CS 214 / 2020-01-25
==========

```
// you generally want these in every source file
// include brings in declarations from libraries
// the '#' is there because of the preprocessor (more on that later)
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* traditional C comment */

// this is my main function
int   // why do we return an int?
main(int argc, char **argv)   // what are thoooose?
{
    int i;

    printf("Hello, world!\n");

    printf("We got %d arguments\n", argc);

    for (i = 0; i < argc; ++i) {
        printf("Argument %d: %s\n",
            i, argv[i]);
    }

    return EXIT_SUCCESS;  // exit status

    // this is also acceptable
    // it means the same thing
    // but EXIT_SUCCESS is clearer to humans
    // return 0;
}
```

Quick intro to gcc

Usage: gcc [options] source_files

```
eg. gcc hello.c
                <- compiles C program to an executable named "a.out"
        gcc -o hello hello.c
                <- compiles C program to an executable named "hello"
```

We will be primarily using C89/C90, but we will allow some C99 features

Compiling vs Linking

```
In C, compiling turns source code to an "object file"
        <- almost executable, but some references aren't filled in
        <- e.g., calls to functions in the main library just say "call this function"

The next step is "linking"
        -> turn indirect references into direct references

Once a program is linked, it can be executed

Use -c to compile without linking

        gcc -c hello.c
                <- compiles hello.c to a "relocatable object" called "hello.o"

        gcc hello.o
                <- links hello.o to the standard library (libc.a) and creates an
                   executable program called a.out

Why have a two step process?
-> save time during development
        when we make changes, only recompile the files that changed,
        then link everything at the end

For simple projects, we can just compile everything in one step
For larger projects, we can save time/keep things simple by breaking project
into smaller files and subprojects
        -> use make to manage compilation (only recompile when necessary)
```

Data in C

```
all data in C is either an integer, a floating-point rational, a pointer,
or a bundle of these

        what about Bool? We don't have them in C; we just use integers
                0 is false, everything else is true (use 1 by default)

        what about char? these are just (one-byte) integers
                'A' <- character literal; behaves the same as 65

        what about strings? there is no string type in C; we use arrays of chars

we have many sizes of integer

        char       (1 byte)
        short int  (at least 2 bytes, 2 on the iLab)
        int        (at least as big as short but not bigger than long; 4 on iLab)
        long int   (at least 4 bytes, 8 on the iLab)

        "short" or "long" by themselves mean short int and long int

we have two kinds of integer: signed and unsigned
        signed is default
        unsigned means no negative values

        "signed" or "unsigned" by themselves mean signed int and unsigned int

        unsigned x = -1;  <- will not do what you expect


"unsigned long" means "unsigned long int", etc.

        unsigned ints can have larger positive values than signed ints

                unsigned x = 1025U;  <- the "U" means unsigned
                you almost never have to worry about this

we have two sizes of floating-point value

        float and double

        C does not specify what these are
        all modern compilers use IEEE floating point

                float is 4 bytes
                double is 8 bytes

        the only reason to use float is to save space; don't bother in this class
        just use double when you need floating-point

        1.0             floating point literal
        1.0e-8          floating point literal with scientific notation
                                1 * 10^-8
```

```
declaring variables
        declare a variable by giving a type and a name

                int i;  <- declares a variable named "i" of type "int"

        we can also initialize a variable when we declare it

                int i = 0;  <- declares i and sets it to 0

Difference from Java: in C, uninitialized variable has an indeterminate value
        if we don't say what is in it, then it contains "garbage"
                <- whatever happened to be in that part of memory

        it is possible to read from a variable that has never been initialized
        <- but you never want to do this

Why does this exist?
        Originally, C required you to declare all local variables at the top of the function, and there
might be variables that you don't end up using
        initializing variables you don't end up using is inefficient (but safer)

        C always chooses speed over safety!

It is never wrong to initialize a variable, so you might as well do it
```