CSE 374 Programming concepts and tools

Winter 2024 Instructor: Alex McKinney

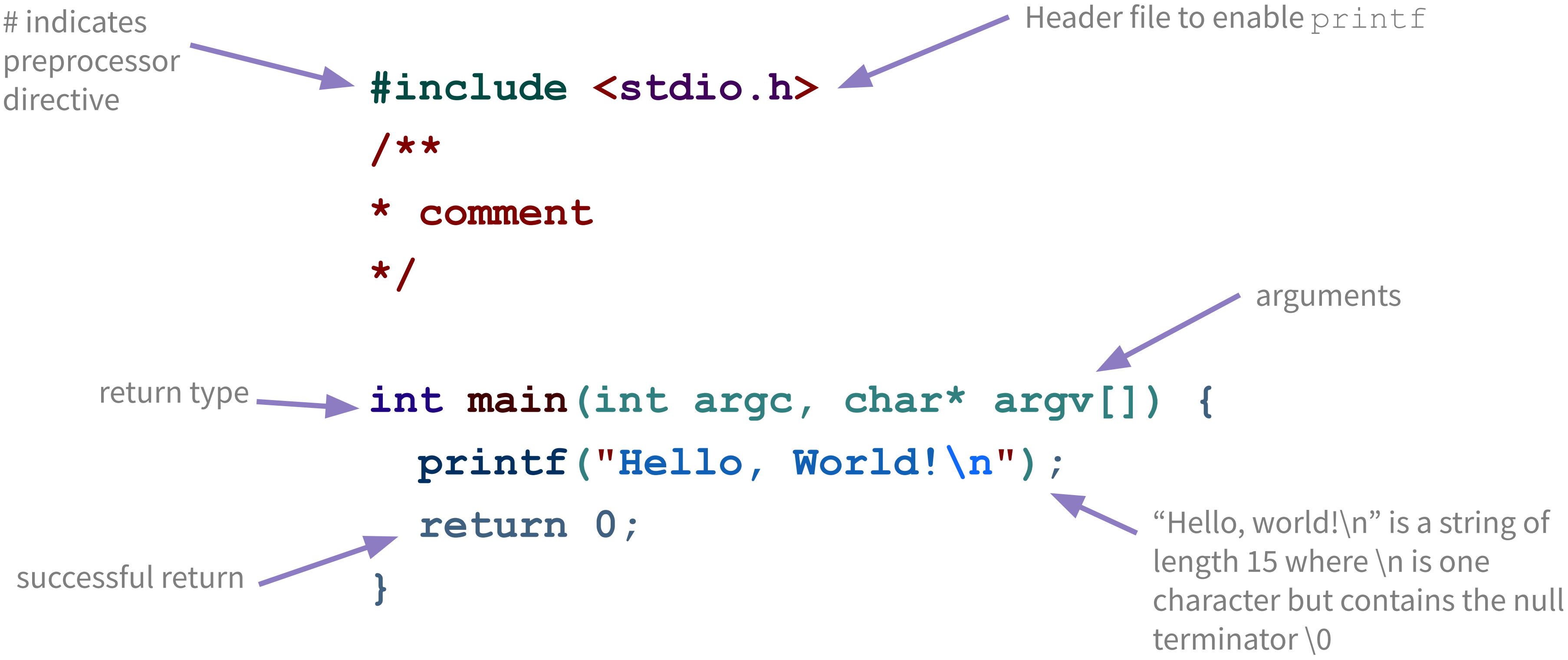
Today

Data representation

Memory

Pointers

Review: Hello World



# indicates preprocessor directive

Header file to enable `printf`

```c
#include <stdio.h>
/**
* comment
*/

int main(int argc, char* argv[]) {
  printf("Hello, World!\n");
  return 0;
}
```

arguments

return type

successful return

“Hello, world!\n” is a string of length 15 where \n is one character but contains the null terminator \0

Data Representations

How Do Computers Store Data?

Large sequences of numbers!

All data is binary - a list of 1's and 0's

- A single digit is called a **bit**
  - The smallest unit of computer memory
- Bits come in groups of 8, called a **byte**
  - Just big enough to store useful data (e.g., a character, or **char** in C)
  - 1 Kilobyte = 1 thousand bytes (KB), 1 Megabyte = 1 million bytes (MB), 1 Gigabyte = 1 trillion bytes (GB)

Binary is a number system, just like how we count 0, 1, 2, 3, ...

Decimal Numbering System

Ten symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

- Called digits
- The base 10 number system

Represent larger numbers as a sequence of digits

- 7061 in base 10
    - $7061 = (7 * 10^3) + (0 * 10^2) + (6 * 10^1) + (1 * 10^0)$

Binary Numbering System

Two symbols: 0, 1

- Called bits
- The base 2 number system
- Convention: start with **0b**

What is **0b**110 in decimal?

- 0b110 = (1 * 2^2) + (1 * 2^1) + (0 * 2^0) = 4 + 2 = 6

Hexadecimal Numbering System

Binary can be very long to write out

How can we make it shorter? Use a bigger base: 16

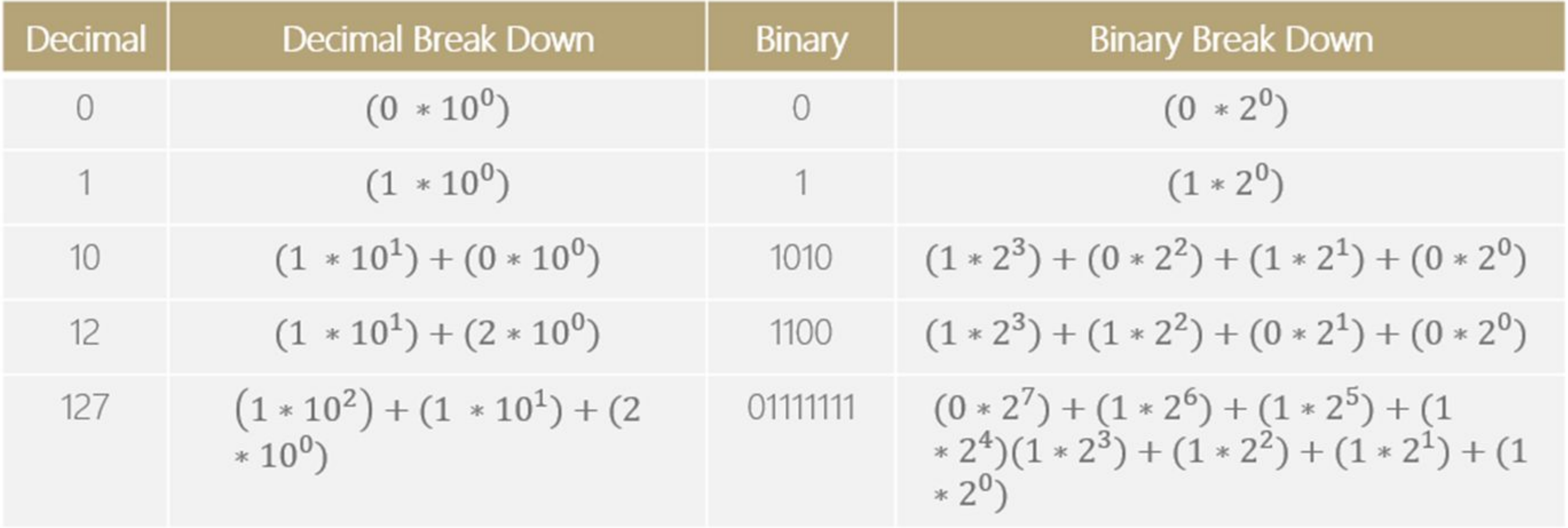
Hexadecimal has 16 symbols: 0-9, A, B, C, D, E, F

- Convention, start with **0x**
- One digit is a nibble
    - Why? Half a byte!

What is 0xF in decimal?

- A = 10, B = 11, C = 12, D = 13, E = 14, F = 15

Binary, Bits and Bytes



Decimal	Decimal Break Down	Binary	Binary Break Down
0	$(0 * 10^0)$	0	$(0 * 2^0)$
1	$(1 * 10^0)$	1	$(1 * 2^0)$
10	$(1 * 10^1) + (0 * 10^0)$	1010	$(1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (0 * 2^0)$
12	$(1 * 10^1) + (2 * 10^0)$	1100	$(1 * 2^3) + (1 * 2^2) + (0 * 2^1) + (0 * 2^0)$
127	$(1 * 10^2) + (1 * 10^1) + (2 * 10^0)$	01111111	$(0 * 2^7) + (1 * 2^6) + (1 * 2^5) + (1 * 2^4)(1 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0)$

Numbers Can Represent Anything

Text files

- "ASCII": uses one byte to represent a single character; Each number corresponds to a different character
  - Ex: 65 = `'A'`, 66 = `'B'`, ...
  - Special ones: 10 = new line (written as `'\n'`), 0 = null (written as `'\0'`)
- "Unicode": similar encoding structure to ASCII but covers a wider range of characters including non-English characters, emojis etc…
  - 好, あ, 😀 (2+ bytes to represent)

Images: represented by a 2D array of “pixels”

- Each pixel is represented by 3 numbers: Red, Blue and Green values 0-255

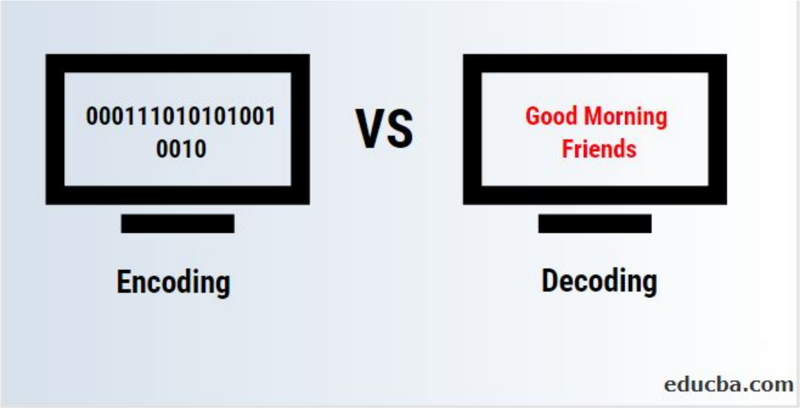
Numbers Are Everywhere

Everything around us, from the **words** we speak to the **colors** we see, can be represented with numbers.

Systems are built to understanding a particular *encoding* of this information (ears and eyes).

Without these systems, the information is meaningless!

By understanding this, we gain insights into how different systems communicate and process information



So What's It Mean?

- *A sequence of bits can have many meanings!*

- Consider the hex sequence 0x4E6F21
  - Common interpretations include:
    - The decimal number 5140257
    - The characters "No!"
    - The background color of this slide
    - The real number 7.203034 × 10^{-39}

- It is up to the program/programmer to decide how to interpret the sequence of bits

Questions?

Memory

Where Do Computers Store Data?

Two ways: files and memory

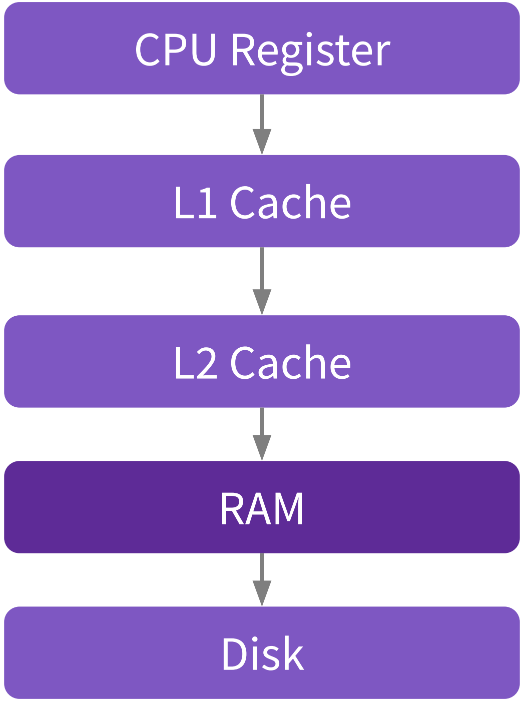
Each file stores a string of data

- Long term - these stay around indefinitely and can be modified by different processes
- This memory is physically stored in the hard drive (AKA disk) or SSD

Each **process** has **memory** to store data

- Short term - when the process ends, that memory goes away
- This memory is physically stored in RAM (main memory)

Quick View on Memory Architecture

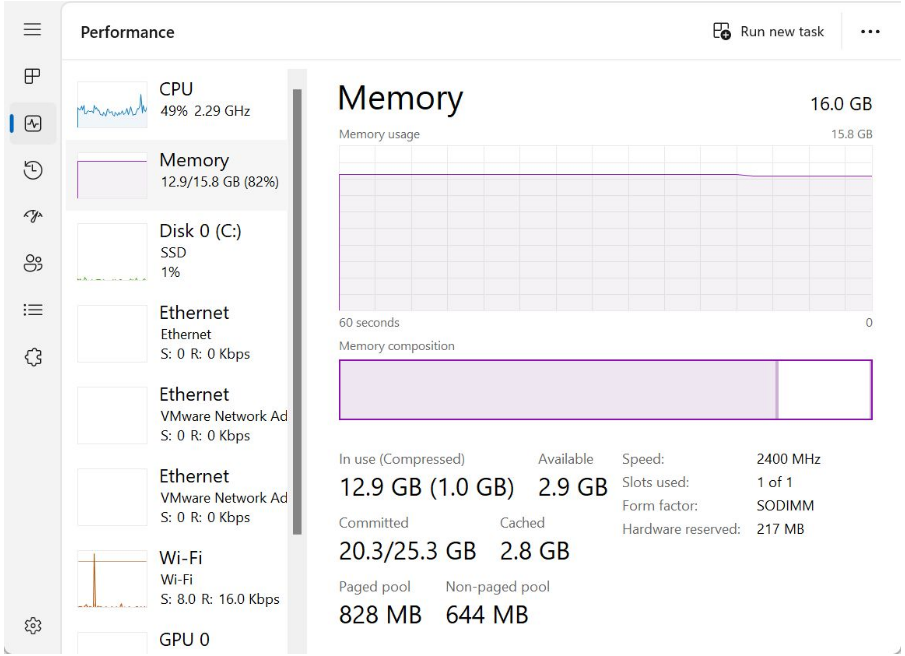


What is this?	Typical Size	Time
Small, high speed storage location in CPU (the brain of the computer)	64 bits	≈free
Extra memory to make accessing it faster	128KB	0.5 ns
Extra memory to make accessing it faster	2MB	7 ns
Working memory, what your program need	8GB	100 ns
Large, longtime storage	1TB	8,000,000 ns

RAM (Random-Access Memory)

RAM is where data gets stored for the programs you run.

RAM goes by a ton of different names: memory, main memory, RAM are all names for this same thing.



CPU
49% 2.29 GHz

Memory
12.9/15.8 GB (82%)

Disk 0 (C:)
SSD
1%

Ethernet
Ethernet
S: 0 R: 0 Kbps

Ethernet
VMware Network Ad
S: 0 R: 0 Kbps

Ethernet
VMware Network Ad
S: 0 R: 0 Kbps

Wi-Fi
Wi-Fi
S: 8.0 R: 16.0 Kbps

GPU 0

Memory 16.0 GB

Memory usage 15.8 GB

60 seconds 0

Memory composition

In use (Compressed) 12.9 GB (1.0 GB)
Available 2.9 GB
Committed 20.3/25.3 GB
Cached 2.8 GB
Paged pool 828 MB
Non-paged pool 644 MB

Speed: 2400 MHz
Slots used: 1 of 1
Form factor: SODIMM
Hardware reserved: 217 MB

RAM can be represented as a huge array

RAM

- addresses, storing stuff at specific locations
- random access

Arrays

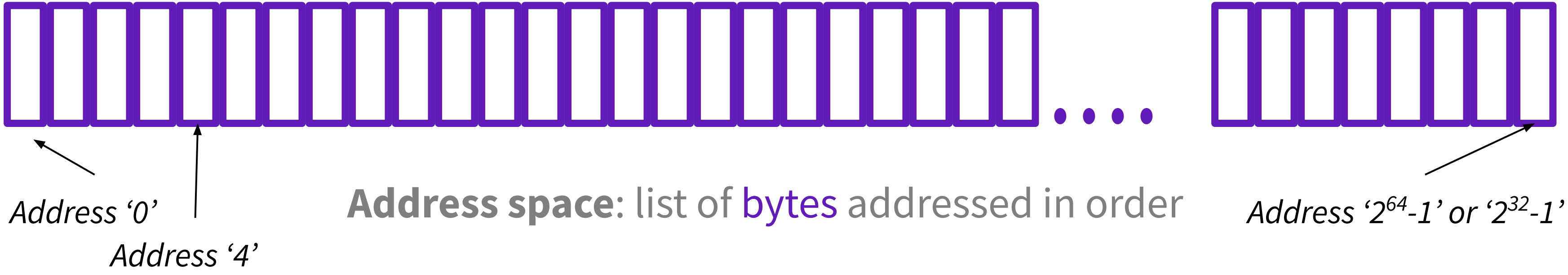
- indices, storing stuff at specific locations
- random access

=

Think of it like a giant array, each element is one byte (8 bits).

Each element of this array has an index, called an “address”.

Working memory



Address space: list of bytes addressed in order

Programs are said to have access to this 2^{64} byte space on a 64-bit system

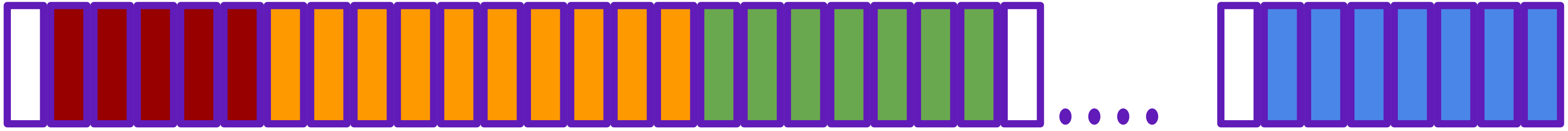
- '64 bit' system refers to needing 64 bits to index the space (18.4 Exabytes!)

Location in array is the address of a byte

Programs keep track of addresses of each of their pieces of memory

Accessing unused address causes a "segmentation fault"

Working memory



code *globals* *heap ->* ***Program address space*** *<- stack*

As a program executes it interacts with the computer's working memory:

- **Code**: space for the code compiled instructions
- **Globals**: space for global variables, static constants, string literals, etc.
- **Heap**: holds dynamically allocated variables (`new` or `malloc` variables)
- **Stack**: holds current instructions, each function in a frame

The heap and stack grow dynamically. Meet in the middle ?= "out of memory" error

Review: The Stack

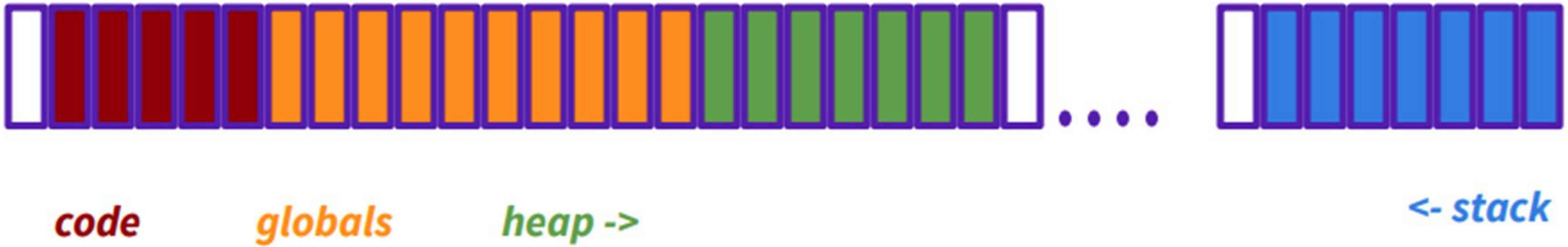
The "stack" is an area of **memory** that holds the local variables

Similar idea to the stack data structure (LIFO), but for local variables

When we call a function, it **allocates** memory on the stack for those local variables

- Size of memory depends on the data type
- If a recursion goes wrong… **Stack overflow**!

When that function returns, it **deallocates** its space on the stack

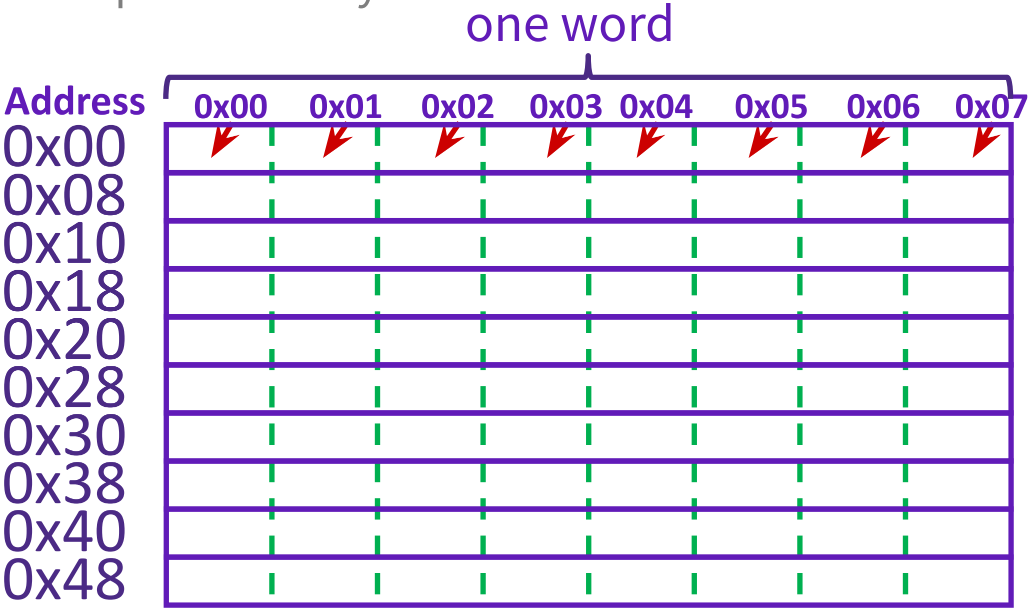


A Picture of Memory (64-bit view)

We can choose to view memory as a series of word-sized chunks of data instead.

A “64-bit (8-byte) word-aligned” view of memory:

- In this type of picture, each row is composed of 8 bytes
- Each cell is a byte
- An aligned, 64-bit chunk of data will fit on one row



Questions?

Poll Question (PollEv.com/cs374)

Where is address `0x1A`? Let's count!

Hint: Hexadecimal has 16 symbols: 0-9, A, B, C, D, E, F

Address	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00								
0x08								
0x10								
0x18								
0x20								
0x28								
0x30								
0x38								
0x40								
0x48								

Where is address 0x1A?

✓ 0

Row: 0x18, Column: 0x01

0%

Row: 0x18, Column: 0x02

0%

Row: 0x18, Column: 0x03

0%

Row: 0x18, Column: 0x04

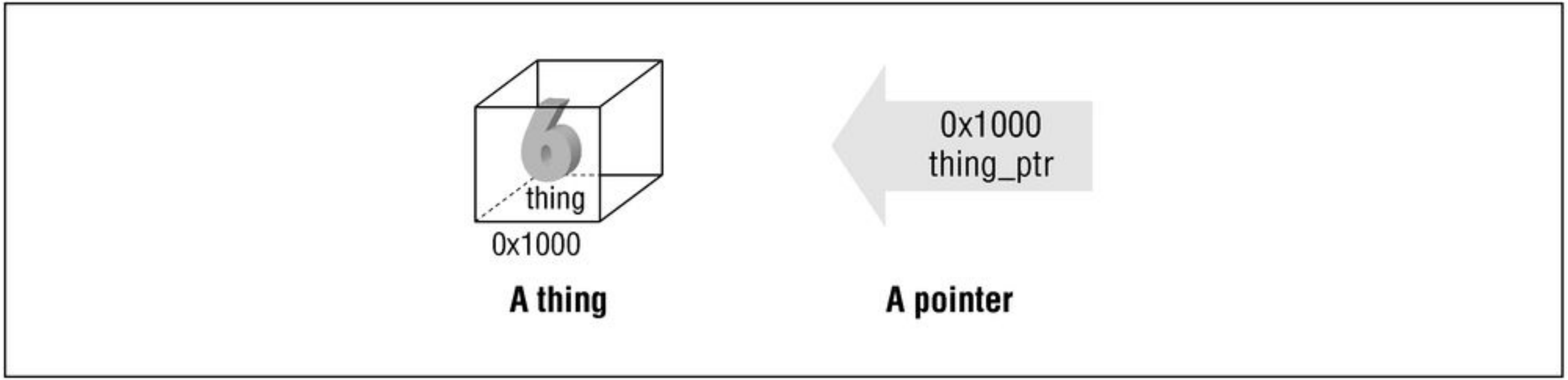
0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Pointers

Quick view of Pointer

The address of `thing` is 0x1000. Addresses are automatically assigned by the C compiler to every variable. Our pointer (`thing_ptr`) points to the variable thing. Pointers are also called address variables because they contain the addresses of other variables.



Addresses and Pointers

An address refers to a location in memory.

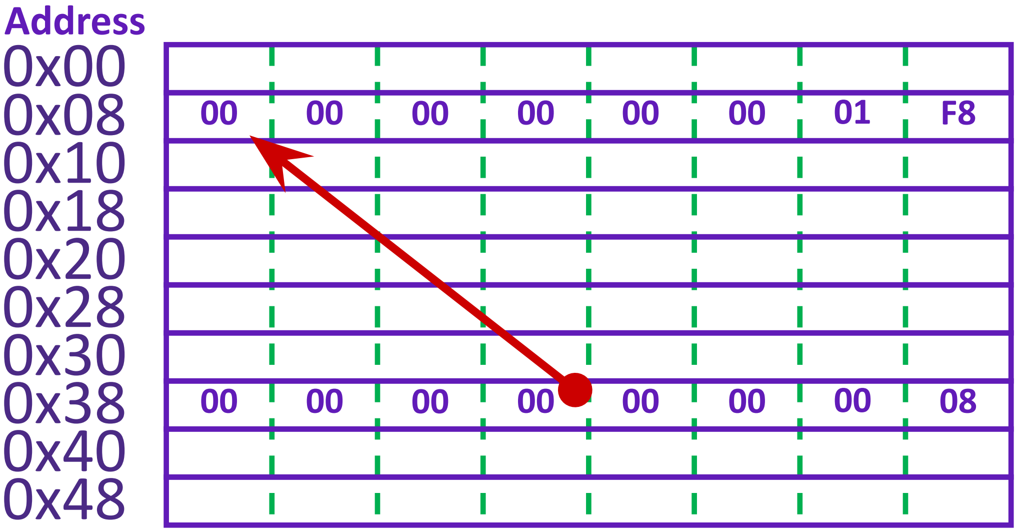
A **pointer** is a data object that holds an address.

- Address can point to *any* data, because they simply point to any space in memory
- Like a "contact", object that stores someone's phone number, doesn't store the actual person

Value `504` stored at address `0x08`

- 504 = 0x1F8 = 0x 00 ... 00 01 F8

Pointer stored at `0x38` points to address `0x08`



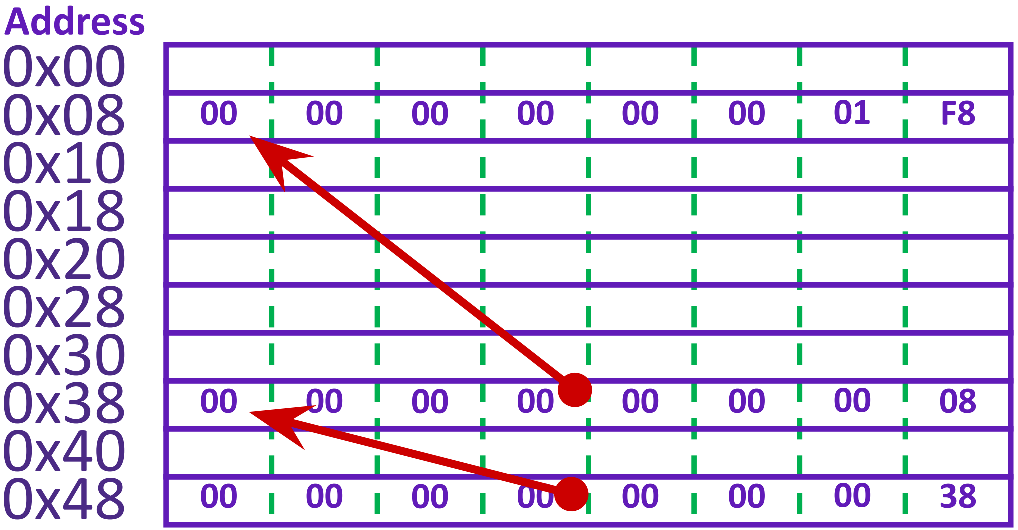
Address								
0x00								
0x08	00	00	00	00	00	00	01	F8
0x10								
0x18								
0x20								
0x28								
0x30								
0x38	00	00	00	00	00	00	00	08
0x40								
0x48								

Addresses and Pointers

Pointers can point to other pointers! <follow down the rabbit hole>

Pointer stored at `0x48` points to address `0x38`

- Pointer to a pointer!
  - = “double pointer”



Address								
0x00								
0x08	00	00	00	00	00	00	01	F8
0x10								
0x18								
0x20								
0x28								
0x30								
0x38	00	00	00	00	00	00	00	08
0x40								
0x48	00	00	00	00	00	00	00	38

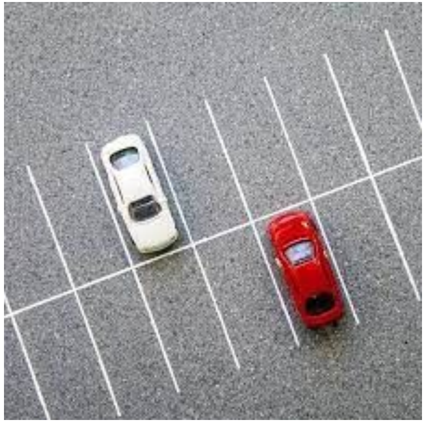
Analogy: Parking Lot

Imagine your computer’s memory as a parking lot full of cars.

Each parking spot has a unique address.

The pointer *points* at an individual parking spot (e.g. B5, spot 12)

“Where’s your car parked?” -> *Point* your finger at the parking spot



Pointer and Address Syntax in C

```
int* ptr;           // a variable of type “pointer to int” without assignment
int x = 123;        // an int variable called “x” that stores “123”
ptr = &x;           // store the address of “x” in “ptr”
```

Adding a ***** (star) after the type means "**pointer to type**"

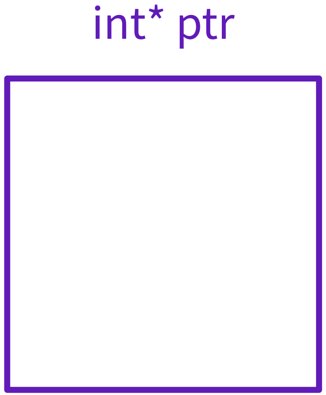
- Similar in java if you add `[]` after type you declare an array of that type
- `int*` means "pointer to int"
  - `int *ptr;` also works! Programmer preference

Placing an **&** (ampersand) before a variable means "**address of variable**"

- Placing an `&` before a variable name will give you the address in memory of that variable
  - `&y` means "address of y"

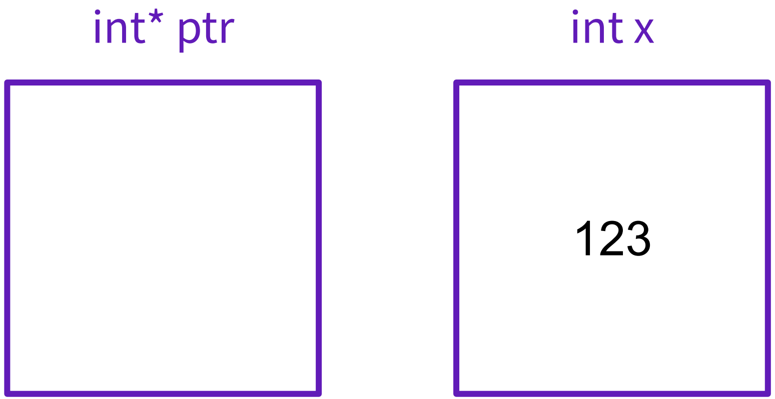
Pointer and Address Syntax in C

```
int* ptr;            // a variable of type “pointer to int” without assignment
int x = 123;         // an int variable called “x” that stores “123”
ptr = &x;            // store the address of “x” in “ptr”
```



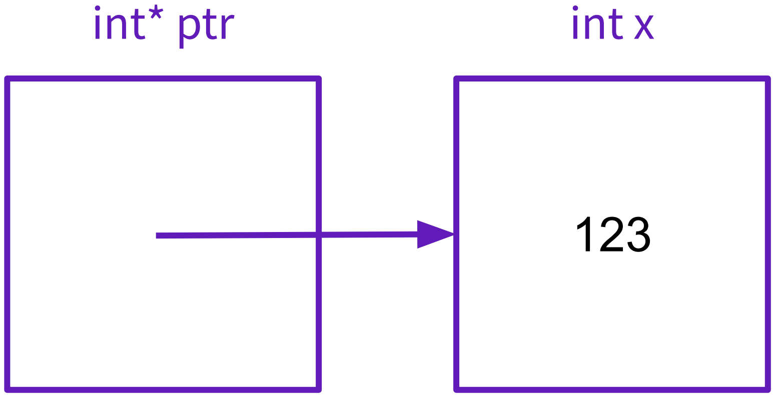
Pointer and Address Syntax in C

```
int* ptr;          // a variable of type “pointer to int” without assignment
int x = 123;       // an int variable called “x” that stores “123”
ptr = &x;          // store the address of “x” in “ptr”
```



Pointer and Address Syntax in C

```
int* ptr;            // a variable of type “pointer to int” without assignment
int x = 123;         // an int variable called “x” that stores “123”
ptr = &x;            // store the address of “x” in “ptr”
```



Dereferencing Pointers

```
int x = 123;
int* ptr = &x;
*ptr = 456;
printf("New value of y: %d\n", *ptr);
```

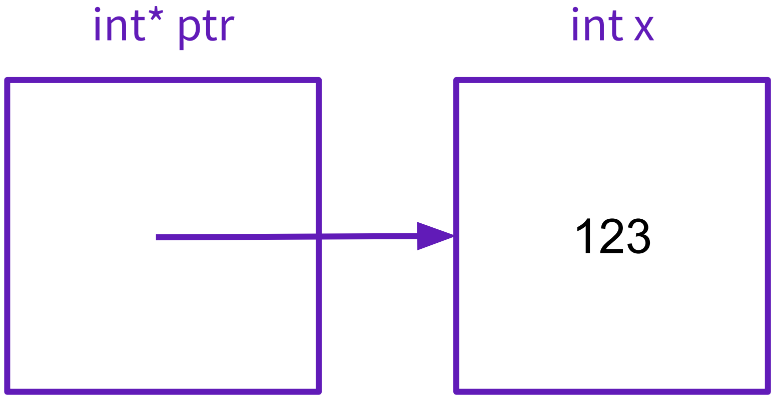
Placing a `*` before a pointer means **dereferences** the pointer

- Means “follow this pointer” to the actual data
- Can be used for read and writing
- `*ptr = <data>` will update the data stored at the address the pointer is referring to, ie “write to memory”
- `<var> = *ptr` will read the data stored at the address indicated by the pointer

Accessing unused addresses causes a “segmentation fault”

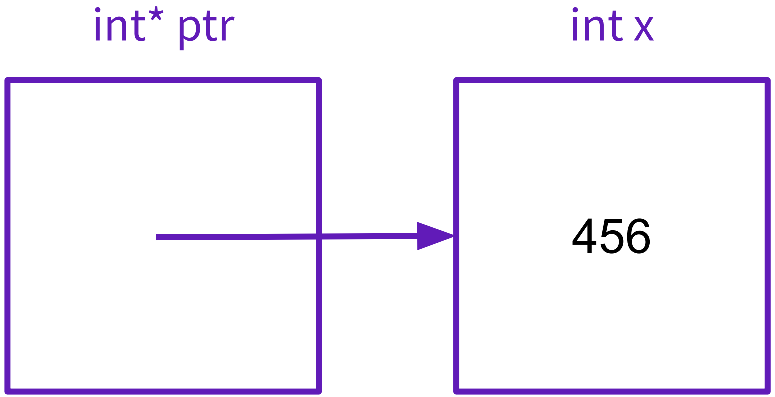
Dereferencing Pointers

```
int x = 123;
int* ptr = &x;
*ptr = 456;
printf("New value of y: %d\n", *ptr);
```



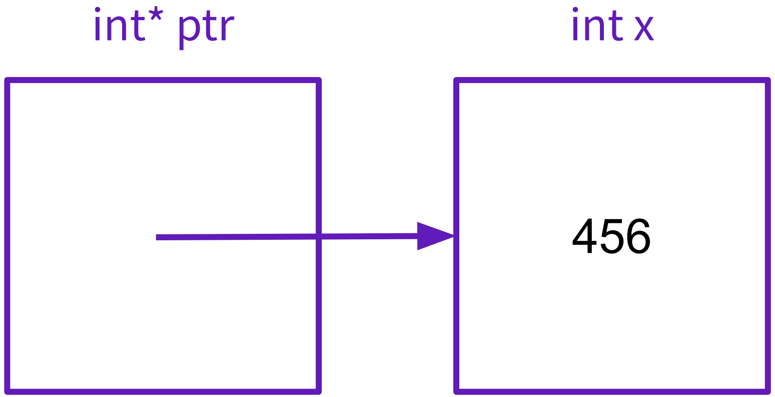
Dereferencing Pointers

```
int x = 123;
int* ptr = &x;
*ptr = 456;
printf("New value of y: %d\n", *ptr);
```



Dereferencing Pointers

```
int x = 123;
int* ptr = &x;
*ptr = 456;
printf("New value of y: %d\n", *ptr);
```



`NULL` in C

Java allows objects to be null

Similarly, you can assign pointers to be `NULL` in C

`NULL` is literally just the number 0

What happens if you dereference a `NULL` pointer?

- i.e. `*ptr = x;` or `x = *ptr;` where `ptr == NULL`
- In Java, this causes a NullPointerException
- In C, your program immediately crashes

This is another case that causes a "segmentation fault".

Pointers Recap

Storing in memory an address to another location in memory

```
int x = 4;      // Variable called ‘x’ of type ‘int’ given value ‘4’

int* xPtr = &x; // Variable called ‘xPtr’ of type ‘int pointer’ given value ‘location of x’

int xCopy = *xPtr;

    // Variable called ‘xCopy’ of type ‘int’ given value ‘value found at address xPtr’ (read)

*xPtr = 123;  // Assigning the value '123' to the ‘value found at address xPtr’  (write)

int* noPtr = NULL; // variable called ‘noPtr; of type ‘int pointer’ given value of ‘null’
```

Using Pointers as Output Parameters

C pointers offer a powerful mechanism for **returning multiple values** from a function.

- Pass the memory address of variables to be modified as function arguments.

```c
void initialize(int *a, char *c);
int main(void) {
    int a = 0;
    char c;                // c is undefined (don’t do this)
    initialize(&a, &c);    // a = 10, c = 'A'
}
void initialize(int *a, char *c) {
    *a = 10;
    *c = 'A';
}
```

Questions?

Poll Question (PollEv.com/cs374)

Which is the correct syntax?

A.

```
int x = 123;
int ptr = &x;
printf("x is %d\n", *ptr);
```

B.

```
int x = 123;
int* ptr = *x;
printf("x is %d\n", &ptr);
```

C.

```
int x = 123;
int* ptr = &x;
printf("x is %d\n", *ptr);
```

D.

```
int x = 123;
int& ptr = &x;
printf("x is %d\n", *ptr);
```

What is the correct syntax?

✓ 0

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

hello.c Revisit

Strings, Arrays, and Pointers

Review: Strings in C

All three of these are equivalent ways of defining a string in C:

```
char s1[] = {'c', 's', 'e', '\0'};

char s2[] = "cse";

char* s3 = "cse";     // won’t be a mutable “string” because it’s stored as a literal
```

There are no “string” in C, only arrays of characters

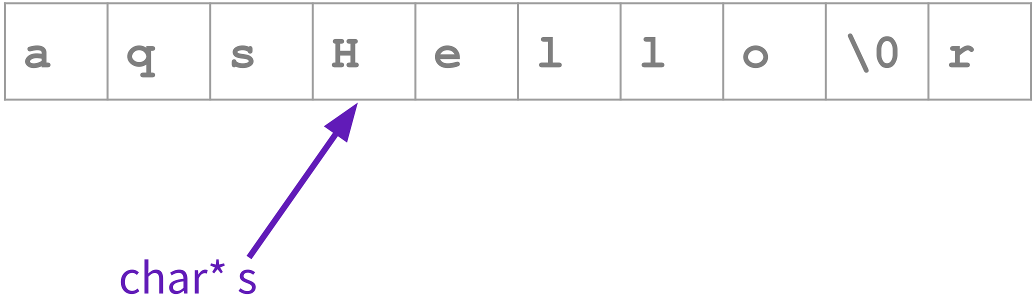
- “null terminated (`\0`) array of characters”

`char*` is another way to refer to strings in C

Strings as Pointers

So are strings just a pointer to one character?

- Yes and no, they point to the **first** character at the **beginning** of the string
- C assumes that there is an array of characters after that, **ending** in a null terminator (`'\0'`)



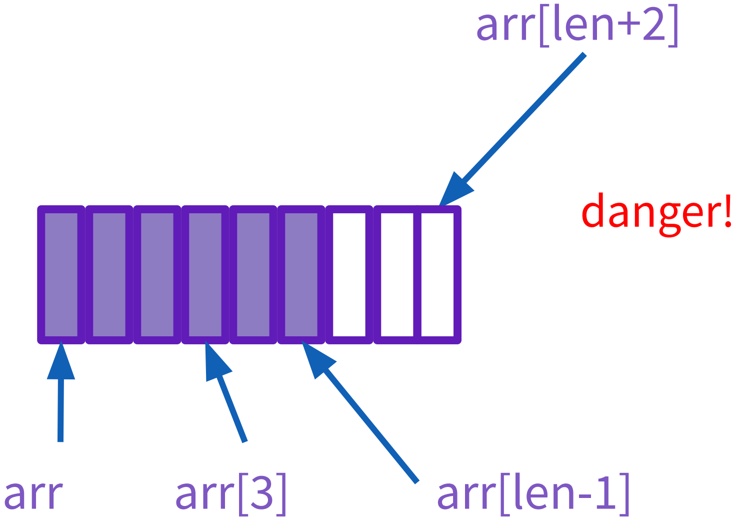
a	q	s	H	e	l	l	o	\0	r

char* s

Array Syntax with Pointers

You can use the bracket notation to index pointers

```
char arr[] = "cse";
char* ptr = arr;
char letter_e = ptr[2];
```



The bracket syntax is just another way of saying this: `letter_e = *(ptr + 2);`

2 * sizeof(char) bytes

- "**Pointer arithmetic**" works with other types like int (4 bytes), long (8 bytes)

Pointer arithmetic

Pointers can be incremented or decremented by a specific number of elements based on their data type.

```
int numbers[] = {10, 20, 30, 40, 50};

int* ptr = numbers;  // Let’s say the first element (10) is at address 0x60

int value = *(ptr + 3);  // Access the value at address 0x6C
```

↓

Address: 0x60 + (3 * sizeof(int)) = 0x60 + (3 * 4) = 0x60 + 12 = 0x6C

```
printf("The value: %d\n", value);  // Output: The value: 40
```

Arrays vs. Pointers

Pointers can either point to a single variable or an array

C uses this property of pointers to pass arrays into functions as pointers

- C cannot actually pass arrays into functions
- Any function parameters which use array syntax are actually just pointers!

```
void foo(int myNumbers[], int len);
```

```
void foo(int* myNumbers, int len);
```

This means that you cannot know how long an array is!

- It is common to pass in an int representing the size of the array

Revisiting `argv`

```
int main(int argc, char* argv[]) { … }
```

```
int main(int argc, char** argv) { … }
```

These are equivalent

- "Array of strings" vs. "Pointer to (an array) of strings"

How do we know that a `char**` points to an array of strings and not just one string?

- Read the documentation for that function 🤷
- In the case of `argv`, it is an array

Experimenting with C

Always best to practice compiling and running the code yourself

But you can quickly check if something compiles/errors without logging onto calgary

Godbolt.org

- Runs instantly, also outputs assembly (not required for 374)

Onlinegdb.com

- Allows you to specify command line arguments as well as take input from `stdin`

EX8 due Friday & HW3 is due on Sunday!

EX8 is due before the beginning of the next lecture

- Link available on the website: https://courses.cs.washington.edu/courses/cse374/24wi/exercises/

HW3 due Sunday 11.59pm!

- START EARLY!
- Instructions on course website: https://courses.cs.washington.edu/courses/cse374/24wi/homeworks/hw3/