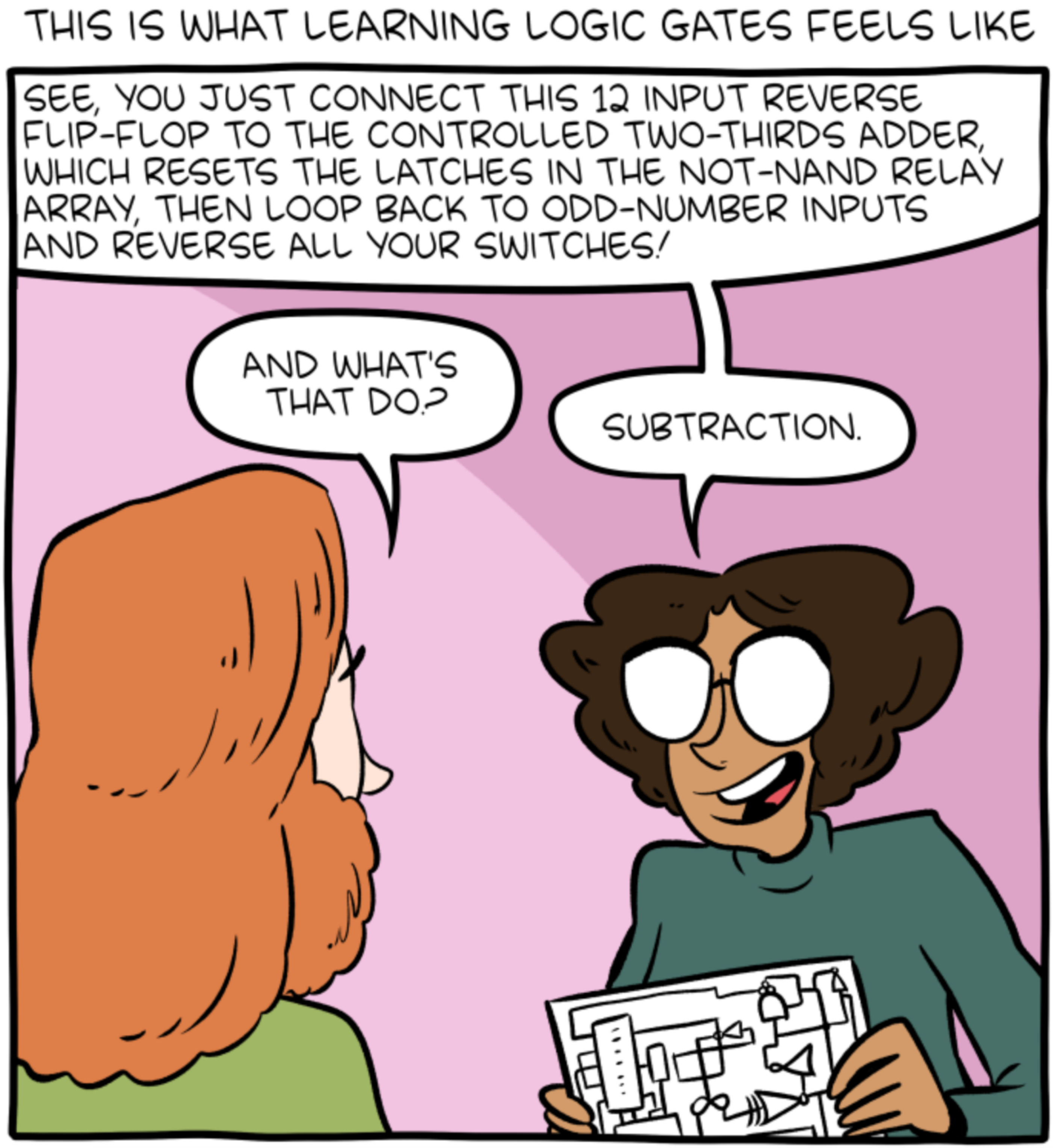
CS 31: Introduction to Computer Systems

09-10: Pointers, Memory

February 19, 21

SWARTHMORE COLLEGE



"If you can do logic gates in your head, please confirm you are not a replicant"

http://smbc-comics.com/comic/logic-gates

Reading Quiz

Last Class

- ISA defines what programmer can do on hardware
  - Which instructions are available
  - How to access state (registers, memory, etc.)
  - This is the architecture's *assembly language*

- In this course, we'll be using IA-32
  - Instructions for:
    - moving data (movl)
    - arithmetic (addl, subl, imull, orl, sall, etc.)
    - control (jmp, je, jne, etc.)
  - Condition codes for making control decisions
    - If the result is zero (ZF)
    - If the result's first bit is set (negative if signed) (SF)
    - If the result overflowed (assuming unsigned) (CF)
    - If the result overflowed (assuming signed) (OF)

Today

- 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

Pointers

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

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

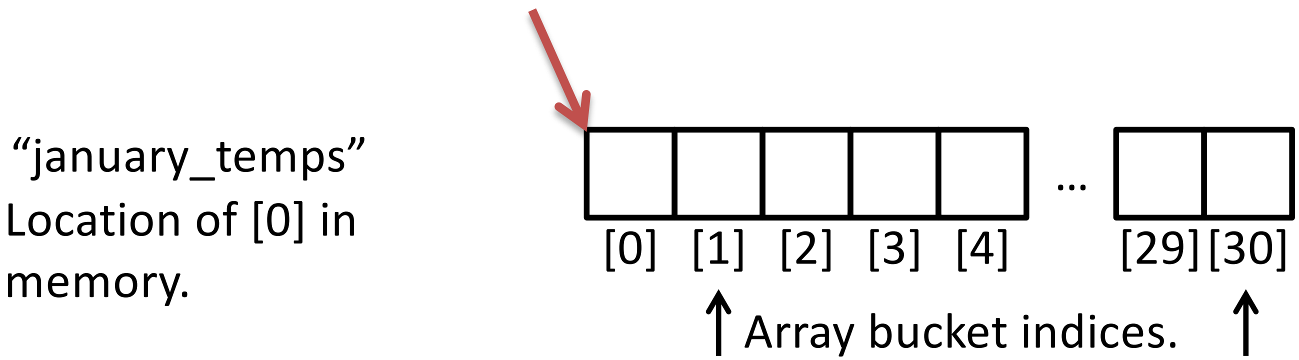
- Where have we seen this before?

Pointers

- Pointer: A variable that stores a reference to a memory location.
- Pointer: sequence of bits that should be interpreted as an index into memory.
- We've seen this examples of this already!

Recall: Arrays

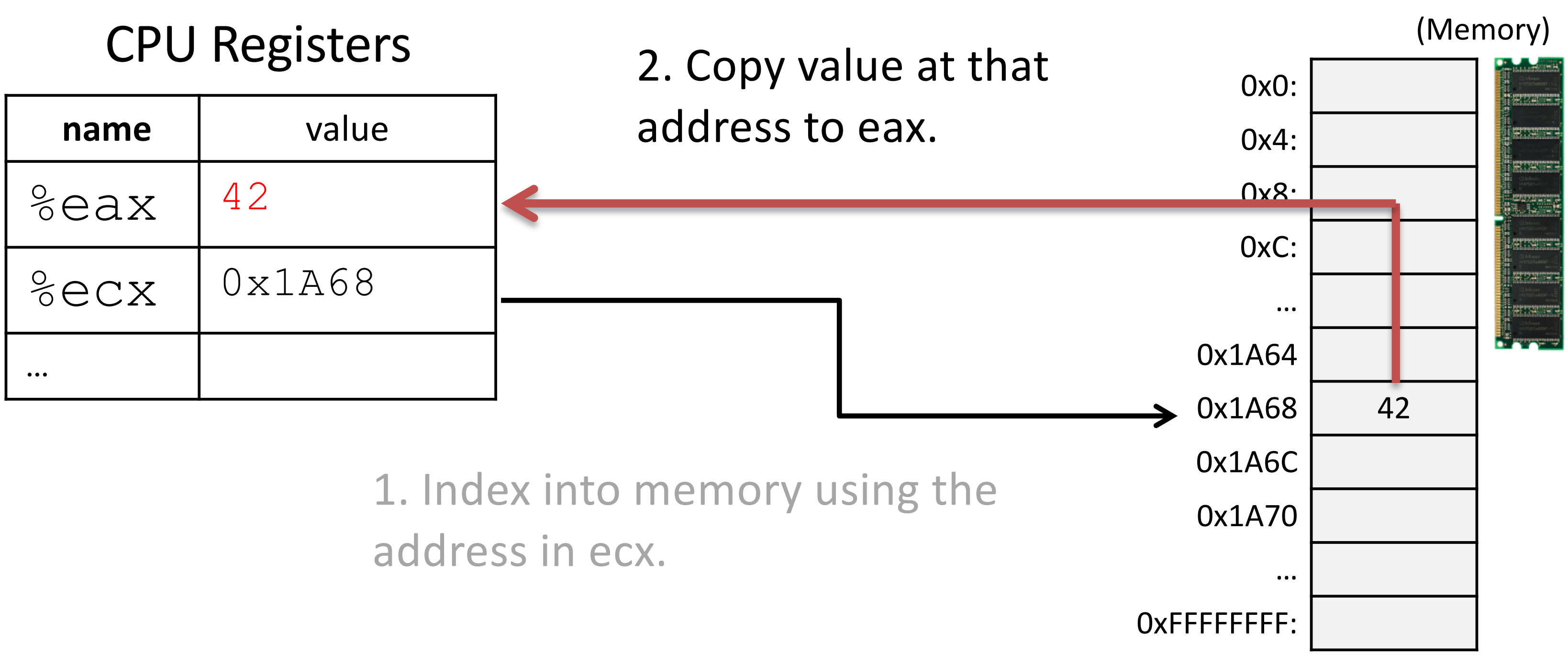
```
int january_temps[31];  // Daily high temps
```



- Array variable name means, to the compiler, the beginning of the memory chunk. (address)

Recall: Addressing Modes

- `movl (%ecx), %eax`
  - Use the address in register ecx to access memory, store result in register eax



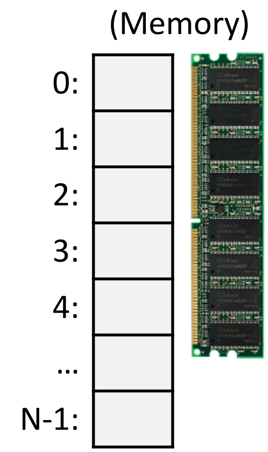
Recall: Program Counter

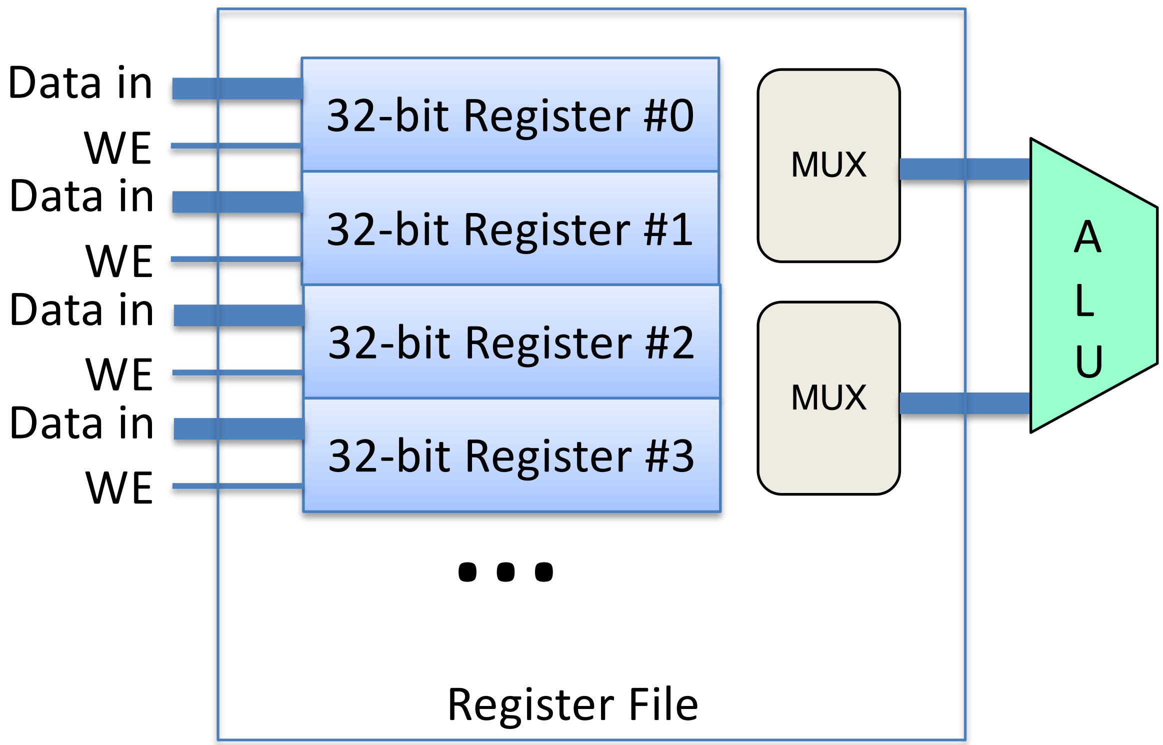
Program Counter (PC): **Memory address of next instr**

Instruction Register (IR): **Instruction contents (bits)**

IA32 refers to the PC as %eip.

Instruction Pointer





Pointers in C

- Like any other variable, must be declared:
  - Using the format: type *name;
- Example:
  - int *myptr;
  - This is a promise to the compiler:
    - This variable holds a memory address. If you follow what it points to in memory (dereference it), you'll find an integer.
- A note on syntax:
  - int* myptr; int * myptr; int *myptr;
  - These all do the same thing. (note the * position)

Dereferencing a Pointer

- To follow the pointer, we *dereference* it.
- Dereferencing re-uses the * symbol.
- If `iptr` is declared as an integer pointer, `*iptr` will follow the address it stores to find an integer in memory.

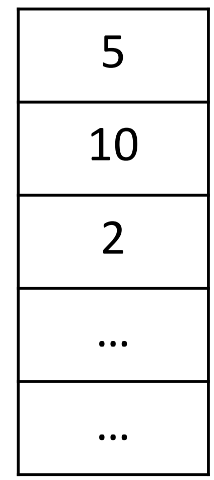
Putting a * in front of a variable…

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

- When you *use* the variable (dereference):
  – Like putting () around a register name
  – Follows the pointer out to memory
  – Acts like the specified type (e.g., int, float, etc.)

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

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

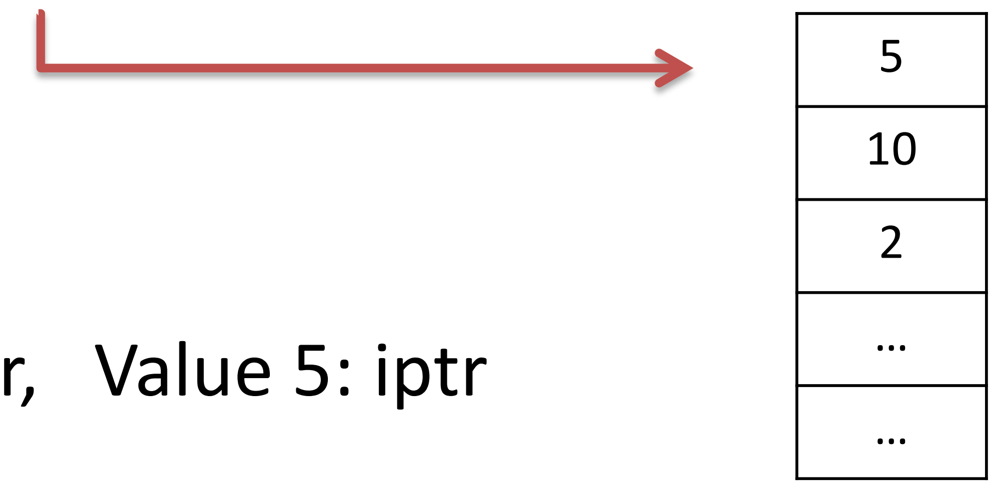


A. Memory address: *iptr, Value 5: iptr

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

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

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



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

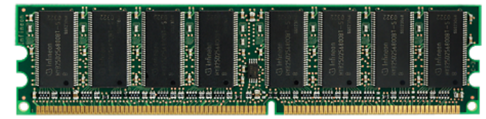
B. Memory address: iptr, Value 5: *iptr

So we declared a pointer…

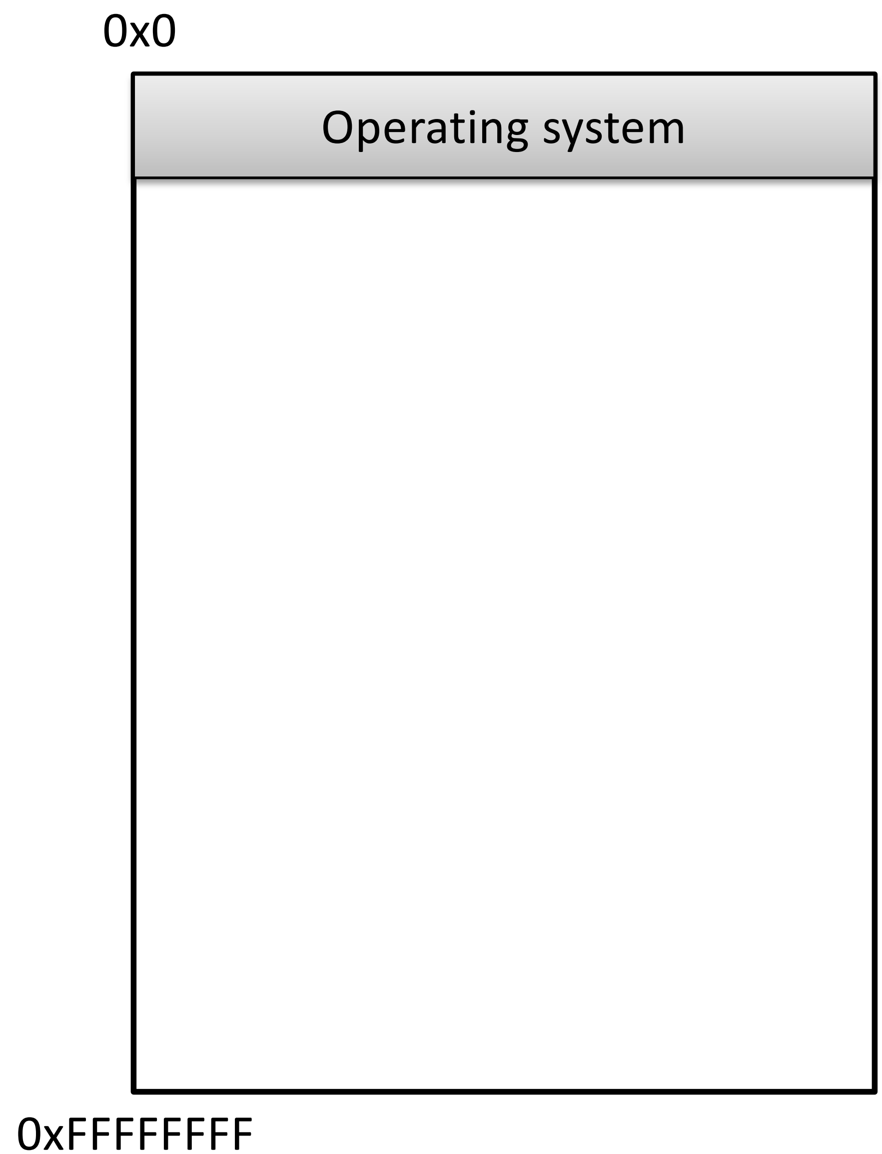
- How do we make it point to something?
  1. Assign it the address of an existing variable
  2. Copy some other pointer
  3. Allocate some memory and point to it

- First, let’s look at how memory is organized.
  - From the perspective of one executing program.

Memory



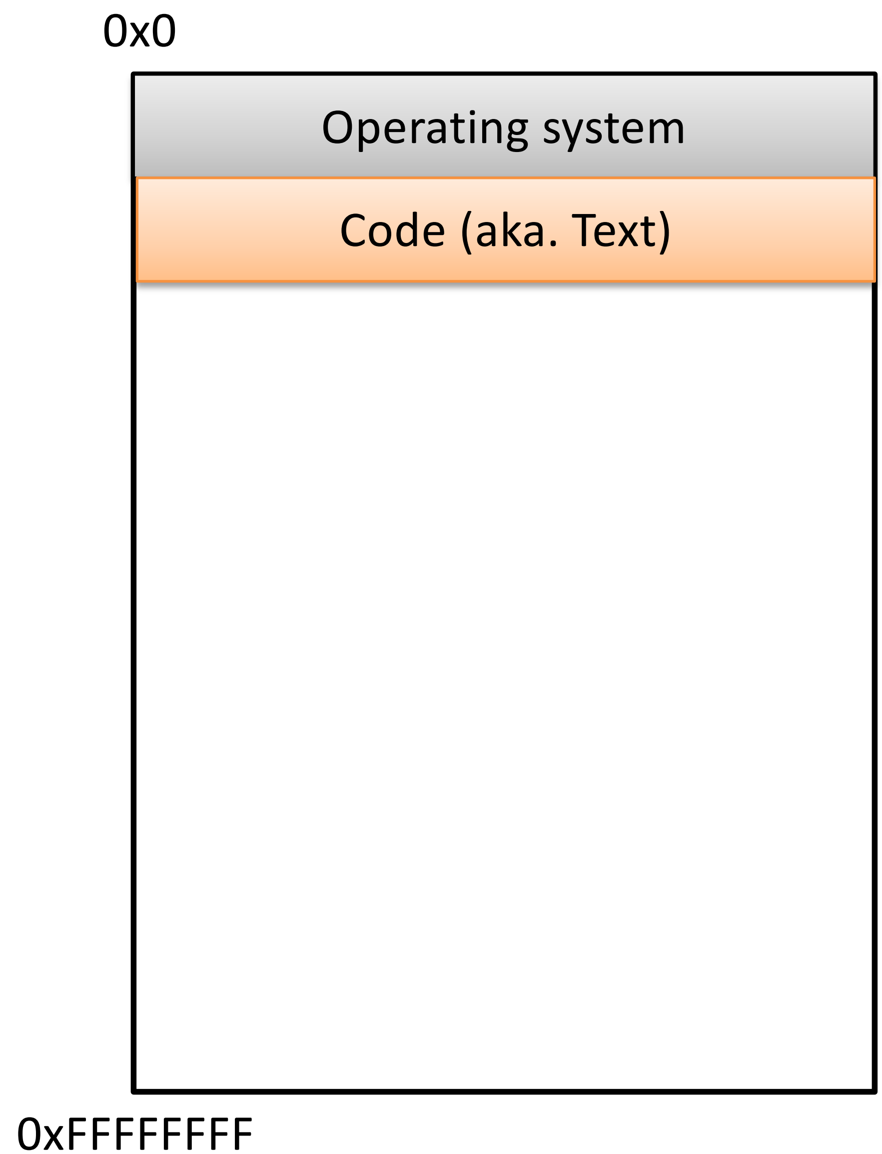
- Behaves like a big array of bytes, each with an address (bucket #).
- By convention, we divide it into regions.
- The region at the lowest addresses is usually reserved for the OS.



Memory - Text

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

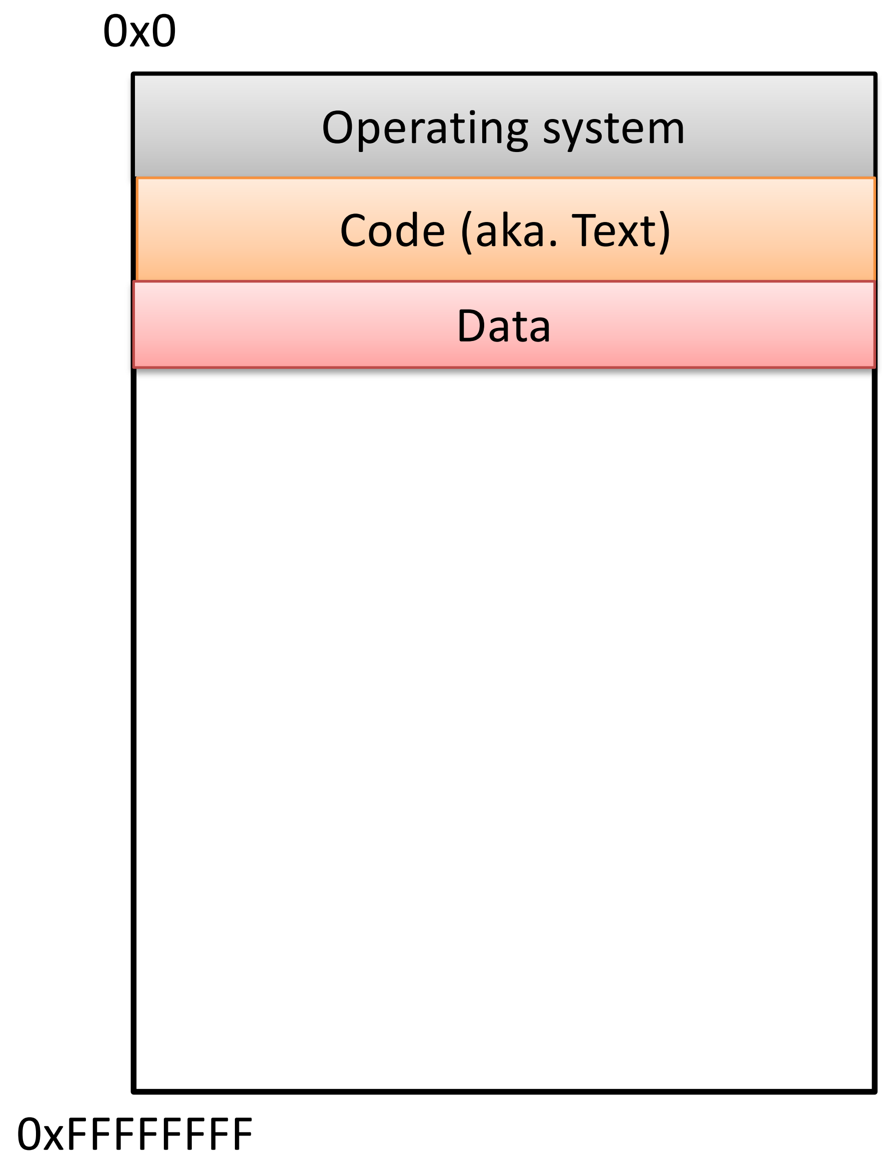
- Instructions generated by the compiler.



Memory – (Static) Data

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

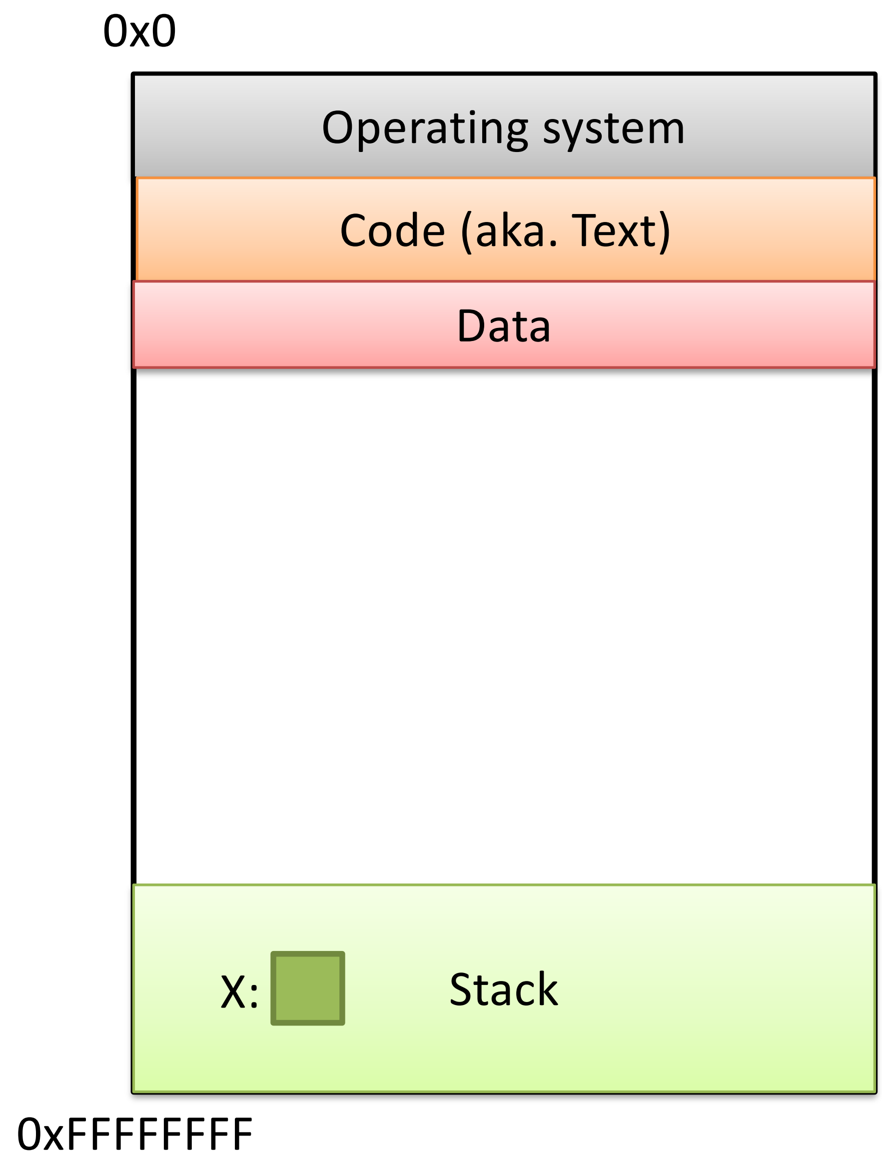
- This stores static variables that are known at compile time.
  – Global variables



Memory - Stack

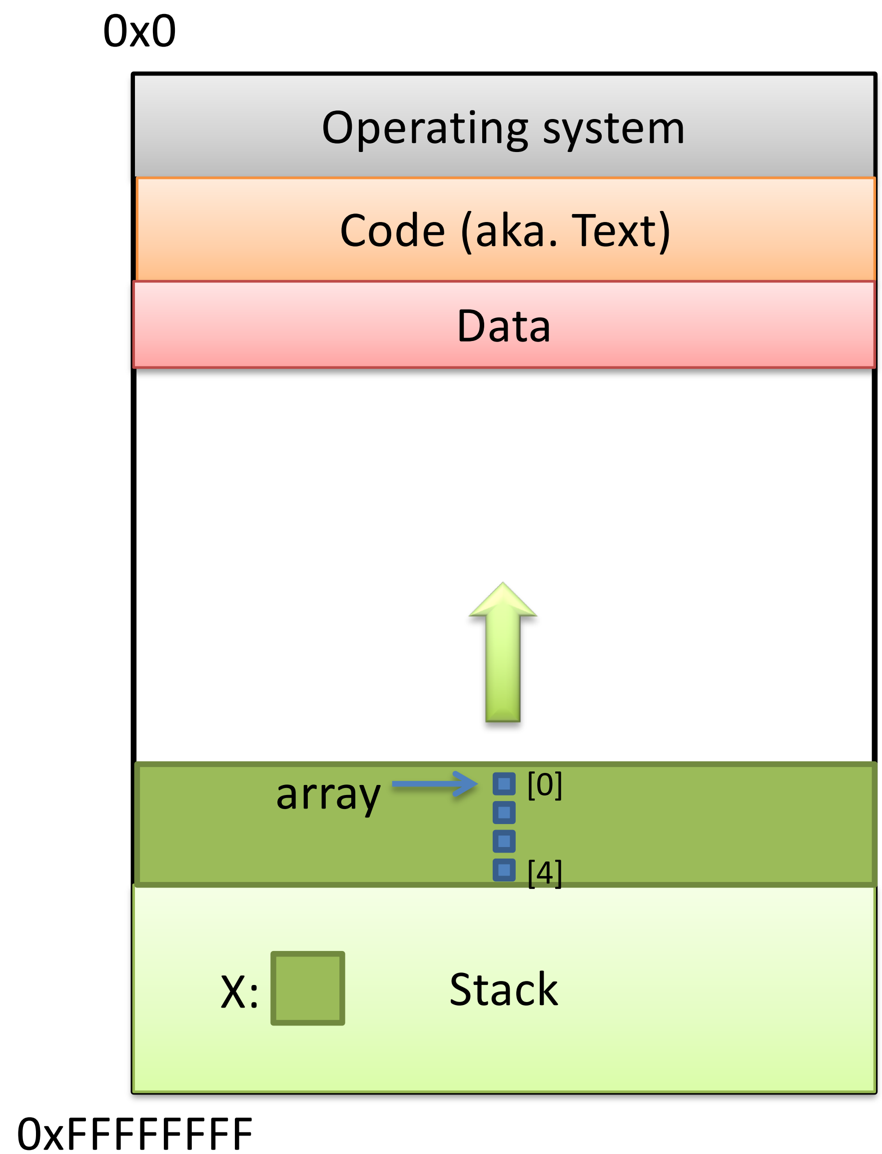
- At high addresses, we keep the stack.

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



Memory - Stack

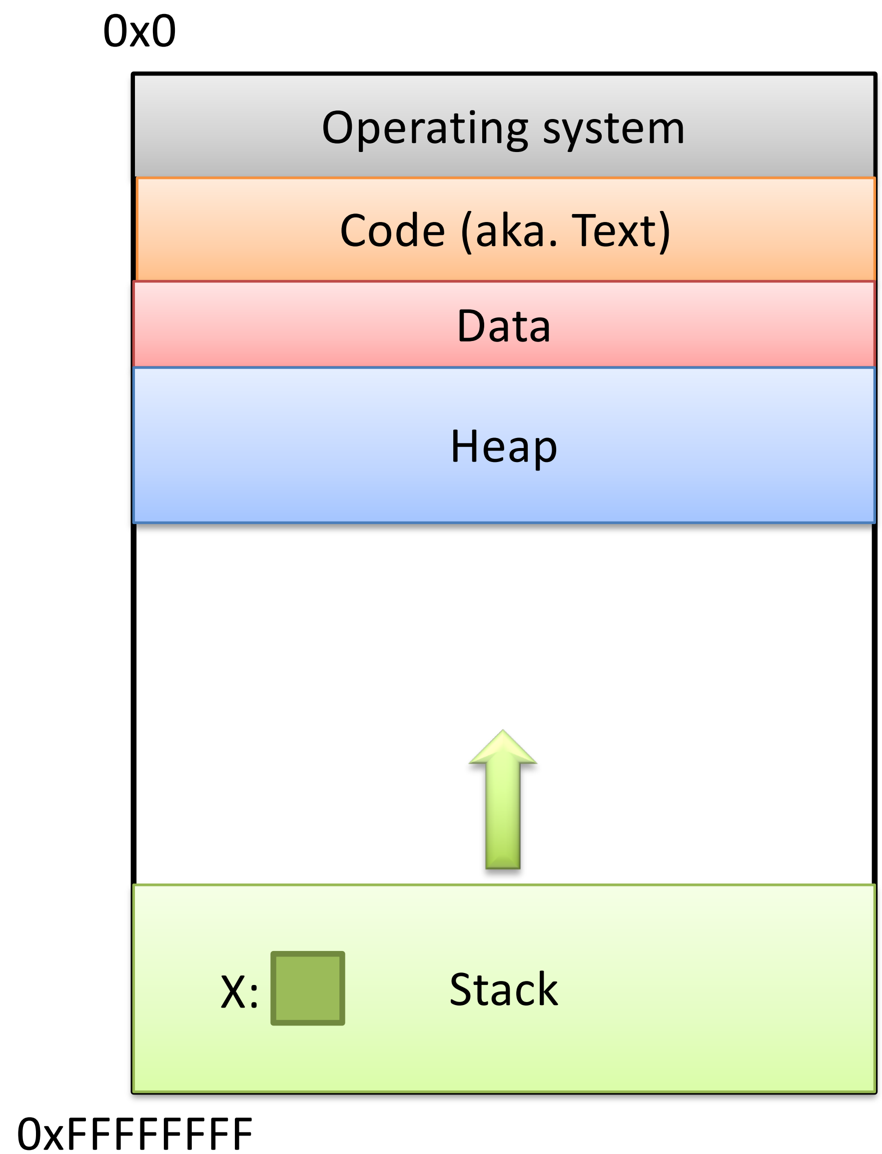
- The stack grows upwards towards lower addresses (negative direction).
- Example: Allocating array
  - int array[4];
- (Note: this differs from Python.)



Memory - Heap

- The heap stores dynamically allocated variables.

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



If we can declare variables on the stack, why do we need to dynamically allocate things on the heap?

A. There is more space available on the heap.

B. Heap memory is better. (Why?)

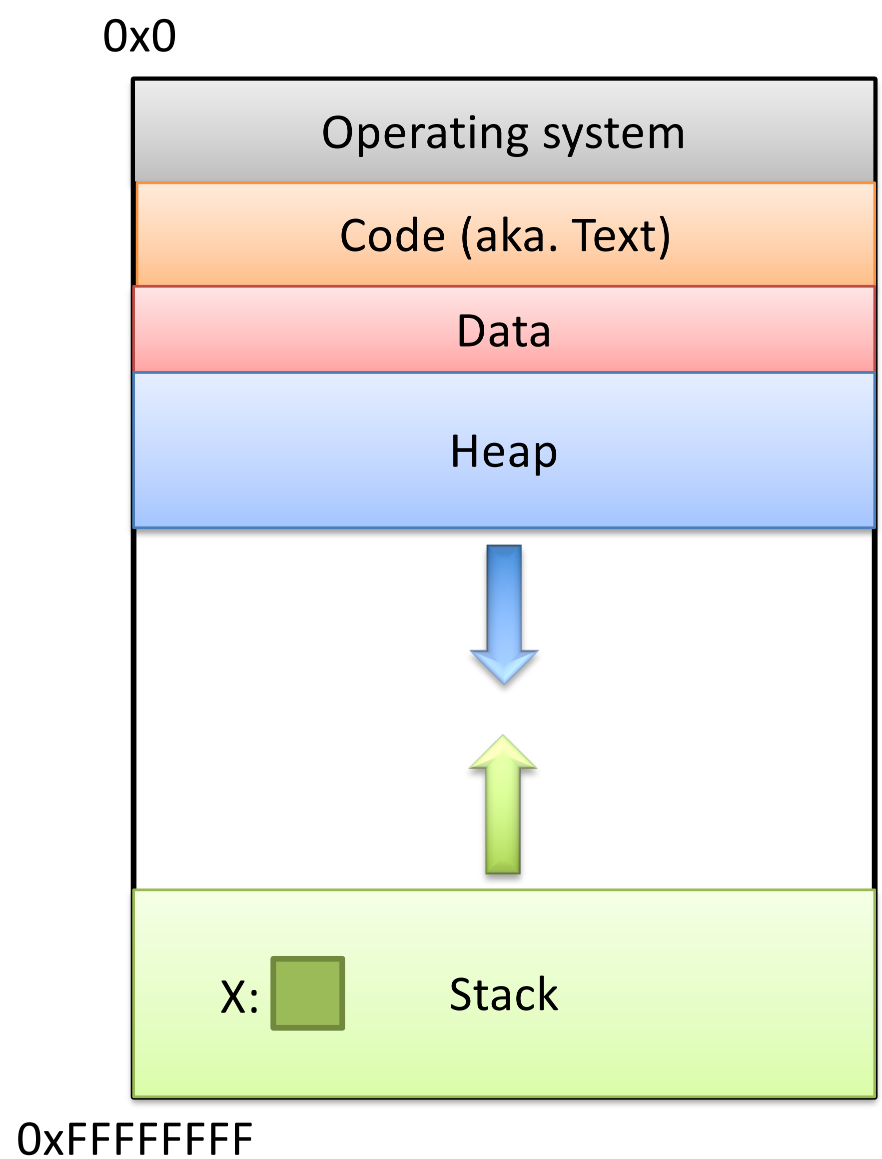
C. We may not know a variable’s size in advance.

D. The stack grows and shrinks automatically.

E. Some other reason.

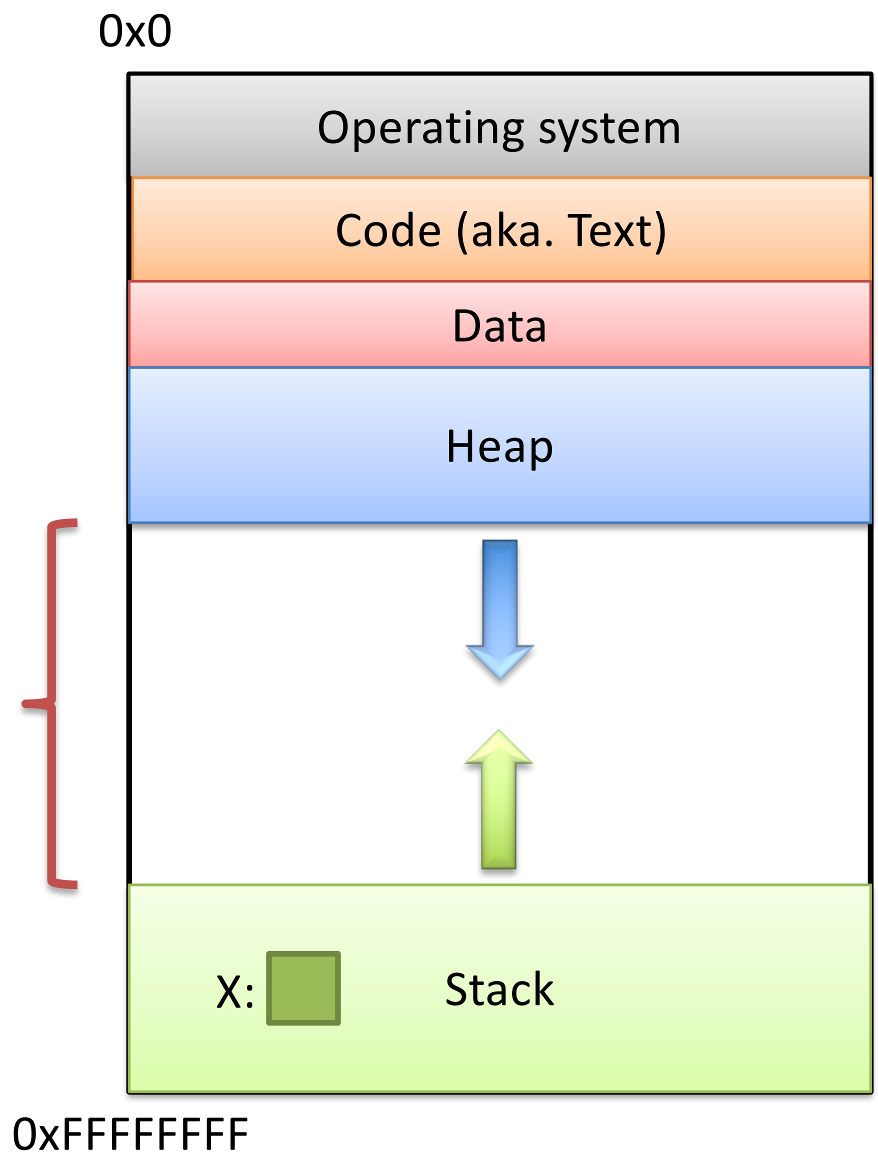
Memory - Heap

- 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?"
- 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.



Which region would we expect the PC register (program counter) to point to?

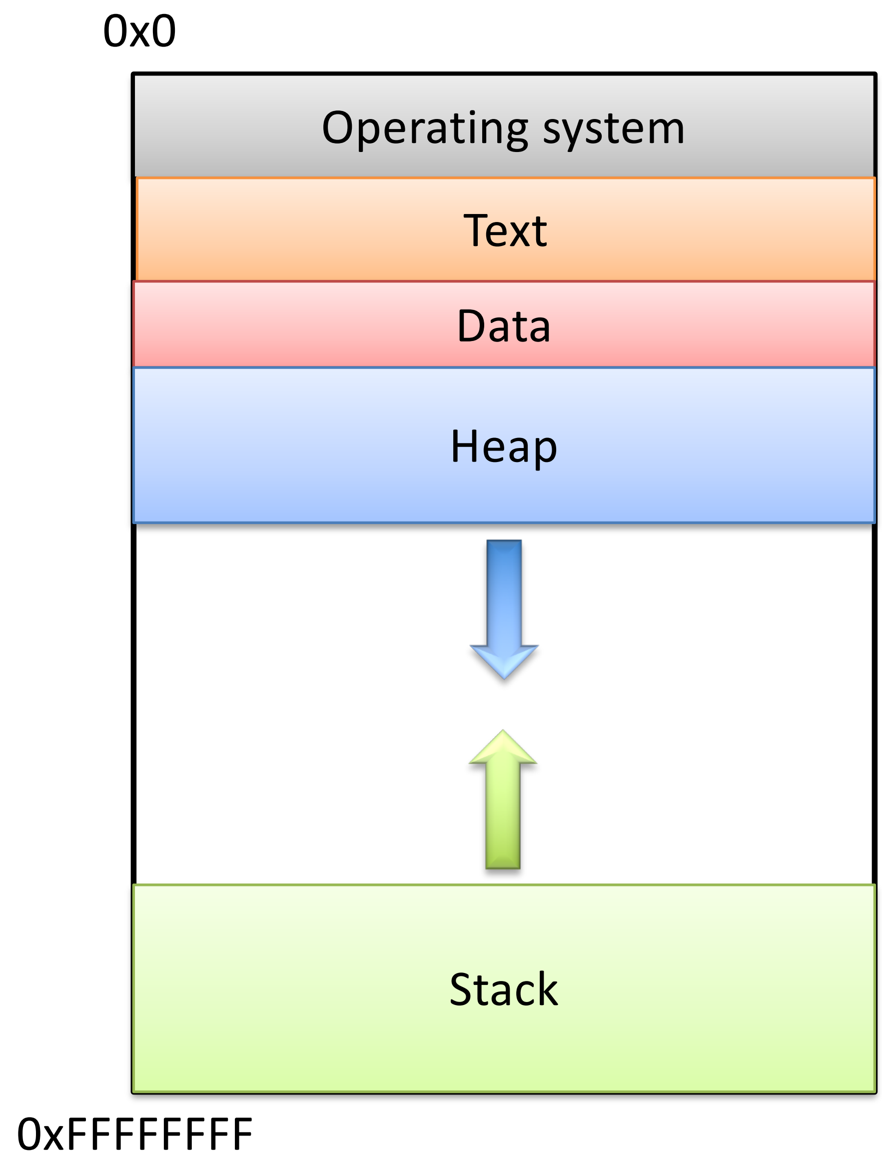
A. OS

B. Text

C. Data

D. Heap

E. Stack



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

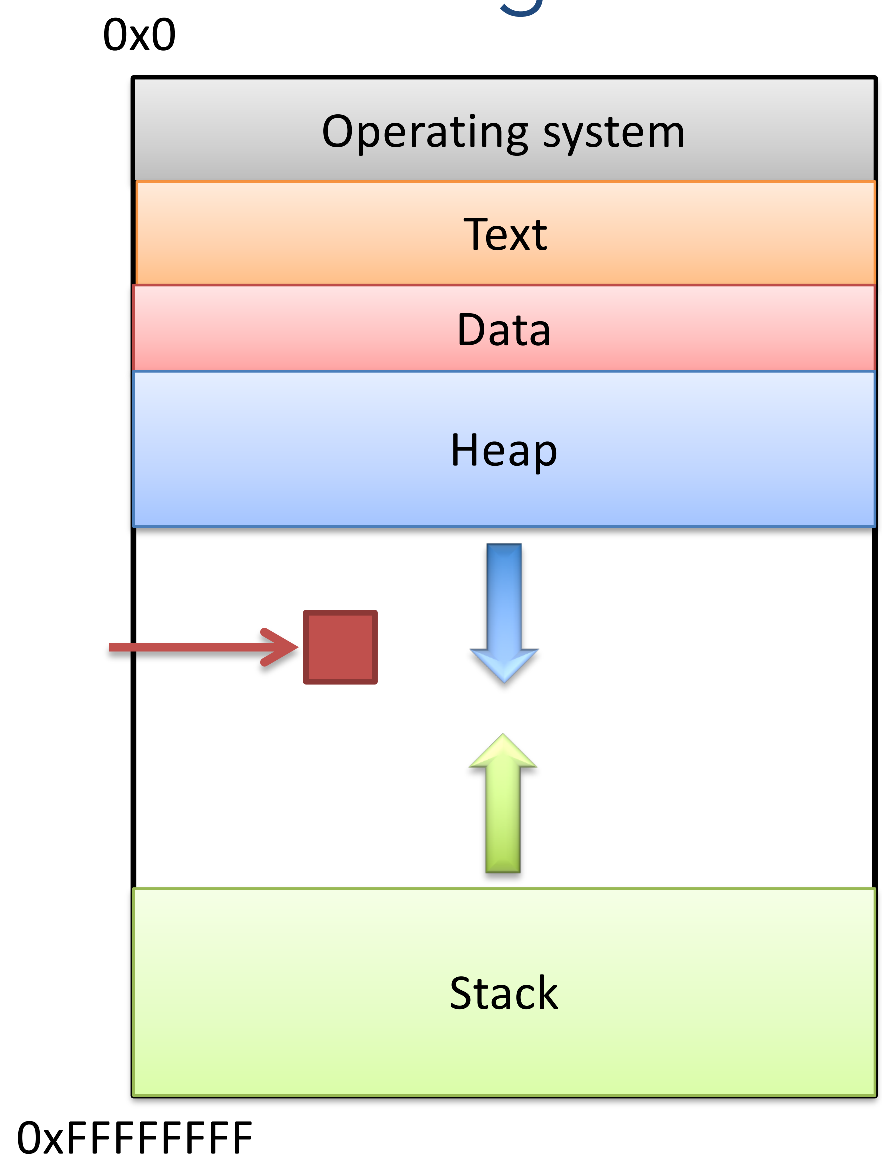
A. The address is allocated to your program.

B. The OS warns your program.

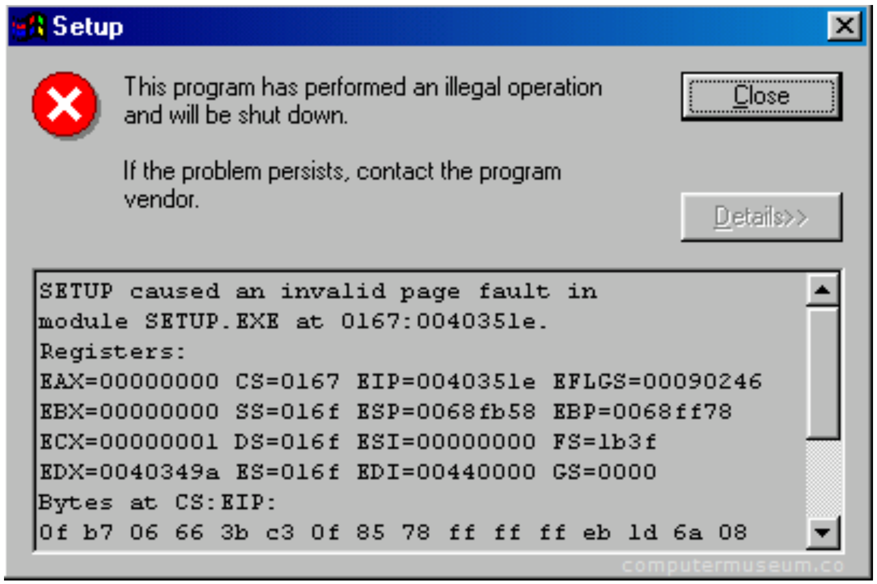
C. The OS kills your program.

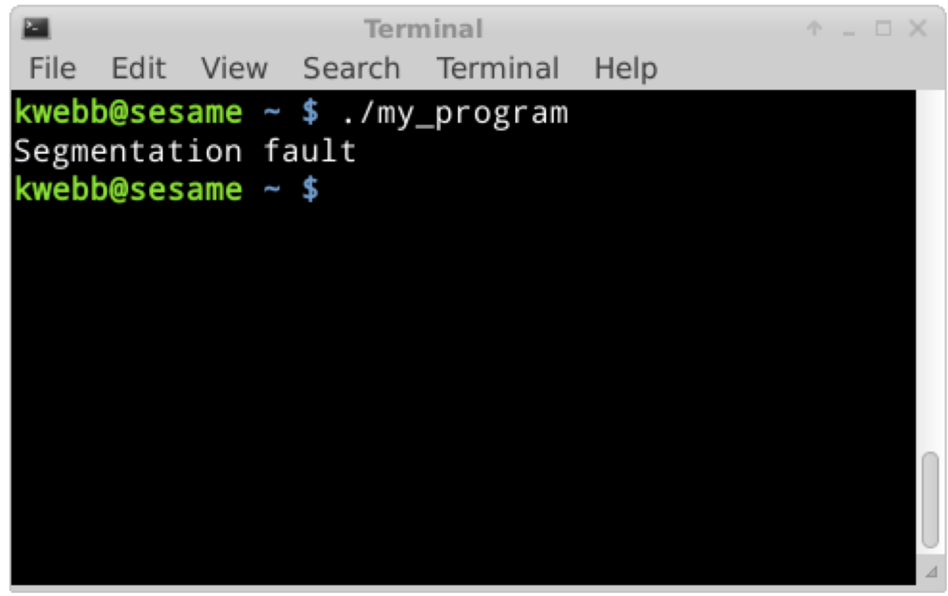
D. The access fails, try the next instruction.

E. Something else



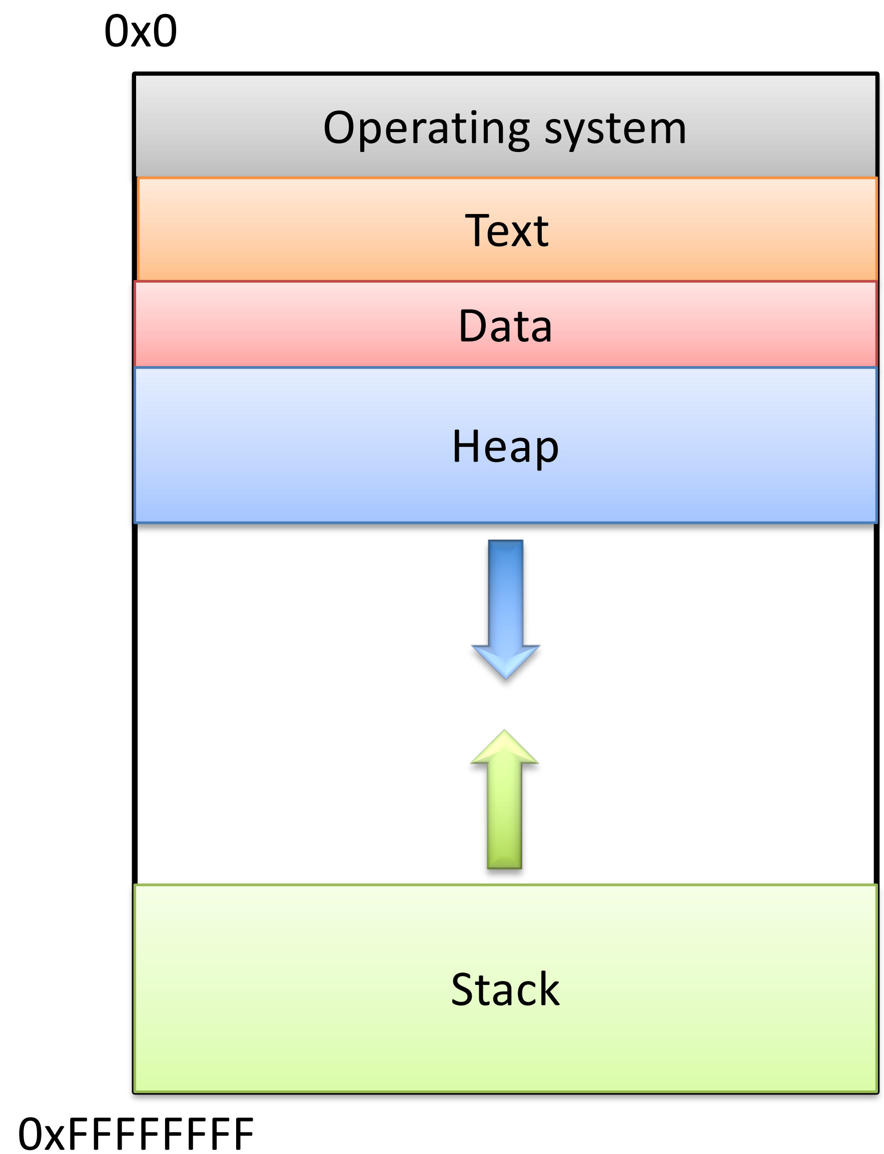
Segmentation Violation





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

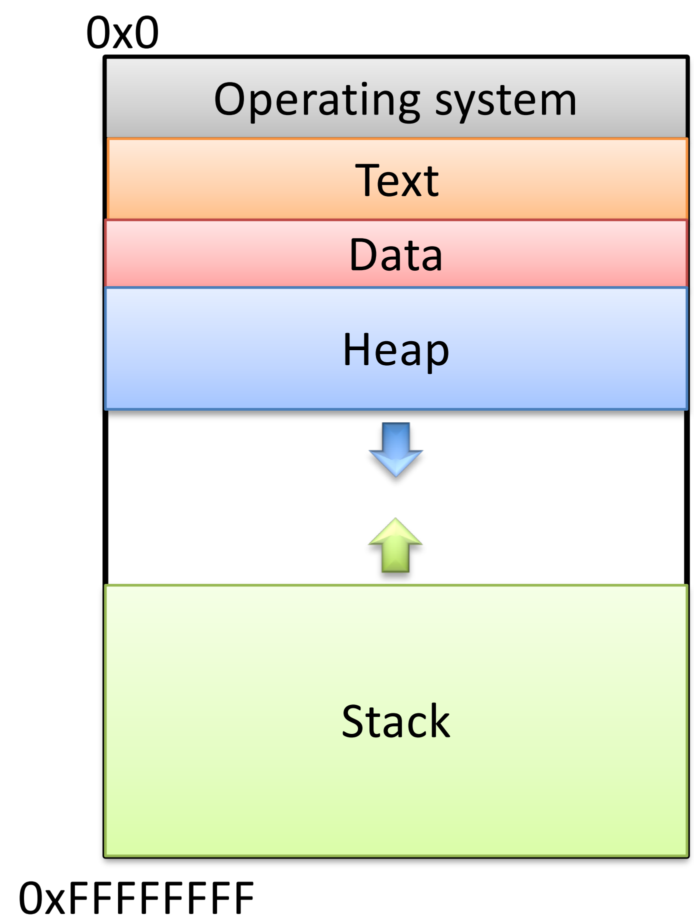


Recap

- & gives us the address of a variable (a pointer)
- * allows us to follow the address to memory, accessing the item (dereference the pointer)

- Memory model:
- So far, all variables on stack.

- Up next: using the heap.
  – We may not know the size of a variable in advance. (dynamic)



So we declared a pointer…

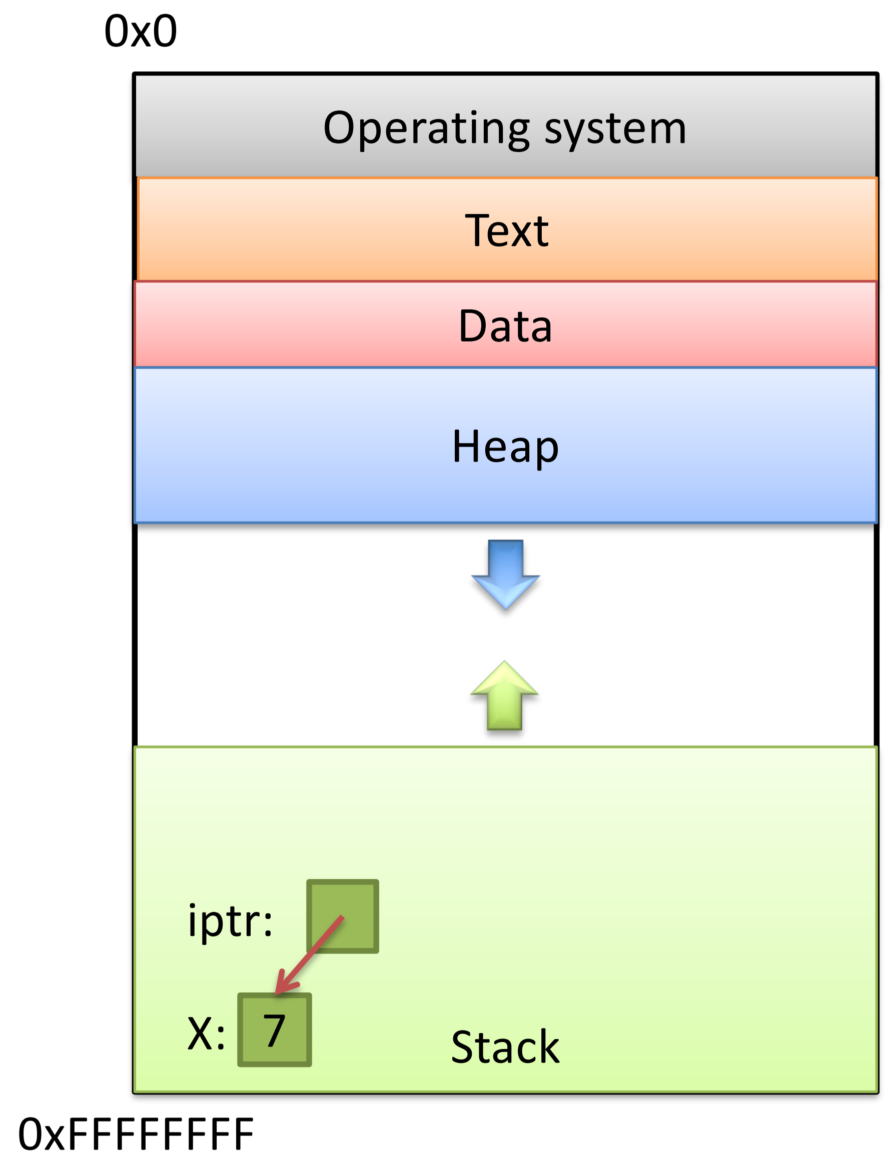
- How do we make it point to something?
  1. Assign it the address of an existing variable
  2. Copy some other pointer
  3. Allocate some memory and point to it

The Address Of (&)

- You can create a pointer to anything by taking its address with the *address of* operator (&).

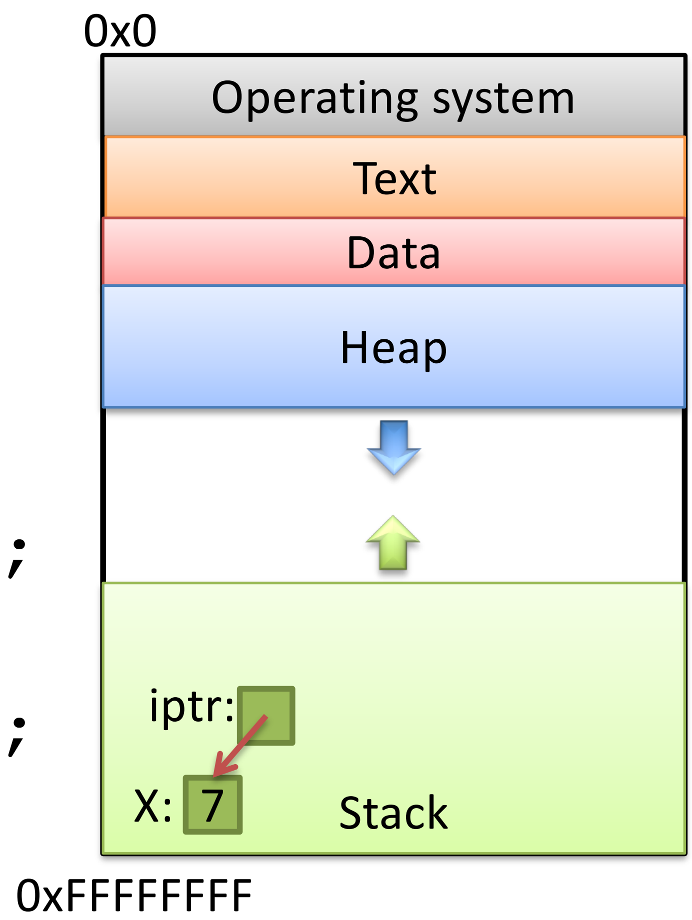
The Address Of (&)

```
int main() {
   int x = 7;
   int *iptr = &x;



   return 0;
}
```



What would this print?

```
int main() {
  int x = 7;
  int *iptr  = &x;
  int *iptr2 = &x;

  printf("%d %d ", x, *iptr);
  *iptr2 = 5;
  printf("%d %d ", x, *iptr);

  return 0;
}
```



A. 7 7 7 7 B. 7 7 7 5 C. 7 7 5 5 D. Something else

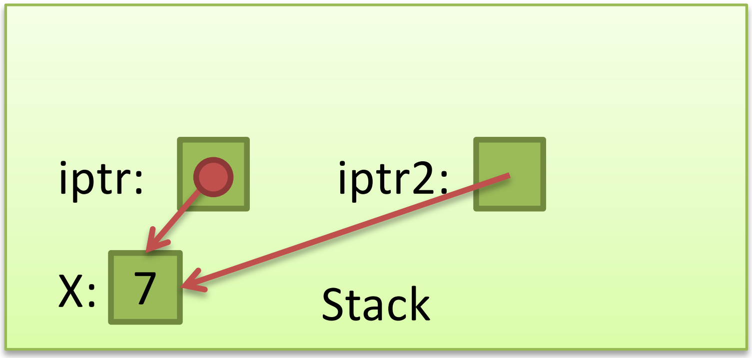
So we declared a pointer…

- How do we make it point to something?
  1. Assign it the address of an existing variable
  2. Copy some other pointer
  3. Allocate some memory and point to it

Copying a Pointer

- We can perform assignment on pointers to copy the stored address.

```
int x = 7;
int *iptr, *iptr2;
iptr = &x;
iptr2 = iptr;
```



Pointer Types

- By default, we can only assign a pointer if the type matches what C expects.

```
int x = 7;
int *iptr = &x;
```

```
int x = 7;
float *fptr = &x;
```

- "Warning: initialization from incompatible pointer type" (<u>Don't ignore this!</u>)

void *

- There exists a special type, void *, which represents "generic pointer" type.
  - Can be assigned to any pointer variable
  - int *iptr = (void *) &x;   // Doesn't matter what x is

- 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.

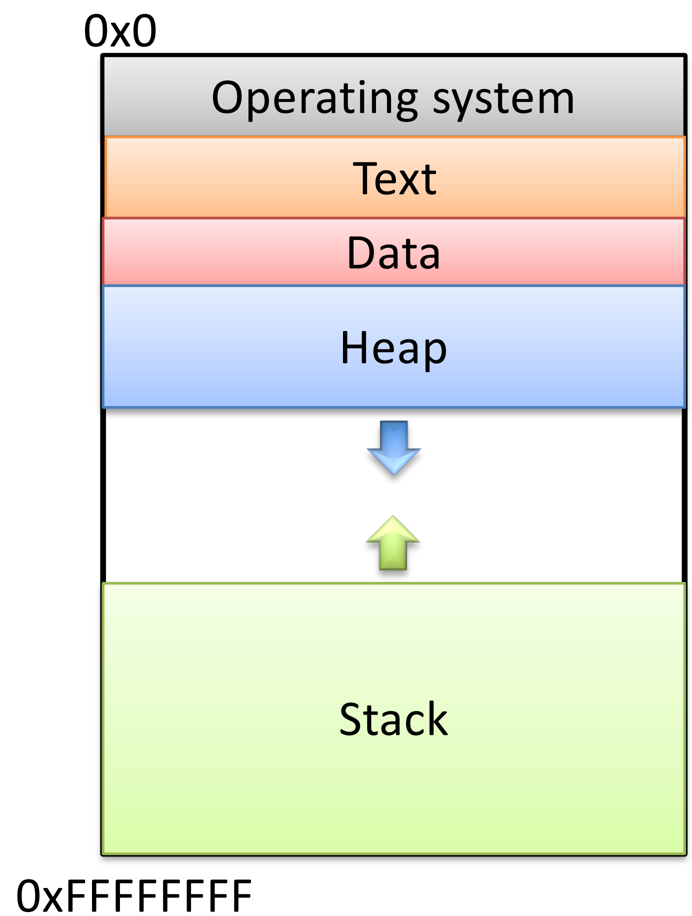
NULL: A special pointer value.

- 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 deref it.
  - Generally a good ideal to initialize pointers to NULL.

What will this do?

```
int main() {
  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 it something when you declare it, initialize your pointers to NULL.

So we declared a pointer…

- How do we make it point to something?
  1. Assign it the address of an existing variable
  2. Copy some other pointer
  3. Allocate some memory and point to it

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

Recall: void *

- void * is a special type that represents "generic pointer".
  - Can be assigned to any pointer variable

- 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…

Size Matters

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

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

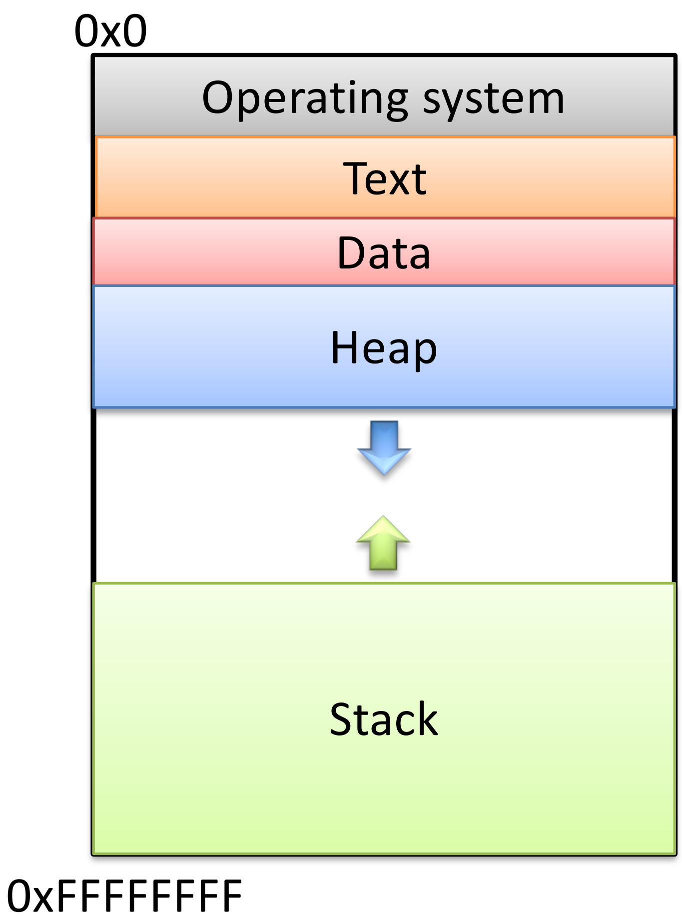
- How much memory should we ask for?

- Use C's sizeof() operator:

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

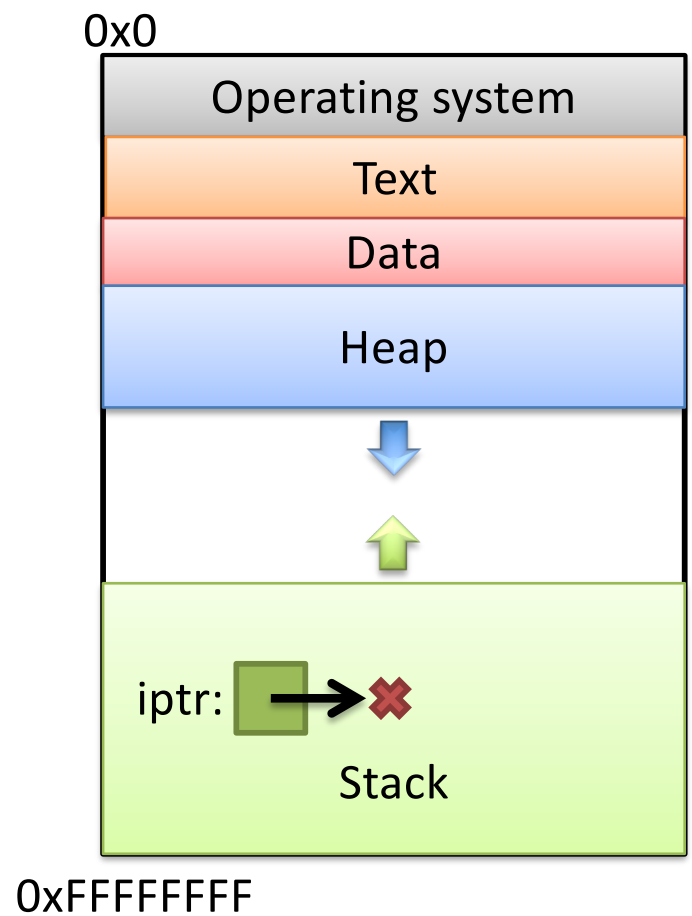
Example

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```



Example

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```

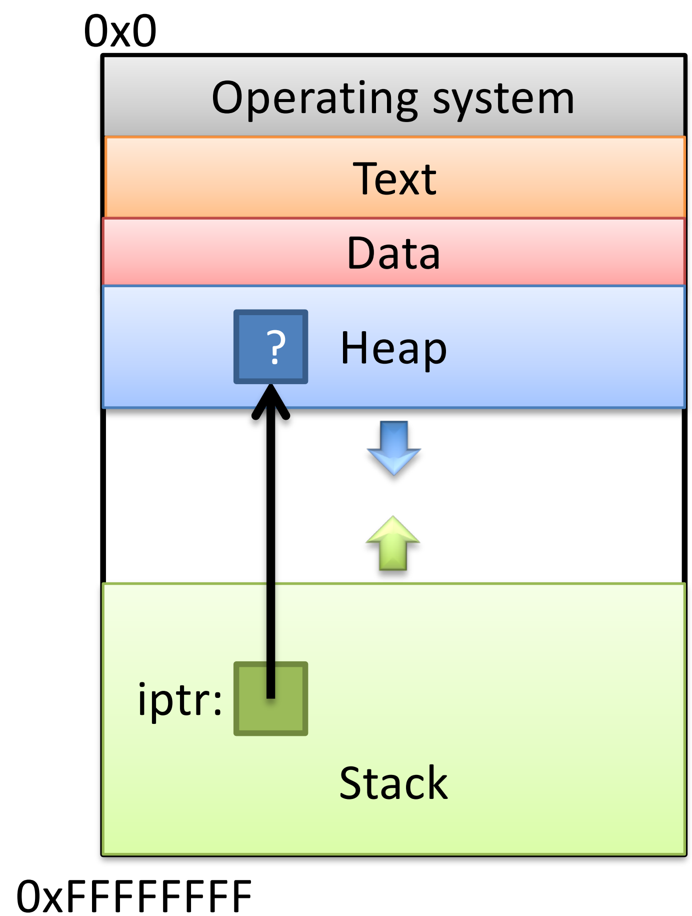


Create an integer pointer, named iptr, on the stack.

Assign it NULL.

Example

```
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.

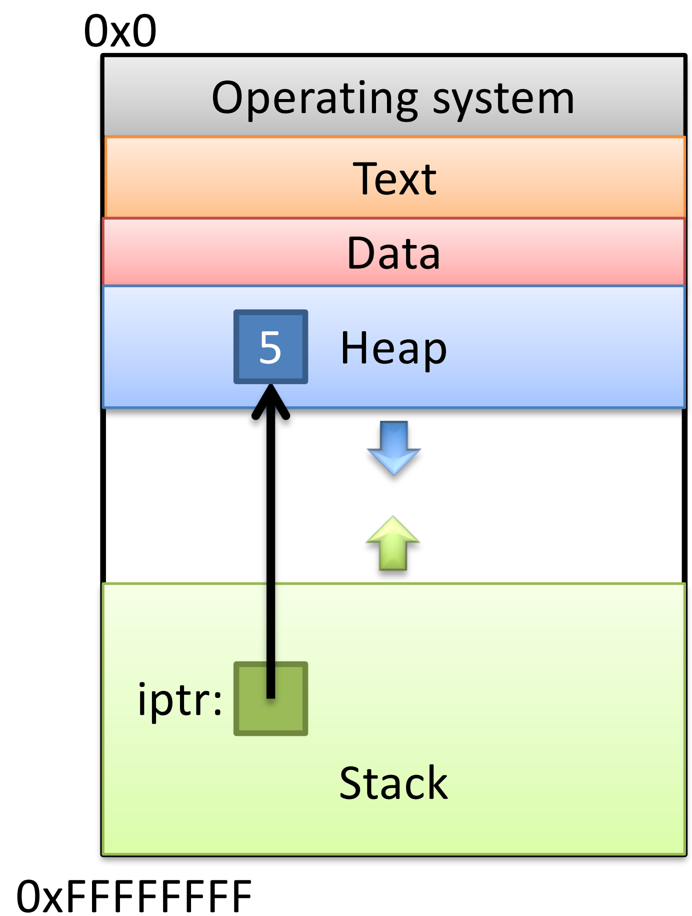
What value is stored in that area right now?

Who knows… Garbage.

Example

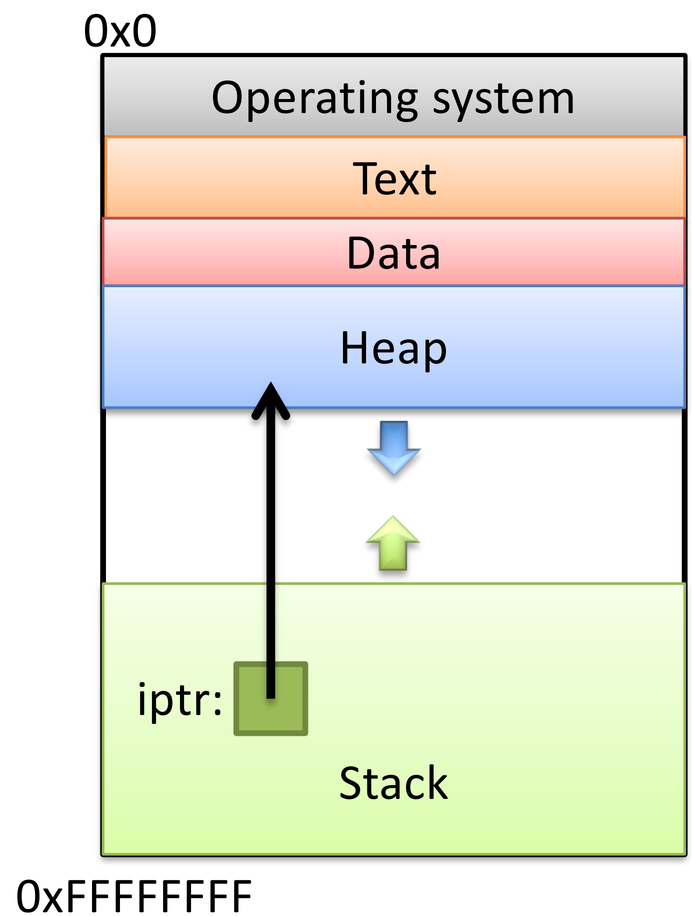
```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;
```

Use the allocated heap space by dereferencing the pointer.



Example

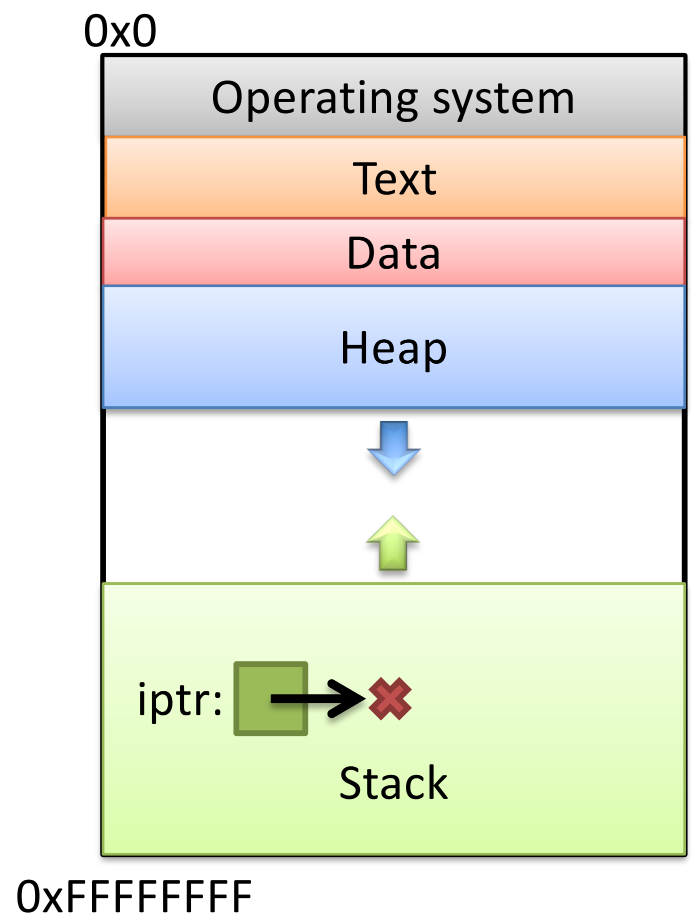
```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;

free(iptr);
```



Free up the heap memory we used.

Example

```
int *iptr = NULL;

iptr = malloc(sizeof(int));

*iptr = 5;

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



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

`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.

`sizeof()` example

```
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 ever want to see a number hard-coded in here!

You're designing a system. What should happen if a program requests memory and the system doesn't have enough available?

A. The OS kills the requesting program.
B. The OS kills another program to make room.
C. malloc gives it as much memory as is available.
D. malloc returns NULL.
E. Something else.

Running out of Memory

- If you're ever unsure of malloc / free's behavior:

```
$ man malloc
```

- According to the C standard:

"The malloc() function returns a pointer to the allocated memory that is suitably aligned for any kind of variable. **On error, this function returns NULL.**"

- Further down in the "Notes" section of the manual:

"[On Linux], when malloc returns non-NULL there is no guarantee that memory is really available. **If the system is out of memory, one or more processes will be killed by the OOM killer.**"

Running out of Memory

- If you're ever unsure of malloc / free's behavior:

```
$ man malloc
```

- According to the C standard:

"The malloc() function returns a pointer to the allocated memory that is suitably aligned for any kind of variable. **On error, this function returns NULL.**"

You should check for NULL after every malloc():

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

if (bob == NULL) {
    /* Handle this.  Often, print and exit. */
}
```

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 (where?)

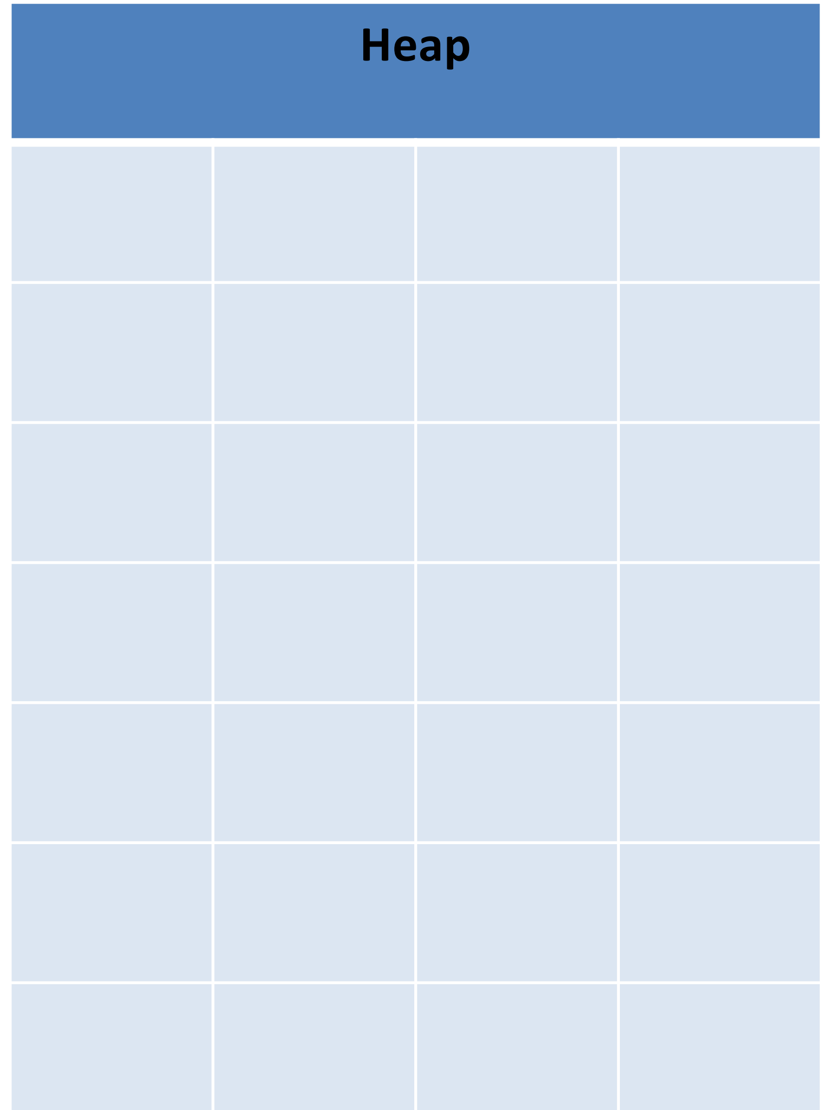
B. In main memory (how?)

C. On the disk

D. Somewhere else

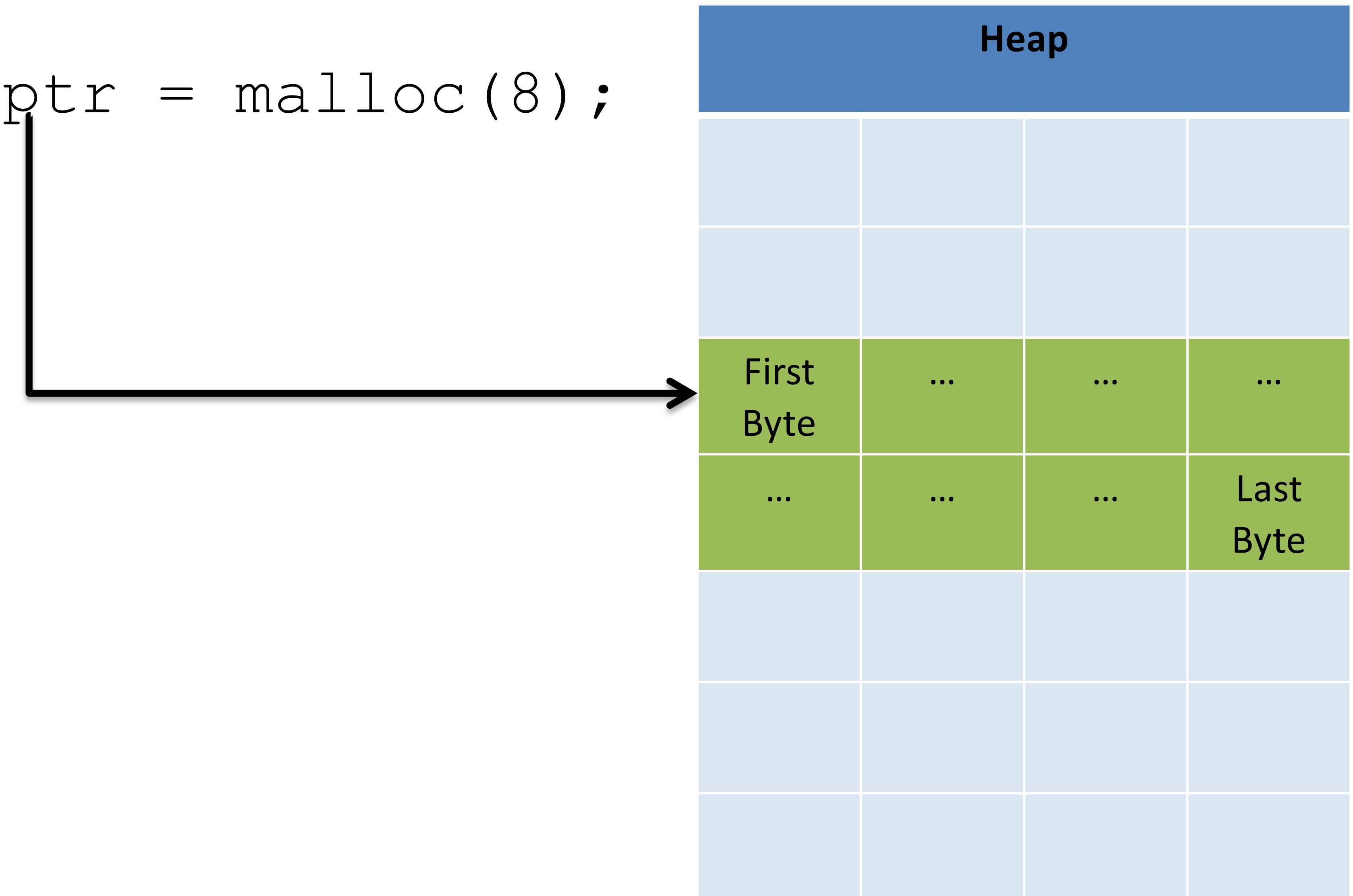
Metadata

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



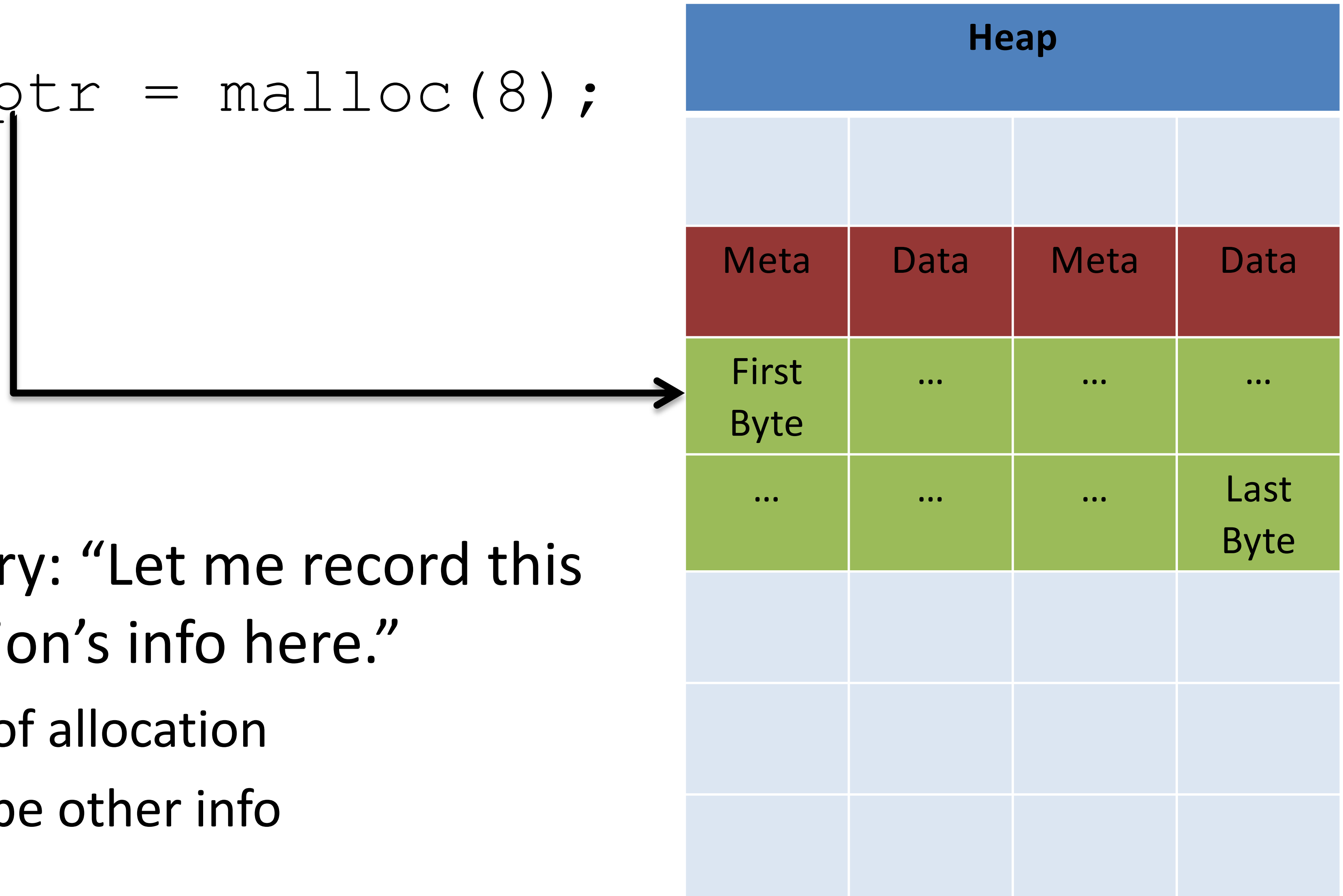
Metadata

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



Metadata

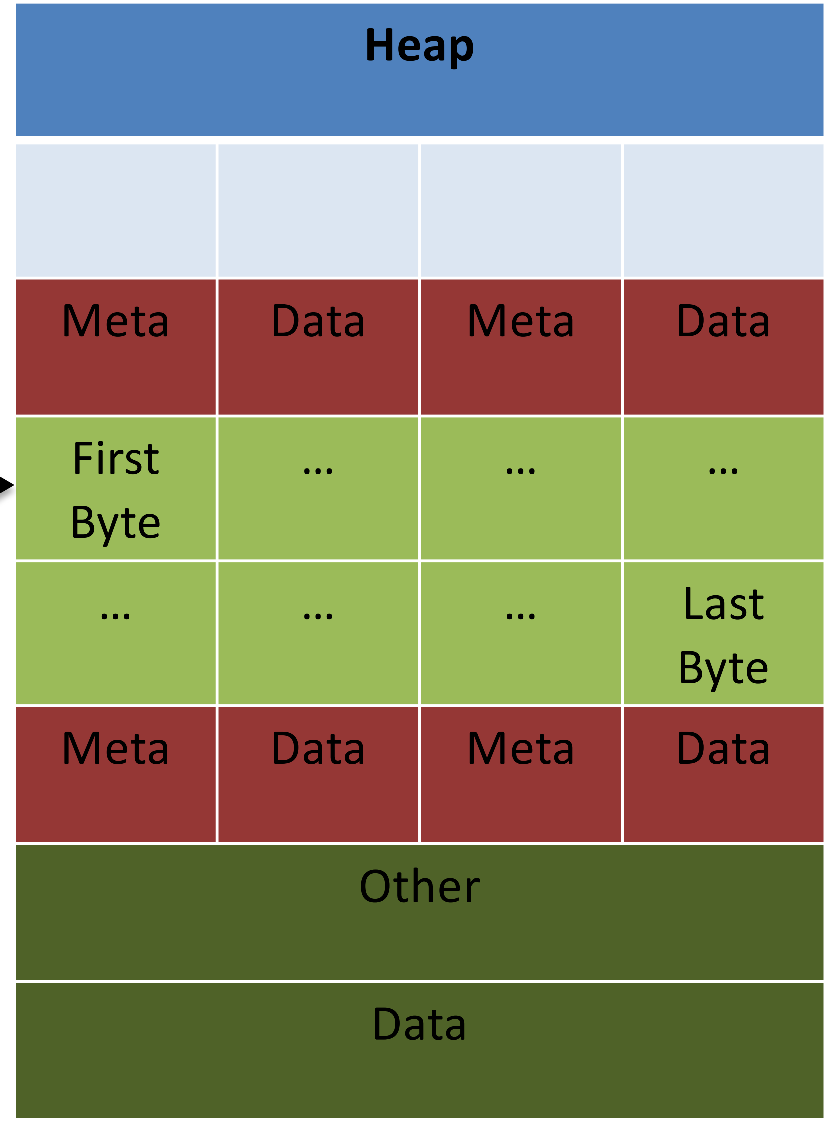
```
int *iptr = malloc(8);
```



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

Metadata

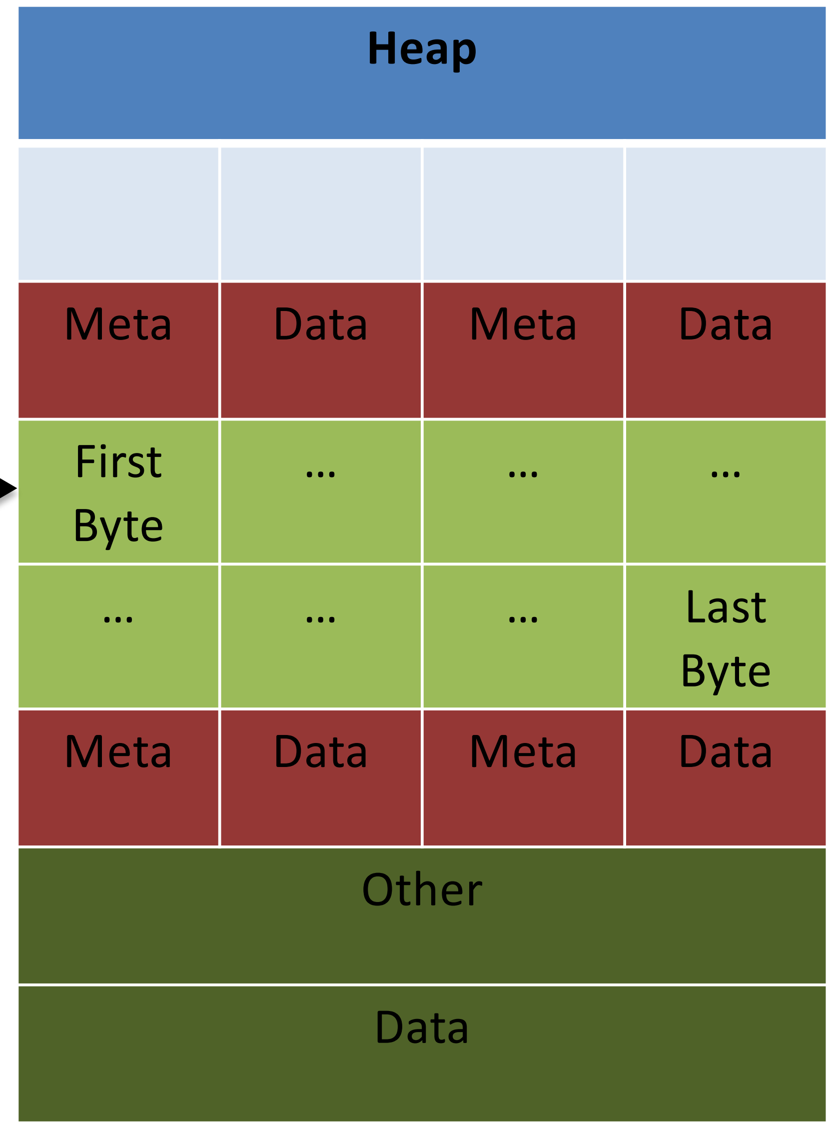
```
int *iptr = malloc(8);
```



- 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!

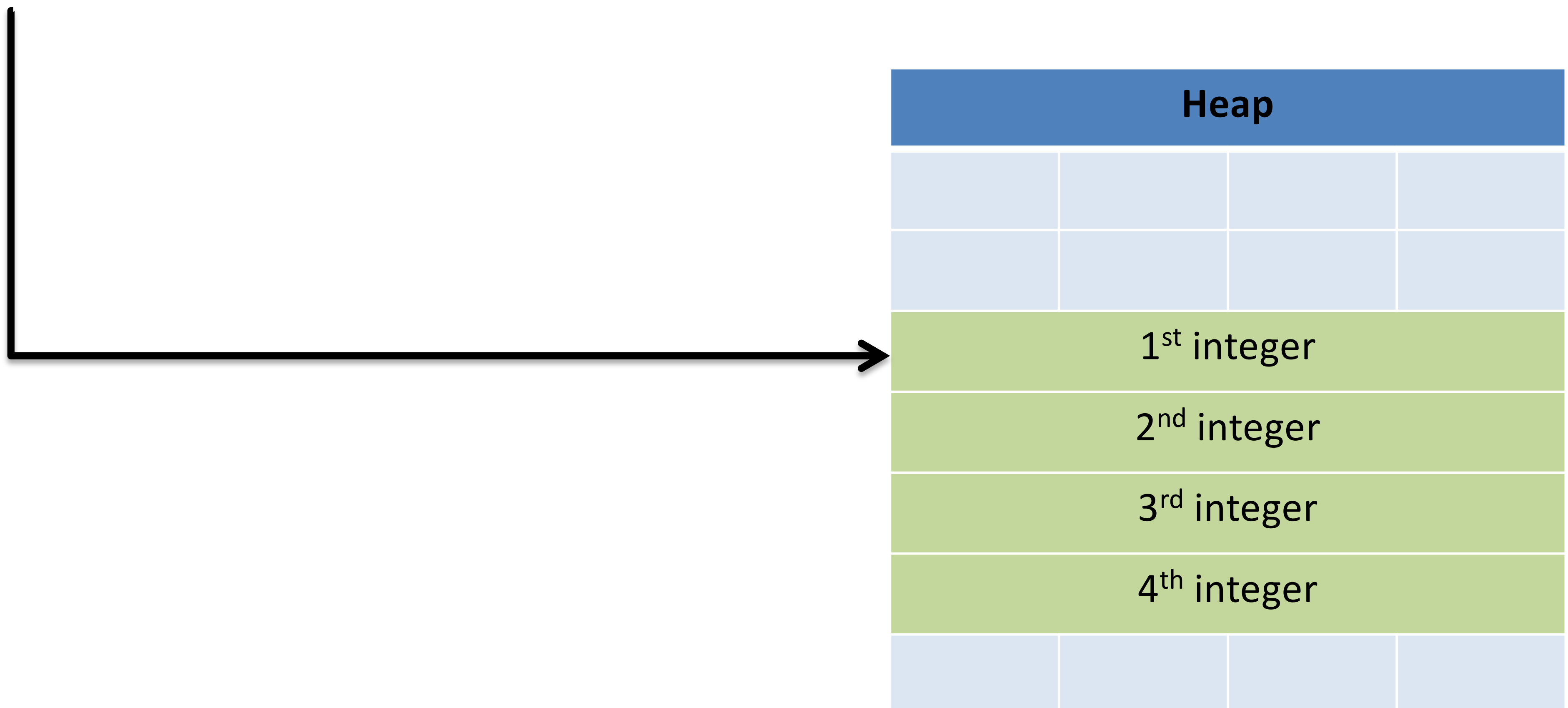
Pointers as Arrays

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

Heap			

Pointers as Arrays

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



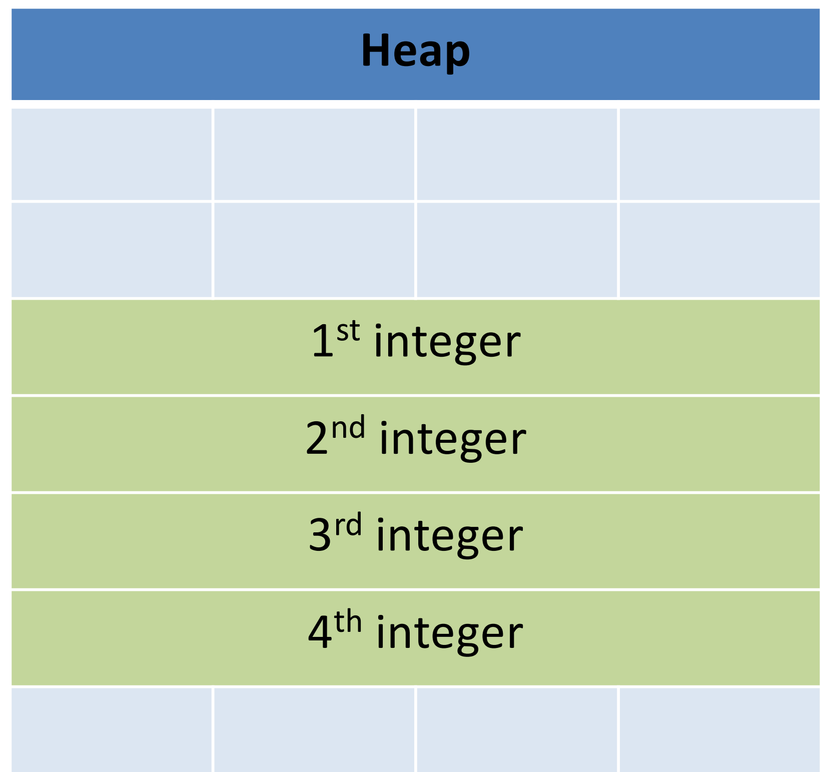
Pointers as Arrays

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

The C compiler knows how big an integer is.

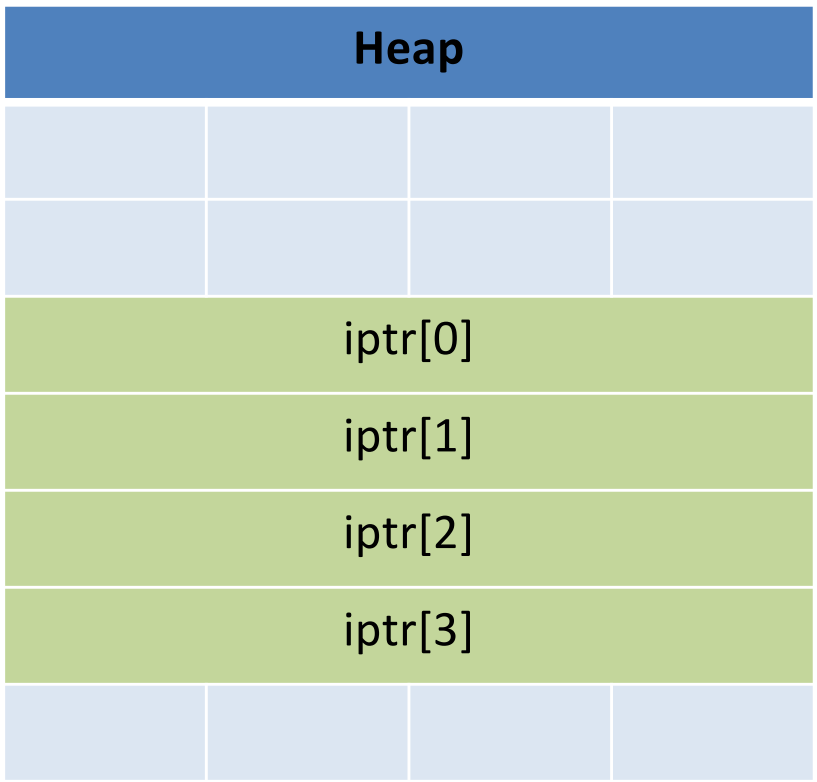
As an alternative way of dereferencing, you can use []'s like an array.

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



Pointers as Arrays

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

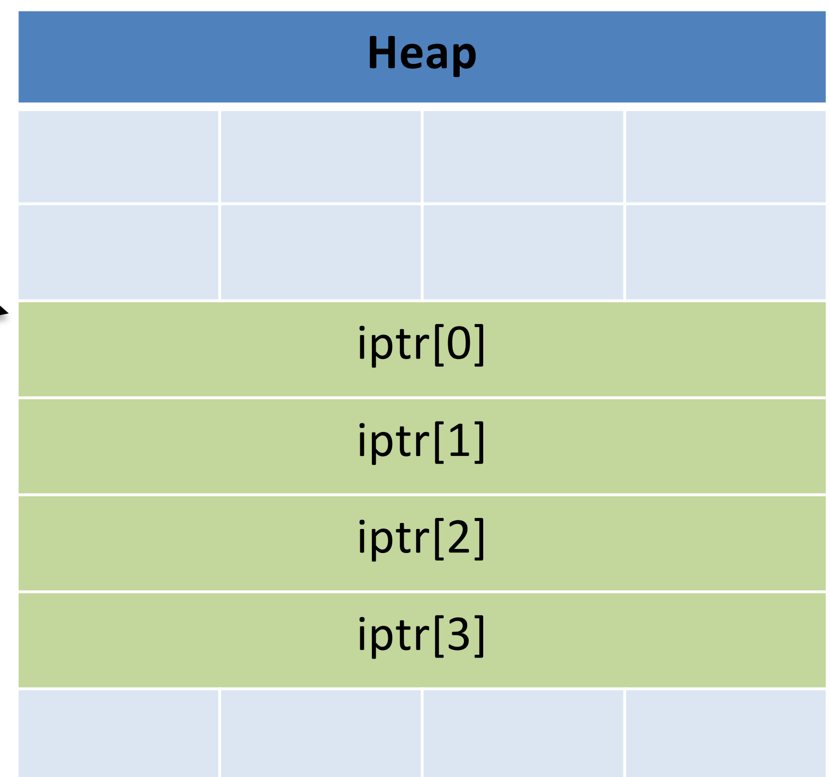


Pointers as Arrays

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

1. Start from the base of iptr.

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



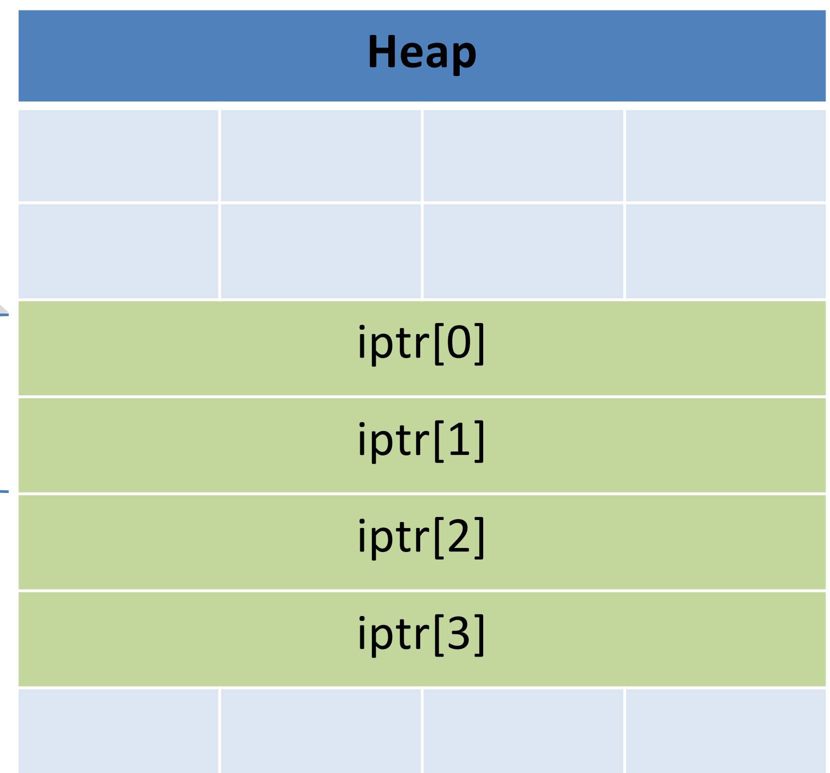
Pointers as Arrays

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

1. Start from the base of iptr.

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

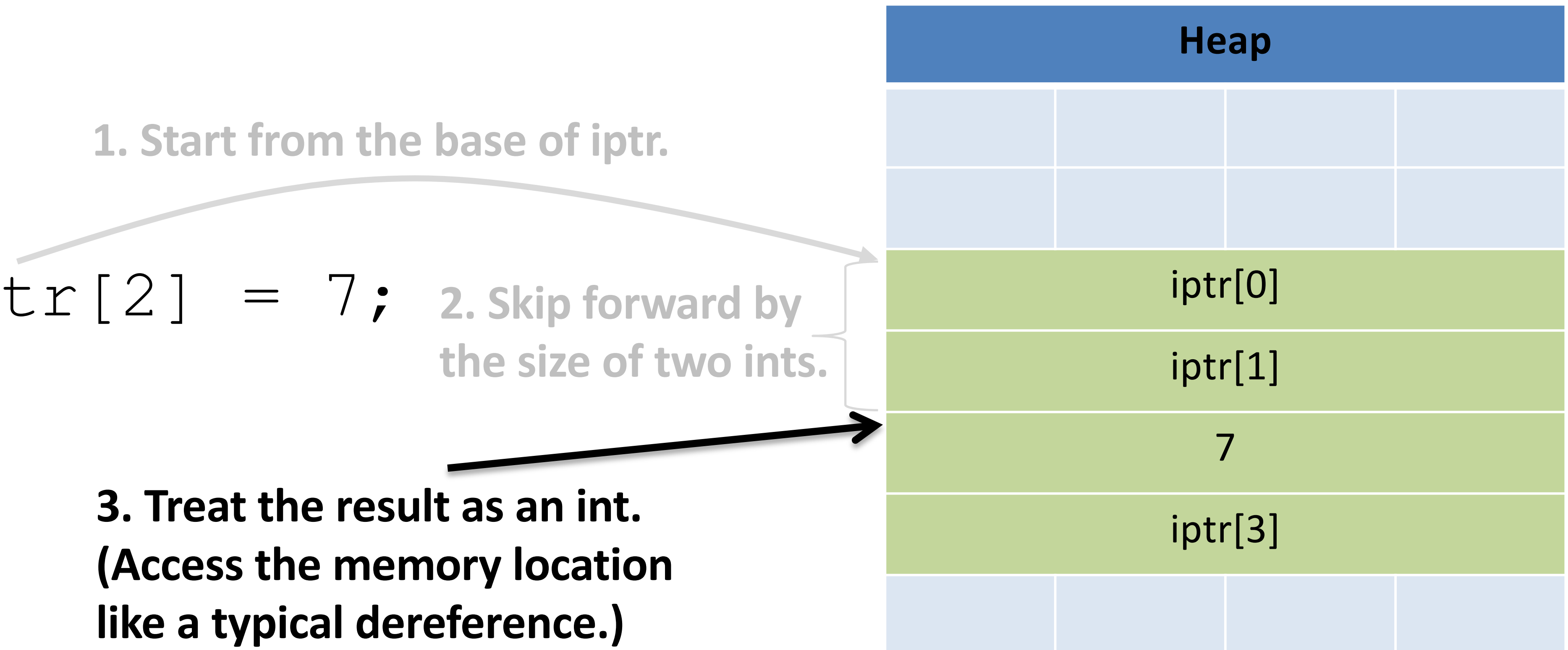
2. Skip forward by the size of two ints.



Pointers as Arrays

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

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



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.

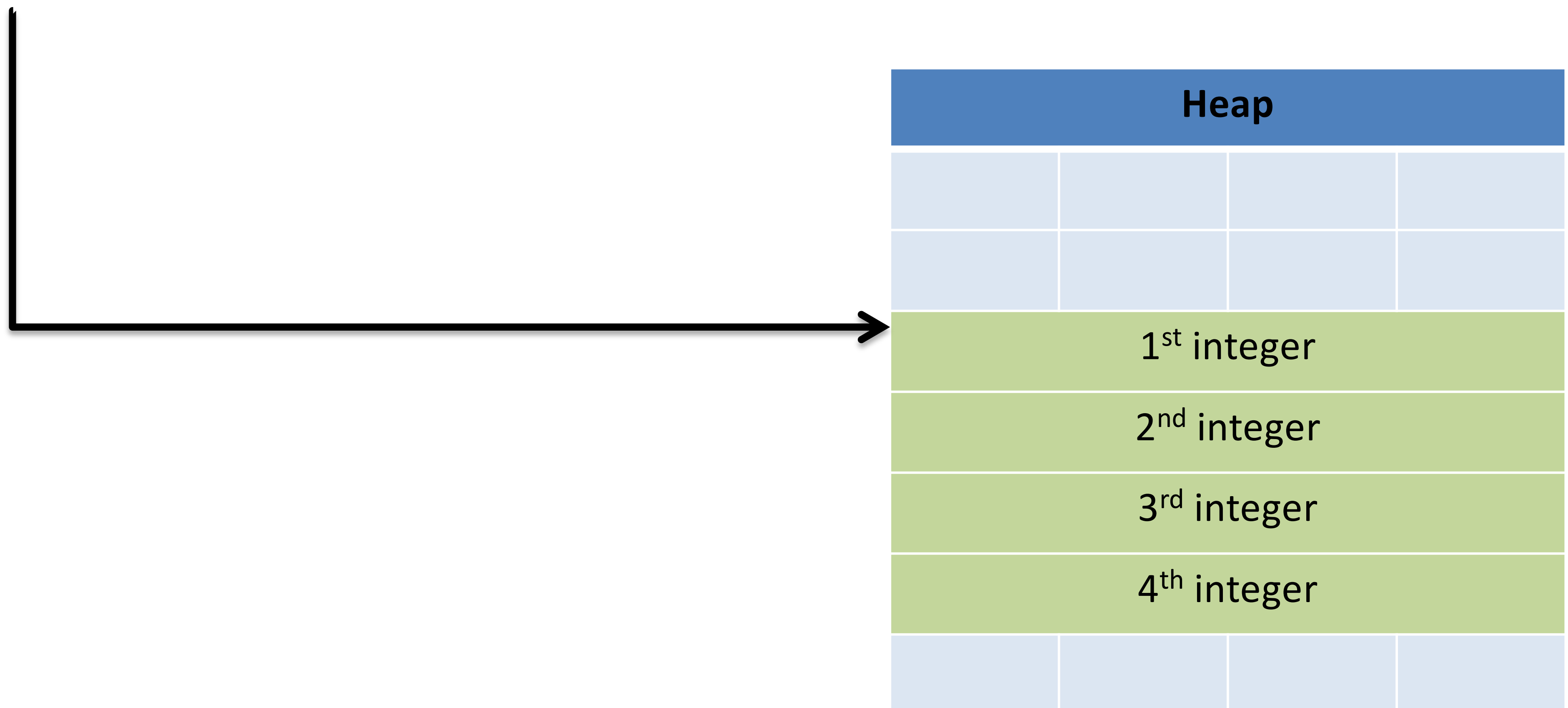
```
float *student_gpas = NULL;
student_gpas = malloc(n_students * sizeof(int));
…
student_gpas[0] = …;
student_gpas[1] = …;
```

Pointer Arithmetic

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

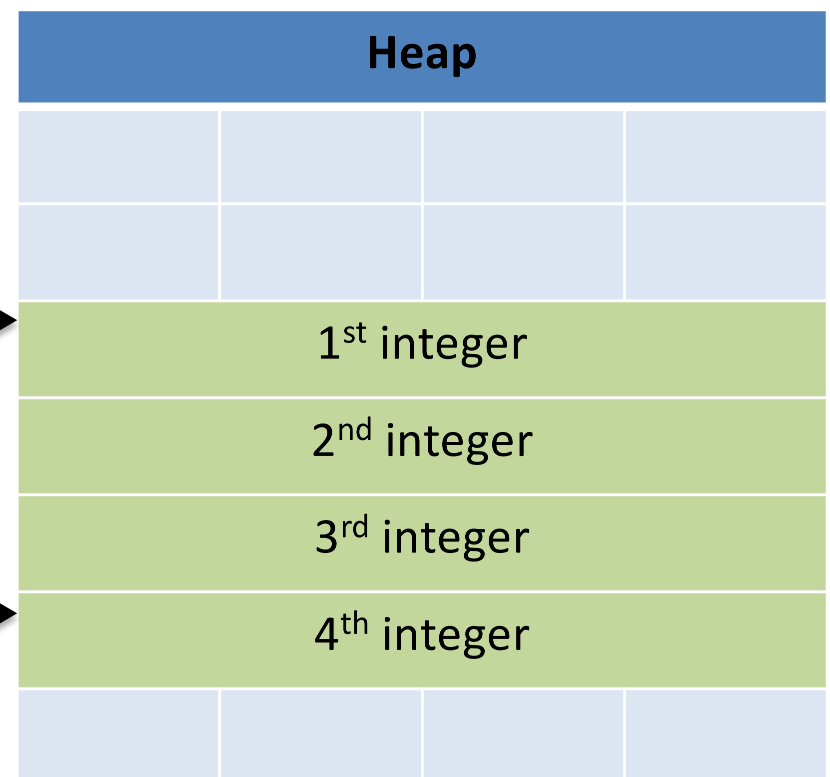
Pointer Arithmetic

```
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.

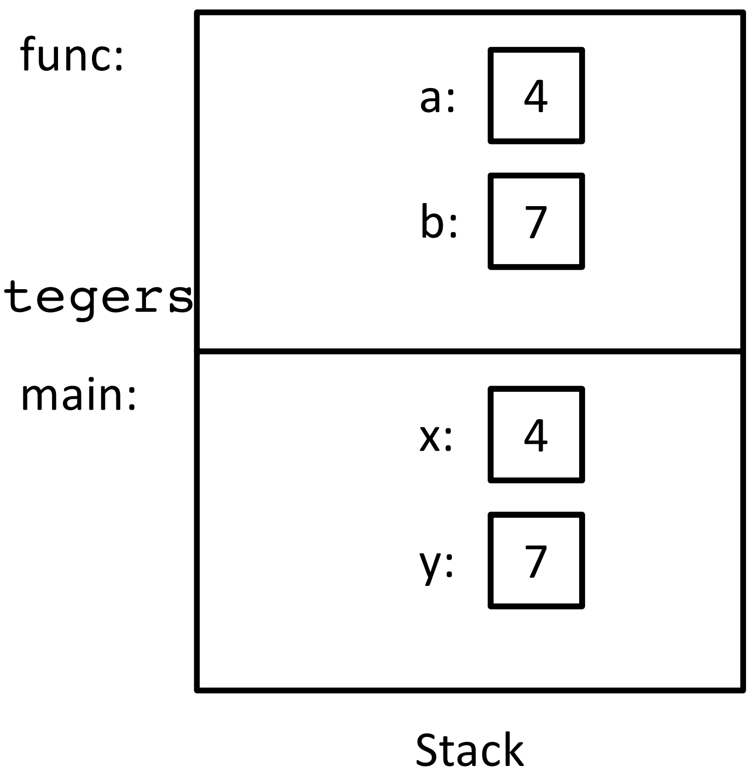
2. Allowing a function to return memory.

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);
}
```

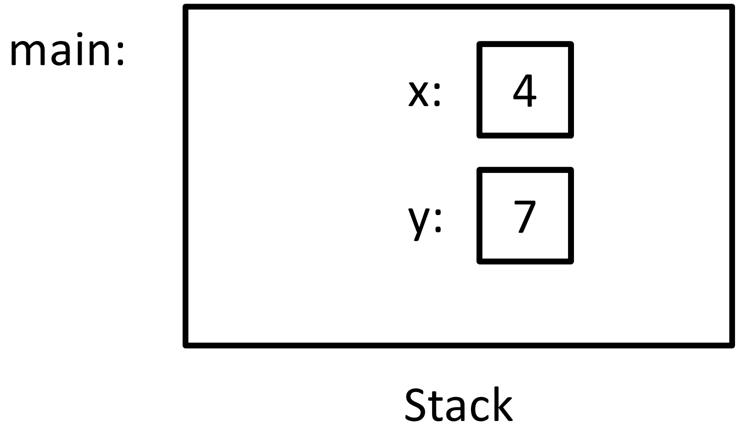


Function Arguments

- 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);
}
```

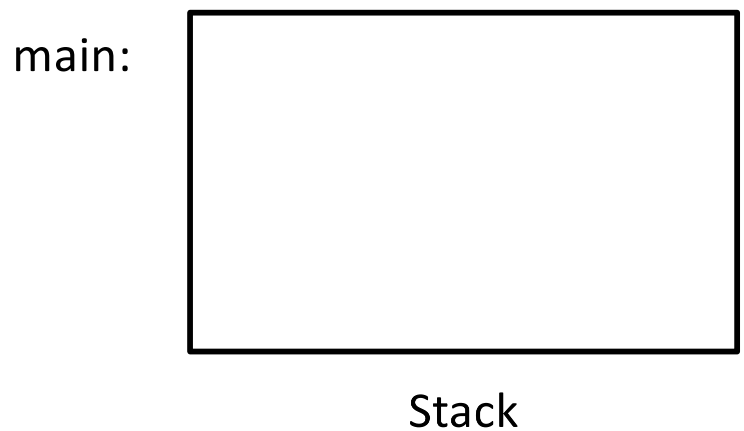
It doesn't matter what func does with a and b. The value of x in main doesn't change.



Function Arguments

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

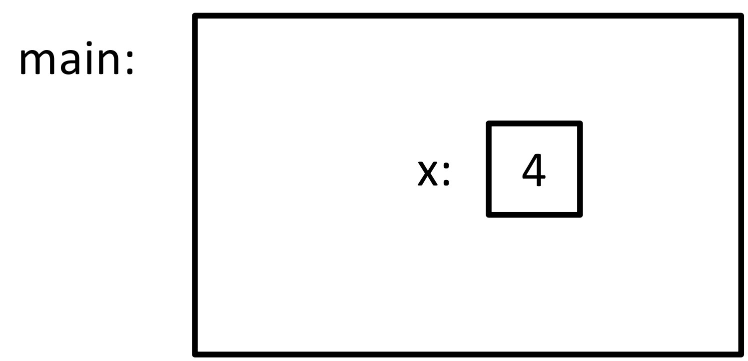
```
void func(int *a) {
    *a = *a + 5;
}

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);
}
```

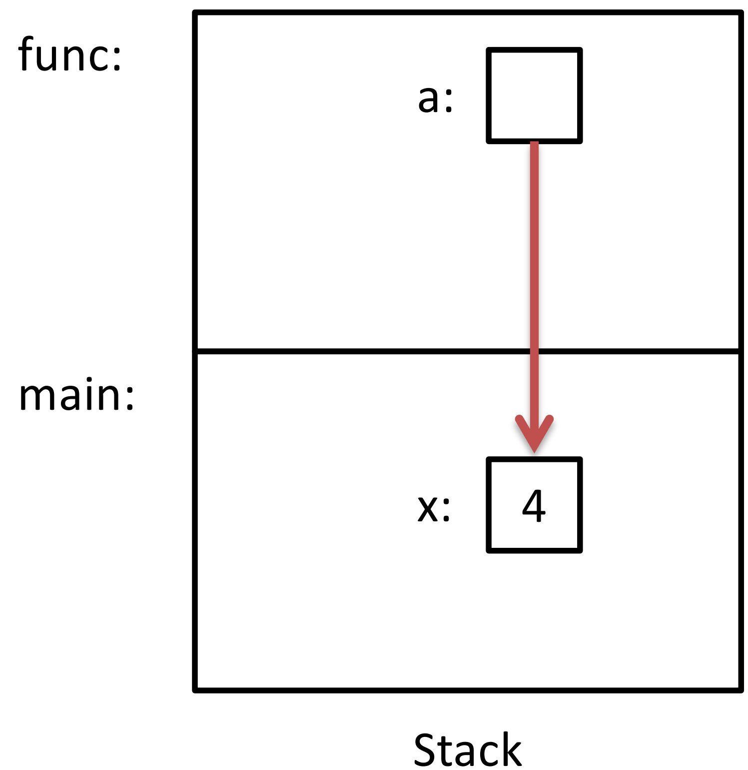


Stack

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);
}
```

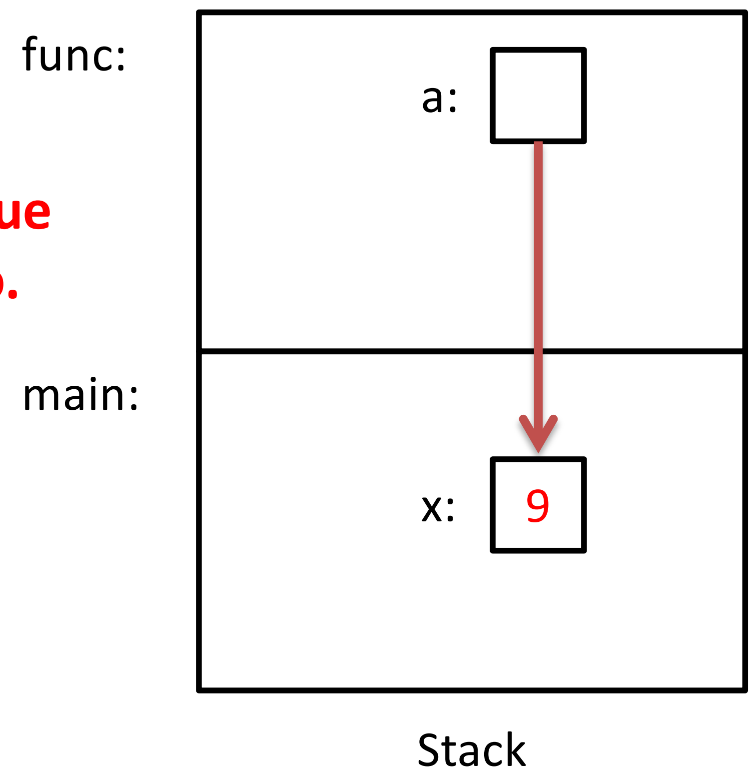


Pointer Arguments

- 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.



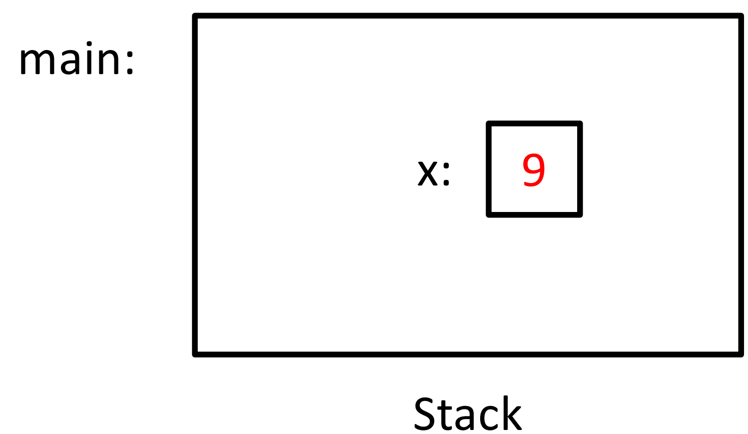
Pointer Arguments

- 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?



Readfile Library

- We saw this in lab 2 with read_int, 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?

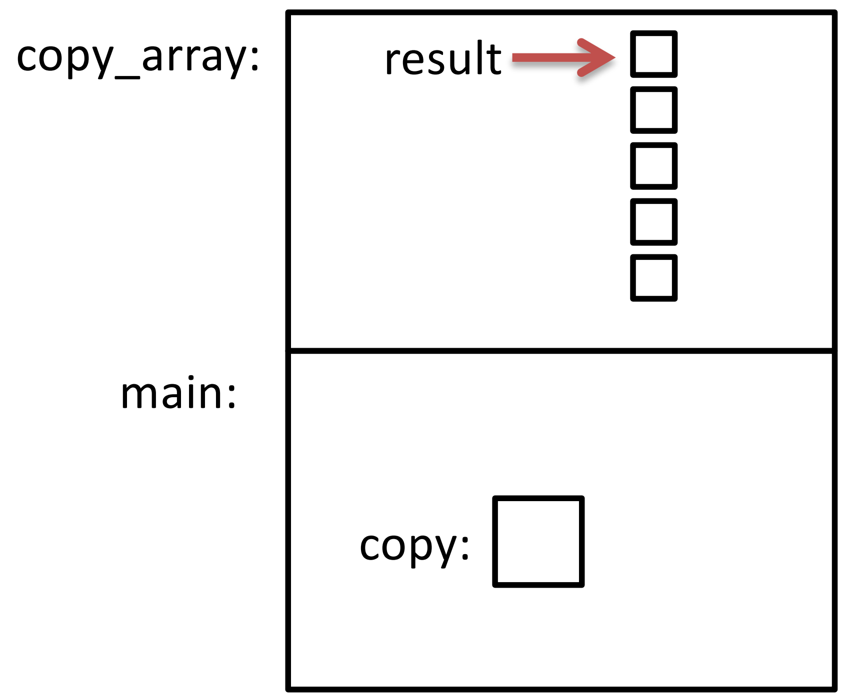
- Suppose you wanted to write a function that copies an array (of 5 integers).
  - Given: array to copy

```
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 wont let you do this anyway.)

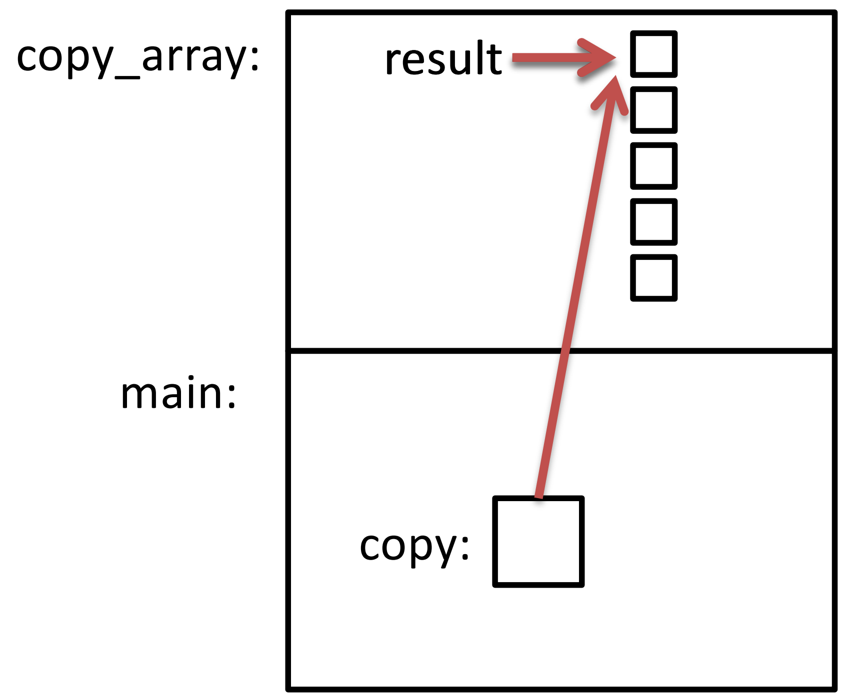
Consider the memory…

```
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(…)
```



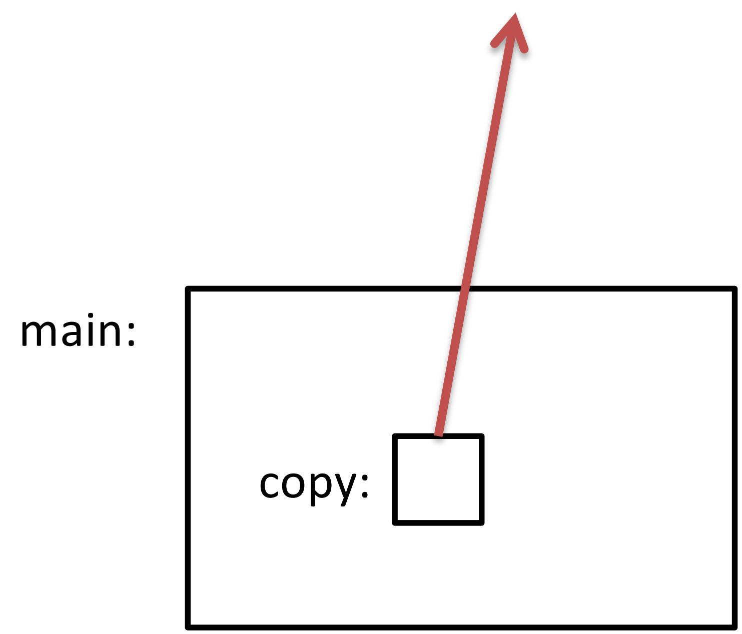
Consider the memory…

Left with a pointer to nowhere.

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

(In main):

```
copy = copy_array(…)
```

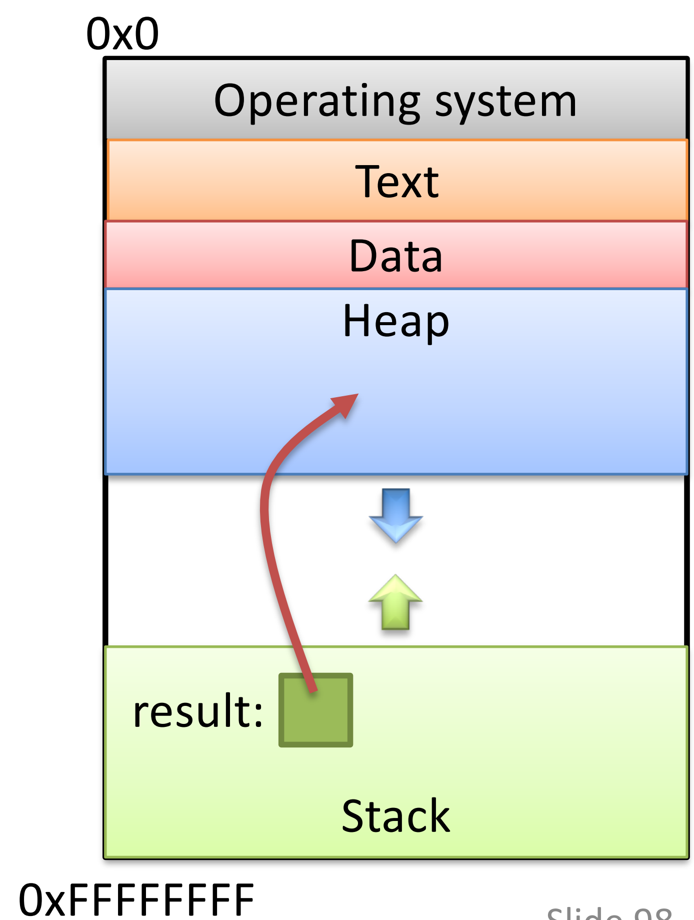


Using the Heap

```
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.)



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'll use them in lab 4.

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.)

Up Next:

- Function calls and stack management