

CS50 Section

Week 4

Attendance Sign In: <https://tinyurl.com/y6qvdwyo>

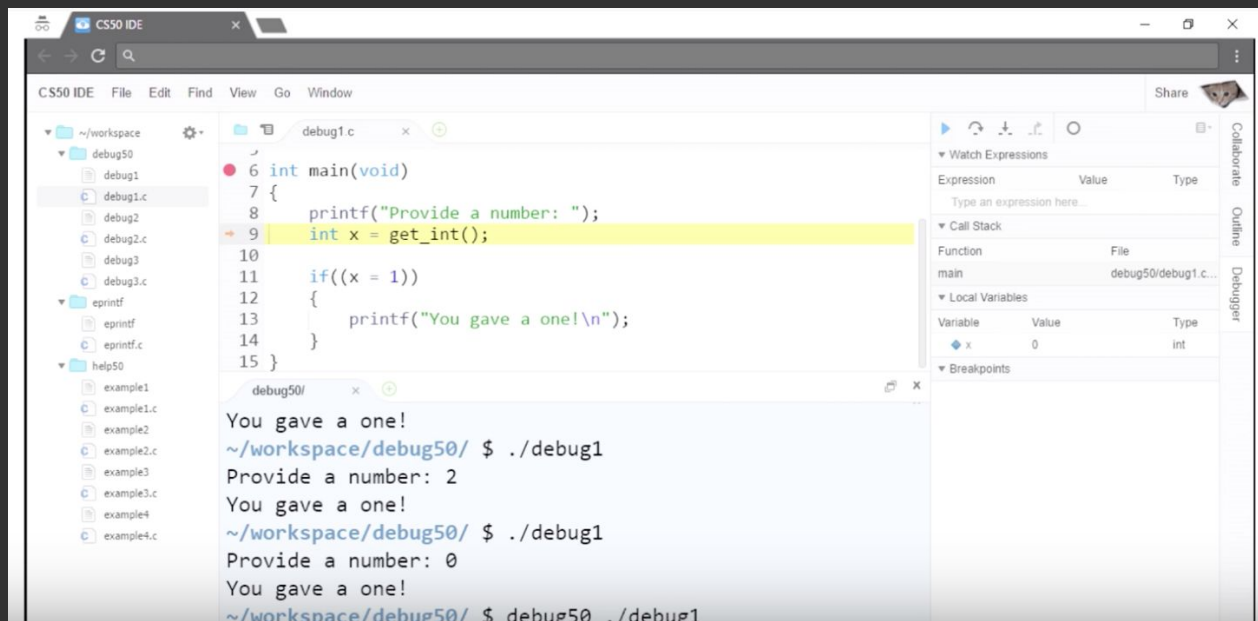
Agenda

- Pset 3 - General
- Debugging Tips
- Week 3 - Remaining Questions
- Week 4
 - Hexadecimal
 - Pointers
 - malloc and free
 - File I/O

Debugging

ONE SIMPLE TRICK TO SAVE HOURS ON DEBUGGING

(BUGS HATE IT!)



The screenshot shows the CS50 IDE interface. The main editor displays the code for `debug1.c`:

```
6 int main(void)
7 {
8     printf("Provide a number: ");
9     int x = get_int();
10
11     if((x = 1))
12     {
13         printf("You gave a one!\n");
14     }
15 }
```

The line `int x = get_int();` is highlighted in yellow, indicating the current step in the debugger. The right sidebar shows the 'Debugger' panel with the following sections:

- Watch Expressions:** A table with columns 'Expression', 'Value', and 'Type'. The first row is empty with the placeholder text 'Type an expression here'.
- Call Stack:** A table with columns 'Function' and 'File'. The first row shows 'main' from 'debug50/debug1.c'.
- Local Variables:** A table with columns 'Variable', 'Value', and 'Type'. The first row shows 'x' with value '0' and type 'int'.
- Breakpoints:** A section for managing breakpoints.

The bottom output window shows the program's execution:

```
~/workspace/debug50/ $ ./debug1
Provide a number: 2
You gave a one!
~/workspace/debug50/ $ ./debug1
Provide a number: 0
You gave a one!
~/workspace/debug50/ $ debug50 ./debug1
```

LINK: <https://www.youtube.com/watch?v=VtkMZjvvKaU>

PROBLEM SOLVING HANDOUT

Based off George Polya's *How to Solve It*, additions by Annie Chen, updated Fall 2017 by Amanda Vostburgh

How to Solve a CS50 Pset

1. Understanding the Problem

"What is the unknown? What is the data? What is the goal? Break the problem into parts"

What is the spec asking you to do? What types of input will you be taking? What should your program be outputting?

Steps: Write down the requirements for the spec in your own words. Watch the walkthrough.

2. Devising a Plan

"Have you seen it before? Do you know a related problem? Tackle the problem one part at a time"

How can you turn the inputs into outputs as a person? How can you get the program to do the same thing? What assumptions/prior knowledge did you as a person use? How can you get your program to know the same thing?

Steps: Using the sample inputs from the spec, figure out (without code) how to get to the outputs. Write down the steps you used to do that. Write down the pseudocode/hints from the walkthrough. Outline the entire program in pseudocode in your source code file.

3. Carrying Out The Plan: Use Examples! Use Google!

Turning concepts into code is often the difficult part, especially if you don't know how to start. What problems have you seen like this before? (In section or lecture?)

Steps: Look back on the source code from the week's lecture/section. How can you change that code to use in your pset? Figure out how the code snippets from the walkthrough fit into your pseudocode. Search reference50 and Google for more information on the functions mentioned in the psets. Understand what inputs they take and what they return. Fit them into your pseudocode.

4. Looking Back: Test Your Code!

"Can you check the result? Can you derive the result differently?"

How does your program handle invalid input? Does it ever crash? Could you have implemented the solution more efficiently?

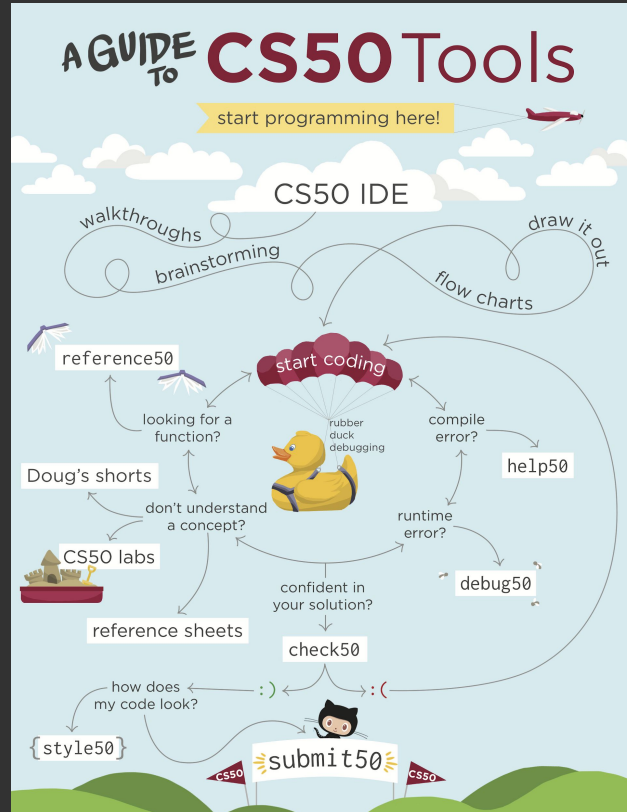
Steps: Run check50. Pinpoint what inputs (if any) your program is failing on, and figure out why they're failing. For help, use CS50 Discourse or Office Hours. Look carefully through your code for areas of improvement. Factor out common code, comment to make intent clear, fix style. When pset is graded, review feedback. Ask for clarification if needed. Watch Postmortem video. Consider how your code could be further improved, and different ways solution could have been implemented.

Checklist

- ❑ **Understand the Problem**
 - ❑ Write down the requirements for the spec in your own words.
- ❑ **Devise A Plan**
 - ❑ Watch the walkthrough.
 - ❑ Write down the pseudocode/hints from the video.
 - ❑ Using the sample inputs from the spec, figure out (without code) how to get to the outputs.
 - ❑ Write down the steps you used to do that.
 - ❑ Outline the entire program in pseudocode in your source code file.
- ❑ **Carry Out the Plan**
 - ❑ Look back on the source code from the week's lecture/section. How can you change that code to use in your pset?
 - ❑ Figure out how the code snippets from the walkthrough fit into your pseudocode.
 - ❑ Search reference50 and Google for more information on the functions mentioned in the psets.
 - ❑ Understand what inputs they take and what they return.
 - ❑ Fit them into your pseudocode.
- ❑ **Check Your Work**
 - ❑ Run check50.
 - ❑ Pinpoint what inputs (if any) your program is failing on, and figure out why they're failing.
 - ❑ For help, use CS50 Discourse or Office Hours.
 - ❑ Look carefully through your code for areas of improvement.
 - ❑ Factor out common code.
 - ❑ Add comments to make intent clear.
 - ❑ Fix style.
 - ❑ When pset is graded, review feedback.
 - ❑ Ask for clarification if needed.
 - ❑ Watch Postmortem video.
 - ❑ Consider how your code could be further improved, and different ways solution could have been implemented.

<https://github.com/rw5614/cs50sectiontf/blob/master/Debugging.pdf>

USE ALL THE TOOLS & RESOURCES AVAILABLE TO YOU



Pset 3 Debrief

Week 3 Review - What Questions do You Have?

- Structs
- Recursion
- Searching
- Sorting
- Running Time

Week 4

Today We'll Cover

- Hexadecimal
- Pointers
- Memory Management: malloc and free
- File I/O

“SHORTS” FOR THE WEEK



A man in a maroon shirt stands against a light gray background. The text 'CALL STACKS' is overlaid in large white letters.

CALL STACKS

https://youtu.be/j_oJoK0LoJY



A man in a maroon shirt stands against a light gray background. The text 'FILE POINTERS' is overlaid in large white letters.

FILE POINTERS

<https://youtu.be/-BNy3eEBGt0>



A man in a maroon shirt stands against a light gray background. The text 'POINTERS' is overlaid in large white letters.

POINTERS

<https://youtu.be/8VAhORT0ZW8>



A man in a maroon shirt stands against a light gray background. The text 'DYNAMIC MEMORY ALLOCATION' is overlaid in large white letters.

DYNAMIC MEMORY ALLOCATION

https://youtu.be/gkA_H8HlwRE



A man in a maroon shirt stands against a light gray background. The text 'HEXADECIMAL' is overlaid in large white letters.

HEXADECIMAL

<https://youtu.be/8okwMK6htKE>

Hexadecimal

Decimal (Base-10)

0 1 2 3 4 5 6 7 8 9

10^7 10^6 10^5 10^4 10^3 10^2 10^1 10^0

10010000

Binary (Base-2)

0 1

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

0b10010000

Hexademical (Base-16)

0 1 2 3 4 5 6 7 8 9 A B C D E F

16^7 16^6 16^5 16^4 16^3 16^2 16^1 16^0

0xFF FFFF

Equivalence

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0

11111111

10^2 10^1 10^0 16^1 16^0

255

FF

Warmup Exercise

11101100101001

→ 0x (Hexademical)

Warmup Exercise

0011 1011 0010 1001

3

B

2

9

Pointers

PASSING DATA IN C

Remember from last week, when we wanted to know what the following code would print:

```
#include <stdio.h>

int main(void) {
    int x = 7;
    int y = x;
    x = 2;

    printf("%i\n", y);
}
```

PASSING DATA IN C

Remember from last week, when we wanted to know what the following code would print:

```
#include <stdio.h>
```

```
int main(void) {
```

```
    int x = 7;
```

```
    int y = x;
```

```
    x = 2;
```

```
    printf("%i\n", y);
```

```
}
```

This will print 7. Why?



PASSING DATA IN C

Remember from last week, when we wanted to know what the following code would print:

```
#include <stdio.h>

int main(void) {
    int x = 7;
    int y = x;
    x = 2;

    printf("%i\n", y);
}
```

This will print 7. Why?

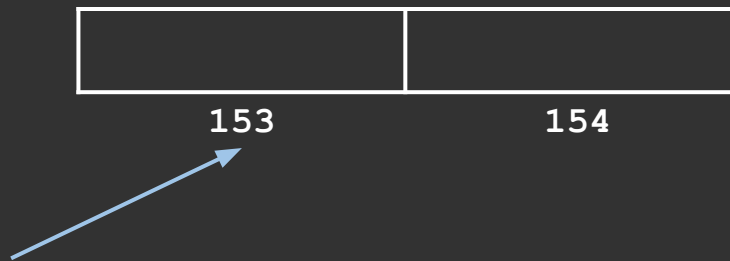
In C, majority of variables are passed by value. That means:

- 1) **x** is set equal to 7.
- 2) Then **y** is set equal to value that **x** represents, which is 7.
- 3) We change the value of **x**, but **y** was set equal to 7, so it prints 7.

PASSING DATA IN C

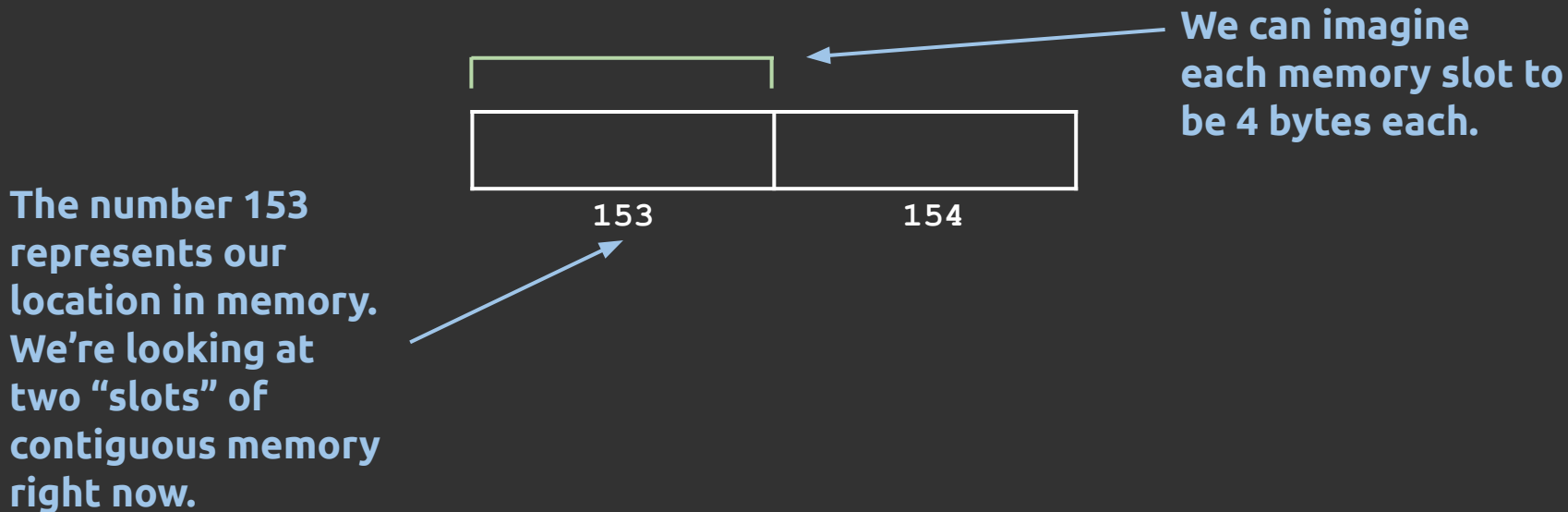
Check out this diagram which represents our computer *memory*.

The number 153 represents our location in memory. We're looking at two "slots" of contiguous memory right now.



PASSING DATA IN C

Check out this diagram which represents our computer *memory*.



PASSING DATA IN C

```
int x = 7;
```

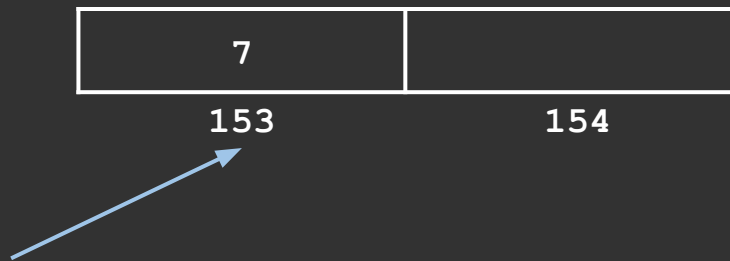
The program looks for a memory slot big enough to hold an integer, finds slot #153 is free, and then assigns `x` to it.



PASSING DATA IN C

```
int x = 7;
```

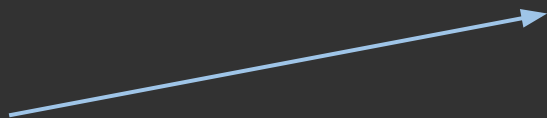
The program looks for a memory slot big enough to hold an integer, finds slot #153 is free, and then assigns `x` to it.



PASSING DATA IN C

```
int y = x;
```

The program looks for another memory slot big enough to hold an integer. It finds slot #154 and reserves it for `y`.



PASSING DATA IN C

```
int y = x;
```



It then goes to the
slot in x and finds
what that value is.



PASSING DATA IN C

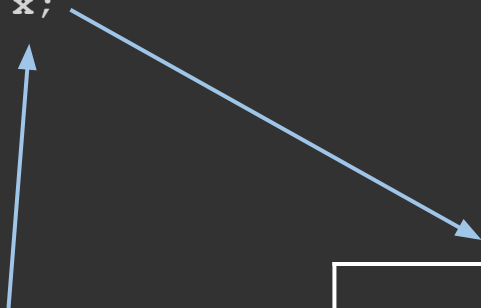
```
int y = x;
```

It then goes to the
slot in x and finds
what that value is.



PASSING DATA IN C

```
int y = x;
```



7	
153	154

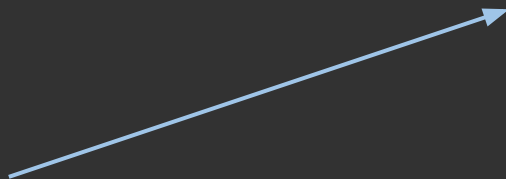
Well, `x` is a variable that holds an integer equal to 7, so it passes that value back to the assignment operator.

PASSING DATA IN C

```
int y = x;
```



Memory location
#154 gets set equal
to the value of `x`.



PASSING DATA IN C

```
x = 2;
```

The assignment operator has been passed the value of 2 to update what is stored in `x`'s memory location.

7	7
153	154

PASSING DATA IN C

```
x = 2;
```



The memory
location for `x` is
found and updated.

PASSING DATA IN C

```
printf("%i\n", y);
```



2	7
153	154

The `printf()`
function is passed `y`
by value.

PASSING DATA IN C

```
printf("%i\n", y);
```



The program looks for the address of `y` in memory and retrieves the value.

PASSING DATA IN C

```
printf("%i\n", y);
```



2	7
153	154

The `printf()` statement prints 7, the value stored at the address for `y`.

PASSING DATA IN C

```
printf("%i\n", y);
```



2	7
153	154

The `printf()` statement prints 7, the value stored at the address for `y`.

SO WHAT ABOUT THIS MATTERS?

Everything you've seen thus far has been passing by value.

But we can also pass by reference. This means you pass the actual address of a memory location rather than the value store there.

How can we accomplish that?

WHAT IS A POINTER?

To learn to pass by reference, we must first learn about **pointers**. Recall our memory diagram from before:



WHAT IS A POINTER?

```
int x = 7;
```



This statement
assigned the value
of 7 to memory
location #153.



WHAT IS A POINTER?

```
int x = 7;
```

```
int *p = NULL;
```



7	NULL
153	154

Now we've declared
a pointer and set it
equal to NULL.

WHAT IS A POINTER?

```
int x = 7;
```

```
int *p = NULL;
```



Now we've declared
a pointer and set it
equal to NULL.

7	NULL
153	154

Pointer p also takes up
a memory location: slot
#154. Pointers are
variables too! They're
just variables that
store memory
locations.

WHAT IS A POINTER?

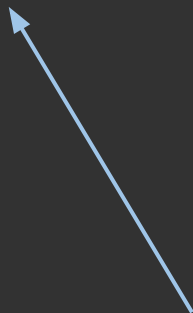
```
int x = 7;  
int *p = NULL;  
p = &x;
```



We now pass `x` by reference and assign it to `p`. This means we get the memory location for `x` and set `p` equal to it.

WHAT IS A POINTER?

```
int x = 7;  
int *p = NULL;  
p = &x;
```



7	0x99
153	154

We now pass `x` by reference and assign it to `p`. This means we get the memory location for `x` and set `p` equal to it.

Note that `0x99` is stored at memory location #154. Memory locations are actually represented as hexadecimal numbers: $153 = 0x99$.

WHAT IS A POINTER?

```
int x = 7;  
int *p = NULL;  
p = &x;  
int y = *p;
```



7	0x99	7
153	154	155

Now we *dereference* our pointer
and assign the value of the
memory address it points to in
y.

WHAT IS A POINTER?

So what do we have?



WHAT IS A POINTER?

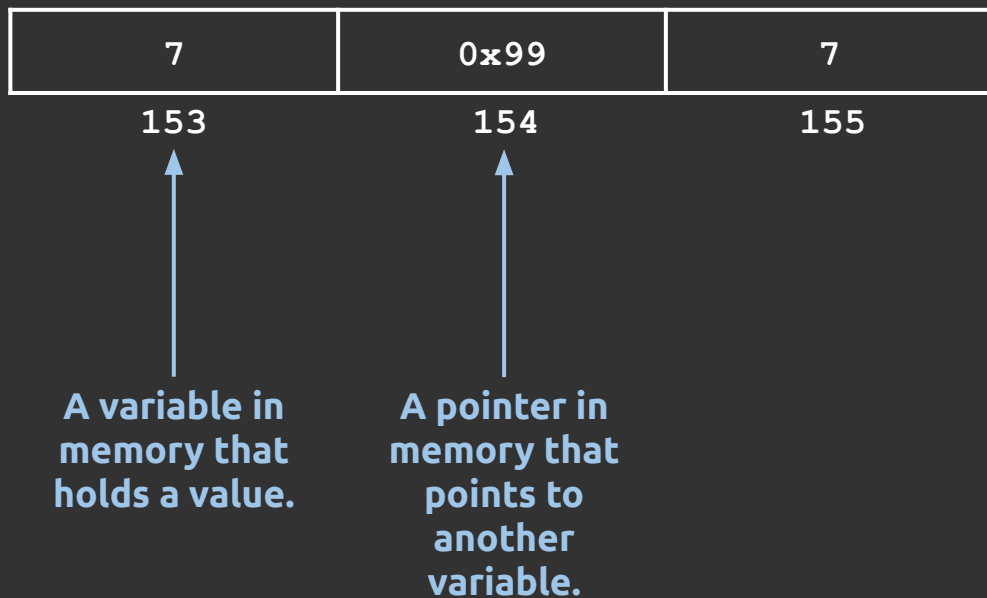
So what do we have?



A variable in
memory that
holds a value.

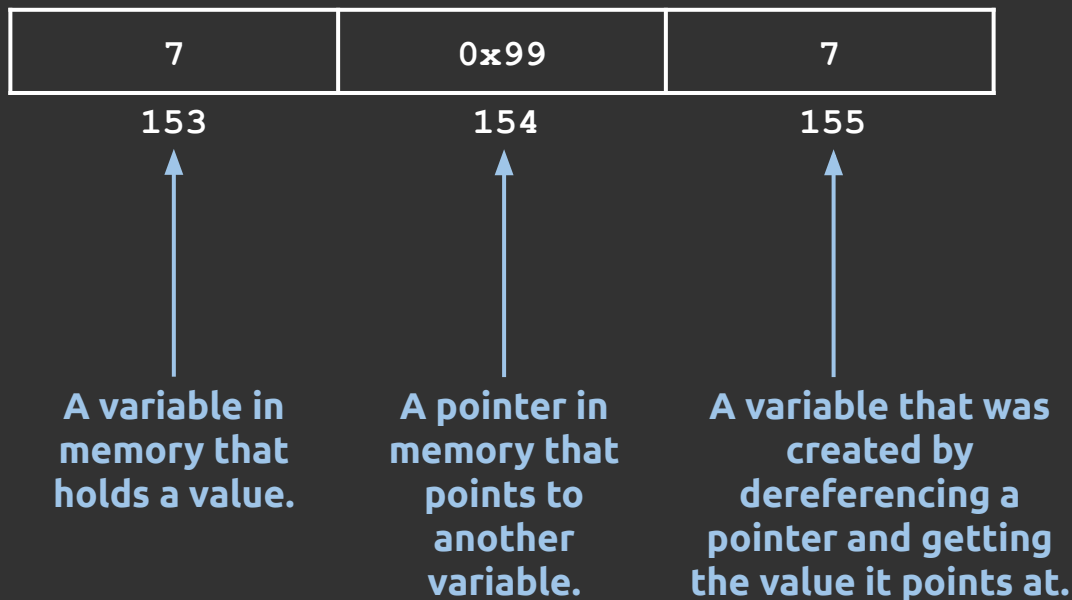
WHAT IS A POINTER?

So what do we have?



WHAT IS A POINTER?

So what do we have?



THE NULL POINTER

The simplest type of pointer in C is the null pointer:

```
int *pointer = NULL;
```

THE NULL POINTER

The simplest type of pointer in C is the null pointer:

```
int *pointer = NULL;
```

If you declare a pointer and don't use it immediately, best practices call for immediately setting it to null.

WORKING WITH POINTERS

So we've established that we can create a pointer the following way:

```
int *pointer;
```

WORKING WITH POINTERS

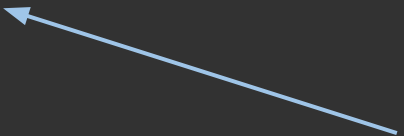
We can get the address of another variable using the `&` operator:

```
int x = 3;  
int *pointer = &x;
```

WORKING WITH POINTERS

We can get the address of another variable using the `&` operator:

```
int x = 3;  
int *pointer = &x;
```



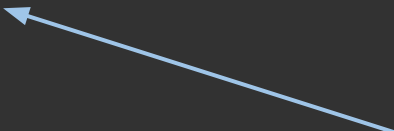
This tells C to get the address of `x` in memory.

If we just wanted the value of `x`, what would we do?

WORKING WITH POINTERS

We can get the address of another variable using the `&` operator:

```
int x = 3;  
int *pointer = &x;
```



This tells C to get the address of `x` in memory.

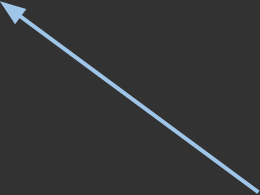
If we just wanted the value of `x`, what would we do?

You just pass by value: `int *pointer = x;`

WORKING WITH POINTERS

We can get the value at the memory address the pointer is referencing using `*`, the **dereference operator**:

```
int x = 3;  
int *pointer = &x;  
int y = *pointer;
```



This goes the memory address that `pointer` points to, retrieves the value, and then returns it.

Note the notation is the same thing we use to declare pointers.

WORKING WITH POINTERS

Take caution: The * you use to declare pointers is part of both the type and variable name:

```
int *px, py, pz;
```

WORKING WITH POINTERS

Take caution: The * you use to declare pointers is part of both the type and variable name:

```
int *px, py, pz;  
int *px, *py, *pz;
```

QUICK QUESTION

If a pointer just holds a memory address, and all memory addresses are just hexadecimal numbers so they take up roughly space in memory, why do we have to specify a pointer's type?

QUICK QUESTION

If a pointer just holds a memory address, and all memory addresses are just hexadecimal numbers so they take up roughly space in memory, why do we have to specify a pointer's type?

Because pointers point to different data types which all vary in size. C wouldn't know where to stop "reading" memory if it couldn't distinguish between the size of memory blocks that its pointing to.

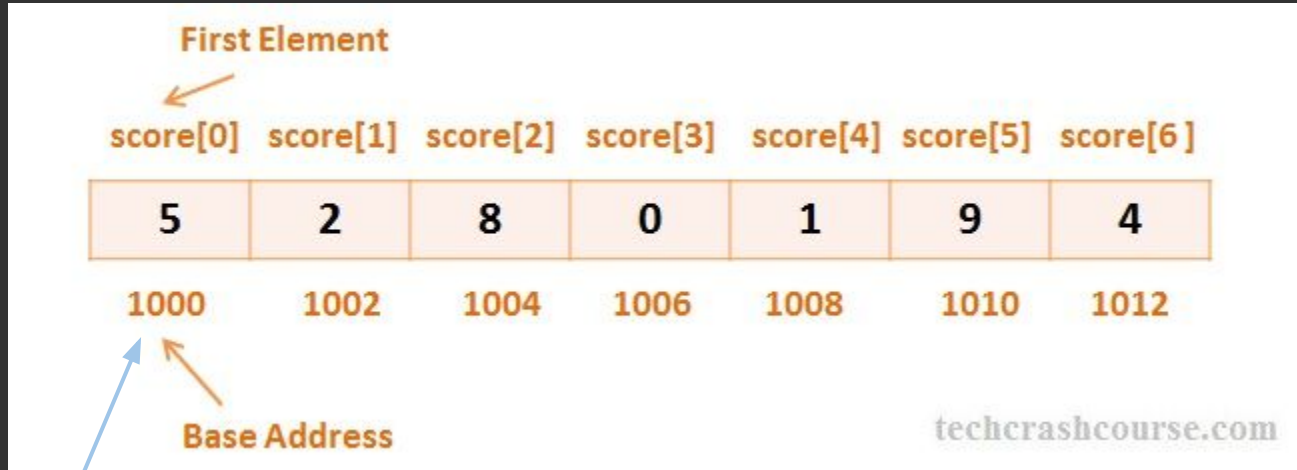
Malloc & Free

THE RELATIONSHIP BETWEEN ARRAYS & POINTERS

We discussed last section the idea of passing by value vs. reference and went into more detail about it today.

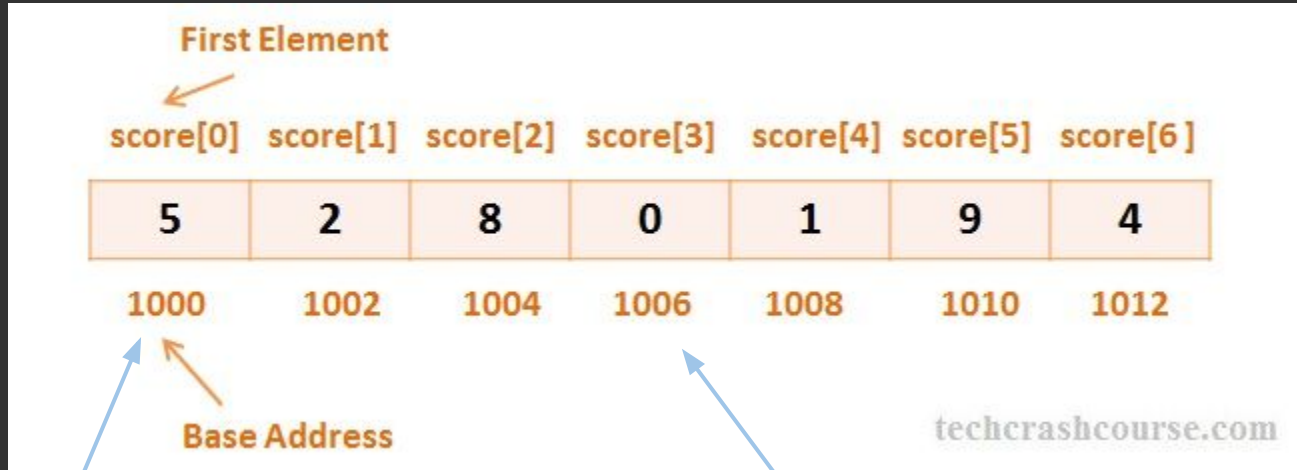
Recall that arrays are passed by *reference*.

THE RELATIONSHIP BETWEEN ARRAYS & POINTERS



The array variable, `score`, is a pointer to the first element in the array

THE RELATIONSHIP BETWEEN ARRAYS & POINTERS



The array variable, `score`, is a pointer to the first element in the array

Requesting a specific index, like `score[3]`, specifies that you want to go to another part of the array in memory

STRINGS ARE ARRAYS AND THUS POINTERS

If a string is really just an array of characters, then it's actually a pointer to where the string starts in memory:

 CLASSROOM

```
char str[6] = "Hello";
```

index	0	1	2	3	4	5
value	H	e	l	l	o	\0
address	1000	1001	1002	1003	1004	1005

dyclassroom.com

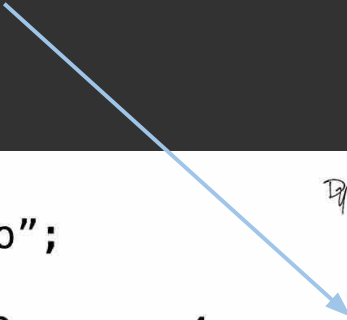
STRINGS ARE ARRAYS AND THUS POINTERS

Why do we null terminate strings in C?

D CLASSROOM

```
char str[6] = "Hello";
```

index	0	1	2	3	4	5
value	H	e	l	l	o	\0
address	1000	1001	1002	1003	1004	1005



dyclassroom.com

STRINGS ARE ARRAYS AND THUS POINTERS

Why do we null terminate strings in C?

Because when we pass a string in C, other parts of our code might not know the length of the string and where to stop “reading” it from memory

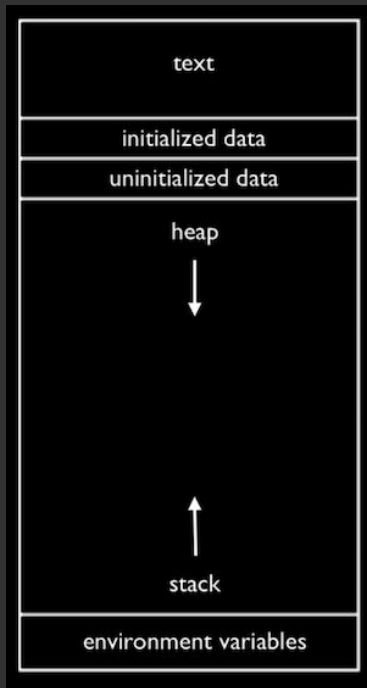
D CLASSROOM

	char str[6] = "Hello";					
index	0	1	2	3	4	5
value	H	e	l	l	o	\0
address	1000	1001	1002	1003	1004	1005

dyclassroom.com

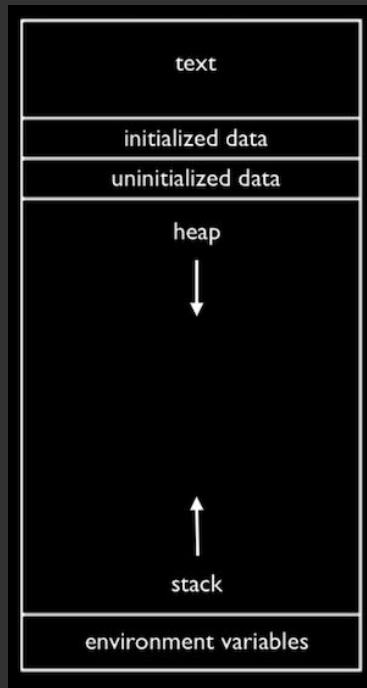
A CRASH COURSE ON MEMORY MANAGEMENT

What purpose do each of these different regions of memory serve?



A CRASH COURSE ON MEMORY MANAGEMENT

What purpose do each of these different regions of memory serve?

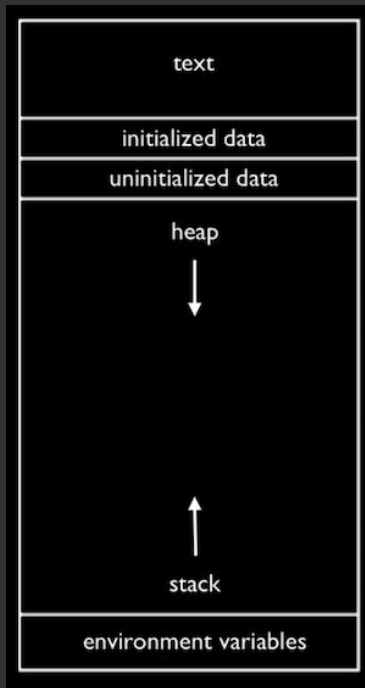


← Raw binary code of program

A CRASH COURSE ON MEMORY MANAGEMENT

What purpose do each of these different regions of memory serve?

Initialized global/static
variables



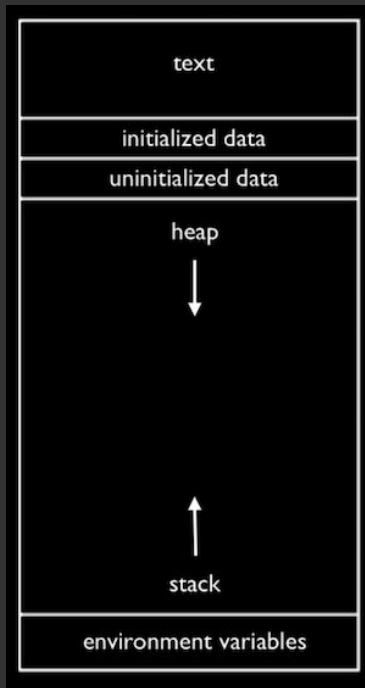
Raw binary code of program



A CRASH COURSE ON MEMORY MANAGEMENT

What purpose do each of these different regions of memory serve?

Initialized global/static
variables



Raw binary code of program

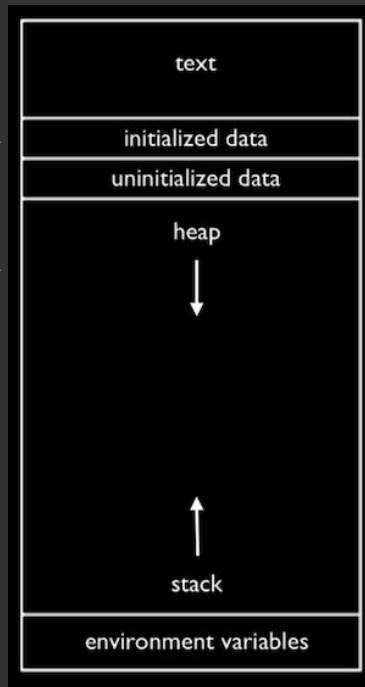
Uninitialized global/static
variables

A CRASH COURSE ON MEMORY MANAGEMENT

What purpose do each of these different regions of memory serve?

Initialized global/static variables

Free memory that can be allocated for our use



Raw binary code of program

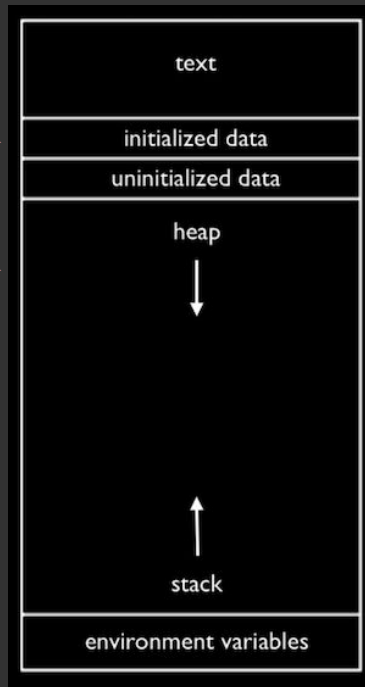
Uninitialized global/static variables

A CRASH COURSE ON MEMORY MANAGEMENT

What purpose do each of these different regions of memory serve?

Initialized global/static variables

Free memory that can be allocated for our use



Raw binary code of program

Uninitialized global/static variables

Memory used by functions in the program

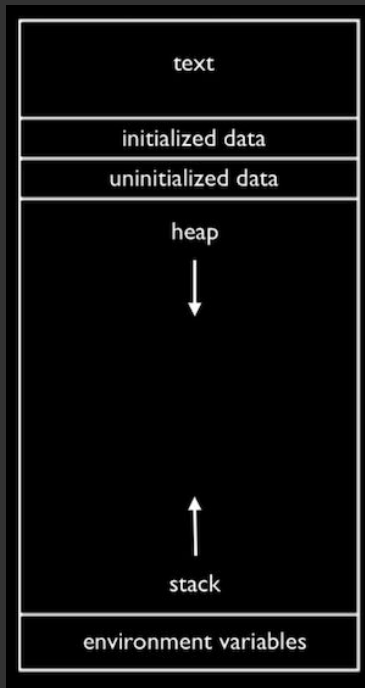
A CRASH COURSE ON MEMORY MANAGEMENT

What purpose do each of these different regions of memory serve?

Initialized global/static variables

Free memory that can be allocated for our use

Command line variables that affect operation of programs



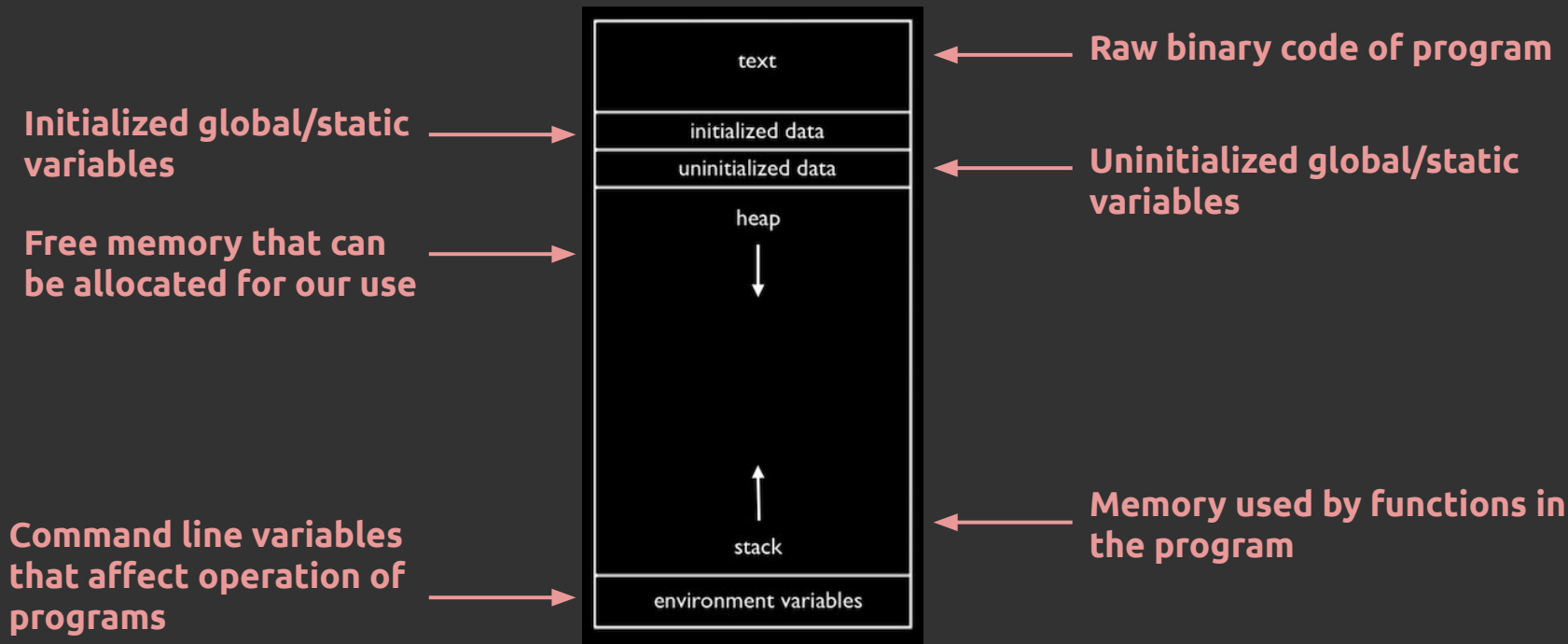
Raw binary code of program

Uninitialized global/static variables

Memory used by functions in the program

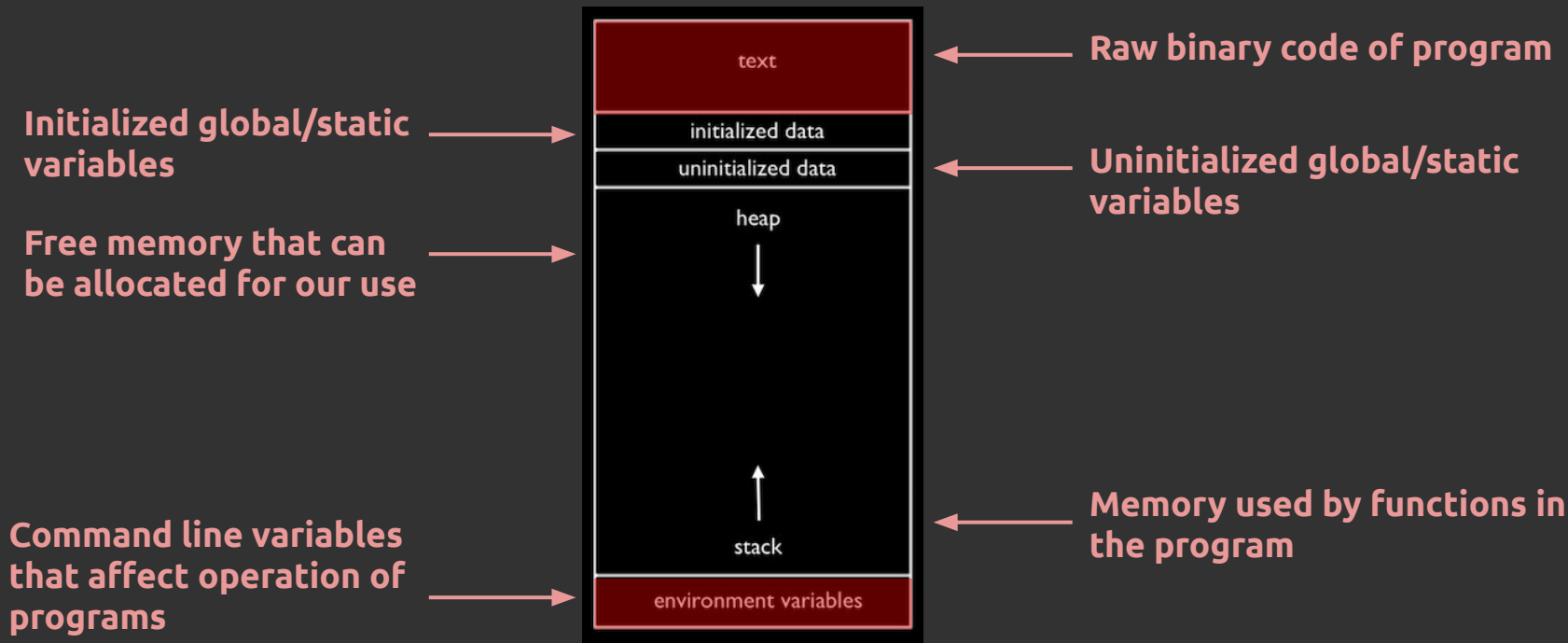
A CRASH COURSE ON MEMORY MANAGEMENT

Which of this memory is read only?



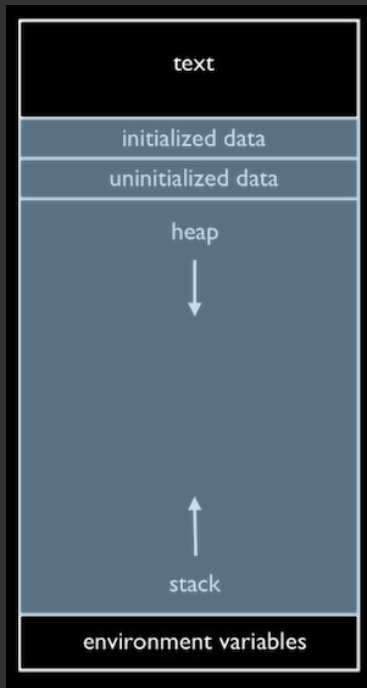
A CRASH COURSE ON MEMORY MANAGEMENT

Which of this memory is read only?



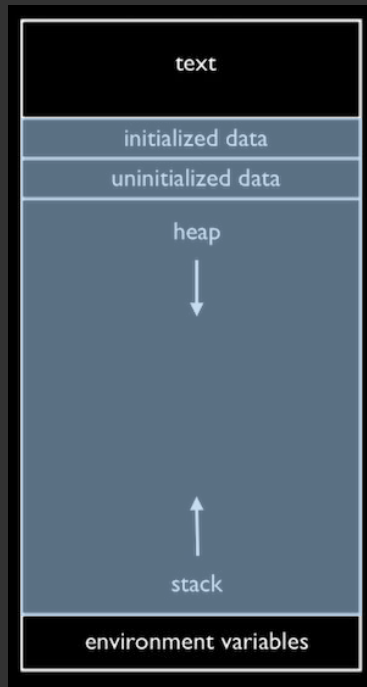
A CRASH COURSE ON MEMORY MANAGEMENT

That means the following memory we can both read AND write from:



A CRASH COURSE ON MEMORY MANAGEMENT

That means the following memory we can both read AND write from:

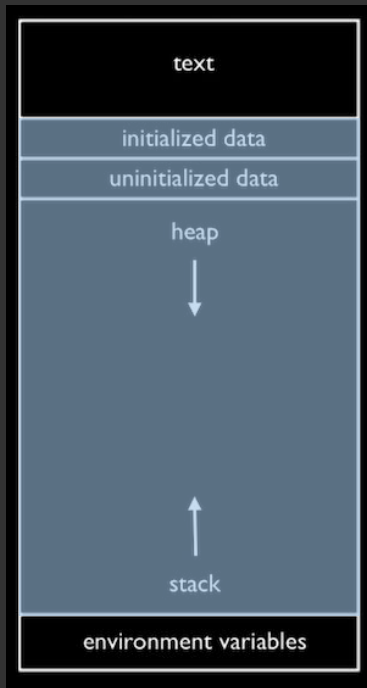


Most of these areas of memory are “compile-time constant.” Once your program is compiled, C knows exactly how much memory to allocate throughout the lifetime of the program.

But this poses a problem...

A CRASH COURSE ON MEMORY MANAGEMENT

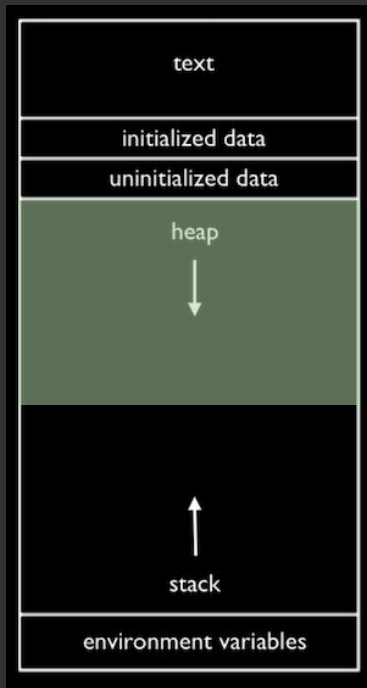
What do you do about the fact that we don't always know how much memory we'll need?



Suppose we have use the `get_string()` function. What if the user enters a paragraph instead of just a word? That's variable in size and not "constant."

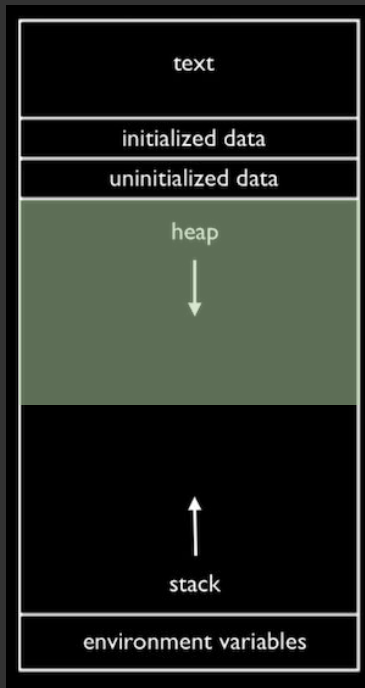
A CRASH COURSE ON MEMORY MANAGEMENT

The solution is to use *dynamically allocated* memory, which occurs in the **heap**.



A CRASH COURSE ON MEMORY MANAGEMENT

The solution is to use *dynamically allocated* memory, which occurs in the **heap**.



We allocate memory from the **heap** as needed and can load things into memory and then free them from memory as we need. It doesn't have to be “compile-time” constant.

A CRASH COURSE ON MEMORY MANAGEMENT

You can utilize `malloc()` to allocate memory from the heap:

```
#include <stdlib.h>

int main(void) {
    int *ptr = malloc(sizeof(int));
}
```

A CRASH COURSE ON MEMORY MANAGEMENT

You can utilize `malloc()` to allocate memory from the heap:

```
#include <stdlib.h>

int main(void) {
    int *ptr = malloc(sizeof(int));
}
```

`malloc()` takes in the number of bytes you want to allocate and returns a memory address. We can use `sizeof()` to determine the size of a data type in C.

A CRASH COURSE ON MEMORY MANAGEMENT

When you're done with that memory, you need to free it:

```
#include <stdlib.h>

int main(void) {
    int *ptr = malloc(sizeof(int));
    free(ptr);
}
```

A CRASH COURSE ON MEMORY MANAGEMENT

When you're done with that memory, you need to free it:

```
#include <stdlib.h>

int main(void) {
    int *ptr = malloc(sizeof(int));
    free(ptr);
}
```

free () takes a pointer to a memory location in the heap and frees it for you.

SOME ERRORS TO WATCH OUT FOR

- **Segmentation Fault** - You've tried to access an "illegal" area of memory or write to a read-only part of memory

SOME ERRORS TO WATCH OUT FOR

- **Segmentation Fault** - You've tried to access an "illegal" area of memory or write to a read-only part of memory
- **Stack Overflow** - Your functions have used up all the space available in the stack so they "overflow" out of it

SOME ERRORS TO WATCH OUT FOR

- **Segmentation Fault** - You've tried to access an "illegal" area of memory or write to a read-only part of memory
- **Stack Overflow** - Your functions have used up all the space available in the stack so they "overflow" out of it
- **Memory Leak** - You forget to free dynamically allocated memory, so you have less of it available as your program runs causing performance/memory issues

Exercise:

String Concatenation

<http://bit.ly/2p2WT0x>

Exercise Solution:

String Concatenation

<http://bit.ly/2noHQxR>

File I/O

FILE I/O

What type of input/output (I/O) have we seen thus far?

FILE I/O

What type of input/output (I/O) have we seen thus far?

We've only seen terminal I/O (a subset of `<stdio.h>`), meaning you can only read/write from the terminal.

FILE I/O

What are the shortfalls of using only terminal I/O?

FILE I/O

What are the shortfalls of using only terminal I/O?

We don't have persistence. Once our program exits, the data is gone!

FILE I/O

What are the shortfalls of using only terminal I/O?

We don't have persistence. Once our program exits, the data is gone!

Luckily, C offers us a data structure called a `FILE` which we can use to do file I/O

FILE I/O

What are the shortfalls of using only terminal I/O?

We don't have persistence. Once our program exits, the data is gone!

Luckily, C offers us a data structure called a `FILE` which we can use to do file I/O

FILE I/O

```
#include <stdio.h>
int main(void)
{
    FILE* in = fopen("input.txt", "r");
    if(in == NULL)
        return 1;
    else
    {
        FILE* out = fopen("output.txt", "w");

        if(out == NULL)
            return 2;

        int c = fgetc(in);
        while(c != EOF)
        {
            fputc(c, out);
            c = fgetc(in);
        }

        fclose(in);
        fclose(out);
    }
}
```

FILE I/O

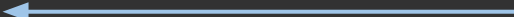
```
#include <stdio.h>
int main(void)
{
    FILE* in = fopen("input.txt", "r");
    if(in == NULL)
        return 1;
    else
    {
        FILE* out = fopen("output.txt", "w");

        if(out == NULL)
            return 2;

        int c = fgetc(in);
        while(c != EOF)
        {
            fputc(c, out);
            c = fgetc(in);
        }

        fclose(in);
        fclose(out);
    }
}
```

Opens the file, "input.txt"
for reading



FILE I/O

```
#include <stdio.h>
int main(void)
{
    FILE* in = fopen("input.txt", "r");
    if(in == NULL)
        return 1;
    else
    {
        FILE* out = fopen("output.txt", "w");

        if(out == NULL)
            return 2;

        int c = fgetc(in);
        while(c != EOF)
        {
            fputc(c, out);
            c = fgetc(in);
        }

        fclose(in);
        fclose(out);
    }
}
```

Opens the file, "input.txt"
for reading

Throws an error if `fopen()` returns `NULL`,
meaning no file could be found

FILE I/O

```
#include <stdio.h>
int main(void)
{
    FILE* in = fopen("input.txt", "r");
    if(in == NULL)
        return 1;
    else
    {
        FILE* out = fopen("output.txt", "w");

        if(out == NULL)
            return 2;

        int c = fgetc(in);
        while(c != EOF)
        {
            fputc(c, out);
            c = fgetc(in);
        }

        fclose(in);
        fclose(out);
    }
}
```

Opens the file, "input.txt" for reading

Throws an error if `fopen()` returns `NULL`, meaning no file could be found

Opens another file, "output.txt," for writing

FILE I/O

```
#include <stdio.h>
int main(void)
{
    FILE* in = fopen("input.txt", "r");
    if(in == NULL)
        return 1;
    else
    {
        FILE* out = fopen("output.txt", "w");
        if(out == NULL)
            return 2;

        int c = fgetc(in);
        while(c != EOF)
        {
            fputc(c, out);
            c = fgetc(in);
        }

        fclose(in);
        fclose(out);
    }
}
```

Opens the file, "input.txt" for reading

Throws an error if `fopen()` returns `NULL`, meaning no file could be found

Opens another file, "output.txt," for writing

Throws an error again if there's an issue (e.g. lack permission to write to disk)

FILE I/O

```
#include <stdio.h>
int main(void)
{
    FILE* in = fopen("input.txt", "r");
    if(in == NULL)
        return 1;
    else
    {
        FILE* out = fopen("output.txt", "w");
        if(out == NULL)
            return 2;

        int c = fgetc(in);
        while(c != EOF)
        {
            fputc(c, out);
            c = fgetc(in);
        }

        fclose(in);
        fclose(out);
    }
}
```

Opens the file, "input.txt" for reading

Throws an error if `fopen()` returns `NULL`, meaning no file could be found

Opens another file, "output.txt," for writing

Throws an error again if there's an issue (e.g. lack permission to write to disk)

Reads a single character from the file and checks if it's "EOF." If it's not EOF, then it runs a while loop which writes a character to the output file and reads the next one from the input file, repeating until it hits EOF.

FILE I/O

```
#include <stdio.h>
int main(void)
{
    FILE* in = fopen("input.txt", "r");
    if(in == NULL)
        return 1;
    else
    {
        FILE* out = fopen("output.txt", "w");

        if(out == NULL)
            return 2;

        int c = fgetc(in);
        while(c != EOF)
        {
            fputc(c, out);
            c = fgetc(in);
        }

        fclose(in);
        fclose(out);
    }
}
```

Opens the file, "input.txt" for reading

Throws an error if `fopen()` returns `NULL`, meaning no file could be found

Opens another file, "output.txt," for writing

Throws an error again if there's an issue (e.g. lack permission to write to disk)

Reads a single character from the file and checks if it's "EOF." If it's not EOF, then it runs a while loop which writes a character to the output file and reads the next one from the input file, repeating until it hits EOF.

Closes the input and output files from read/write access

FILE I/O - Notes

- When you open a file, you're really loading that file into memory and creating a pointer to the first location of it in memory
- You must always close files to free them from memory, just like with pointers
- **EOF** is a "sentinel value" that tells us the file is ended
 - You should use this as the cue to "break" whatever loop you're using to read from a file

Exercise:

Copying a File

<http://bit.ly/2p4nScd>

Exercise Solution:

Copying a File

<http://bit.ly/2nA6bR7>

FILE I/O - Quick Reference

- `fopen()` - creates a file reference (file descriptor)
- `fread()` - reads some amount of data from a file
- `fwrite()` - writes some amount of data to a file
- `fgets()` - reads a single string from a file (typically, a line)
- `fputs()` - writes a single string to a file (typically, a line)
- `fgetc()` - reads a single character from a file
- `fputc()` - writes a single character from a file
- `fseek()` - like rewind and fast forward on YouTube, to navigate around a file
- `ftell()` - like the timer on YouTube, tells you where you are in a file (how many bytes in)
- `fclose()` - closes a file reference, used once done working with the file

PROBLEM SET 4 PREVIEW

Due Sun 10/6 @ 11:59pm

You will need to complete:

- **Filter**
- **Recover**



Section Feedback: <https://tinyurl.com/cs50rwfeedback>