

Welcome; About Computers; Intro to C

CS 350, Lecture 1

A. Why?

- Course guidelines are important.
- Active learning is the style we'll be using in the class.
- Before studying how computers are organized, we should think about what they do.
- We should also think about any fundamental differences between computers.

B. Outcomes

At the end of today, you should:

- Know how the course will be structured and graded.
- Have practiced some of the techniques we'll be using in class.
- Know that computation involves manipulation of symbols.
- Know that (up to memory and speed constraints) all computers can do the same computations.

C. Introduction and Welcome

- The course webpages are at <http://www.cs.iit.edu/~cs350>
- The home page includes news and the calendar of lectures, coursework, and tests.
- There's a Course Plan page that discusses the textbook, the course topics, course structure, and grading, collaboration, disability, and ethics policies.

D. Active Learning [See Activity 1.1]

- Education research shows that students learn better when they are active participants in class, compared to passive listeners of lectures.
- This active kind of learning includes obvious things like answering questions in class, but it also includes team activities and quick feedback to the instructor.
- You'll need to put more effort into class, but you'll learn more/better/faster.

E. What Are Computers?

- So what do computers do, anyway? [See Activity 1.2]

F. Computers as Symbol Manipulators

- Clearly one important use of computers involves manipulating symbols.
- In some sense, computers don't actually manipulate symbols, they move photons and electrons etc.
- We think of them as manipulating symbols or doing calculations because of our interpretation of these activities.

- I.e., we view things from some context and abstract away the parts that aren't part of the context.
- Textbook identifies levels of abstraction and transformations in computer systems:
 - Problem, Algorithm, Language (Program), Machine (Instruction Set Architecture), Microarchitecture, Circuits, Devices.
- Compiler translates high-level program into machine code (a sequence of bits).
- Machine runs machine code that comes from a particular set of instructions (the **ISA: Instruction Set Architecture**).
 - The ISA is the interface between software and hardware.
 - The ISA instructions themselves can be broken down into sub-operations; this is the microarchitecture of the machine.
 - The micro-operations and -operands are implemented using logic and arithmetic circuits (which deal with bits).
 - The circuits are implemented using devices (such as transistors, switches) that fiddle with electrons.

G. Is There An Ultimate Computer?

- Obviously some computers are more powerful than others, assuming we measure power as speed or memory etc.
- If we abstract away from speed and memory constraints, is any computer more powerful than another?
 - Turns out they are equivalent in power.
- But they're equivalent in the sense that they all do the same kinds of calculations.
 - Given enough time and memory, any computer can simulate another computer.
 - E.g., the "Virtual PC" program ran on Power PC Macintoshes and made them simulate Intel hardware, so you could run Windows (very slowly :-).

H. Turing Machines

- Alan Turing was a mathematician & cryptologist; he worked on the British Enigma project that broke Nazi codes and made them readable by the Allies.
- He invented what we call "Turing Machines" (TMs).
 - Very simple theoretical computer.
 - Has a tape with a read/write head.
 - You have instructions like "If we're in state 72 and we see a \$, move the head right one symbol and go to state 53."
- Unsurprisingly, you can simulate a TM with a modern computer.
- Surprisingly, TMs can simulate modern computers (verrrry slowly).
 - You can design TMs that do arithmetic on integers and floating-point numbers, manipulate characters, and so on.

- You could even simulate an Intel PC.
- There even exists a “Universal” TM.
 - The Universal TM is an interpreter/simulator of TM programs: You give it a description of the other TM and the other TM’s data, and the universal machine will do what the other TM would’ve done.
 - In some sense, this is the only TM you would ever need to actually build.
- In the same way, we don’t generally build different computers to (e.g.) edit documents, edit photographs, send/receive email, etc.
 - We might, if we bring real-world considerations back into the picture.
 - E.g. build a small, cheap email machine.
 - What we do is build general computers and then write programs to make them simulate the computations some more-specialized machine might do.

I. Turing’s Thesis and Universal Computation Devices

- Ignoring speed/memory constraints, all computers are equivalent. Rough argument:
 1. Take two computers X and Y.
 2. For each, write a program that simulates the universal TM.
 3. Write a universal TM program that simulates X and another for Y.
- For step 3 to work, you have to assume that any computation that can be done by some computer can be done by some TM. This is **Turing’s Thesis**:
- **Turing’s Thesis**: Every computation that can be done can be done by some TM.
 - This is not formally provable, but it is true for every computer we’ve built or imagined building.
 - Another way to say this is that TMs are **Universal Computation Devices**.
 - TMs can do every conceivable kind of mechanical computation.
- Modern computers are also universal computation devices.
 - TMs and modern computers are equivalent (can solve the same set of problems).
 - Modern computers can do every conceivable kind of mechanical computation.
- Turing’s Thesis sure looks to be true.
 - Nobody’s found a mechanical, computational operation that a TM (or modern computer) can’t do.
 - Even quantum computers (when they get built) will only solve the same set of problems we can solve today (but an awful lot faster!).

J. Overview of C

- Historically, C was developed so that Unix could be written in it. (Mostly early 1970s.)
- Translation from C operations to machine-level operations intended to be simple.
 - C statements/operators/datatypes correspond closely to machine-level operations.
 - Primitive datatypes come from hardware datatypes

- Build up values using arrays, records, pointers.
- Doesn't have objects, modules, classes.
- Should be able to access all hardware features from C language
 - Underlying machine has operations like pre/post-increment/decrement.
 - Add operators to language to let programmer use the operations.
- Versions of C
 - K&R C (Kernighan & Ritchie, from their book *The C Programming Language*).
 - ANSI C (C89 or C90)
 - function prototypes, pointers to void
 - C99
 - inline functions, variable-length arrays, one-line comments via `//`.
 - C11
 - Just came out last year (2011)
 - More-general types, multi-threading library, bounds-checking.

The C Preprocessor

- Preprocessor gets called before actual compilation; does text-to-text translation.
- Preprocessor directives included in program but not part of C language.
- `#include` directive
 - `#include <stdio.h>`
 - Replace line above by contents of header file `stdio.h`.
 - Includes names/types of some standard I/O functions and variables.
 - Can `#include` a file with anything -- could be any C source code.
- `#define` directive
 - Define macro name and value — used for named constants before they were added to 1999 version of C language.
 - `#define STOP 0`
 - After this definition, the preprocessor will replace all instances of `STOP` with `0`.
 - Define macro instruction with arguments
 - `#define NEG(X) -(X)`
 - Explicit parentheses around `X` to avoid precedence/associativity problems.
 - Define a name to exist
 - `#define DEBUG`
 - Used in conditional compilation