Spring 2013
CSE2421 Systems1
Introduction to Low-Level Programming and Computer Organization
Kitty Reeves
TWRF 8:00-8:55am
CSE2421 = Alice in Wonderland

A look into the rabbit hole...
Introduction

- Website
  - Personal: [http://www.cse.ohio-state.edu/~reeves](http://www.cse.ohio-state.edu/~reeves)

- Syllabus
  - Course Description
  - Pre-requisites
  - Objectives
  - Textbook – Safari online an option for C programming
  - Grading Policy
  - Academic Misconduct

- Day2 lab day in BE0310
  - Lab locations

- Discussion Group - Piazza

- Grade Posting – by code given
Why C?

- Age has its advantages
  - C has been around for ~40 years
  - C is a great language for expressing common ideas in programming in a way that most people are comfortable with (procedural language)
- Portable, versatile, simple, straight-forward
- Reasonably close to the machine
  - Low-level access to memory
  - Provide language constructs that map efficiently to machine instructions
  - Requires minimal run-time support

*** C has the best combination of speed, low memory use, low-level access to the hardware, and popularity ***

OK, really... why C?

- Is there a size problem?
  - Size is part of the issue, but so is speed.
  - C is lightweight and fast.

- I love garbage
  - No automatic garbage collection
  - Fun memory leaks to debug

- Wonderfully, yet irritatingly, obedient
  - you type something incorrectly, and it has a way of compiling fine and just doing something you don't expect at run-time.

- Power...
  - To optimize
  - Write drivers
  - Get a job in micro processing technology
  - Write my own OS
Welcome to C

- Going from Java to C is like going from an automatic transmission to a stick shift
  - Lower level: much more is left for you to do
  - Unsafe: you can set your computer on fire
  - C standard library: is much smaller
  - Similar syntax: can both help and confuse
  - Not object oriented: paradigm shift
Happiness is... programming in C

- C is procedural, not object-oriented
- C is fully compiled (to machine code), not to bytecode
- C allows direct manipulation of memory via pointers
- C does not have garbage collection
- Many of the basic language constructs in C act in similar ways to the way they work in Java
- C has many important, yet subtle, details
C vs Java/C++

Programming language rankings
Speed - Portability - Object Orientation

- Pointers to memory
- Platform dependent types
- Programmer allocated memory
- Declare variables at start of block

- References to objects
- Types have well defined sizes
- Automatic garbage collection
- Declare variable anywhere
C does not...

- C is a procedural language, and **does not support objects**. That is, it does not support entities which contain data and model behavior. We can botch together something of the sort in C, but it is still far from what we would ever consider a class.

- C **does not support Encapsulation**. While you may set up a group of data types to only be accessible through a structure collection, it still can be accessed from anywhere, by anything, as long as the collection exists within a scope seen by what is trying to access it.
What is C?
- C is a programming language originally created for developing the Unix operating system. It is a low-level and powerful language, but it lacks many modern and useful constructs.
- C is a simple programming language with few keywords and a relatively simple to understand syntax.
- C is also useless (whaaaaaaaaaaaaattt???). C itself has no input/output commands, doesn't have support for strings as a fundamental (atomic) data type. No useful math functions built in.
- Because C is useless by itself, it requires the use of libraries. This increases the complexity of C. The issue of standard libraries is resolved through the use of ANSI libraries and other methods.

Three traditional aspects of the C language:
- Characters are promoted to integers before being used for any type of arithmetic.
- The default character type, either signed or unsigned, is not specified by the Standard so that the implementer can choose whichever is most efficient for a particular machine.
- There is no range checking on array subscripts.
Controlling complexity is the essence of computer programming.

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.
"Unix for Beginners" (1979)

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?
"The Elements of Programming Style", 2nd edition, chapter 2

Do what you think is interesting, do something that you think is fun and worthwhile, because otherwise you won't do it well anyway.
*An Interview with Brian Kernighan* from the PC Report Romania[1]

Advice to students: Leap in and try things. If you succeed, you can have enormous influence. If you fail, you have still learned something, and your next attempt is sure to be better for it.

Advice to graduates: Do something you really enjoy doing. If it isn’t fun to get up in the morning and do your job or your school program, you’re in the wrong field.
"Leap In and Try Things: Interview with Brian Kernighan"[2] from *Harmony at Work blog*[3]
Your first C program

#include <stdio.h>
void main(void)
{
    printf("Hello, world!\n");
}

Reminder ➔ There are a lot of different ways to solve the same problem.
TO-DO: Experiment with leaving out parts of the program, to see what error messages you get.

Which one is best?

#include <stdio.h>
int main(void) {
    printf("Hello, world!\n");
    return 0;
}

#include <stdio.h>
int main(void) {
    printf("Hello, world!\n");
    getchar();
    return 0;
}

#include <stdio.h>
void main(void) {
    printf("Hello,  ");
    printf("world!\n");
}
C compilation model... hello.c to hello

Type in program using an editor of your choice (file.c); plain text

.c + .h = .i which is the “ultimate source code”? i.e. # includes expanded and #defines replaced

.i → .s which is assembler source code

.s → .o which is an object file; fragments of machine code with unresolved symbols i.e. some addresses not yet known (vars/subrs).

.o + library links → a.out (default name); resolves symbols, generates an executable.

%gcc -o hello hello.c

%hello
Functions, types and macros of the standard library are declared in standard headers:

```c
<assert.h> <float.h> <math.h> <stdarg.h> <stdlib.h>
<ctype.h> <limits.h> <setjmp.h> <stddef.h> <string.h>
<errno.h> <locale.h> <signal.h> <stdio.h> <time.h>
```

A header can be accessed by
- `#include <header>`
- Notice, these do not end with a semi-colon

Headers can be included in any order and any number of times

Must be included outside of any external declaration or definition; and before any use of anything it declares

Need not be a source file
Every full C program begins inside a function called "main". A function is simply a collection of commands that does "something". The main function is always called when the program first executes. From main, we can call other functions, whether they be written by us or by others or use built-in language features.

Java programmers may recognize the main() method but note that it is not embedded within a class. C does not have classes. All methods (simply known as functions) are written at file scope.

The main() method in Java has the prototype ‘main(String[] args)’ which provides the program with an array of strings containing the command-line parameters. In C, an array does not know its own length so an extra parameter (argc) is present to indicate the number of entries in the argv array.
Your first C program (cont)

What is going on?

- `#include <stdio.h>` - Tells the compiler to include this header file for compilation. To access the standard functions that comes with your compiler, you need to include a header with the `#include` directive.
  - What is a header file? They contain prototypes and other compiler/pre-processor directives. Prototypes are basic abstract function definitions. More on these later...
  - Some common header files are `stdio.h`, `stdlib.h`, `unistd.h`, `math.h`.

- `main()` - This is a function, in particular the main block.
  - `{ }` - These curly braces are equivalent to stating "block begin" and "block end". The code in between is called a “block”

- `printf()` - Ah... the actual print statement. Thankfully we have the header file `stdio.h`! But what does it do? How is it defined?

- `return 0` - What's this? Every function returns a value...
  - you should always explicitly declare the return type on the function. If you don’t, it defaults to a type integer anyway.
Your first C program (fyi)

- The return 0 statement. Seems like we are trying to give something back, and it is an integer. Maybe if we modified our main function definition: int main() Ok, now we are saying that our main function will be returning an integer! So remember, you should always explicitly declare the return type on the function! **If you don’t, it defaults to a type integer anyway.**

- Something is still a little fishy... I thought that 0 implied false (which it does)... so isn't it returning that an int signifying a bad result? Thankfully there is a simple solution to this. Let's add #include <stdlib.h> to our includes. Let's change our return statement to return EXIT_SUCCESS;. Now it makes sense!

- Let's take a look at printf. Hmm... I wonder what the prototype for printf is. (btw, what’s a prototype?) Utilizing the man pages we see that printf is: int printf(const char *format, ...); printf returns an int. The man pages say that **printf returns the number of characters printed.** Now you wonder, who cares? Why should you care about this? It is good programming practice to **ALWAYS** check for return values. It will not only make your program more readable, but in the end it will make your programs less error prone. But in this particular case, we don't really need it. So we cast the function's return to (void). fprintf, fflush, and exit are the only functions where you should do this. More on this later when we get to I/O. For now, let's just void the return value.

- What about **documentation**? We should probably doc some of our code so that other people can understand what we are doing. Comments in the C89 standard are noted by: /* */. The comment begins with /* and ends with */.

  - **Comments cannot be nested!**
  - // is a single line comment i.e. from the location of // to the end of the line is considered a comment
Your first C program...

New and Improved?

#include <stdio.h>
#include <stdlib.h>

/* Main Function
 * Author(s) and percent of work:
 * Purpose: Controls program, prints Hello, World!
 * Input: None
 * Output: Returns Exit Status
 */

int main(int argc, char **argv) {
    printf("Hello, world!\n");
    return EXIT_SUCCESS;
}

Much better! The KEY POINT of this whole introduction is to show you the fundamental difference between correctness and understandability. All of the sample codes produce the exact same output in "Hello, world!" However, only the latter example shows better readability in the code leading to code that is understandable. All codes will have bugs. If you sacrifice code readability with reduced (or no) comments and cryptic lines, the burden is shifted and magnified when your code needs to be maintained.