Day 1

Course website is `http://courses.cms.caltech.edu/cs11/material/c/mike/index.html`. Assignments are due Monday night, midnight. K&R is the bible.

C was made to be a very thin level above machine code, making it very fast/memory efficient. Hello World program is below

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

Assume the above code is in `hello.c`, then to compile we use `$ gcc hello.c -o hello` and then run the executable.

Two kinds of source code files, code `.c` and header `.h` files. Compiler turns source code into object code `.o` which is "linked" to executable. `gcc` is both a compiler and linker. Then what we did above has three steps, compiles to object code, links it together to an executable, and deletes the object file (can be kept via an arg to `gcc`).

We can do them individually via `gcc -c hello.c` to generate `hello.o`, then we can link via `gcc hello.o -o hello`. Next lecture when we talk about make files we will see how the object file can be useful.

C is not object-oriented, and programs are built up of *functions*. The `main()` function is the big daddy of functions, where execution starts/ends.

C has data types `int, char, float, double`. Variable names can have numbers as long as they don't start with a number. Booleans are 0 is false, 1 is true. Strings are arrays of characters, so `char some_string[9] = "woo hoo!";` (note that the last character of the string is implicit `\0`). C has operators, surprise surprise, all the usual shabang.o

If no parameters accepted, define function using `void`. Declare local variables at top of function. Global variables suck unless constants. `printf()` function to print things. `#include <stdio.h>` is a preprocessor directive. Conditionals are like Java.

Day 2

More preprocessor commands, `#define` is an *absurdly* common one. It performs a purely textual substitution, no typing. *Magic numbers* are terrible! Ternary operators are commonly coupled with `#define` for simple functions, i.e. `#define MAX(a, b) (((a) > (b)) ? (a) : (b)`.

Functions must be defined before use; if call function1 in function2 and function1 source is later in source than function2 then bug compiler! We use function prototypes to get around this, just prototype is enough.

`break` exits nearest enclosing loop, `goto` can exit more loops labelled `<label>:`.

IO. `printf()` outputs to `stdout`. `fprintf(stderr, <message>)` is the "correct" way to write error messages. Note that `printf()` and `fprintf(stdout,)` are equivalent. Recall `scanf()` which rceads from `stdin`. Note that `scanf()` returns whether successful, *not* what was actually read; `EOF` will be returned by `scanf()` on invalid read.

Write comments! Function of function, args, return value. `make` is a program that takes care of compiling/everything management. `Makefile` has compile info. Example makefile

```
program: program.o
    gcc program.o -o program
program.o: program.c program.h
    gcc -c program.c
clean:
    rm program.o program
<target>: <dependencies>
    <commands>
```

Only will compile changed files!

Day 3

Arrays! And Command line args. And assertions. Today. Yes.

Arrays are linear sequences of data objects, can be different types! Uninitialized array space contains junk! Arrays can be initialized

```
int my_array[10];
int my_array[5]; = {1, 2, 3, 4, 5};
int my_array[] = {1, 2, 3, 4, 5};
/* OKAY! */
int my_array[10] = {1, 3, 5};
/* partially initialized, rest of array is ZEROED! */
```

Note that `int my_array[10] = 0;` zeroes the array! So clever. If we try to access outside array bounds, C doesn't check array bounds, so works just fine, no errors. Note that the curly brace syntax is only used during declarations, cannot be used on a separate initialization line.

2D arrays are okay, but are laid out in memory just like extra long 1D array. Initializing 2D array, can be written

```
int my_array[2][3];
/* not initialized */
int my_array[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
/* initialized */
int my_array[2][] = { { 1, 2, 3 }, { 4, 5, 6 } }; /* okay */
int my_array[][3] = { { 1, 2, 3 }, { 4, 5, 6 } }; /* okay */
int my_array[][] = { { 1, 2, 3 }, { 4, 5, 6 } }; /* NOT OKAY */
/* initialized */
```

C passes by value, not by reference.

Command line args! Recall that `main` function includes `argc, argv` in header! `argc` is number of arguments, `argv` are the actual arguments, as an array. As expected, `argv[0]` is program name. `atoi(), atof()` converts string to `int, float` respectively, in `<stdlib.h>`. Then `strcmp()` returns integer, 0 if equal, in `<string.h>`. Do not use `==` because compares pointers!

Assertions can be used easily! Make sure to include `<assert.h>`. Assert very useful to check whether elements are sorted. Use via `assert(<value>)`, will halt if `<value> == 0`. We will have `sorted(int arr[], int nelems)` to check whether our sorted array is sorted, for our assignment.

Assertions should only be used for logical correctness of code, debugging purposes. Next week pointers!

Day 4

Keep in mind that C was written to eliminate machine code, but needs to be low level too. Has low-level access to addresses/pointers. Address of variable is written `&x` and pointers are variables that hold addresses. Consider then when we write the following code

```
int i = 10;
int *j = &i
```

then there will be an address with name `i` and some address pointing to a location in memory containing 10. Then therewill also be an address with name `j` with some address pointing to a location in memory containing contents equivalent to the address of `i`! Also dereferencing operator `*`.

Correct way to print hex numbers in `printf()` is using `%x`.

Pointers allow us to call by reference. Short lecture! Next is pointers/arrays untold story.

Day 5

Pointer arithmetic exists and works and can be used! Keep in mind though that the pointer increments/decrements based on its typing; can be dangerous if we're out of scope! Wwe can get the size of a type using `sizeof()`; note that this is an operator not function b/c it takes a type not an argument.

Arrays are actually just syntactic sugar for pointers; there exist only memory chunks and pointers to memory chunks in C! Note that pointer addition is pretty slow compared to pointer increments; can get speedups by keeping track of pointers manually and incrementing rather than looking for the 900th element of an array.

DAM! Dynamic memory allocation. If we want dynamically sized arrays we have to use DAM. Three library functions

```
void *malloc(int size)
void *calloc(int nitems, int size)
void free(void *ptr)
```

which we have to include `stdlib.h` to use. `malloc` will just return the pointer to the address in memory, and it will reserve as many bytes as passed as argument. IT CAN FAIL; we will talk about this in a second. `calloc` takes two arguments, number of "things" to be allocated and number of bytes per "thing." It will zero the momory, so it's slightly slower.

Sometimes these will fail! Maybe no memory or something. If so they will return `NULL`; ALWAYS ALWAYS **ALWAYS** check for this! For example

```
if(arr == NULL)
/* can also do "if(!arr)" since NULL=0 */
{
    fprintf(stderr, "out of memory!\n");
    exit(1);
}
```

Lastly comes `free`, which frees the DAM'd memory. Curious note: when `free()` is called it doesn't instantly erase the memory, only marked for removal by the OS, so sometimes program can still access `free`'d memory! Note that only memory on the heap can be returned; memory on the stack is marked for removal as soon as out of scope.

Memory leaks suck poop, but don't free memory on the stack either! Blah stuff on statically vs dynamically allocated memory; Brian makes just as much sense, he should be a lecturer. Memory leaks are one of the *worst* kinds of bugs because no harm is done! Hard to track down and can slow down program. `valgrind` is the holy grail of memory leak checking, but Mike will supply us with a cheap version. TOO BAD I'M A VALGRIND USER ALREADY **YES**.

Day 7

Last lecture we covered structs, `typedef` and linked lists. I missed it by accident sorry.

This week we will talk about hash tables, more on the C preprocessor, `extern` and `const`.

Hash tables are a new data structure that conceptually is just an array indexed with strings. This yields $O(1)$ constant time. What we want is associate a string (key) with a *value*. We generate what is called a *hash value* from the key, such that different keys generate different hash value. Hash value is an index to an array of linked lists (array of pointers to `Node`s); the array starts with `NULL` pointers. Note that no linked list should ever get large in the hash table else hash table is ineffective.

Let's do a hashing example, turning a string into a key for the hash table. Take the string as a list of `char` which are each integers $0 - 127$ (C allows us to treat chars directly as integers) and we can just sum over the string. Note that strings are terminated with `NUL` as opposed to `NULL` the null pointer (They are numerically equivalent but conceptually different so please be careful else Mike be sad). We can take this sum to be our index into the array.

Obviously there will be many hash collisions, so an easy extension is to multiply each character by some constant; subject of much research!

To find a value in a hash table then we generate the value with the key and lookup; if the location is `NULL` return "not found", else we iterate through the linked list and compare directly to our value. To add a value we just append it to the correct LL! **Note: It is probably easiest to append to LL at the beginning.**

Lab 7 will be just a routine application of hash tables; one particular memory leak will abound! Be careful.

Let's now discuss the C preprocessor. Sometimes want to *conditionally compile code*, decide whether to compile code while compiling! An example of debugging code is

```
#define DEBUG
    \dots
#ifndef DEBUG
    do stuff
#endif
```

Note that while `DEBUG` isn't defined to be anything, it is defined! This is easy when debugging code pops up in a lot of places. An alternative way to do this is `gcc -DDEBUG` to define DEBUG (useful for things like `Makefile`!). We can also use `ifndef`.

We can actually use `#if` and `#elif` and not just for definitions! For example `#if REVISION == 1`, though this is now obselete with version control systems.

We can also talk about include guards! Multiple inclusion of header files *can* cause problems; defining a struct twice is an error. Include guard

```
#ifndef FOO_H
#define FOO_H
/* code */
#endif
```

Clever right? Oftentimes people will use underscores on either side of the filename, not necessary.

Last feature, `extern`. Global variables `extern int a` such that value is shared between files that include it!

Last-er feature, `const`, newer way to define constants `const int C = 100;` gives typechecking!

Next week is the last lecture; we will not cover certain things (function pointers), but we will discuss virtual machines and more integer types `short, long, unsigned` and wrap up!

Day 8

Last week we covered hash tables and C preprocessor. We will talk about other integral types, bitwise operators, `switch` operator, virtual machine assignment.

We usually use `int` to represent integers, but there are a few more that we can use: `short, long, char` and `unsigned` varieties. Note that `char` is always 8-bit number (ASCII).

This raises two questions: why use `unsigned` or shorter/longer integer types? Using shorter type is to save memory, and unsigned types get an extra bit. `unsigned` types can also be used as an array of bits. Shorter integer type just saves space. Guaranteed is char < short < int < long in terms of bit length, and in general they are $8, 16, 32, 32$ bits. `unsigned` assumes `int`.

A cool application is to use bitwise operators for bit arrays in `unsigned`'s! There are bitwise operators OR, AND, XOR, NOT being `|, &, ^ ,  ~`. Don't use short-circuiting like logical `||, &&`, but if want no short circuiting then can use these.

XOR has an interesting property; XOR-ing with 0001000 will flip that bit; this is called a *mask*. Two more bitwise operators, left/right shift `< <. > >`, faster way to multiply/divide by two.

Then `switch` statements. Syntax

```
switch (i)
{
```

```
    case 0:
        do;
        break;
    case 1:
        do;
        break;
    default:
        do;
        break;
}
```

Looks much nicer than `else if`s! Can only be used on an integer types. Sometimes we will want to omit the `break` to *fallthrough* the switch statement cases. In general we want to comment the `/* FALLTHROUGH */`; some compilers will recognize this and not warn about the lack of break statement!

We now discuss virtual machines. The biggest example of VM is J(ava)VM. We will implement such a sort of virtual machine for a mythical computer's assembly programming language. Our machine will have only `int`s and instructions will act on `int`s. Each instruction memory contains 65536 memory locations, each location being a single byte (`unsigned char`). We will then use an `unsigned short` to refer to memory. We will have 16 registers, so we only need 4 bits to determine location; we will use 8, i.e. `unsigned char`.

We then have a stack which is 256 deep, so we need 8 bits to represent all locations in the stack. We have a *stack pointer* to keep track of where we are in the stack (for our VM, it will be just an index). Stacks have only two operations, `push, pop`. Push will add to stack pointer and repoint, pop will pop from stack pointer etc. The stack pointer points to the first unused location and the *TOS* (top of stack) refers to the first used location. Note that when we pop, we can either choose to erase or not the old stack element; we will overwrite it anyways.

VM instructions are often referred to "bytecode" because all instructions can be indexed by a byte (8 bits, 256 operators). Our VM will have only 14 operands. These are

- `NOP` (0x00) - does nothing
- `PUSH` (0x01) - `PUSH <n>` pushes integer ont stack
- `POP` (0x02) - removes top element from stack
- `LOAD` (0x03) - `LOAD <r>` pushes contents of register onto stack
- `STORE` (0x04) - `STORE <r>` pops top of stack and stores it onto register
- `JMP` (0x05) - `JMP <i>` sets instruction pointer to <i>
- `JZ` (0x06) - JMP only if TOS is zero (also pops stack).
- `JNZ` (0x07) - JMP only if TOS is nonzero (pops stack).
- `ADD` (0x08) - pops top two elements and pushes back their sum
- `SUB` (0x09) - pops top two elements and pushes back their difference
- `MUL` (0x0a) - pops top two elements and pushes back their product
- `DIV` (0x0b) - pops top two elements and pushes back their quotient
- `PRINT` (0x0c) - pops stack and prints it
- `STOP` (0x0d) - halts program

We will write the program in our assembly language (actually, he gives us this) and we write the interpreter. We will only use two registers; register 0 starts at 10 and register 1 starts at 1, and at each step register 1 multiplies by register 0 and decrement 0. The assembly is

```
1 load 0
jz 2
/* multiply them */
load 1
load 0
mul
store 1
/* subtract register 0 */
load 0
push 1
sub
store 0
jmp 1
2 load 1
print
stop
```

We must pull off the byte-code interpreter; can do in about 70 lines of code. Error checking: popping empty stack, stack overflow.

Natural next-steps are CS11 tracks, CS11 project track, CS24.