Applied Algorithms

Lecture 1: Welcome, Pointers, and C

Welcome!

- Can everyone see me and the projector?

About the class

- Goal: bridge the gap between theory and practice
- Are there theoretical models that better predict practice? (Yes, sometimes)
- How to implement ideas efficiently in practice
- Using algorithmic knowledge to become better coders!

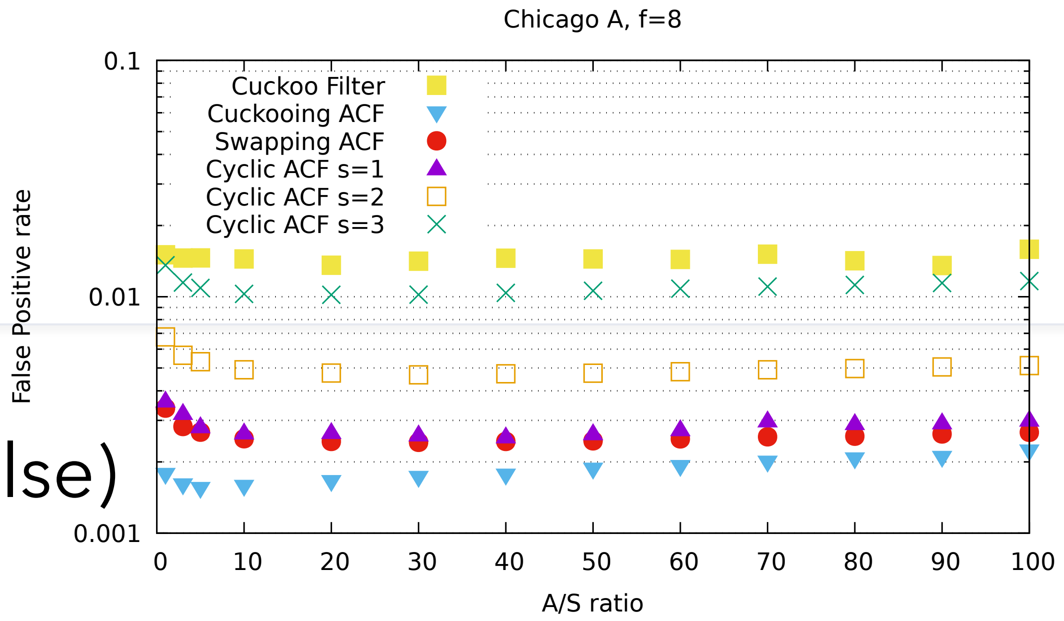
Pantry Algorithms

- Algorithms that you should always have handy because they are incredibly useful
- Bloom filters, linear programming, Lloyd's k-means

Coding

- Friendly, optionally-anonymous competition for bonus points
- Code review occaisionally!
- Collaboration (with citation) encouraged
- Make C code run fast
- Mostly no parallelism (sorry)

About me



- Call me Sam (or something else)
- My research is in algorithms
    - Data structures, randomized algorithms, similarity search
    - Some practice!
- Office is TCL 209
- Office hours in 312 *Unix Lab* (not my office) Wednesdays 1-4
- Office hours in my office Tuesday 3-4

History of the course

- I taught a few times before
- But not here!
- Potential minor scheduling adjustments
- Stay in touch if there are problems!

- No TAs – ask me questions and collaborate with each other

About the course

- Hopefully: half theory, half coding
- In terms of time and in terms of grading
- Probably more focus on ”theory” in lectures

Theory

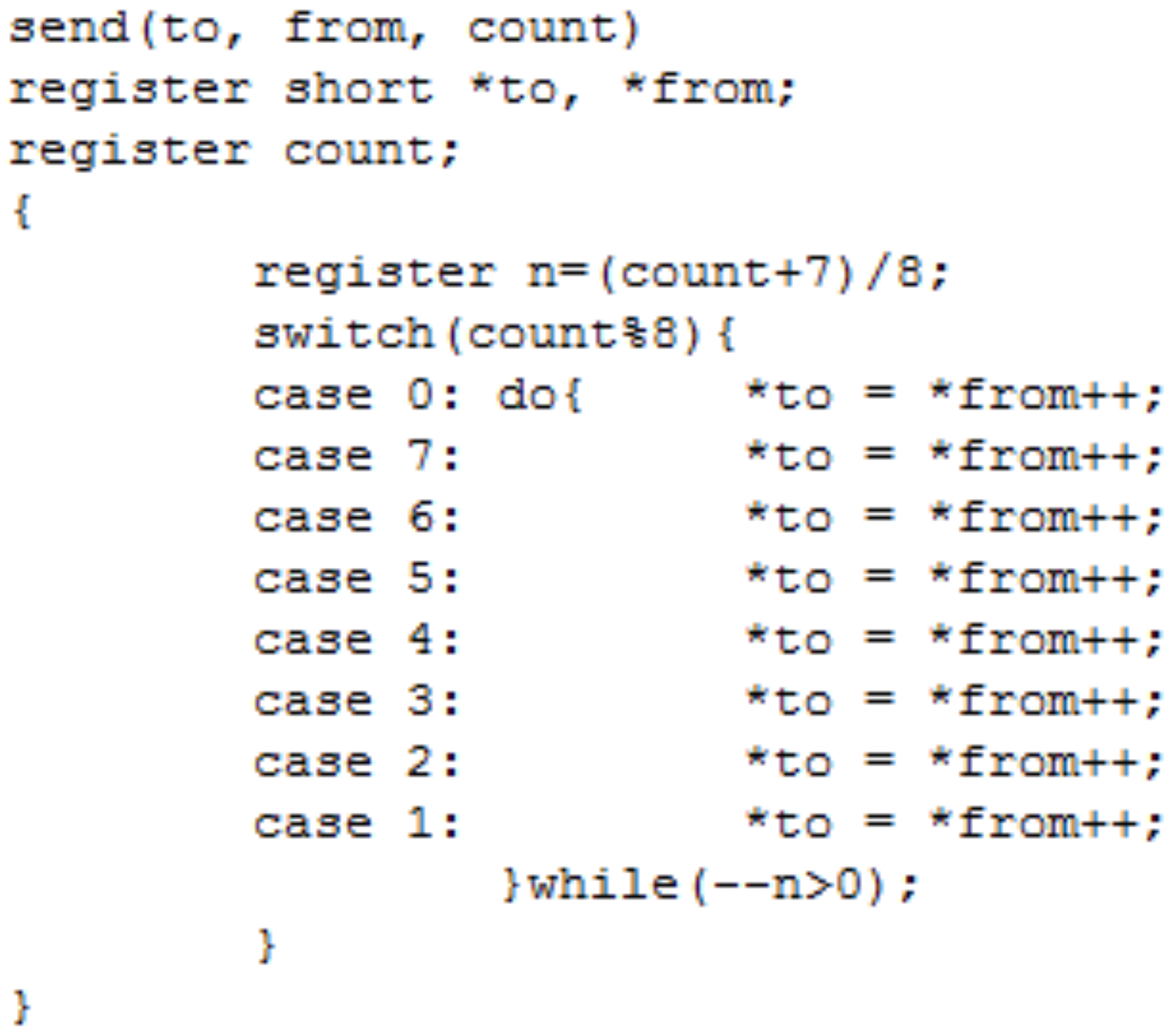
- Algorithms is (technically) a prerequisite
- If you haven't taken 256, might need some catchup
  - We're doing dynamic programming week 2
  - Second section of course (in March) is probability
- Slightly different focus:
  - Design
  - New models/considerations
  - Think 136 more than 256

Coding

- We'll be coding in C
- Weekly assignments
- First assignment is intended to give you a chance to catch up
- Office hours!
- Grading should not be too strict, collaboration is encouraged

Why C?

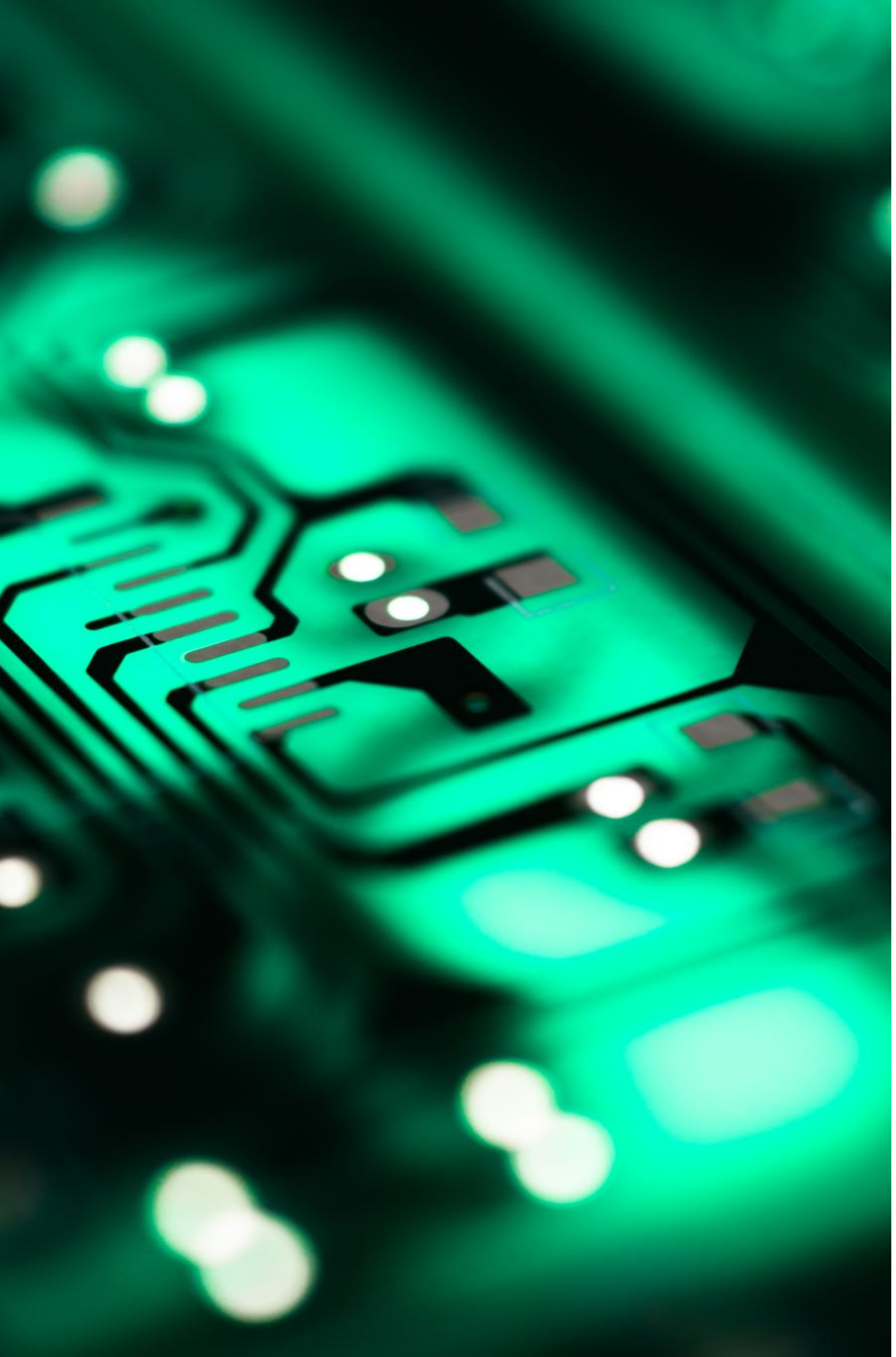
- Familiarity!
  - Seen C/Looks like Java
- Low-level
  - See impact of course concepts
- Fast!
- Useful to know!



```
send(to, from, count)
register short *to, *from;
register count;
{
        register n=(count+7)/8;
        switch(count%8){
        case 0: do{     *to = *from++;
        case 7:         *to = *from++;
        case 6:         *to = *from++;
        case 5:         *to = *from++;
        case 4:         *to = *from++;
        case 3:         *to = *from++;
        case 2:         *to = *from++;
        case 1:         *to = *from++;
                }while(--n>0);
        }
}
```

Course website

- Can access from CS webpage, or my site
  - Hopefully from Google soon
- Are you registered?
  - Please email me if not!
- Go through site and syllabus



Crash course in C

- Intro/refresher
- Readings and practice available on website
- First assignment is in pairs, intended to give a chance to catch up on C (as well as learn a new algorithmic concept)

- If you are experienced in C, let others answer questions

About C

- Lifetime of information to learn
- I am not an expert (even though I’ve used it a lot)
- Many features, many interesting effects behind the scenes

Simple program

- Hello world
- Preprocessor/include
- Print sum of two variables
- Loop
- If, modulo

- Compile

Arrays

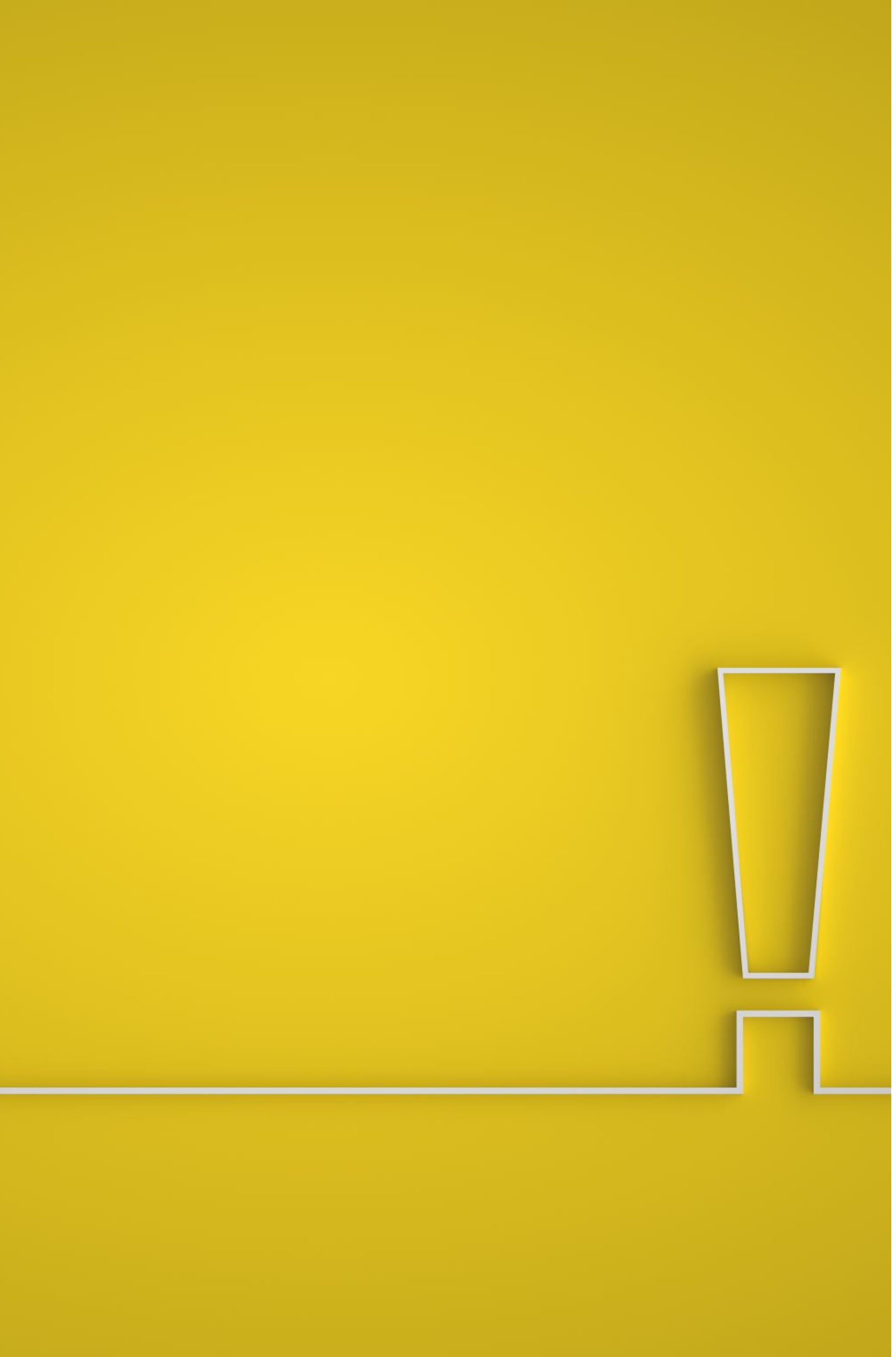
- Arrays work largely like Java
  - We'll talk about "new" equivalent in a second
- No bounds checking!!! (also, no boolean)
- sizeof for fixed-size array (C replaces at compile time; easier to read)

Structs

- No classes, structs instead
  - No member functions
  - Sequence of variables stored contiguously in memory
  - Use . operator to access member variables
- Semicolon after declaration
- Use “struct” to refer to your structs
- OR use typedef

Pointers

- Manually get the address of variables
- Addresses can be stored, printed, manipulated
- int* stores a pointer to an int; char* stores a pointer to a char
- & operator gets address
- * operator returns *value at* address
- Changes between executions
- Arrays



Careful coding

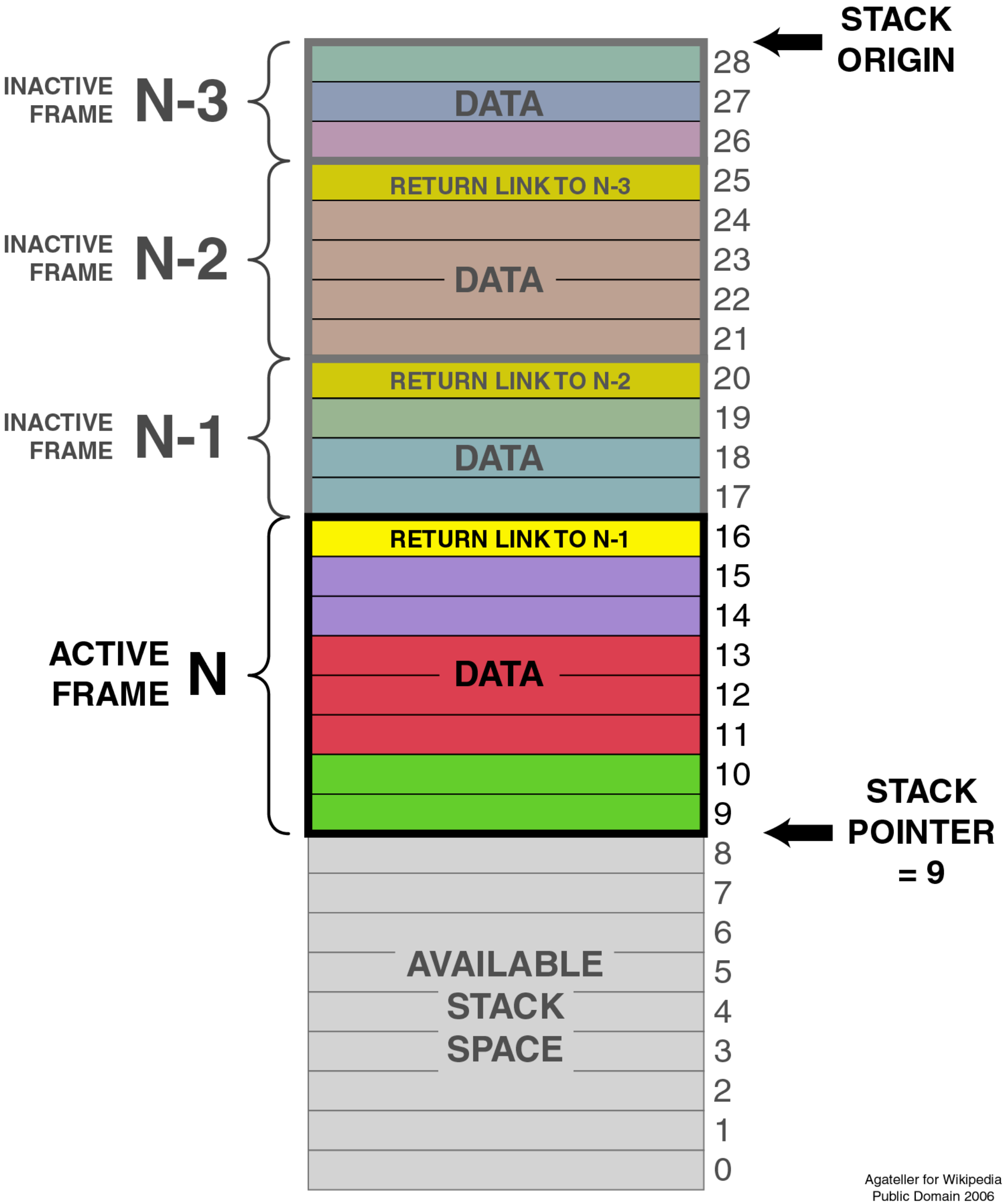
- Good coding practice is much much much more important than ever
- Include asserts to check array ranges
- Code, test, code, test
- Split into functions and test separately!
- Check your pointers!
- Corner cases! (Is this pointer null? Is this value 0?)
- Speed is not your first priority, correctness is

Pointers, functions, and structs

- Creating function
- Passing is *always* by value. Can pass struct instances
- How do we change a variable inside a function?
  - Pass the address–the address doesn't change, but the value does!
- -> operator
- Structs stored contiguously in memory

Allocation

- “new” in Java and C++ allocates space for a new instance of a variable
- C uses “malloc”
- Very much user-controlled: you set the space, no garbage collection



Agateller for Wikipedia
Public Domain 2006

Where are things stored?

- First place: in CPU register, never in memory
  - Temporary variables like loop indices
  - Compiler decides this
- Second place: call stack
  - Small amount of dedicated memory to keep track of current function and local variables
  - Pop back to last function when done
  - Temporary!

The heap

- Very large amount of memory (basically all of RAM)
- Using new in Java or C++ puts variable on the heap
- We use malloc
    - Does not zero out memory. calloc does
    - C will not make you instantiate your variables
- Needs stdlib.h
- Returns pointer; don’t need to cast to pointer type

Ways to store things

- Speed: registers > stack > heap
- Size: heap > stack > registers
- Longevity: heap > stack > registers

- Java rules work out well: store "objects" and arrays on heap, just declare small "primitive types" and let the compiler work it out

Allocation, pointers, and arrays

- What is an array?
- Can we use arrays without using array-like things?
  - Using pointers and malloc instead?
- Does this allow us to allocate arrays dynamically?

- Pointers and arrays are (mostly) *equivalent* in C

Memory leaks

- C does not have a garbage collector
  - Fast, efficient, you actually really want to be able to control this
  - But, obviously, huge pain and difficult to debug
- free() releases memory
  - Can be used for another variable
  - Not zeroed out
- Every malloc() should have a free()!
- After your program ends all memory is released

Segmentation faults

- Access “illegal” memory
  - Address that the OS didn’t give your program
- Given very very little information
- Debug using gdb (checkpoints, etc.)
- valgrind is useful for checking memory
- We’ll see some examples of these next week

Compiling and building

- Compile: convert code into machine-executable code
  - gcc –c [file name]
- Link: stitch together function calls between files
- Build: whole process
  - What gcc actually does when given file
  - Need to list compiled object files

What happens when we change one file?

- Need to recompile that file
- Need to build final output file

- Can we do this automatically?

Makefile

- Lists dependencies
- Lists what you actually want to build
- Entire command: make
- If a file changes, compiles only what's necessary

- Very very useful!

In this class

- I will give you makefile
- Don’t need to change unless you use multiple files
    - You can, but probably won’t ever need to
    - Projects in this class are fairly small and self-contained