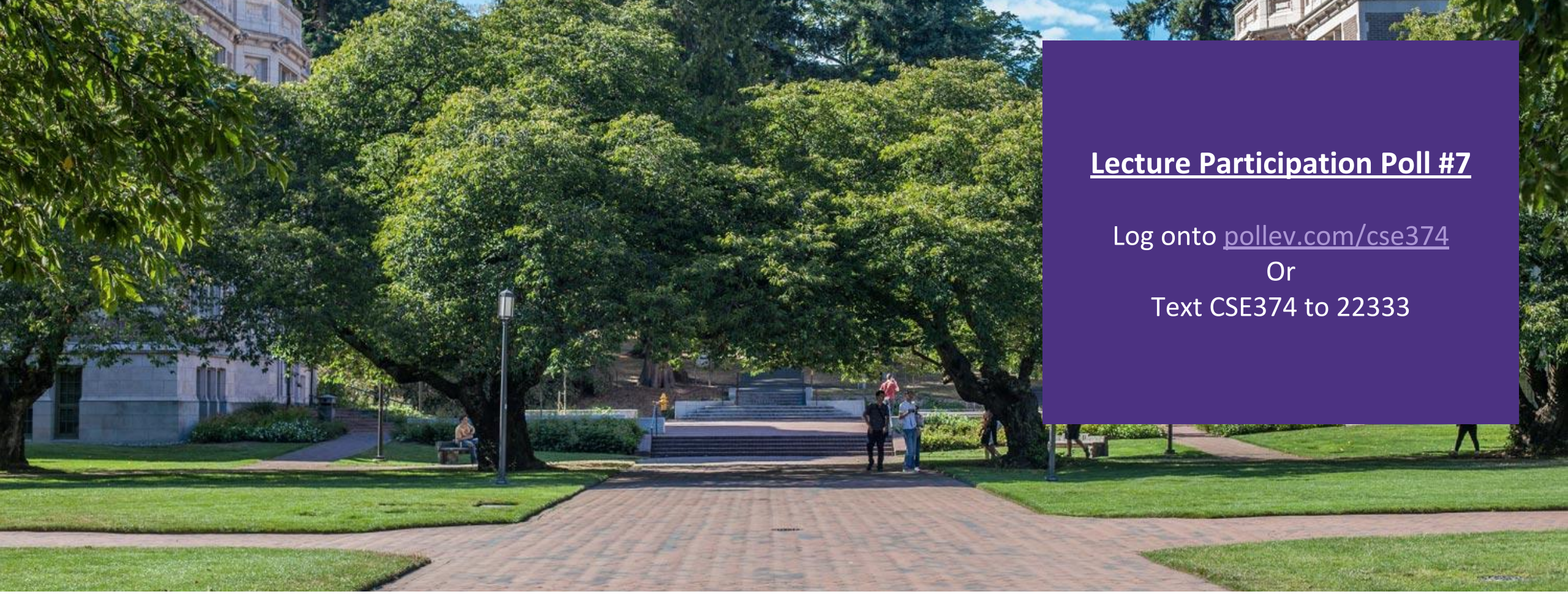


Lecture Participation Poll #7

Log onto pollev.com/cse374
Or
Text CSE374 to 22333

Lecture 7: Intro to C Programming

CSE 374: Intermediate Programming Concepts and Tools

Administrivia

Assignments

- Hw1 turn in live
- EX 4 did not release
- Poll Everywhere is being mean
- Review assignment coming- find groups!
- Use tickets on discord

Sorry Kasey is behind on messages- will get back to you today!

Regex

- Regular expressions (regex) are a set of rules for matching patterns in text
  - Used across programming languages and math
  - Different applications might have slightly different rules (yeah, it's frustrating…)
- Regex patterns can include characters, anchors and modifiers
  - Characters = the literal characters you are trying to match
  - Anchors – set the position in the line where a pattern may be found
    - ^ anchor to front
    - $ anchor to end
  - Modifiers – modify the range of text pattern can match
    - * matches any number of characters
    - [set of chars]
- Regex basics, let P be our pattern and S be a string to match
  - P can be a single character (ex: a) to match S of the same single character
  - P_1P_2 matches S if $S=S_1S_2$ where $P_1 = S_1$ and $P_2 = S_2$
  - $P_1|P_2$ matches S if P1 or P2 matches S
- grep –e finds using regex
  - By default grep matches against .*p.*

https://regex101.com/ https://regexcrossword.com/ https://regexone.com/

Regex special characters

`\` – escape following character

`.` – matches any single character at least once
- `c.t` matches `{cat, cut, cota}`

`|` – or, enables multiple patterns to match against
- `a|b` matches `{a}` or `{b}`

`*` – matches 0 or more of the previous pattern (greedy match)
- `a*` matches `{, a, aa, aaa, …}`

`?` – matches 0 or 1 of the previous pattern
- `a?` matches `{, a}`

`+` – matches one or more of previous pattern
- `a+` matches `{a, aa, aaa, …}`

`{n}` – matches exactly n repetitions of the preceding
- `a{3}` matches `{aaa}`

`()` – groups patterns for order of operations
- `(abc)` matches `{abc, 1abc2, 123abc}`

`[]` – contains literals to be matched, single or range
- `[a-b]` matches all lowercase letters

`^` – anchors to beginning of line
- `^//` matches lines that start with `//`

`$` – anchors to end of line
- `;$` matches lines that end with `;`

`\d` – matches one digit
- `\d+` matches `{1, 2, 3, 4, …}`

`\s` – matches whitespace character
- `\s` matches `{' ', \t, etc…}`

Useful patterns

- [a-zA-Z] - matches all English letters
- [0-9]* - matches list of numbers
- (abc)* - match any number of "abc"s
- (foo | bar) – matches either "foo" or "bar"
- ^\d+$ - whole numbers (\d stands in for digit, +one or more digits) (regexpal)
- ^\d*\.\d+$ - numbers with decimals (regexpal)
- ^\b\d{3}[-.]?\d{3}[-.]?\d{4}\b$ - phone number (regexpal)
- [^([a-zA-Z0-9._%-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}*$] – emails (regexpal)

https://digitalfortress.tech/tricks/top-15-commonly-used-regex/

Regex Practice

- Regex for date in format YYYY-MM-DD
- Year - [12]\d{3} – start with 1 or 2 followed by 3 digits
- Month - (0[1-9]|1[0-2]) – 0 followed by a digit 1-9 OR 1 followed by a digit 0-2
- Day – (0[1-9]|[12]\d|3[01]) – 0 followed by digit 1-9 OR 1 or 2 followed by any digit OR 3 followed by 0 or 1
- Final - ([12]\d{3}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01]))

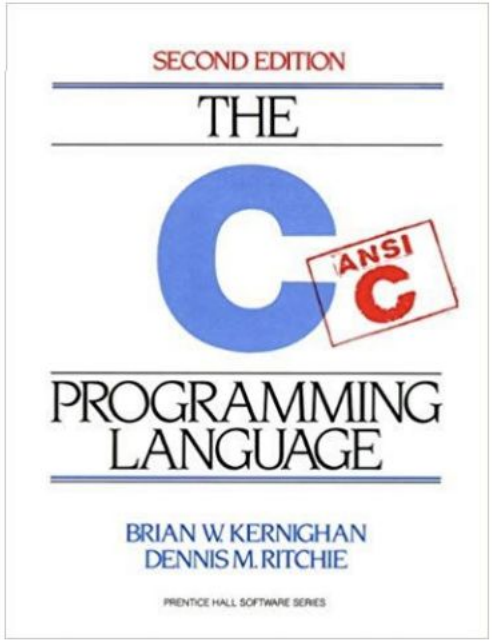
(regexpal)

Meet C

- Invented to rewrite the Unix OS, successor to B
- A "low level" language gives the developer the ability to work directly with memory and processes
  - Low level means it sits closer to assembly, the language the CPU uses
  - Java is a "high level" language, compiles to bytecode, has a garbage collector that manages memory for you
- Useful for software that requires low-level fOS interaction
  - Robotics, mobile, high performance software, drivers
  - Compact language, human readable but few features compared to Java
- Ancestor of most modern languages
  - Java, C++, C#
  - Much syntax is shared

C reference books

The standard reference. Available on Kindle and in the UW library.

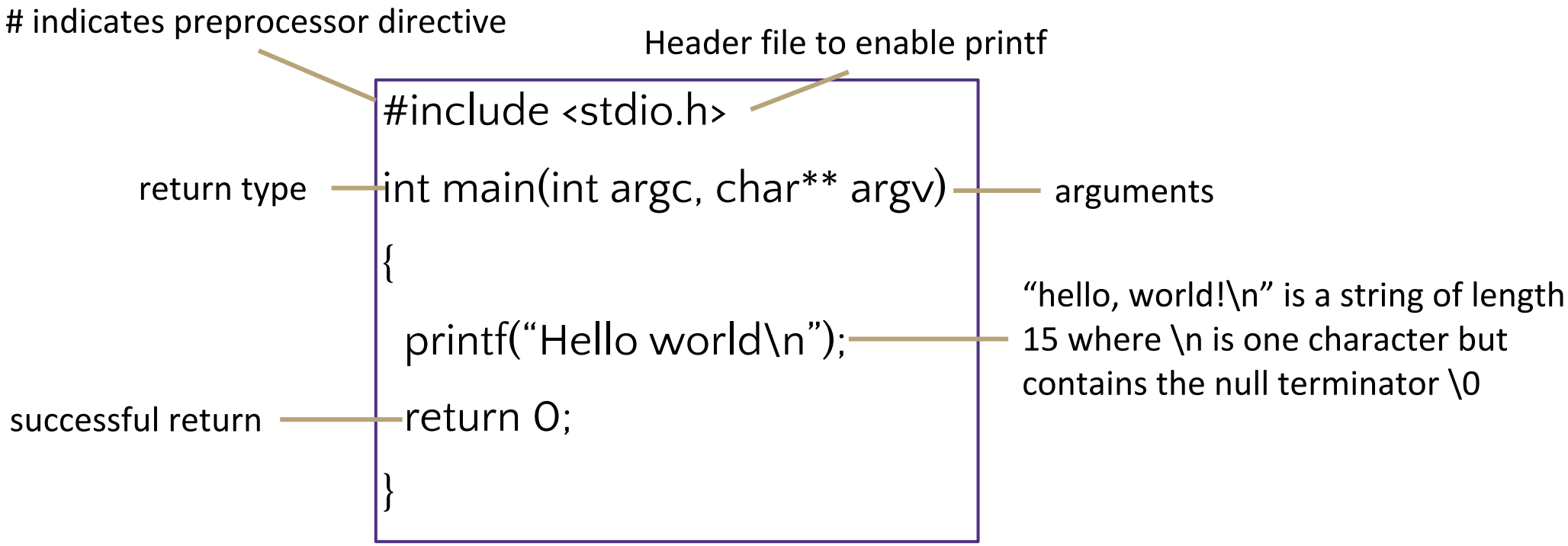


http://cslibrary.stanford.edu/101/EssentialC.pdf http://www.cplusplus.com/

GCC

- GCC is the C compiler we will use
  - Translates C into assembly code
    - Java compiler takes java code and turns it into Java bytecode (when you install JDK you teach your computer to understand javanite code)
    - Assembly is the language of your CPU
- gcc [options] -o outputName file1.c file2.c
- gcc --version
- Can provide warnings for program crashes or failures, but don't trust it much
- Before compiling your code, gcc runs the C preprocessor on it
  - Removes comments
  - Handles preprocessor directives starting with #
- Options
  - -g enables debugging
  - -Wall checks for all warnings
  - -std=c11 uses the 2011 C standard, what we will use for this class

C Hello World



Save in file "hello.c"
Compile with command gcc hello.c
    creates executable a.out
Compile with command gcc –o hello.exe hello.c
    creates executable hello.exe
Run ./hello.exe

Hello World in C

#include

- Provides access to code in another file, similar to Java import statements
- `#include<somefile.h>` will insert code in somefile.h into your C file
  - .h files are called "header files"
  - `#include <foo.h> // standard libraries`
    - searches for foo.h in "system include" directories
  - `#include "foo.h" // developer files`
    - searches current directory, lets coder break project into smaller files (java does this automatically)
- Executed by preprocessor
  - Pulls in code before it is compiled
  - Includes work recursively, pulls in includes from headers that were directly included
- stdio.h provides foundational set of input and output functions
  - printf, stdout

http://www.cplusplus.com/reference/cstdio/

Functions

- C programs are broken into functions
  - Named portion of code that can be referenced by code elsewhere
  - Similar to methods and classes in java

```
returnType functionName (type param1, …, type paramN) {
    // statements
}
```

Declaration – specifies the function name, return type and parameters

```
//declaration
int square (int n);
```

-The function header ending in ;
-Similar to interfaces in Java
-exist so you can call a function before you fully define it

Definition – declaration plus the code to run

```
//definition
int square (int n) {
    return n * n;
}
```

-You will get a Linker-error if an item is used but not defined (java equivalent of "symbol not found")

Main function

```
void main(int argc, char** argv) {
   printf("hello, %s\n", argv[1]);
}
```

- argv is the array of inputs from the command line
  - Tokenized representation of the command line that invoked your program
- argv[0] is the name of the program being run
- argc stores the number of arguments ($#)+1
- Like bash!

Main is the first function your program executes once it starts

Expect a return of 0 for successful execution or -1 for failure

Variables

- C variable types: int, char, double, arrays (details)
  - No Booleans, use int values of nonZero=true and 0=false instead,
    - WARNING: opposite of bash

`<type> <name> = <value>` – Left side evaluates to locations = right side evaluates to values

```
int x = 1; // stores value 1 at location labeled x
char c = 'a'; // stores value a at location labeled c
double d = 2.5; // stores value 2.5 at location labeled d
int* xPtr = &x; // stores value of location x at location xPtr

x = 2; // stores value 2 at location x
*xPtr = 3; //stores value 3 at location xPtr
```

Much more on * and & tomorrow!

Global vs Local Variables

- Variables defined inside a function are local to that function
  - Can only be used by function within which they are defined
  - May have multiple instances (recursion)
  - Only "lives" until end of function
    - Space on stack allocated when reached, deallocated after block
- Variables defined outside functions are global and can be used anywhere in the file and by any function
  - Will only ever be a single instance of a global variable
  - Lives until end of program
    - Space on stack allocated before main, deallocated after main
  - Should be avoided if possible for encapsulation

example.c

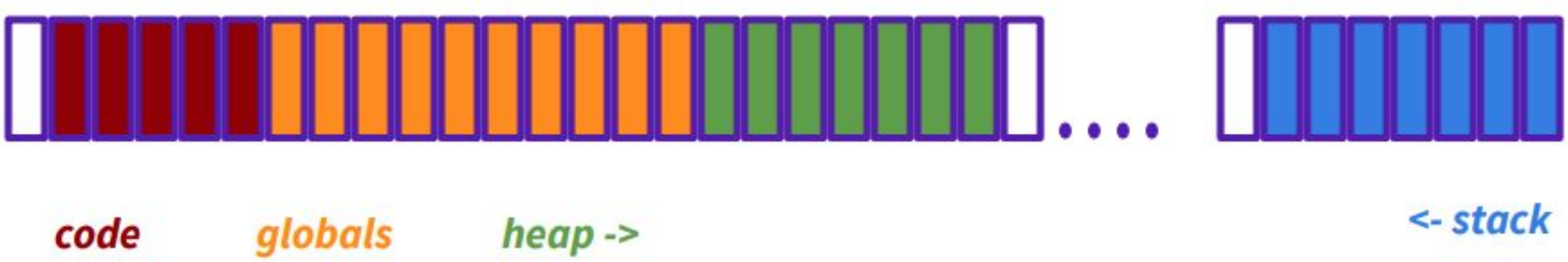
global

```
int result = 0;
int sumTo(int max) {
  if (max == 1) return 1;
  result = max + sumTo(max - 1);
  return result;
}
```

local

The Stack

- An area of local memory set aside to hold local variables
- Functions like the stack data structure – first in first out
- When we call a function it **allocates** memory on the stack for all local variables
  - Size of memory depends on datatype
- When the function returns the memory for the local variables is **deallocated**
- Java has been doing something similar in the background for you all along- garbage collector



Strings in C

```
char s1[] = {'c', 's', 'e', '\0'};
char s2[] = "cse";
char* s3 = "cse";
```

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09
a	q	s	h	e	l	l	o	\0	r

All are equivalent ways to define a string in C

There are no "strings" in C, only arrays of characters

- "null terminated array of characters"

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

- Technically is a pointer to the first char in the series of chars for the string

Strings cannot be concatenated in C

```
printf("hello, " + myName + "\n"); // will not work
```

Printf – print format function

- Produces string literals to stdout based on given string with format tags
  - Format tags are stand ins for where something should be inserted into the string literal
  - %s – string with null termination, %d – int, %f – float
  - Number of format tags should match number of arguments
    - Format tags will be replaced with arguments in given order
- Defined in `stdio.h`
- printf("format string %s", stringVariable);
  - Replaces %s with variable given
  - `printf("hello, %s\n", myName);`

https://en.wikipedia.org/wiki/Printf_format_string

Demo: echo.c

Example: echo.c

```
#include <studio.h>
#include <stdlib.h>
#define EXIT_SUCCESS = 0;
int main (int argc, char** argv) {
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]);
    }
    printf("\n");
    return EXIT_SUCCESS;
}
```

Arrays in C

- `datatype name[length]`
- Contiguous block of memory
- C doesn't pass arrays around like ints, but rather passes the references to the array
  - Just like Java
- Each item in array has an address based off of initial start item which is at 0
- Arrays must be declared with a known length (so compiler can allocate space)
  - This size is not stored like in Java, you have to save length as a separate variable you pass around
- No default values, arrays will hold whatever was in that spot before you declared it so accessing those addresses will cause errors

```
char arr[] = "cse";
char* ptr = arr;
char letter_e = ptr[2]; // synonymous to *(ptr + 2)
int myArr[10];
```

C style

- C curly brace style
  - Each curly brace is on its own line, not at the end of an instruction
- C naming conventions
  - Constants are ALL_CAPS with underscores for spaces
- C white space conventions
  - One declaration per line

Anatomy of a C program

```
// includes for functions & types
#include <stuff.h>
// symbolic constants
#define TRUE 1
#define FALSE 0
//global variables (if any)
Int x = 1;
// Function declarations
Void do_this(char, int)
Function definitions
Void do_this(char s, int m)
{
  //statements
}
<main method at end of file?>
```