Brown University
Alex Khosrowshahi (Class of 2027)

# CS300: A Student's Guide

Collected Notes and Reflections for my Favorite Class

Yes, that is me in the photo

October 5, 2024

# The Purpose of this Document

This document is not meant to replace CS300's lecture notes, nor is it meant to replace lecture itself. If you want to do well, attend all lectures and ask questions. Go to hours. Go to section. Do your work. Your efforts will not go unrewarded. This document is also not restricted to in-order reading. Refer to whatever you need help with at a given time, set it down, and come back once you need help again. The lecture notes are probably better than I am.

# Preface

**CS300 is not an easy class.**

Quite the contrary—you will more than likely find yourself pulling long nights of head-scratching work for it if you share the lack of time management skills typical to computer science students. That said, it is one of the finest introductions to computer systems design, implementation, and, most importantly, principles that a student can experience. CS300 is hard because computer systems is a challenging subject—one that many of the greatest minds in computer science have and continue to grapple with daily.

I say this because I do not want you to be discouraged by the hardships you may face. My advice is that when you become frustrated, step away for a moment and think about why you are here—what this whole systems thing is about. To build a computer has never been an easy task, and yet we do it despite adversity.

To borrow the words of the brilliant Clifford Stoll, "Math ain't about numbers. If you think math is about numbers, you probably think that Shakespeare is all about words. [...] Math ain't about numbers. Math is about logic, it's about beauty, it's about connections, it's about how you get from one place to another." Though we may not wear our mathematician hats in this course, I encourage you to adopt a similar sense of intellectual whimsy navigating this subject. As you learn about the internal mechanisms that drive electronic devices you use—of caching buffers, page tables, and other implementations we will discuss, you will find that your computer is full of stories; these stories are of mammoth challenges and the great minds that overcame them. Take moments to appreciate these recondite narratives—there is much beauty in the everyday machines we use.

# Contents

# 1 Memory, Data Types, and Pointers, Oh My!

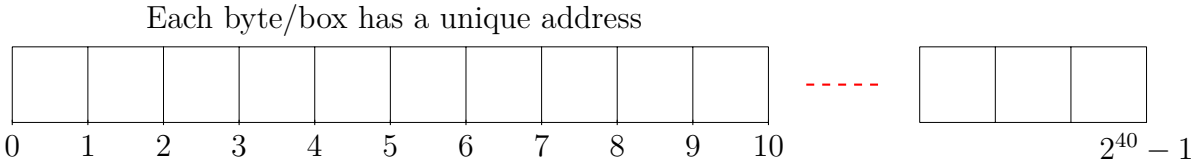
By the time you take CS300, you hopefully have some fluency with a computer. You probably can make some decent projects with some boilerplate code or from scratch (good on you), you probably know the names of basic data types and what a constructor does, and you probably have some experience using an IDE, whether VSCode, IntelliJ, vim, emacs, or some other development tool.

These are all imperative skills to have and ones that you'll need going forward, but up until now, you have been unknowingly working through many levels of abstraction. When you initialize a String, you don't think much about the space that string takes up. When you delete a node from a LinkedList in Java, inbuilt garbage collection handles the nastiness of cleaning up your computer memory. In this course, you'll learn what's going on under the hood of your computer—firstly, with understanding how data is stored.

## 1.1 Bits and Bytes

Your computer's memory is formed out of **bytes**. Bytes are segments of 8 bits. A bit is an atomic unit of data with two modes, on or off, 1 or 0. Each byte, comprised of 8 bits, can hold $2^8 = 256$ values. This means that within every byte we can store **any number between 0 and 255.**

Every byte of memory in your computer has an address, one for each byte of memory. Professor Schwarzkopf and Professor DeMarinis like to describe these like post-office boxes, each storing some mail (value) and each having an address.




## 1.2 Types and Sizes

Looking at a computer's memory this way, we stumble upon a surprisingly profound realization: all computer memory is the same. The data "types" we use are simply expressions of how much memory we're operating with and what information we store in said memory. For example, we can fit all commonly used English characters* within one byte of memory, with 256 unique identifiers, so a char is only one byte. An integer uses 4 bytes, giving us a range (from $2^0 \rightarrow 2^{32}$ if unsigned). One of the greatest strengths of C is the ability to work with memory on a deep level, so knowing your types and exactly how much memory they take up can be helpful.

Table 1. Data Types in C

| Type | Size | Value Range |
|---|---|---|
| char (signed) | 1 Byte | -128 to 127 |
| char (unsigned) | 1 Byte | 0 to 255 |
| short (signed) | 2 Bytes | -32,768 to 32,767 |
| short (unsigned) | 2 Bytes | 0 to 65,535 |
| int (signed) | 4 Bytes | -2,147,483,648 to 2,147,483,647 |
| int (unsigned) | 4 Bytes | 0 to 4,294,967,295 |
| long (signed) | 4 or 8 Bytes | See here, very big |
| long (unsigned) | 4 or 8 Bytes | 0 to $2^{64}$ if 8 bytes |
| pointer | 8 Bytes | 0 to $2^{64}$ (all addressable memory) |

Note: long sizes differ by architecture, in CS300 assume 8 byte longs in C. Signed/unsigned means inclusive of negatives or noninclusive. See section 4 for more on signedness.

You may see something a little unfamiliar here—that being a "pointer," along with the conspicuous absence of Strings. Don't worry, we'll get there soon. What you need to understand right now is that all memory is the same, memory types are determined by the size of memory blocks allocated to store some data, and each memory type has a certain range of values it can store.

## 1.3 What's a Pointer?

We can store more than just data types like ints, longs, floats, and shorts in our 8-byte memory boxes—we can also store the addresses of other memory! To do this, we use pointers.

**Pointers are 8-byte segments of memory that hold other memory addresses within them.** The name "pointer" comes from thinking of these memory boxes as containing arrows pointing to other memory boxes. *Pointers must also be stored in memory, and take up 8 bytes per pointer.*

### 1.3.1 Thank your lucky stars (dereferencing)

In C, we use two operators with pointers. The ampersand, &, which gives the address of an item of data and the asterisk, *, which can mean two things:

1. An asterisk after a data type refers to a pointer, the type of data it points to depending on the type before the asterisk (e.g. an int* points to an integer)
2. An asterisk before a pointer 'dereferences' that pointer, giving us access to the information stored in the address it points to.

Pointers can be useful in C, which is a pass-by-value language, to pass memory between functions without duplication (among many other uses).

### 1.3.2 Array Pointer Duality

Arrays are not what they seem...

In reality, **arrays in C are silently replaced with a pointer to the first element of an array in memory**—this means that in an array of 10 items, say 10 chars, the name itself for the array is actually a pointer to the start of a 10-byte region of memory with ten 1-byte blocks sequentially allocated in memory.

**Strings are a lie too!**—a String is actually just a char array of however many characters are in the string + 1. Why the extra one? We reserve this spot for the null bit, indicating the location where a string terminates.

```
import<stdio.h>

int main(void) {
    // myArr actually decomposes to a pointer to the beginning of a
    // 40-byte segment of memory.
    int myArr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    // This means that if we define a pointer p
    int* p;
    // And assign it to myArr
    p = myArr;
    // C goes behind the scenes and decomposes the name myArr
    // to something like
    p = &myArr[0];
}

```

The other tool you likely have or will use in class provided by array-pointer duality is pointer arithmetic.[1] In case you need a reminder of how to use pointer arithmetic, we use the equation

$$p[i] = *(p + (i \cdot \text{sizeof}(p)))$$

To navigate memory, where i is the number of steps we want to move forward in memory (or the index), p is the pointer we're performing pointer arithmetic on, and sizeof($p$) is the size of the data type we're performing pointer arithmetic on. C will, by default, include the size of your unit, so unless you're incrementing by a different data type's size (which you may need), you can only include $*(p + i)$.

Table 2. Pointers: What to Remember

| Operator + Name Combination | Meaning |
|---|---|
| &<name> | The address of the item with a given name |
| <type>* <name> | A <type>pointer named <name> |
| *<name> | Dereferencing a pointer |

[1] Alex actually used pointer arithmetic in his entire initial implementation of DMalloc, only realizing he could use C arrow notation a few days before the due date. Please don't do this!

# 2 Placing things (Where does it all go?)

Computers have very purposeful rules not only for how memory is used (you'll notice that *purpose* is a common theme in this guide), but where it is used. In the following section, we'll quickly cover the different sections of memory, their uses, and how segments of memory are aligned for efficient use.

## 2.1 Sections of Memory

Your computer memory is separated into sections, each with different use cases. Why is this? There are many reasons:

1. For organization and consistency - Your computer can know where to go to find certain variables, values, etc.
2. For safety of memory - Isolating segments of memory makes sure processes that shouldn't have access to certain memory remain contained.
3. For performance - Again, reduction of searching Scoped
4. For internal fragmentation - Related to performance, see here

The four segments of memory are the **Code (or Text), Data, Stack, and Heap**. The uses of these segments are as follows:

1. The **Code/Text** is the section of memory that includes *executable instructions* and *constant global variables*.
2. The **Data** is the section of memory that includes *global variables*.
3. The **Stack** is the section of memory that includes *local variables*.
4. The **Heap** is the section of memory that includes*dynamically allocated memory*.

## 2.2 Alignment and Why We Do It

As we discussed in section 1, different types of memory of different sizes are stored in "blocks" of bytes. Different types will have blocks of different sizes.

This is all well and good, but actual operations on memory in your computer are often done in CPU cache. This is a small, hyper-fast segment of memory used for frequent, small operations. In x86 systems, the CPU cache is generally 64 bits, but different systems may vary. What we call *alignment* is the manner by which your compiler aligns data in memory such that the CPU cache is optimally used.

Table 3. Alignment of Common Types in C

| Type | Size | Address restriction |
|---|---|---|
| hline char | 1 Byte | No restriction |
| short | 2 Bytes | Multiple of 2 |
| int | 4 Bytes | Multiple of 4 |
| long | 4 or 8 Bytes | Multiple of 4 or 8 |
| float | 4 Bytes | Multiple of 4 |
| double | 8 Bytes | Multiple of 8 |
| pointer | 8 Bytes | Multiple of 8 |

Note: long sizes differ by architecture, in CS300 assume 8 byte longs in C.

### 2.2.1 Alignment Rules for Common Types

In order to keep alignment efficient, the C compiler aligns data in memory dependent on the data's type. This is done automatically when compiler optimizations are turned on, but it's good to keep these rules in mind when programming in memory-unsafe languages.

For common types, alignment is relatively simple—common types must be aligned on a multiple of their size. Think of the boxes of bytes we use to represent memory—we want 4-byte integers to fall on multiples of 4, 8-byte longs to fall on multiples of 8, etc.[2]

### 2.2.2 Struct Alignment Rules

1.

[2]Keep in mind, these multiples will be in hexadecimal since memory is in hex.

# 3 Remember Me (Topics in Memory)

## 3.1 How does C allocate dynamic memory?

### 3.1.1 Regarding malloc()

### 3.1.2 Regarding free()

## 3.2 Common Issues

### 3.2.1 Avoiding the SEGFAULT Menace

### 3.2.2 Leak Prevention + Good Memory Practice

# 4 Defining Undefined Behavior

## 4.1 What is Undefined Behavior?

## 4.2 Common Areas for Undefined Behavior

## 4.3 Dealing with Undefined Behavior

# 5 Avengers, Assembly!

## 5.1 How Compiling Works

## 5.2 Reading Assembly

### 5.2.1 Assembly Cheat Sheet

# 6 Racks on Racks, Stacks on Stacks (How the stack works)

## 6.1 What is the Stack

## 6.2 Stack Organization

### 6.2.1 Special Registers

## 6.3 Stack Smashing

### 6.3.1 Buffer Overflows

# 7 Operating the System

## 7.1 What is an Operating System?

## 7.2 Privilege Separation/Memory Protection

## 7.3 Virtual Memory/Address Translation

## 7.4 Process Creation

### 7.4.1 The fork() syscall

### 7.4.2 The exec() syscall

# 8 Inter-Process Communication

## 8.1 Introduction to Multiprocessing

## 8.2 Methods of Inter-Process Communication

### 8.2.1 waitpid()

### 8.2.2 Let's talk Pipes

# 9 Beyond the Pipe: Concurrency, Parallelism, and Threads

## 9.1 Principles of Concurrency and Parallelism

## 9.2 Introduction to Multithreading

## 9.3 Race Conditions and Mutexes

## 9.4 Synchronization and Atomics

### 9.4.1 Bounded Buffer

## 9.5 Scoped Locks

# 10 Wrapping Things Up

## 10.1 Our History as Computer Scientists

## 10.2 Philosophical Reflections

## 10.3 Further Reading

# 11 Acknowledgements