

Contents

1	Administrative Details	1
2	What is Computer Science?	2
2.1	What Computer Science is Not	2
2.2	So what is CS?	2
2.3	Various Fields of CS	4
2.4	So where does one start?	4
3	A Brief History of CS (1st Day)	5
3.1	Early Days	5
3.2	Turing	5
3.3	Development of Software & Programming Languages	7
3.4	Interpreter vs. Compiler	8
3.5	The Myriad of Programming Languages	8
4	Diving in	9
4.1	Course Outline & Book	9

1 Administrative Details

- Review Syllabus - any questions?
- Does everyone have a good idea of what to expect in this class?
- Office hours... what time is everyone available?

2 What is Computer Science?

2.1 What Computer Science is Not

- Becoming proficient in MS Office or other applications
- Programming a VCR (are those still around?)
- Learning every programming language in existence

2.2 So what is CS?

- CS is relatively “new” . . . having roughly 80 years of history
(Aside from the theoretical backgrounds, which are important)
- As such, *many* people have *many* different ideas about what CS really is
- Educators and employers typically have very different ideas
- *This is getting better!*

- The core CS curriculum, as employed by UC is as follows
 - Focus on good programming techniques and data structure usage
 - *Heavy* focus on algorithm design and analysis
 - *Heavy* focus on mathematical backgrounds and tools
 - Study of algorithms is central – separates CS from programmer

- Programming is not the end goal for CS
- Programming is best viewed as just a tool, used to interact with a computer
- As such, it's a good first step... *That's why you're here!*

- Why C++?
- Why not the *best* programming language ever! Which is (insert name here)?
- Although we may have preferences for our programming languages, the above question is *meaningless*
- A more appropriate version would be “what is the best programming language *for this particular project?*”

2.3 Various Fields of CS

- Algorithms
- AI
- Data Mining

2.4 So where does one start?

1. First, you learn the *tool*, programming, and become proficient using it.
2. Next, you learn data structures; how to represent complicated structures of data in an easy, intuitive, and relevant manner.
3. Next, you learn algorithms... and plenty of them. Given a problem to solve, what's the *best* way to solve it? Here, best may mean "fastest" or "uses the least memory", or perhaps some other measure. Depending on what is best for you, the choice of data structures (how to represent your problem / data) may have a huge impact (this is why DS is studied first).
4. Also learn the basics of how a computer works.
5. Various science/mathematics help tremendously
6. After that... it's your choice (cover the core stuff, but also describe some of the free electives. Not too much detail here?)
 - Note that the study of data structures and algorithms are *theory*, and those studies are *fairly independent of any programming language*.
 - However, to apply it, we need to work in *some* language.

3 A Brief History of CS (1st Day)

3.1 Early Days

Even before the days of punch-cards...

- To solve a problem, we construct an *algorithm* for how to solve it
- People begin building machines that *act* as a specific algorithm.
- We say that the machine *implements* a specific algorithm.
- NOTE: At this time, there was no such thing as “software”. Each machine was built for a specific problem, using a specific algorithm, and that was all it could do.

Imagine the price of MS Office if you had to buy one machine for Word, another for PowerPoint, and a third for Excel!

3.2 Turing

Then there was Turing.. who had 2 great insights (an overview at 30,000 feet)...

1. **ALL** machines could work using data represented in a single format, binary, and every machine we could build, could be built using the same 4 simple parts.

So, rather than building each new machine using specialized hardware, we could just make 4 parts, and figure out how to arrange these 4 parts to implement a given algorithm.

Price of MS Office comes down dramatically! With student discount, only \$250,000 for the whole suite!

2. An then, the big one...

- Turing was playing with the ways of arranging these 4 pieces.
- Noticed that he could “represent” any arrangement *in binary*
- I.e. He could represent a machine *as data*

- **THEN** he built a single machine, that could read in such a representation, and then *behave as that machine!* This is called the UTM.
- For example... we want to solve some problem, so we devise an algorithm to solve it. Next, we layout the structure of a machine that will implement this algorithm (and thus solve the problem). Then, we represent the machine as binary, and feed-it-into the UTM. Viola! The UTM *acts like* the machine we devised.
- The birth of software! No more building a machine for every purpose! We build *one*, and we build that well... Then we simply write software to do the rest.
- MS Office now affordable to most people!

3.3 Development of Software & Programming Languages

- There were (and still are) many different architectures of a computer.
- Since each architecture is different, each has its own format for representing machines (i.e. how we encode machines as data).
- Thus, in the beginning each architecture had its own “language” for interacting with it... for telling it what to do.
- This was (and still is in some cases where it is required) extremely cumbersome.
- Then, (*nearly*) standard languages were developed that could be used across different architectures.
 - You could write your program in such a language, and then run it on different platforms, and hope to have the same result.
 - To run a program in such a language, you must use either a *compiler* or an *interpreter* for that language, specific to the architecture you want to run the program on.

3.4 Interpreter vs. Compiler

- We write programs in text files, similar to how you write a Word document
- By itself, that file does nothing...
- We need some way of telling the computer to “do what your program (the file) tells it to do”
- There are two main ways...
 1. Use a *compiler* to turn your file(s) into a format that can be directly run on the computer.
 2. Use an *interpreter* to read your file(s), line by line, translate each line to the language of the machine, and give the command to the machine to be run.
- In this class, we’ll be using a compiler

3.5 The Myriad of Programming Languages

Over the years, many languages have been developed...

- COBOL / Fortran (*Very popular in the 70’s and early 80’s*)
- C (*Very popular from the mid 80’s until even today*)
- C++ (*A successor (of sorts) for C, which is still very popular*)
- Java (*Very popular from late 90’s onward*)

Once you learn *one* of these languages, it’s relatively easy to learn another of the similar kind.

4 Diving in

- We'll be learning C++, and using the MS Visual C++ compiler and IDE for writing our programs.
- MS Visual Studio is *very popular* in industry for writing C++ programs
- As such, *we're not just learning a language, but also Visual Studio!*
- What's an IDE?
 - Stands for *Integrated Development Environment*
 - Basically, it's a piece of software that tries to help you write programs
 - An IDE usually makes it very easy to write, compile, run, and debug programs

- Most programs you are familiar with have a *GUI*... A Graphical User Interface
- I.e. you use your mouse to move through menus and “do stuff”
- Although almost all modern software has a GUI, we'll be doing something more basic at first
- Instead of a GUI, we'll be running things at a command line

- Don't worry, *once you know the basics of writing software, creating a GUI for your program is relatively easy*

4.1 Course Outline & Book

The following reading is officially assigned...

- Before this week is over, you should have read all of Chapter 1 and the first half of Chapter 2 (up to the section on the if/else structure)
- By the second week, you should have read all of Chapter 2
- For the third week, you should have all of Chapter 3 read
- For the fourth week, you should have all of Chapter 4 read
- For the fifth week, you should have all of Chapters 5 & 6 read