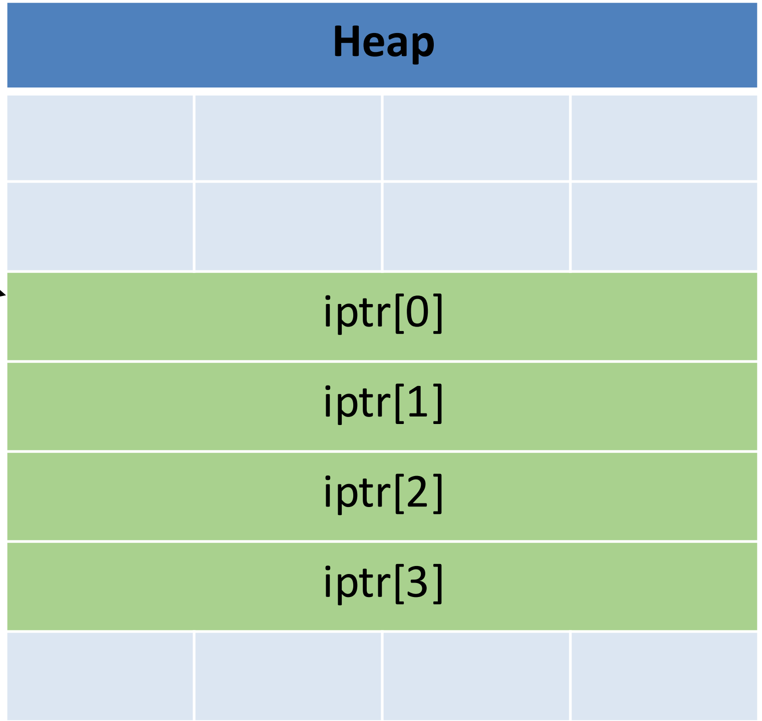


Pointers as Arrays

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

1. Start from the base of iptr.

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

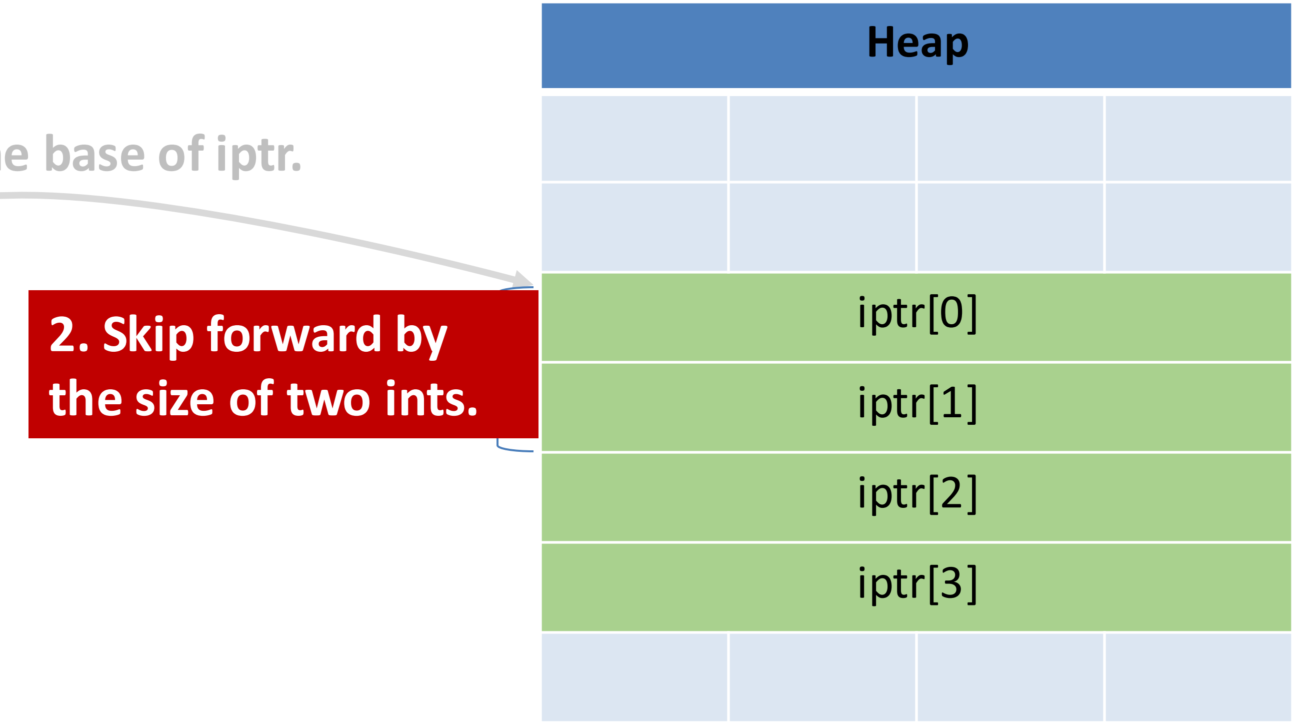


Pointers as Arrays

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

1. Start from the base of iptr.

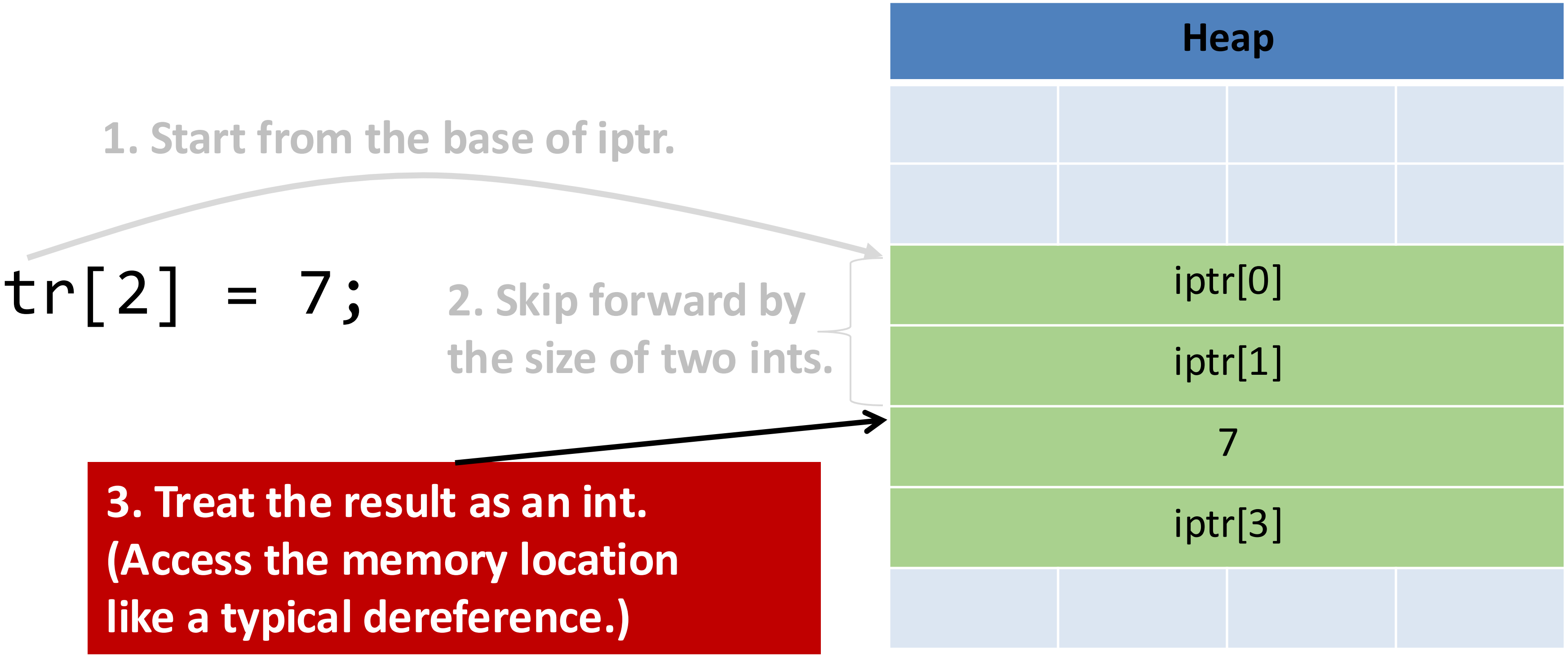
```c
iptr[2] = 7;
```



Pointers as Arrays

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

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



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] = …;
```

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.)