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
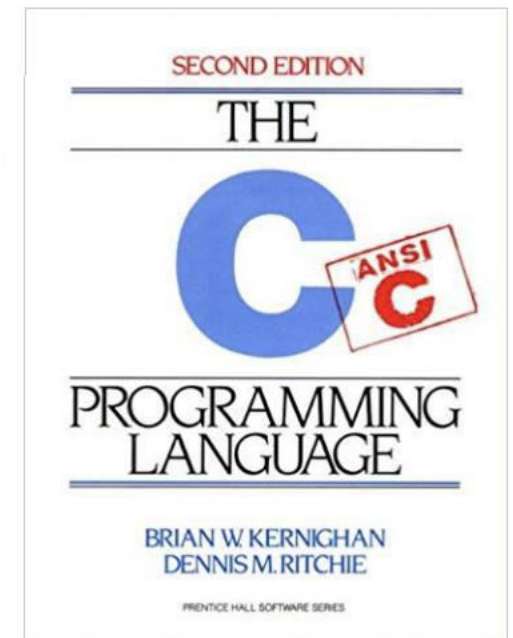
Administrivia

- Bash sample code + demo videos added to course calendar on website
- Schedule 1:1 time with Kasey via Calendly
  - https://calendly.com/kasey-champion/1on1
  - Link posted on OH page of course website
- HW1 turn in fixed
  - grading scripts misbehaving
    - due date will be "flexible"
  - HW1 Individual Assignment open on gradescope
- HW2 posting later today, more Bash

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
  - Still used for:
    - Embedded programming
    - Systems programming
    - High-performance code
    - GPU programming

- 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.

SECOND EDITION

THE

C
ANSI C

PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

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

C vs Java

C

- low level
- user responsible for memory
- "functions"
- No classes – NOT object oriented
- compiled
- conditional controls
- modern syntax (human readable)
- small standard library

Java

- high level
- memory managed (garbage collection)
- "methods"
- classes define objects
- compiled
- conditional controls
- modern syntax (human readable)
- large standard library, HUGE extended libraries

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
  - 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 #

- `gcc <options> -o <output exe> <c file to compile> <c file to compile>`
  - `gcc -o hello.exe hello.c`

- 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

# indicates preprocessor directive

Header file to enable printf

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

int main(int argc, char** argv)
{

    printf("Hello world\n");

    return 0;

}
```

return type

arguments

successful return

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

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
- other useful standard libraries
  - stdlib
  - math
  - assert

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

Arguments to Main

char =   datatype

char* = pointer to a place in memory that stores a char

char** = pointer to a place in memory that stores pointers to chars

int argc = number of pointers stored in argv

char** argv = "array" of pointers to program input arguments from command line

- Access values with argv[index] Ex: argv[1]
  - argv[0] = program name, just like bash
- Array of chars = String
- Arrays do not store their length as a field (not an object), must be passed in argc

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

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

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

code   globals   heap ->   <- stack

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

```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;
}
```