Introduction to C

Joe Raskind

Compiled: August 04, 2026

What is a Program?

Introduction

> “
>
> *Computational processes are abstract beings that inhabit computers…People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells. — Harold Abelson and Gerald Sussman*

I can claim with absolute certainty that if you are reading these notes you have used a computer program before. I can claim with less certainty, although I think there's pretty decent odds considering undergraduates are often American citizens, that you have driven, or have been driven by, a car. I'd say there's a good chance that if that's the case you likely don't know all about the mechanics involved in make a car go: the inner workings of the combustion engine, the way the transmission changes the direction of rotation, how the break fluid transfers pressure from your foot to stop the wheels, etc. This is much the same how you may not have known anything about how when you boot up your laptop that there's an entire world of transistors that begin firing off to let you eventually launch your browser of choice to browse the internet.

It's fine to not know how a car works if you're not in the market to build cars. It's less fine to not know how a computer program works if you're in the market to make computer programs. The point of this course is to get you closer to understanding how a computer program works and to get you on the right track to build up your knowledge in the future.

Why Do We Program?

Computer programming is a practical effort in real-world problem solving. It is primarily an intellectual exercise involving the careful balancing of the rules of computation with the desired outcomes envisioned by the programmer. In the most obvious way, the fact that computers dominate our daily lives implies that they must have *some sort of use value*, i.e. there must be a reason to program them. When asked, sophomore computer science students often come up with the following points:

- To **automate** tasks
- To **simplify** tasks
- To **store** the results of tasks

Once a computer program has been written and transformed into a runnable state, a human being no longer has to perform the same mental calculus that was needed in order to write it in the first place. They simply run the program and record the results (or maybe the results are recorded for them!). For example, the first computer programs were written to help speed up ballistics calculations for naval warfare—all the user needed to do was provide information about the variables involved and the machine would perform all necesarry computations.

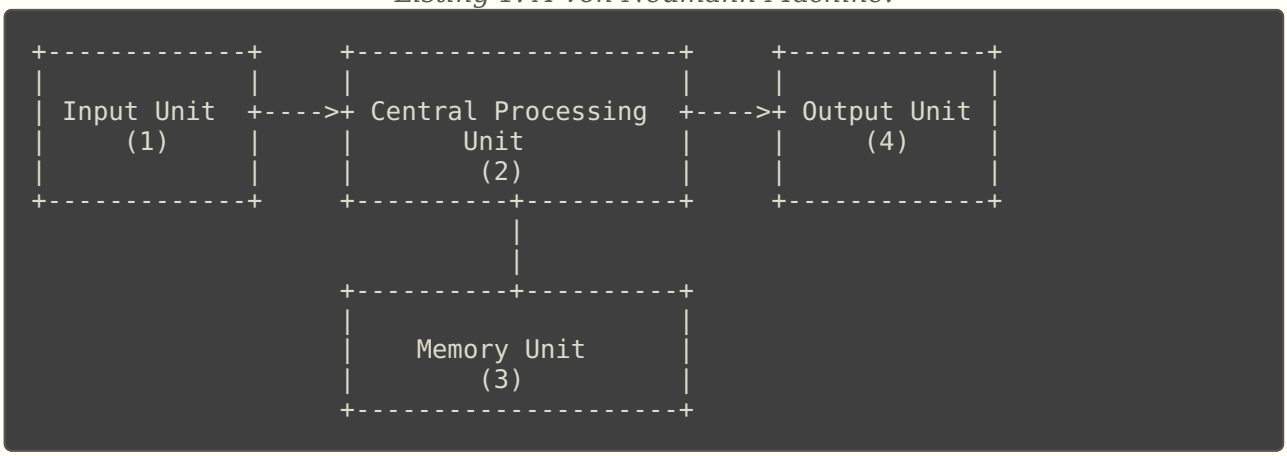
What Are the Components?

Nearly all modern day computers follow a design philosophy which was established in the late 1940s: the Von Neumann Machine. Named after the Hungarian-American mathematician Jon von Neumann, the Von Neumann Machine is a logical abstraction of a computation machine which, in its simplest formulation, is constituded of just four parts:

1. Input Unit
2. Central Processing Unit
3. Memory Unit
4. Output Unit

These four "components" are often represented as such:

Listing 1: A Von Neumann Machine.



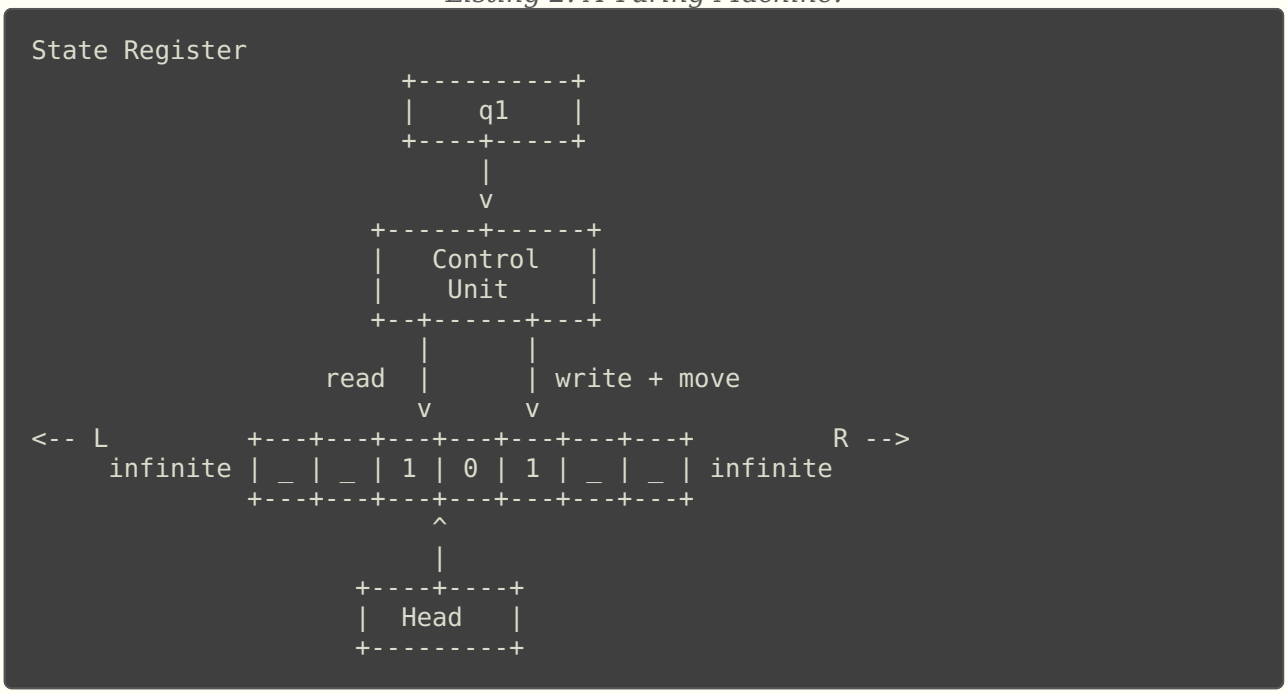
(1) Takes input from the user, without which we would have no way of interacting with the machine. (2) The "brain" of the machine that decodes instructions and carries them out. (3) Provides temoporary storage for larger computations. (4) Displays output to the user, without which we would have no way to get our results.

Real world machines are often more complicated than Listing 1, but more often than not they can still be reduced to the same abstraction from a programmer's point of view[1].

What's Special About the Von Neumann Model?

Von Neumann developed his famous abstraction with inspiration derived from another titan of computer science, Alan Turing. Turing in his watershed paper, *On Computable Numbers, with an Application to the Entscheidungsproblem*, gave the outline for a way to model computation in a pure, formal system. In it he states that "it is possible to invent a single machine which can be used to compute any computable sequence"—this powerful, abstract machine would later be named a Turing Machine. A diagram is shown below:

Listing 2: A Turing Machine.



The machine is quite simple, it takes input from tape and writes output to that tape. Its instruction set, i.e. it's *language*, dictates how the head will move around the tape. Some instructions move it to the left, others to the right; some instructions skip spaces, others write to spaces; etc. Turing's great insight was that this abstract machine could represent *every conceivable computation a human being could think up*. There is not a single actionable mathematical computation that can't be represented by a Turing Machine. Taken one step further, a Turing Machine can be constructed which takes as its input other Turing Machines. This "super" Turing Machine is called a Universal Turing Machine and it was what Von Neumann based his computer design on. There are theoretical limitations of the Turing Machine, but that is outside the scope of this class[2].

The uniqueness of the Turing Machine is that the tape does not distinguish between *instructions* and *data*, i.e. instructions can be manipulated *like anything else*.

How Do We Get the Instructions?

It should be clear now that in order to program a computer we must provide it with instructions. Here's an example you saw in the previous chapter:

Listing 3: A simple "Hello World" C program.

```
1:  #include <stdio.h>
2:  int main(){
3:    printf("Hello, World!\n");
4:  }
```

Listing 3 is a small program that prints `"Hello, World!"` out to the screen. But *how* does it print out to the screen? How does this strange, sentence-like structure turn into something the computer can "understand"? This is something we'll learn later in the course.

So, What is a Program?

A program is nothing more than a set of instructions which tells a machine what to do. They can be as complicated as Linux or as simple as a single HALT instruction. By the end of this course you'll be able to write C programs that fall somewhere in-between those two extremes.

Exercises

1. Think up more reasons one might program. Does it fall into one or more of the given categories?
2. Assume we have a Turing Machine with tape that has letters of the English alphabet as shown in Listing 4, further assume the head will point at *A* to start. Each time the head moves it will print the character it lands on. Write down the instructions needed to print out your first name.

Listing 4: Alphabet Tape.

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

1. Think about a computer program you use every day. While doing so, try to think about all of the steps that might be involved in getting that program to *actually run* on real hardware. (Try not to give yourself a migrane...)

Footnotes:

1. This, of course, varies depending on the sorts of programming one engages in. A software engineer dealing with high-minded business logic can think the computer "looks like" a few floating rectangles, but a computer engineer working on an embedded device will need an entirely different mental model. For this class, we'll stick to the high-minded *abstraction* provided by the Von Neumann machine. At least for now…

2. One such impasse being the "Halting Problem". It is impossible to construct a Universal Turing Machine capable of knowing whether or not an incoming Turing Machine is going to "halt" (stop processing instructions).

UNIX

What is UNIX?

UNIX is an operating system (OS), or really a collection of OSs, which was originally developed at Bell Labs by in the Late 1960s by Ken Thompson and Dennis Ritchie[1]. UNIX was developed to be a successor to MULTICS which was the first time-sharing OS, also created by Thompson and Ritchie. UNIX caught fire because it was the first[2] portable OS written—this was because it was written entirely in *C*.

Note: Dennis Ritchie's name might sound familiar. This is because he is the Ritche from Kernighan and Ritchie's *C Programming Language* book!

What is an Operating System?

In order to understand what UNIX is, we first have to understand what an Operating System is in the first place. Simply put, an operating system is a *special program* that operates as a sort of middle man between the user and the hardware. Prior to the development of operating systems, users would interact with computers by submitting programs for execution *one-at-a-time*. If Alice needed neutron diffusion calculations and Bob needed to simulate blast wave propagation, each would have to write a separate program and wait in line to submit it. Once Alice's program finished, Bob's program could run. Since computers prior to the explosion of integrated circuits were expensive[3] users would have to contend with one another for access time. This creates an obvious problem: the owner of a computer needs to have multiple technicians on staff dedicated to handling program submission requests.

The first "operating systems" were little more than hoppers. A technician would take a stack of punchcards representing a program from a user and place it on the queue. Once a previous program finished calculating, the next would be taken off the queue. This too presents problems like: what happens when the program crashes? Or, what happens if the program stalls? Eventually, programmers decided to come up with programs that could solve these logistical issues. And thus the Operating System was born.

Modern OSs provide more than just the ability to queue up a few programs. They supply powerful abstractions that developers are constantly making use of, whether they recognize it or not. When we start programming in C we'll find we have to do very little besides write in C to get something runnable. This is unfathomable without the intervention of an OS.

What is the UNIX Philosophy?

The UNIX philosophy was outlined by Brian Kernighan in 1984's *The UNIX Programming Environment*, "The power of a system comes more from the relationships among programs than from the programs themselves". What he meant was that UNIX provides many useful abstractions to allow for communication between multiple processes. Instead of one super program which is in charge of many different aspects of the system, UNIX is structured in such a way to allow for many specialized programs to do simple jobs well. Later in this chapter we'll see some practical examples showcasing this behaviour.

Is Linux UNIX?

Most people no longer directly use UNIX like they did fifty years ago. Instead, we use OSs *heavily inspired by UNIX*. In this course we'll be using Linux. Linux itself isn't a singular OS, it instead refers to a family of OSs which more-or-less share a kernel (key subroutines in an OS). There are many different distros (or distrobutions) of Linux: Ubuntu, Debian, Red Hat, Arch, etc. Each distribution has its own flavour and goals, but all are *fundamentally Linux*.

Note: The name Linux is a portmanteau of the first name of its creator, Linus Torvalds, and UNIX. Linux + UNIX = Linux.

At Binghamton, we use Linux for a variety of reasons, but the most important is that it is open-source software. This means that anyone can take a look at the Linux source code. This makes it perfect for educational purposes! There's no guesswork when it comes to understanding how Linux works, we can simply take a tour through the actual code itself.

Using a UNIX-like Machine

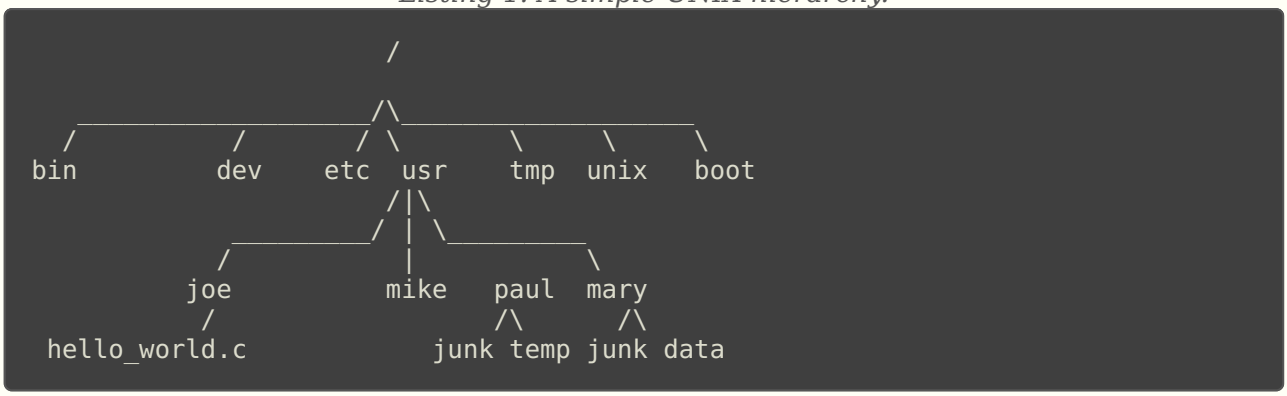
Most students who take this course have never used Linux before and have also never used the command line. The purpose of this portion of the chapter, and the first lab, is to familiarize you with some terminology and get you a bit more used to interacting with a UNIX-like system.

Note: Keep in mind, getting used to using the command line is not easy. *You will have to work at it the whole semester.* Take my word for it: it's worth the effort!

The Filesystem

Before we start looking at how to interact with data on Linux, we need to understand a little bit about the filesystem. A *filesystem* is an abstraction provided by the OS which organizes data stored in memory (often in secondary storage). This is not an OS course, so there's no need to get into *how exactly that works* here—we're just going to take for granted that it does.

Listing 1: A simple UNIX hierarchy.



In Linux, all data is stored as a *file*. A file is a sequence of bytes. Files will contain *meta-data*, or information about the file itself (size, location, permissions, etc.). Files are arranged in a hierarchy as shown by Listing 1. There are special files in the filesystem called *directories* which are files that contain other files. In Listing 1, `/`, `usr`, and `joe` are all directories and `hello_world.c` is a regular file.

The Shell

The *shell* is a program whose key purpose is to provide the user with direct access to the underlying system. There are many different possible shell programs that can be loaded onto a Linux/UNIX-like machine. You may or may not be familiar with `sh`, `bash`, `zsh`, `fish`, `pwsh`, etc. All of which have their own pros and cons associated with them. For this class we will be focused on using `bash` terminals[4].

A shell is a *command interpreter*. This means that when you're interacting with a shell program you are actually *programming*. Shell programs have their own languages that they use to understand the commands given to them by their users.

Note: Opening up a terminal will be different for different OSs. If you're on a Mac device, you can follow along by searching for the "Terminal" application. Most Linux distros

When you open up a shell it will look something like the following:

Listing 2: Freshly opened shell.

```
josephraskind@stargazer:~$
```

`josephraskind` is the user, `stargazer` is the name of the machine, and `~` is the shell's current working directory. The `$` indicates the beginning of the prompt, or the command that you'll be sending in. Try typing `ls` into the shell and hitting `ENTER`. Here's what it looks like for me:

Listing 3: Using ls.

```
josephraskind@stargazer:/tmp/example$ ls
homer
```

In Listing 6, I am in the subdirectory `example/` of the `tmp/` directory, which itself is a subdirectory of the root directory, `/`. What I see are the files contained within `/tmp/example/` which happens to be a directory called `homer`. You can probably guess what `ls` is supposed to do, but you can doublecheck your guess by using the command `man ls`. When I do so my terminal changes to show the following output:

Listing 4: Using man.

```
LS(1)                                                 User
Commands                                                LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information about the FILEs (the current directory by default).
Sort entries alphabetically if
       none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..
...
```

Note: You can exit `man` with the "q" key.

`man` shows the man page, or manual entry, for commands/programs installed on the machine. We can see here that `ls` is a program that lists information about files! You'll also notice that the man page lists a whole bunch of "options" that can be used to modify the behaviour of `ls`. Try out `ls -a` and see what it prints out. Here's mine:

Listing 5: Using ls -a.

```
josephraskind@stargazer:/tmp/example$ ls -a
.  ..  homer
```

You'll notice that `ls` now prints out two additional files: `.` and `..`. Believe it or not, those are also directories. Every single directory in Linux has those two directories contained within it. They represent connections to the current directory, `.`, and the parent directory, `..`. Watch what happens when I use `ls ../`:

Listing 6: Using ls.

```
josephraskind@stargazer:/tmp/example$ ls ../
example
```

We can change directories using the `cd` command:

Listing 7: Using cd.

```
josephraskind@stargazer:/tmp/example$ cd homer
josephraskind@stargazer:/tmp/example/homer$
```

My working directory has now changed from `/tmp/example/` to `/tmp/example/homer`. If I `ls` now I should see different files:

Listing 8: Using ls in homer/.

```
josephraskind@stargazer:/tmp/example/homer$ ls
iliad.txt  odyssey.txt
```

Note: `cd` is a *built-in command* whereas `ls` is a *program*. When `ls` is invoked the program must be turned into a *process* by the OS and loaded onto the CPU.

If I want to peak inside of the `iliad.txt` file I can use the `head` program:

Listing 9: Using head.

```
josephraskind@stargazer:/tmp/example/homer$ head iliad.txt
The Project Gutenberg eBook of The Iliad

This eBook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever. You may copy it, give it away or re-use it under the terms
of the Project Gutenberg License included with this eBook or online
at www.gutenberg.org. If you are not located in the United States,
you will have to check the laws of the country where you are located
before using this eBook.
```

Additional Functionality

Linux adheres to the UNIX philosophy by providing communication protocols that can be used to send information from one program to another. These come in two forms on the command line: *pipes* and *redirects*. Pipes *pipe* information from one program to another. Syntatically, we must place a `|` between the first in second program to indicate the direction of the dataflow. Here's an example where I count all of the lines in the *Iliad* where the word "water" is used:

Listing 10: Using a pipe.

```
josephraskind@stargazer:/tmp/example/homer$ grep -ie "water[ .,-;]" ./
iliad.txt | wc -l
12
```

First I use the `grep` program to look for instances of "water" then I take the output of that program and send it to `wc` which counts the number of lines in the text it takes as input!

Redirects can be used to send the output of a program that would normally be printed to the screen to go to a file instead. Here's an example where I use `echo` to print out a phrase to a new file:

Listing 11: Using a redirect.

```
josephraskind@stargazer:/tmp/example/homer$ ls
iliad.txt  odyssey.txt
josephraskind@stargazer:/tmp/example/homer$ echo "The Simpsons" > simpson.txt
josephraskind@stargazer:/tmp/example/homer$ ls
iliad.txt  odyssey.txt  simpson.txt
```

`echo` normally prints out to the screen, but the redirect tells the bash shell to make sure it goes to the new file simpson.txt instead.

Exercises

1. Using the `man` command, look up the `pwd` command. What does it do? Run it and describe the output.
2. Navigate to your home directory using `cd` and use `ls -a` to list all files including hidden ones. How many hidden files (files starting with .) do you see? What do you think they might be used for?
3. Create a directory structure that mirrors the filesystem diagram from Listing 1 — specifically the `usr` subtree with at least two user directories beneath it. Use `cd` and `ls` to verify your work.
4. Use `head` and `wc -l` together in a pipe to count the number of lines that `head` prints by default. What is that number?
5. The `cat` command prints the entire contents of a file to the screen. Use `echo` and a redirect (>) to create a file called `hello.txt` containing the text `"Hello, UNIX!"`. Then use `cat` to verify its contents.

6. What is the difference between `/usr` and `./usr`? Think about what `/` and `.` represent before answering.

Footnotes:

1 There were more, but Thompson and Ritchie often get the lion's share of the credit.

2 This is not strictly true, but it was the first portable OS to be shared around more than a local lab group.

3 For example, the ENIAC cost about $8.9 million in today's money (2026).

4 There are two reasons for this. (1) `bash` is what is installed on the remote servers. (2) `bash` is the terminal scripting language I am most familiar with. It also happens to be incredibly popular, so you don't lose anything learning a little bit about it.

Compilation & Execution

Introduction

In a previous chapter, we discussed a little bit about computer architecture. We learned that we can think of a computer as a machine which takes input, performs calculations based on that input, and returns output. What we're primarily interested in in this class is how we can write the "input" that the computer will use to perform calculations.

What is Machine Code?

Computers come in many different shapes and sizes, but, at base, they all follow the exact same design philosophy: there will be some internal unit which will read instructions to perform actions. In the case of a modern personal computer that philosophy is concretized by the Central Processing Unit (CPU). A familiar refrain is used by both the lowliest hacks and the holiest hackers in computer science: "*The CPU is the brain of the computer*". What they mean is that the CPU "*drives*" all of the components connected to the motherboard. All peripherals "listen" to what the CPU has to "say". But the CPU itself is "listening". What it's "listening" to are blocks of binary digits called words[1].

A typical, simplified CPU loop is as follows:

1. Fetch a binary word representing an instruction[2]
2. Decode that word to determine the desired operation
3. Execute the operation

This loop is performed for as long as the CPU is powered on.

All of these steps use a language tailored to the particular components of the given computer called *machine code*. This sort of code refers to the binary-to-instruction mappings provided by the instruction set manual of the target CPU. Below is a snapshot of the instruction set, in octal, that came with the CDC160A (the first microcomputer[3]):

OPCODE	F	E	G	Description
NOP	00	0X	----	No operation
HLT	77	00	----	Terminates
ADN	06	XX	----	Add XX to A
LDN	04	XX	----	Load XX into A
STM	41	00	YYYY	Store A into (i)YYYY

Where **F** is the Function Opcode, **E** is the "Execution Address", **G** is the second instruction word, **A** is the accumulator register, **X** is an octal operand, and **Y** is an octal address. So, if I wanted to perform the operation `x = 2 + 1` I could write the following:

Listing 1: CDC160A simple addition

```
04 02       # Load 2 into A
06 01       # Add 1 to A
41 00 0001  # Store A in memory (x)
```

In an unformatted binary blob it would look like:

Listing 2: CDC160A blob

```
000100000001000110000010100001000000000000000001
```

which is closer to what the machine will "see". It is very difficult for human beings to parse the latter example which is why practically no one writes machine code directly when programming[4]. Many people, however, write in what's called an *assembly language*.

What is an Assembly Language?

Assembly languages are one rung up the ladder of abstraction relative to machine code. They provide readable, symbolic abstractions for the pure binary strings directly processed by the CPU. Below is a snapshot of x86-64 assembly code:

Operation	Op1	Op2	Description
mov	src	dst	dst = src
add	src	dst	dst = dst + src

If I wanted to repeat the code from Listing 1 I could write:

Listing 3: x86-64 simple addition

```
1:  mov $2, %rdi  # Move constant 2 into register rdi
2:  add $1, %rdi  # Add 1 to register rdi
3:  mov %rdi, 0x1 # Store value of rdi into mem address 1 (x)
```

Here, instead of octal or binary values, I have symbols representing familiar words ("mov" for "move", "add" for addition, etc.). If I store the above code in a file called "simple_add.s", I can use an assembler to turn it into a binary file:

Listing 4: Using an assembler

```
1:  as simple_add.s -o simple_add.o
2:  objcopy -O binary --only-section=.text simple_add.o simple_add.bin
```

The first line invokes the GNU assembler, `as`, which turns the assembly code into an ELF file; the second line invokes objcopy which copies the contents of an object file to pure binary (the .text section is where code is stored in compiled files). If we try to directly `cat` out the **.bin** file we'll see a strange pseudo-binary string. This is because `cat` looks for UTF-8 encoded characters to print out to the screen, not unencoded binary strings! We can properly view the binary contents of the file with:

Listing 5: Hexdump command

```
xxd -b -c 1 simple_add.bin | awk '{printf("%s",$2)} END{print ""}'
```

Which should spit out the following:

Listing 6: Binary output

```
01001000110001111100011100000010000000000000000000000000010010001000001111000110000000101001000100010010011110000100101000000010000000000000000000000000
```

Wow! Over a hundred bits for only 3 short lines of assembly! Again, this should show you why it's not easy to program in machine code directly…but, even assembly looks a bit tough on the eyes. Both Listings 1 and 3 write more than one line of code to represent the logic of `x = 2 + 1`. The process that translates the `x = 2 + 1` to the machine code seen above is called *compilation*.

What is Compilation?

Put simply, compilation is the translation of one computer language to another (Language A → Language B). Compilation is a rich and complex topic in computer science, and therefore not one we can spend much time on in class; however, it is important that you are aware of the basic steps of compilation needed to turn C source code into a runnable executable. Compilation begins with the *compiler*.

What is a Compiler?

A compiler is a program which translates computer code from one language to another. All compilers roughly follow the below path[5]:

Listing 7: Compilation Stages


position = initial + rate * 60
Lexical Analyzer
<id,1> <=> <id,2> <+> <id,3> <*> <60>
Syntax Analyzer
=
<id,1> +
<id,2> *
<id,3> 60
Semantic Analyzer
=
<id,1> +
<id,2> *
<id,3> inttofloat
60
Intermediate Code Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator

```
              |
              v
    LDF   R2, id3
    MULF  R2, R2, #60.0
    LDF   R1, id2
    ADDF  R1, R1, R2
    STF   id1, R1
```

For this class we're mainly interested in the first three stages: lexical analysis, syntactical analysis, and semantic analysis. These represent three facets of a programming languge: its symbol set, structure, and rules. In other words, what "words" we can use, how we arrange those "words", and what those arrangements "mean". In the programming language world we group the first two under "syntax" and the one remaining under "semantics"—in reasonably complicated programming languages, like C, neither can be totally separated from one another.

Let's use an example in a language we all know very well: "The dog flew quite orangely this Mars." Each word exists within the symbol-set of the English language, i.e., they can all be found in a dictionary. Additionally, we can analyze the *structure* of the sentence:

- "The dog"
  - noun phrase acting as subject. "The" is a definite article modifying the noun "dog."
- "flew"
  - simple past tense intransitive verb, the predicate.
- "quite orangely"
  - adverb phrase acting as adverbial modifier of "flew." "Quite" is an adverb of degree modifying the manner adverb "orangely."
- "this Mars"
  - noun phrase acting as adverbial of time/place, modifying "flew." "This" is a demonstrative determiner modifying "Mars."

The sentence is well-formed, but it is completely devoid of *semantic meaning*—it's complete nonsense![6]

Much the same, I can construct lines of C code which appear to be correct, but have no actual place in the language: `"hello world" + 1`. *Syntactically*, I have placed two operands on either side of an infix operator, but *semantically* I've tried to add the number one to a constant that represents a string of characters. Compilers exist to catch errors of this nature.

GCC

As this is a C course we will be using a C compiler. There are many different C compilers available to use, but we choose `gcc` or the GNU Compiler Collection due to its popularity and wide availability on Linux platforms.

Listing 8: "hello world" program

```
1:  #include <stdio.h>
2:  int main(){
3:    printf("Hello, World!\n");
4:  }
```

For our purposes, using `gcc` is pretty easy[7]. If we have a C program, say `hello_world.c` (from Listing 8), we can compile it into an executable with the following command: `gcc hello_world.c`. `gcc` will automatically create an executable called `a.out` which can be run by specifying the path to it in the shell:

Listing 9: Compilation and execution of hello_world.c

```
josephraskind@stargazer:/tmp/hello$ gcc hello_world.c
josephraskind@stargazer:/tmp/hello$ ls
a.out  hello_world.c
josephraskind@stargazer:/tmp/hello$ ./a.out
Hello, World!
```

In the blink of an eye `gcc` expanded the `include` macros, went through every stage detailed in 7, called the assembler, `as`, and called the linker, `ld`! Each step in the process could be studied for weeks. We'll have to make do with occasionally referencing them when they're relevant.

What are Compilation Tools?

Compilation can be rather tricky to wrap one's head around, and it can get really overwhelming when you are working on a project that spans not just two files, but two hundred or two thousand files![8] This is where getting familiar with *compilation tools* can really pay off.

Make

Make is a popular compilation tool for C projects in Linux environments. It is yet another GNU tool which is often pre-installed in most Linux distros. Make has its own language, albeit one much more limited in scope than C, whose programs are stored in *makefiles*. The structure of a makefile looks a bit like this:

Listing 10: Makefile syntax

```
1:  VARIABLE = lorum
2:  target: dependencies
3:        commands
```

Let's assume I still have my file `hello_world.c`. I can write a makefile to compile my program as follows:

Listing 11: "hello world" makefile

```
1:   hello_world: hello_world.c
2:           gcc hello_world.c -o hello_world
```

I define the **rule** "hello world" (before the ":") which is dependent on the file `hello_world.c`. When the rule is triggered, make checks to see if `hello_world.c` needs to be compiled, i.e. "has it changed since the last compilation attempt?", then performs the set of commands defined beneath the rule. As long as there is a file named `makefile` or `Makefile` in my working directory I can invoke the file with a simple call to the `make` executable:

Listing 12: Calling make

```
josephraskind@stargazer:/tmp/hello$ make
gcc -Wall hello_world.c -o hello_world
josephraskind@stargazer:/tmp/hello$ ./hello_world
Hello, World!
josephraskind@stargazer:/tmp/hello$ make
make: 'hello_world' is up to date.
```

Note: You'll notice that a second call to make did not call `gcc`. This is significant for projects which have hundreds or thousands of C files which need to be compiled prior to producing an executable.

We can also add extra rules which perform various non-compilation tasks:

Listing 13: "hello world" makefile with clean

```
1:   hello_world: hello_world.c
2:       gcc hello_world.c -o hello_world
3:   clean:
4:       rm hello_world
```

This makefile has a new "dummy" rule which will remove the executable from the working directory:

Listing 14: Calling make clean

```
josephraskind@stargazer:/tmp/hello$ ls
hello_world  hello_world.c  makefile
josephraskind@stargazer:/tmp/hello$ make clean
rm hello_world
josephraskind@stargazer:/tmp/hello$ ls
hello_world.c  makefile
```

Note: `make` always calls the first rule by default!

We can use variables to further abstract our rules:

Listing 15: "hello world" makefile with variables

```
1:  FILE = hello_world
2:  FLAGS = -Wall
3:  CC = gcc
4:  $(FILE): $(FILE).c
5:      $(CC) $(FLAGS) $(FILE).c -o $(FILE)
6:  clean:
7:      rm $(FILE)
```

Now, instead of rewriting a makefile for every new single-shot file I write, I can just replace the "hello_world" at the top.

Note: Be careful when copy-pasting makefile rules, `make` expects a TAB before every command!

If you do enough C programming, you **will** encounter makefiles. It's best to get familiar with them now in a low(er) stakes environment.

Exercises

1. Using `gcc`, compile the hello world program from Listing 8 and run it. Then remove the quotations and see what happens. Reason about the output.
2. Write a makefile for the hello world program that includes three rules: one to compile the program, one `clean` rule to remove the executable, and one `run` rule that compiles and immediately runs it. What happens when you call `make run`?
3. Using the CDC160A instruction set from the chapter, write the machine code (in octal) to perform the operation `x = 5 + 3`. What would the equivalent x86-64 assembly look like?
4. Assemble the x86-64 addition code from Listing 4 using `as` and `objcopy`. Then use the `xxd` command from Listing 6 to dump the raw binary. How many bits does the output contain? Does this match what you'd expect?
5. Take the hello world makefile from Exercise 2 and convert it to use variables for the filename, compiler, and flags as shown in Listing 15. Then use it to compile a second program of your choice by only changing the variable at the top.
6. The chapter states that `cat` cannot properly display a `.bin` file. Try it yourself, run `cat simple_add.bin` after assembling. Describe what you see and explain why it looks that way.
7. The chapter mentions that `gcc` internally calls `as` and `ld`. Try compiling hello world manually step by step: first use `gcc -S` to produce assembly, then `as` to assemble it, then `ld` to link it. What errors or surprises do you encounter?
8. A makefile rule only recompiles when its dependency has changed. Compile `hello_world` with `make`, then run `make` again without changing anything. Now `touch hello_world.c` (which updates its timestamp without changing its contents) and run `make` again. What happens and why?

Footnotes:

1 Word size varies depending on architecture. For this class, we focus on x86-64 which has a word size of 64 bits. Instructions are not always word length.

2 Instructions will come from a component called the Instruction Cache, or I-Cache.

3 *A History of Modern Computing*, Paul E. Ceruzzi.

4 There are exceptions, but this is true in the broad strokes.

5 Diagram transcribed from *Compilers: Principles, Techniques, and Tools* by Aho et al.

[6] Noam Chomsky famously used the sentence "Colorless green ideas sleep furiously" in *Syntactic Structures*.

[7] You will likely not feel like that at first. That's okay! The more you practice, the more it will start to make sense.

[8] In this course we do not go beyond more than a few files in one project.

Variables

Introduction

Think back to your first time encountering an algebraic formula; say something as simple as $y = x + 1$. You likely learned that **x** is an unknown value, or at least a value which isn't known the moment you glanced at the formula. **x** is a stand-in, a signifier, a symbol, a name, i.e., something that *represents* the presence of some magnitude. **x** is a *variable*. The value attached to **x** is subject to change. The potential value attached to it is external to the formula itself.

Variables in Programming Languages

Variables within programming languages satisfy a similar, although not identical, position to variables within algebraic formulas[1]. Variables are *names* for data stored in the computer. When we write `x = 1` we are saying that `x` is a name which represents the computational object `1` which exists somewhere within the hardware of the machine running the program. As the name suggests, variables can change values. For example, we can write `x = x + 1` which increments the values of `x`.

Scope

All identifiers, including variable names, will have a *scope* associated with them. Scope has two facets in C: (1) the *lexical scope*; (2) the *linkage scope*. Here, we focus on hte *lexical scope* which refers to "the region of the program text within which the identifier's characteristics are understood"[2]. An example should be instructive:

Listing 1: Examples of scope

```c
//Block 0, Global scope
int g = 4;
int main(){
  //Block 1, Outer main method
  int x = 1;
  {
    //Block 2, Inner main method
    int y = 2;
    {
      //Block 3, Innermost main method
      int z = 3;
    }
  }
}
```

Listing 4 shows a C program which defines 4 `int` type variables: **g**, **x**, **y**, **z**, each one within a different code *block*. In C, braces, "{}", indicate the starting and closing of a code block, i.e., a collection of statements (you can think of statements as the sentences of programming languages). These blocks are the arbiter's of a variable's *lexical scope* (hereon referred to simply as "scope"). When a variable is defined in a given block, it can be "seen" by all statements/expressions within the same block as well as interior blocks. For example, **g** is visible at Block levels 0, 1, 2, *and* 3, despite it only being declared in Block 0. We call Block 0 the *global scope*—this refers to anything declared outside of a function block (such as the function `main`). Furthermore, **x** is visible at Block levels 1, 2, and 3, but *not* at Block 0. Every time a block is exited, all of the names defined therein are forgotten. To further emphasize this I will update Listing 4 so that **z** is referenced in Block 1 after it has been defined in Block 3:

Listing 2: Examples of scope

```
 1:  //Block 0, Global scope
 2:  int g = 4;
 3:  int main(){
 4:    //Block 1, Outer main method
 5:    int x = 1;
 6:    {
 7:      //Block 2, Inner main method
 8:      int y = 2;
 9:      {
10:        //Block 3, Innermost main method
11:        int z = 3;
12:      }
13:    }
14:    x = z + 1;
15:  }
```

When I try to compile this code, I get the following error:

Listing 3: Scope error

```
josephraskind@stargazer:/tmp/scope$ gcc scope.c
scope.c: In function ‘main’:
scope.c:14:7: error: ‘z’ undeclared (first use in this function)
   14 |   x = z + 1;
      |       ^
scope.c:14:7: note: each undeclared identifier is reported only once for each
function it appears in
```

Why does the compiler say that **z** is undefined? I clearly define it in Block 3, no?

Again, this has to do with the way scope works in C. The moment we exit a block, we lose the ability to reach all of the identifiers defined within that block. This means **z** effectively vanishes from the perspective of the compiler the moment we exit Block 3 (Line 12).

Scope rules also allows us to reuse old identifiers so long as we do not use the same identifier in the same scope. For example, take a look at the following code which defines **x** twice in `main`:

Listing 4: Examples of scope

```
1:   int main(){
2:     int x = 1; //Declaring x
3:     int x = 2; //Declaring x, again
4:   }
```

If I try to compile that code:

Listing 5: Redeclaration error

```
josephraskind@stargazer:/tmp/scope$ gcc dec.c
dec.c: In function ‘main’:
dec.c:3:7: error: redefinition of ‘x’
    3 |   int x = 2; //Declaring x, again
      |       ^
dec.c:2:7: note: previous definition of ‘x’ was here
    2 |   int x = 1; //Declaring x
      |       ^
```

I get an error that tells me I "redefined" **x**. This happens because all identifiers within a given scope must be *unique*. I cannot have two different memory locations tied to the same name, there must be a one-to-one mapping within a scope between name and memory location at all times. If I wanted to fix the above error, I would need to define each **x** in their own, mutually exclusive scopes (I will leave that as an exercise for the reader).

Note: Scope will become relevant again when we discuss functions!

What a Variable *Really* Is in C

In C, variables are not purely symbolic names. They are intimately tied up with the *memory semantics* of the C runtime system. They are truly just standins for *locations in memory*[3].

To illustrate this I will write a simple C program which declares two `int` variables and adds them together and stores the result in a third variable:

Listing 6: Adder

```
1:   int main(){
2:     int a = 100;
3:     int b = 200;
4:     int c = a + b;
5:   }
```

Then I will compile the program down to its assembly representation with `gcc -O0 -fno-omit-frame-pointer -fcf-protection=none -S add.c`[4]. We can see the resultant assembly file here:

Listing 7: Adder (x86-64 assembly)

```
1:          .file   "add.c"
2:          .text
3:          .globl  main
4:          .type   main, @function
5:  main:
6:  .LFB0:
7:          .cfi_startproc
8:          pushq   %rbp
9:          .cfi_def_cfa_offset 16
10:         .cfi_offset 6, -16
11:         movq    %rsp, %rbp
12:         .cfi_def_cfa_register 6
13:         movl    $100, -12(%rbp)
14:         movl    $200, -8(%rbp)
15:         movl    -12(%rbp), %edx
16:         movl    -8(%rbp), %eax
17:         addl    %edx, %eax
18:         movl    %eax, -4(%rbp)
19:         movl    $0, %eax
20:         popq    %rbp
21:         .cfi_def_cfa 7, 8
22:         ret
23:         .cfi_endproc
24: .LFE0:
25:         .size   main, .-main
26:         .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
27:         .section        .note.GNU-stack,"",@progbits
```

The `main` label shows where our program's code begins. We can see our constants clearly on Lines 13 and 14 whic must mean that `-12(%rbp)` and `-8(%rbp)` are our **a** and **b**, respectively. How can that be? It turns out that local variables, i.e. variables defined within a non-global scope, are placed on something called "the stack". It is nothing more than a location in memory which is pointed to by the `%rbp` register in x86-64. By the time we reach assembly language land, we no longer have symbolic names like **a** and **b**, but rather purely locations in memory managed by the C runtime constructed by the compiler.

Note: We'll learn more about this when we learn calling conventions for C functions!

Exercises

1. Look at Listing 4 again, and edit the code such that **x** is defined twice in `main`, but the compiler does not throw any errors.
2. Take Listing 4 and compile it with `gcc -O0 -fno-omit-frame-pointer -fcf-protection=none -S add.c`. Identify which lines in the assembly output correspond to each of the three variable declarations. What register is used as the base for all three variables? What is the offset of each variable from that register, and why do you think the offsets are multiples of 4?
3. Modify Listing 4 to add a fourth variable `int d = c * 2` and recompile to assembly. What new instructions appear? Has the offset pattern changed? What does this tell you about how the compiler allocates space on the stack for local variables?
4. Write a C program that declares a global variable `int g = 10` and a local variable `int g = 20` inside `main`. Print both. Does it compile? Which value gets printed and why? Now try to print the global `g` from inside the block that defines the local `g`—is it possible?
5. The chapter states that once a block is exited its variables "vanish from the perspective of the compiler." Write a program that demonstrates this

by attempting to access a variable from an inner block after that block has closed. Record the exact compiler error you receive and explain in your own words what it means in terms of scope.

6. Earlier in the chapter, I wrote the code snippet `x = x + 1` and referred to it as "incrementing" **x**. Outside of computer science, `x = x + 1` likely looks like a nonsense statement. From what you now know about variables, how is that possible in a programming language? You can see for yourself by writing a program that defines an integer **x**, assign it some value, say, 100 and increment it by, say, 200. Check out the assembly code produced by `gcc` to help aid your understanding.

Footnotes:

1 All subsequent references to "variables" will be references to "variables" in the computer science sense.

2 *C Programming Language - Second Edition*, K&R.

3 This will return when we talk about pointers.

4 `-O0` prevents `gcc` from performing optimizations, `-fno-omit-frame-pointer` forces `gcc` to show the frame pointer for all functions, `-fcf-protection=none` turns off branch protection, and `-S` stops compilation at the assembler.

Data

Introduction

> “
>
> *The world is the totality of facts, not of things — Ludwig Wittgenstein*

In order to perform computations we must have something to perform computations on. After all, what good is addition if there is nothing to add? In computing, we call these computational necessities "data". Conceptually, data is an information abstraction. This abstraction has analogues to phenomena in the real world—meaning data *represents something*. For example, I may have data in the form of numbers which represent the dimmensions of a shipping container, or I may have data in the form of words which represent the list of students who made the Dean's list, or I may have data in the form of a graph which represents related artists on a music platform, etc.

What interests us most in this class is not what data is philosophically, but rather how it is stored within a computer.

How is Data Stored in a Computer?

Fundamentally, data can be stored in any way a computer architect sees fit. The earliest computers stored numbers as *decimals* within their circuitry. For example, the ENIAC used what was called Binary Coded-Decimal (BCD) wherein 10 binary digits (bits) were used to represent a single value, `0000000001` for 0, `0000000010` for 1, ..., and `1000000000` for 9. It wasn't until the EDVAC that it became standard for computers to store numeric information purely in binary (`0` for 0, `1` for 1,..., and `1001` for 9). As late as 1968, of the 139 available general purpose digital computers, "46 were decimal, 72 binary, 4 octal, 15 binary/decimal, 2 biquinary"[1].

The switch to pure binary came from a desire to cut down on the amount of space taken up by numerical data. In the famous *Report on EDVAC*, John Von Neumann writes,

- "If one contemplates using a decimal system with either the Iconoscope or delay line memory one is forced into a binary coding of the decimal system—each decimal digit being represented by at least a tetrad of binary digits. Thus an accuracy of ten decimal digits requires at least 40 binary digits. In a true binary representation of numbers, however, about 33 digits suffice to achieve a precision of 10^10."

We'll talk more about memory later in the semester, but right now it will suffice to say that there must be something to "hold" our bits when we run out of register space—that is what we call *memory*.

In this chapter we'll answer two questions: "How are numbers stored in a computer?" and "How is text stored in a computer?".

How are Numbers Stored in a Computer?

In order to answer this, we first have to define what we mean by "numbers". In an effort to avoid any sort of philosophical detour, we'll assume a "common sense" definition of "number": numbers are objects used to measure phenomena. A human being has two eyes, one nose, ten fingers, etc. We may also want to distinguish between simple, countable numbers and infintesimal, continuous numbers. In C, these kinds of numbers are distinguished by having different *types* (we will talk about this extensively in the future). `int` matches to integers, or countable numbers, and `float` matches to reals, or continuous numbers.

Integers

When I write the C code `x = 17`, what is *actually* being stored inside the computer? The answer is that there will exist a set of binary digits which will be arranged to represent the idea of the number seventeen within the machine. A trip to the C specification will tell us that,

- "A 'plain' int object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range INT_MIN to INT_MAX as defined in the header <limits.h>)." (*ISO/IEC 9899:2024 Draft*, § 6.2.5)

This likely raises more questions than it answers. What is the "natural size" of an integer? How is anything about a computer natural? What is <limits.h>?

A lot of this has to do with the fact that C was made to be a language that could target hardware as best as possible by removing as much overhead as possible. Some systems may not benefit from an `int` being a fixed 4 bytes—for instance, smaller systems might want a 3 byte `int`. If I look in my `/usr/include/limits.h` file, I see the following:

Listing 1: int boundaries

```
1:  /* Minimum and maximum values a `signed int' can hold.  */
2:  #  define INT_MIN       (-INT_MAX - 1)
3:  #  define INT_MAX       2147483647
```

According to Listing 1, an `int` on my machine is between -2147483648 and 2147483647. Taking the log_2 of 2147483648 yields 31 (2^31 = 2,147,483,648) which means it takes 31 bits to encode that maximum

number. From what you already know about computers it might seem odd to default to 31 bits for an integer. Why not 32? The reality is that an `int` *does* default to 32 bits due to the use of *two's compliment*. In two's compliment the most significant bit is a *signed bit*, meaning its presence/absence indicates a negative/positive value. For example,

- 0000 0000 0000 0001 = 1
- 1111 1111 1111 1111 = -1

It can be represented by the formula:

- (d^n-1 * -2^n-1) + (d^n-2 * 2^n-2) +...(d^1 * 2^1) + (d^0 * 2^0)

where **n** is the number of bits.

This means that 17 is stored as `0000 0000 0001 0001`.

Reals

How about `x = 17.25`? Let's take another look at the spec,

- "There are three standard floating types, designated as float, double, and long double. The set of values of the type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set of values of the type long double." (*ISO/IEC 9899:2024 Draft*, § 6.2.5)

Somehow even more vague! Further on we find,

- "The float type matches the ISO/IEC 60559 binary32 format" (*ISO/IEC 9899:2024 Draft*, § F.2.1)

This means that a `float` follows the ISO/IEC 60559 standard which is equivalent to the IEEE 754 standard. It follows the following formula:

- (-1)^s + (1 + m) * 2^e
- Exponent bias = 127
- Exponent = e + bias
- m = fraction bits (start at 2^-1)

Listing 2: 17.25 in IEEE 754 Standard

```
  IEEE 754 Floating Point Specification

sign   exponent (8 bits)                           fraction (23 bits)
 |   |                    |                                          |
|                                                                     |
 v   v                    v                                          v
v                                                                     v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 1| 0| 0| 0| 0| 0| 1| 1| 0| 0| 0| 1| 0| 1| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 31  30                   23 22                        (bit
index)                                 0
```

First we see if the number is positive or negative, clearly it's positive so the signed bit is set to 0. Next, we need to represent 17.25 as a true binary number. We already know the non-fractional half is 10001. The fractional half is easy to get, all we need to do is follow the algorithm:

1. Set f equal to the fractional half.
2. Set f equal to f * 2.
3. If f >= 1 record a 1; if not, record a 0.
4. Repeat 1 until pattern emerges or we run out of bits.

.25 is rather simple, we multiply by 2 to get .5 (and tally a 0), then multiply by 2 to get 1 (and tally a 1). Since the remaining fractional half is 0, there's no further need to multiply. This gives us .01 in binary. Combine that with our 17 and that gives us `10001.01`.

Next we convert `10001.01` to scientific notation, `1.000101 * 2^4`. However many spaces we moved the decimal up is what we add to the bias. This gives us 127 - (-4) = 131 which is equivalent to 100000011 in binary. We then take the remaining bits after the decimal and store that in the fractional portion. Which gives us the IEEE 754 32-bit floating point number shown in Listing 2.

How is Text Stored in a Computer?

Text is where bit-based data representation and storage becomes much more abstract. There is an obvious direct mapping between representing the number 17 as decimal number and 10001 as binary number. It becomes much less obvious when trying to map 97 to 'a'.

One traditional way of representing characters is using ASCII encoding:

Char	Decimal	Octal	Hex	Binary
A	65	101	41	01000001
B	66	102	42	01000010
C	67	103	43	01000011

Char	Decimal	Octal	Hex	Binary
…	…	…	…	…
Z	90	132	5A	01011010

ASCII encoding using a single byte (really 7 bits, 0 to 127) to encode all of the english alphabet, including uppercase and lowercase letters, common punctuation, symbols and digits, alongside specialty characters like the null character '\0' and the new line character '\n'. For example, the character string "101" would be encoded as the binary string `001100010011000000110001` ("0" is 0x30 and "1" is 0x31).

ASCII encoding works well for the English language, but begins to fall apart the moment more than 128 characters are needed. For example, languages like Japanese or Chinese need far more symbols than the basic 26 required for English works (52 if counting uppercase letters). The most popular solution nowadays is to adopt newer encoding standards like UTF-8 or UTF-16 encoding.

UTF-8

UTF-8 is a character encoding standard based on the unicode standard. It is by far the most popular text encoding and has the ability to encode 974,530 unique symbols[2] (in reality, only a small fraction of that potential is realized). The encoding rules are simple:

First Code Point	Last Code Point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0zzzzzzz			
U+0080	U+07FF	110xxxyy	10yyzzzz		
U+0800	U+FFFF	1110wwww	10xxxxyy	10yyzzzz	
U+010000	U+10FFFF	11110uvv	10vvwwww	10xxxxyy	10yyzzzz

where the characters u to z, each representing a hexadecimal digit, are replaced by their constituent four bits uuuu to zzzz, from the positions U+uvwxyz.

Say I wanted to encode the thumbs up emoji (👍), I would first need to look up its UTF-8 value. Using the link here, we can see the code is U+1F44D. That places the emoji squarely in the last row of the above table. First I need to convert the code to binary U+1F44D → `0 0001 1111 0100 0100 1101` such that it matches with `u vvvv wwww xxxx yyyy zzzz`. Next we need to match the letters with their bits in the template: `11110000 10011111 10010001 10001101`. In order to verify that this in C we should convert the binary into hexadecimal: `0xF0 0x9F 0x91 0x8D`. The following C program can be used to double check our work:

Listing 3: UTF-8 Checker

```c
#include <stdio.h>
int main() {
  char thumbs_up[] = {0xf0, 0x9f, 0x91, 0x8d, 0x0};
    printf("%s\n", thumbs_up);
    return 0;
}
```

Compiling and running it we get:

Listing 4: Thumbs up!

```
josephraskind@stargazer:/tmp/thumb$ gcc thumb.c -o thumb
josephraskind@stargazer:/tmp/thumb$ ./thumb
👍
```

Note: This only works if your terminal is UTF-8 encoded. This is almost always the case for Linux devices, but is not guaranteed for Windows systems (which tend to favor UTF-16 for historical reasons).

Hopefully, this gave you a bit more insight into what's going on behind the scenes when your using your computer!

Exercises

1. Write a C program that stores 2147483647 in a variable of type `int`, adds 1 to it, and prints it out. What do you see? Explain why. (You can print out numbers using `printf("%d\n", variable_name)`.)
2. Convert the number -17 to its 32-bit two's complement binary representation by hand. Verify your answer by writing a C program that stores -17 in an `int` and prints it out using `printf("%d\n", x)`. Then print it again using `printf("%u\n", x)` (unsigned). What do you see and why?
3. Convert 0.1 to IEEE 754 32-bit floating point by hand following the algorithm given in the chapter. What do you notice about the fractional part? What does this imply about storing 0.1 in a `float`?
4. The string `"101"` is given in the chapter as the binary string `001100010011000000110001`. Using the ASCII table from the chapter, encode your first name as a binary string by hand. Then verify by writing a C program that prints each character of your name alongside its decimal ASCII value using `printf("%c %d\n", c, c)`.
5. Using the UTF-8 encoding table, encode the character 'é' (U+00E9) by hand. Which row of the table does it fall in? Write a C program to verify your encoding using the same technique as Listing 3.
6. The chapter mentions that ASCII uses 7 bits (0 to 127) despite being stored in a full byte. What happens to the 8th bit? Look up "extended ASCII" and write a short explanation of how different systems took advantage of that extra bit, and why this caused compatibility problems that UTF-8 was designed to solve.

Footnotes:

1 *History of Binary And Other Nondecimal Numeration*, Anton Glaser.

2 According to Wikipedia.

Types

Introduction

In a previous chapter, I wrote that *data* is an essential "information abstraction" that falls under the category of "computational necessity". This idea was further developed when we looked at how data is stored in a machine, both as constant values and as *variables*. Now, we'll explore how different kinds of data are interpreted by the C programming language and the *rules* that govern them.

Type System

We now know that all data in a computer is stored as a collection of binary digits (bits). We also know that in C, if I want to store numeric information I can store it in something called an `int` variable. I could declare one such variable using the statement `int x = 65`, which says that **x** is a symbolic name for a location in memory holding the bits `1000001`. But we also know that `1000001` can be interpreted as the letter 'A' in ASCII. What determines how those bits are interpreted?

In programming languages, one way of managing the interpretation of data is through the construction of a *type system*. In Benjamin Pierce's seminal book, *Types and Programming Languages*, he provides a possible definition, **"a type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."** This is a highly technical and accurate, albeit a bit bureaucratic, definition of a type system. I can give a bit more of a looser, intuitive definition,

- **A type system provides the *rules* for different kinds of data.**

This means that type systems lay out all of the ways that data can interact during a program's execution.

Nearly every high-level programming language in use today[1] has a type system. Type systems provide a variety of benefits to programmers:

- Error checking
  - Converting a string into a number before taking the square root
- Maintenance
  - Update the definition of a data type and its updated everywhere
- Enforces contracts
  - Guarantees behaviors of two variables
- Documentation
  - Easier to read code when types are involved

- Language safety
    - A safe language is one that protects its own abstractions

The C programming language has a type system and it is important you understand how it works.

C is *Statically* Typed

Firstly, C is *statically* typed. This means that every single variable and function within a program must have a type assigned to it at compile-time. Here is an example of a small C program:

Listing 1: Variable declarations

```
1:  int main(){
2:    int i = 10;
3:    float f = 10.5;
4:    const char * s = "Hello, World!";
5:  }
```

In order for the above program to compile, variables **i**, **f**, and **s** must all have *types* associated with them. **i** is declared with the type `int`, **f** with the type `float`, and **s** with the type `const char *`. Here's what happens if I remove the `int` from line 2 in Listing 1:

Listing 2: Removing int type

```
josephraskind@stargazer:/tmp/types$ gcc types.c
types.c: In function ‘main’:
types.c:2:3: error: ‘i’ undeclared (first use in this function)
    2 |   i = 10;
      |   ^
types.c:2:3: note: each undeclared identifier is reported only once for each
function it appears in
```

We can see in the error message from Listing 2 that something went wrong in the compilation process. The error message, however, looks a bit strange. It says that **i** must have been *undeclared*. Again, this is because all variables in C must be declared with a type—any reference to a variable without a type identifier behind it immediately tells the compiler that the function is *not being declared*, but is being used.

The compiler does not just check to see that every variable/function has a type, but it also checks to make sure that variables/functions follow the rules of their declared types. For example if I alter Listing 2 to:

Listing 3: Invalid type assignment

```
1:  int main(){
2:    int i = 10;
3:    float f = "Hello, World!";
4:    const char * s = "Hello, World!";
5:  }
```

When I try to compile the code I get the following message:

Listing 4: Compilation error from Listing 3

```
josephraskind@stargazer:/tmp/types$ gcc types.c
types.c: In function ‘main’:
types.c:3:13: error: incompatible types when initializing type ‘float’ using
type ‘char *’
    3 |   float f = "Hello, World!";
      |             ^~~~~~~~~~~~~~~
```

This happens because the type assigned to the expression on the right side of the `=` is not the same as the type assigned to the variable on the left side (and cannot be implicitly converted).

C is *Weakly* Typed

Although C is *statically* typed, it's often not considered *strongly* typed. This is because it is very easy to "break" the type system. For example if I slightly change the sort of alteration performed in Listing 3, to the following:

Listing 5: Valid type assignment?

```
1:  int main(){
2:    int i = "Hello, World!";
3:    float f = 10.5;
4:    const char * s = "Hello, World!";
5:  }
```

Then I compile it:

Listing 6: No compilation error from Listing 5

```
josephraskind@stargazer:/tmp/types$ gcc types.c
types.c: In function ‘main’:
types.c:2:11: warning: initialization of ‘int’ from ‘char *’ makes integer
from pointer without a cast [-Wint-conversion]
    2 |   int i = "Hello, World!";
      |           ^~~~~~~~~~~~~~~
```

I dont get an error?

Note: Technically, the above does not "break" the type system, but it does break reasonable expectations from the programmers perspective. It seems nonsensical to be able to assign an integer to the string constant "Hello, World!". In reality, that's not exactly what's happening...

C Types

We will now go through, almost, all of the different types in C. Starting with the basic types.

Basic Types

C defines four basic types:

1. char
2. int
3. float
4. double

char

According to the ISO/IEC 9899:2024 Standard, "An object declared as type char is large enough to store any member of the basic execution character set". It also states that "the representation of each member of the source and execution basic character sets shall fit in a byte". This means that a char must be one byte large.

Note: This *does not mean* that a char must be 8 bits in size! As a "byte" is defined in the standard as an "addressable unit of data storage large enough to hold any member of the basic character set of the execution environment". In x86-64 systems, it so happens that a char is actually one byte large (8 bits).

int

We went on a long investigation of ints in the data chapter where we learned that an int represents whole numbers between a range defined by the given machine. In x86-64 systems an int is 4 bytes long.

float

Similarly to ints, we looked at floats in the data chapter where we learned that a float is an IEEE 754 32-bit rational number.

double

Doubles follow the same standard as floats, except that they are 64-bit instead of 32-bit.

Signed/Unsigned Types

All integer types (char, int) can be explicitly assigned as "signed" or "unsigned" numbers. By default both types are considered signed. Here is an example:

Listing 7: Signed and unsigned integers

```
1:  #include <stdio.h>
2:  int main() {
3:      int s_i = 0xFFFFFFFF;
4:      unsigned int u_i = 0xFFFFFFFF;
5:      printf("signed   / 2: %d\n", s_i / 2);
6:      printf("unsigned / 2: %d\n", u_i / 2);
7:  }
```

If I compile and run the above program I get:

Listing 8: Results from running Listing 7

```
josephraskind@stargazer:/tmp/types$ gcc signs.c -o signs
josephraskind@stargazer:/tmp/types$ ./signs
signed   / 2: 0
unsigned / 2: 2147483647
```

The signage meaningfully changes computations during the program's execution.

Type Conversion

As shown in Listing 6, sometimes mixing and matching types is *legal* in the language despite intuition pointing the opposite direction. This sort of type fluidity is called type conversion. One data type is converted to another based on a set of rules by the language specification. There are two kinds of type conversion: explicit and implicit.

Explicit Conversion

Explicit type conversion occurs when the programmer uses direct syntax to indicate the change from one type to another. Here is an example:

Listing 9: Explicit type conversion

```
1:  int main(){
2:    int i = (int) 10.5;
3:  }
```

What we see on Line 2 is an explicit type conversion from the constant 10.5's `double` type to an `int`. Placing the name of the desired type in parentheses prior to the expression explicitly tells the compiler to convert the `double` to an `int`—this is called a "cast". In Line 2 we cast the double 10.5 to the int 10.

Implicit Conversion

Implicit type conversion occurs in situations where a type issue arises, but can be resolved with the compiler's intervention between compatible types. An example:

Listing 10: Implicit type conversion

```
1:  int main(){
2:    int i = 10.5;
3:  }
```

Listing 10 works exactly the same as Listing lst:explicit even though there is no explicit cast. The compiler recognizes that there is a type compatibility between floats and ints as they are both numeric types (*arithmetic* types in the spec).

Conversion Rules

As we saw above, the C programming language allows for conversions, both implicit and explicit, between data types. In order for this to be deterministic the language specification details exactly how certain conversions are meant to be achieved/implemented by the compiler. You are not responsible for memorizing all of the edge cases provided by the language specification for handling type conversion, but you should be aware that it can happen. Some rules you should be aware of:

- When converting from real values to integer values the fractional portion is always discarded
- When a larger data type is converted to a smaller data type (i.e., int to char), the least significant bits are retained.
- "Narrower" types will be automatically converted to "wider" types
    - `float` + `int` → `float` + `float`

Type Qualifiers

C also provides the ability for programmers to assign "extra" rules to a variable by providing type *qualifiers*. As of C11 there are four type qualifiers: `const`, `volatile`, `restrict`, and `_Atomic`. In this course, we will only concern ourselves with `const`. The const qualifier turns the variable into a "constant", meaning once it has been assigned a value, that value cannot be changed. For example:

Listing 11: const qualifier

```
1:  int main(){
2:    const int i;
3:    i = 10; // No problem
4:    i = 20; // Problem!
5:  }
```

The above code will not compile because the integer **i** cannot be assigned a second value. You may think that it's odd that C provides the ability to write constant values. After all, why not just write 10 for ever instance of **i**? The reason is that sometimes you want to change constants when testing your code. For example, if I'm testing to see how many students pass CS211 it's easier for me to change a constant variable `const PASS_SCORE` than it is for

me to individually change it throughout my entire source code file. This is doubly true if the constant is propagated throughout dozens of files.

Exercises

- The chapter states that narrower types are automatically converted to wider types in mixed arithmetic expressions. Write a C program that divides two `int` variables where the result has a fractional component (e.g. `7 / 3`), stores it in a `float`, and prints it. Does the fractional part appear? Now try storing the result of `(float) 7 / 3` in a `float` and print again. Explain the difference in terms of implicit and explicit conversion.
- Write a C program that declares a `const int` representing the passing grade for this course (60). Attempt to change it after declaration and record the compiler error. Then remove the `const` qualifier and confirm it compiles. Why might using `const` be preferable to just writing the number 60 throughout your code?
- Write a C program that explicitly casts each of the following and prints the result: `(int) 3.9`, `(int) -3.9`, `(char) 65`, `(float) 7 / 2`, `(float) (7 / 2)`. Before running, predict each result. Explain any surprises.
- The chapter states that when converting from a larger type to a smaller type, the least significant bits are retained. Write a program that assigns `int i = 256` to a `char c` via explicit cast and prints both. Explain the result in terms of bit representation.
- `sizeof` is an operator that will report the size of a given type. Use `sizeof` to print the size in bytes of a `char`, `int`, `float`, and `double` on your machine. Do they match what the chapter states?
- Listing lst:surprisingly_valid_assignment assigns a string constant to an `int` and produces a warning rather than an error. Run this program and print the value of `i` with `printf("%d\n", i)`. What value do you get? Based on what you know about how variables are stored in memory, reason where that value might come from. (We'll come back to this when we discuss pointers.)
- Write a C program that demonstrates implicit conversion in an arithmetic expression: declare an `int` and a `float`, add them together, and store the result in both an `int` and a `float` variable. Print all four variables. What do you observe about the results?
- The chapter mentions that C is considered *weakly* typed despite being statically typed. In your own words, explain the difference between static typing and strong typing using examples from the chapter.
- Using the `limits.h` header, print `INT_MAX`, `INT_MIN`, `CHAR_MAX`, and `CHAR_MIN`. Verify that the ranges match what you would expect given the bit sizes stated in the chapter.
- Write a C program that stores `0xFFFFFFFF` in both a `signed int` and an `unsigned int` and performs the following operations on each: addition of 1, division by 2, and multiplication by 2. Print all results with the correct format specifier ("%d" for signed integers, "%u" for unsigned integers). For each operation, explain why the signed and unsigned results differ.

Footnotes:

1 I'm writing this on 7/7/26. Exceptions I'm aware of are Brainfuck, which is intentionally esoteric, and BCPL which was the progenitor to C—neither of which are seriously used.

Operators

Introduction

Up to this point, we've discussed data, how that data is stored in variables, and how those variables must all have types. This gives us everything we need in order to understand how we can write C code to make these typed variables *interact* with one another. These interactions can be effaced through the use of *operators* in C, i.e., special symbols which have semantic meaning for computations involving data.

Infix Notation

When language designers begin the long, exciting process of building out a language, they have to make a choice on how exactly that language is going to look. One of the first considerations is what sort of notation will be used for parsing operations: prefix, infix, or postfix. A quick demonstration will show what I mean:

- Prefix → `+ 2 1`
- Infix → `2 + 1`
- Postfix → `2 1 +`

All of the above are ways of representing the semantic action of "add together the constants two and one". For the programmer, the difference is purely visual. For the language designer, or, more accurately, the compiler developer, the difference will manifest itself in how the language is *parsed*[1]. (We'll come back to this when we discuss stacks and queues.) For now, all we need to know is that C uses infix notation for all of its binary operators.

Operators in C

Here, I will enumerate all of the operators in C and provide examples for how each one operates.

Assignment

Assignment operators are used to capture values and place them into variables.

=

= is the basic assignment operator. The left operand must be a *lvalue* while the right operand must have a *rvalue*. For the following examples, assume `int x` and `int y` have already been declared"

- `x = 1` → `1` is now stored in the memory location tied to `x`
- `x = y` → The value tied to the memory location of `y` at the time of evaluation is now stored in the memory location tied to `x`

Note: Without getting into too much theory, *lvalues* are locations in memory (variables, or pointer dereferences) whereas *rvalues* are data (which can include memory locations). The variable `x` returns an *l* or *r* value based on its surrounding context.

Arithmetic

Arithmetic operators are only capable of being used with arithmetic types (char, int, float, double, and their variants[2]).

+

+ is the addition operator. It is used to add two numeric values.

- `4 + 2` → `6`

-

- is the subtraction operator. It is used to subtract two numeric values.

- `4 - 2` → `2`

Note: `-` is also a unary operator to indicate a negative number. For example, `-2 + 4` will also return `2`.

*

* is the multiplication operator. It is used to multiply two numeric values.

- `4 * 2` → `8`

/

/ is the division operator. It is used to divide two numeric values.

- `4 / 2` → `2`

Note: When two integers are involved integer division is performed. Functionally, that means the fractional result of the operation is removed. `2 / 4` equals `0`.

%

% is the modulo operator. It is used to get the remainder after the left operand is divided by the right operand.

- `4 % 2` → `0` (`2` goes into `4` two times with `0` remainder)
- `2 % 4` → `2` (`4` goes into `2` 0 times with `2` remainder)

Implicit Conversion, Again

Something that often trips students up is the fact that implicit conversion kicks in during the use of binary arithmetic operators. For example,

- `1 + 1.5` → `double`
- `1 + 1.5f` → `float`
- `1 / 2.0` → `double`

And so on. In all of the above cases `1` is converted to the *wider* type.

Relational

Relational operators, like arithmetic operators, are only capable of being used with arithmetic types. They are used to perform comparisons between two values. All relational operations return either a 1 for *true* or a 0 for *false*.

>

> is the "greater than" operator. It is used to check if the left operand is greather than the right operand.

- `2 > 4` → `0`

<

< is the "less than" operator. It is used to check if the left operand is less than the right operand.

- `2 < 4` → `1`

>=

>= is the "greater than or equal to" operator. It is used to check if the left operand is greater than or equal to the right operand.

- `2 >= 4` → `0`

- `4 >= 4` → `1`

<=

<= is the "less than or equal to" operator. It is used to check if the left operand is less than or equal to the right operand.

- `2 <= 4` → `1`
- `4 <= 4` → `1`

Equality

Equality operators too are only capable of being used with arithmetic types. They are used to check equality between two values. All equality operations return either a 1 for *true* or a 0 for *false*.

==

== is the equality operator. It is used to check if the left operand is equal to the right operand.

- `2 == 4` → `0`
- `2 == 2` → `1`

Note: Strict equality is dangerous when comparing floating point values! Relational operators should be used instead when dealing with floating points.

!=

!= is the inequality operator. It is used to check if the left operand is not equal to the right operand.

- `2 != 4` → `1`
- `2 != 2` → `0`

Logical

Logical operators can only take scalar values as input. All reals will be implicitly converted to scalars.

&&

&& is the logical **AND** operator. It is used for boolean conjunction where any non-zero value is considered to be *true*.

- `1 && 0` → `0`
- `2 && -1` → `1`

Note: `&&` is guaranteed to be evaluated from left to right. This means that if the left operand is 0, the right operand will not be evaluated. This is often called "short circuiting".

||

|| is the logical **OR** operator. It is used for boolean disjunction where any non-zero value is considered to be *true*. Much like **&&**, **||** will short circuit if the left operand resolves to a *true*.

- `0 || 0` → `0`
- `2 && 0` → `1`

Bitwise

Bitwise operators require an integer type (this includes char, long int, and short int). They are used to perform "bit-twiddling", i.e. setting/unsetting bits in integer values.

&

& is the bitwise **AND** operator. Each bit in the resulting value is set if and only if each of the corresponding bits in the converted operands is set.

- `0xff & 0xf0` → `0xf0`
- `0xfc & 0xf3` → `0xf0`
- `0xff & 0x0f` → `0x0f`

|

| is the bitwise **OR** operator. Each bit in the resulting value is set if one of the corresponding bits in the converted operands is set.

- `0xff | 0xf0` → `0xff`
- `0xfc | 0xf3` → `0xff`
- `0xff | 0x0f` → `0xff`

^

^ is the bitwise **XOR** operator. Each bit in the resulting value is set if and only if exactly one of the corresponding bits in the converted operands is set.

- `0xff ^ 0xf0` → `0x0f`
- `0xfc ^ 0xf3` → `0x0f`
- `0xff ^ 0x0f` → `0xf0`

~

~ is the bitwise **NOT** operator. Each bit in the resulting value is set if and only if the corresponding bits in the operand is not set.

- `~((char)0xff)` → `0x00`
- `~((char)0xfc)` → `0x03`
- `~((char)0x0f)` → `0xf0`

Note: If I don't explicitly cast the hexadecimal constant to a `char`, the C compiler will keep it as an `int` and flip more bits than I wanted to show.

<<

<< is the left bitshift operator. From the standard: "The result of E1 << E2 is E1 left-shifted E2 bit positions; vacated bits are filled with zeros. If E1 has an unsigned type, the value of the result is E1 × 2^E2, wrapped around. If E1 has a signed type and nonnegative value, and E1 × 2^E2 is representable in the result type, then that is the resulting value; otherwise, the behavior is undefined."

- `0xf << 4` → `0xf0`
- `0x80000000 << 1` → `0x00`

>>

>> is the right bitshift operator. From the standard: "The result of E1 >> E2 is E1 right-shifted E2 bit positions. If E1 has an unsigned type or if E1 has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of E1/2^E2. If E1 has a signed type and a negative value, the resulting value is implementation-defined."

- `0xf0 >> 4` → `0x0f`
- `0xf >> 4` → `0x00`
- `0x80000000 >> 1` → `??`

Note: There are certain edge cases in the C programming language which produce *undefined behavior*. We'll learn a bit more about it when we get to pointers. For now, think of it as something that is impossible for progammers to know without reading compiler documentation.

Operators in Other Programming Languages

It just so happens that C-style syntax is rather popular and many of the symbols chosen by C are used in other programming languages. For example, almost all of the operators and their symbols in C are identical in Java. Even if they weren't, many programming languages share the exact same *logic* and abstractions that C does (at least in the general sense). Both in Python and in

C, `2 * 3` represents the multiplication of the integers 2 and 3. Knowing the ins and outs of one programming language prepares you for learning another!

Exercises

1. Predict the result of each of the following expressions before writing a C program to verify them. Pay close attention to integer division:
   - `7 / 2`
   - `7 / 2.0`
   - `7.0 / 2`
   - `(float) 7 / 2`
   - `(float) (7 / 2)`
2. Write a C program that declares two `int` variables `a = 15` and `b = 4` and prints the result of all five arithmetic operators applied to them. What type is the result of each operation?
3. The chapter notes that strict equality is dangerous with floating point values. Write a C program that computes `0.1 + 0.2` and checks if it equals `0.3` using `==`. Print the result. Then print the raw value of `0.1 + 0.2` using `printf("%.20f\n", 0.1 + 0.2)`. Explain what you observe.
4. Write a C program that prints the results of all six relational operators (`>`, `<`, `>=`, `<=`, `==`, `!=`) comparing `a = 5` and `b = 5`. Predict each result before running.
5. The chapter states that `&&` and `||` short circuit. Write a C program that demonstrates short circuiting by placing a division by zero on the right side of a `&&` where the left side is `0`, and on the right side of a `||` where the left side is `1`. Does the program crash? Explain why or why not.
6. Using only bitwise operators, write a C program that:
   - Sets the 4th bit of `0x00` to 1 using `|`
   - Clears the 4th bit of `0xff` using `&`
   - Flips the 4th bit of `0xff` using `^`

   Print each result in hexadecimal using `printf("%x\n", result)`.
7. Write a C program that demonstrates left and right bitshifting on the value `0x01`. Left shift it 4 times, printing the result each time. Then right shift the final result 4 times back, printing each time. What do you observe? Express each result in both hex and decimal.
8. The chapter states that `1 + 1.5` produces a `double`. Write a C program that uses `sizeof` to verify the type widths and then performs the following mixed-type operations, printing each result with an appropriate format specifier:
   - `int + float`
   - `int + double`
   - `float + double`
9. Evaluate the following expressions by hand using the bitwise operator rules, then verify in C using `printf("%x\n", result)`:
   - `0xAB & 0x0F`
   - `0xAB | 0x0F`
   - `0xAB ^ 0x0F`

    - `~((char)0xAB)`
10. The chapter introduces three different notations for writing operations: prefix, infix, and postfix. Rewrite the following C expressions (which use infix notation) in both prefix and postfix notation:
    - `2 + 3`
    - `10 - 4`
    - `3 * 5`
    - `8 / 2`

Footnotes:

1 Most popular languages nowadays default to infix notation; however, there are still some languages which do not. For example, Lisp, a *relatively* popular functional programming language, and all of its various dialects use prefix notation. To my knowledge, Forth is the only programming language that anyone uses which sticks to postfix notation.

2 We'll find out later that includes pointers.

Expressions

Introduction

Up to this point, we've learned what variables are and how to use operations to perform computations using those variables. What we haven't explored are the fundamental units within the code that allow for the compiler to return the values of said computations. These units are called *expressions*.

Evaluation

A commonality across all programming languages is that *expressions return values*. Take, for example, the simple expression `3 * 2`. We know now that it will *evaluate* to `6` because the `*` operator represents multiplication and the constants `3` and `2` are equivalent to their decimal symbols. What might be less obvious is that there are actually *three* expressions in that one, simple expression:

1. The constant value 3 in the left operand.
2. The constant value 2 in the right operand.
3. The operation of multiplying the left and right operand.

This can be made more clear if we were to replace `3` and `2` with the integer variables `x` and `y`, giving us `x * y`:

1. The value of the variable `x` (what is stored at the memory location mapped to the symbol `x`) in the left operand
2. The value of the variable `y` (what is stored at the memory location mapped to the symbol `y`) in the right operand
3. The operation of multiplying the left and right operand.

In order for the multiplication operation to be successful, the runtime must *evaluate* what is stored in the variables.

Precedence

All expressions must be able to be evaluated *deterministically* and *unambiguously*. What do I mean by that? Well, take for example a more complicated set of expressions, `5 * 2 + 3 * 4`. How exactly should that be evaluated by the compiler? Grade school math teaches us that it should evaluate to 22, and it turns out that's also the case in C, but why is that? If I read the symbols from left to right it tells me I should multiply 5 by 2, then add 4, and then multiply by 4, giving me 52. This is because, without *precedence rules*, infix notation is inherently *ambiguous*—there's no way to

know how to order the expressions without external information about the symbols.

All programming languages that use infix notation must define precedence rules that govern expression evaluation order. It follows then that the C standard has exactly that:

Operators	**Associativity**
() [] -> .	left to right
! ~ ++ -- + - * (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Note: In the second row, `+`, `-`, and `*` are unary operators.

When using the above rules, there can be absolutely no abiguity about the expression `5 * 2 + 3 * 4`. We can see that binary `*` takes higher precedence than binary `+`. There are quite a lot of rules here, but most will become second nature after you get some experience under your belt.

Using Parentheses

Oftentimes, it can become a bit confusing to parse out how exactly a certain expression will be evaluated. Case in point, let's take a look at this monstrosity `1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8`. What we see here *is* valid C, and any working C compiler will be able to evaluate the given expression. In fact, I had Claude Sonnet generate a python script, which you can download here, to show exactly how that expression gets parsed:

Listing 1: Abstract Syntax Tree for `1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8`

```
TRANSLATION_UNIT: complicated_expr.c
    └── FUNCTION_DECL: main
        └── COMPOUND_STMT
            └── BINARY_OPERATOR [|]
                ├── BINARY_OPERATOR [&]
                │   ├── BINARY_OPERATOR [+]
                │   │   ├── BINARY_OPERATOR [+]
                │   │   │   ├── INTEGER_LITERAL [1]
                │   │   │   └── UNARY_OPERATOR [~]
                │   │   │       └── INTEGER_LITERAL [2]
                │   │   └── BINARY_OPERATOR [*]
                │   │       ├── BINARY_OPERATOR [/]
                │   │       │   ├── BINARY_OPERATOR [*]
                │   │       │   │   ├── INTEGER_LITERAL [3]
                │   │       │   │   └── INTEGER_LITERAL [4]
                │   │       │   └── INTEGER_LITERAL [5]
                │   │       └── INTEGER_LITERAL [6]
                │   └── INTEGER_LITERAL [7]
                └── INTEGER_LITERAL [8]
```

Visually, there's so much noise in the original expression that it's difficult for even a veteran C programmer to understand what's happening at first glance. This is where using parenthesis can be very helpful to the new C programmer. If I wrap the complicated expression in parentheses instead, it becomes more obvious what's going on:

- `(((1 + (~2)) + (((3 * 4) / 5) * 6)) & 7) | 8`

Parentheses can make things a bit more visually noisy, but many text editors, like Emacs, come preequipped with parentheses highlighting which makes the task a bit easier.

Expressions Contra Statements

Expressions can be constrasted with statements insofar as expressions always return values whereas statements do not. For example, an `if` statement, which we'll discuss next chapter, handles control flow, it does not return a value. This is in direct contradistinction with the `+` operator which will always return the value equal to the sum of the two operands on either side of it.

Exercises

1. Evaluate the following expressions by hand using the precedence table, then verify in C using `printf("%d\n", expr)`:
    - `5 * 2 + 3 * 4`
    - `5 * (2 + 3) * 4`
    - `10 - 4 / 2 + 1`
    - `10 - 4 / (2 + 1)`

2. The chapter identifies three sub-expressions in `3 * 2`. How many sub-expressions can you identify in `5 * 2 + 3 * 4`? List each one and the order in which they are evaluated according to the precedence table.
3. Download the AST builder script linked in the chapter and run it on a C file containing the expression `1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8`. Verify the output matches Listing 1. Then add parentheses to the expression to change the evaluation order and observe how the AST changes.
4. Write the fully parenthesized version of the following expressions as the compiler would evaluate them, then verify by computing the result:
    - `8 >> 1 + 1`
    - `1 | 2 & 3`
    - `!0 + !1 * 2`
    - `3 * 4 + 5 * 6 - 7 / 2`
5. The second row of the precedence table notes that `+`, `-`, and `*` appear as unary operators. Write a C program that demonstrates the difference between unary and binary `+`, `-`, and `*` by using each in context. What does unary `*` do?
6. Write a C program that declares `int x = 5` and `int y = 3` and evaluates `x = y = 2`. Print `x` and `y` after the assignment. Using the associativity column of the precedence table, explain why the result is what it is.
7. The chapter states that expressions always return values but statements do not. Classify each of the following as an expression or a statement and explain why:
    - `x + 1`
    - `int x = 5`
    - `x = x + 1`
    - `3 * 4`
8. Write a C program that evaluates `(((1 + (~2)) + (((3 * 4) / 5) * 6)) & 7) | 8` and the equivalent unparenthesized version `1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8`. Do they produce the same result? What does this confirm about the precedence rules?
9. Using the precedence table, determine which of the following pairs of expressions are equivalent without running them. Then verify in C:
    - `a + b * c` and `a + (b * c)`
    - `a & b | c` and `(a & b) | c`
    - `!a && b` and `(!a) && b`
    - `a = b = 0` and `a = (b = 0)`
10. The chapter gives `~1 + ~2 + 3 * 4 / 5 * 6 & 7 | 8` as an example of a confusing but valid C expression. Write your own complex expression using at least five different operators, write its fully parenthesized form, compute the result by hand, and then verify with a C program.

Conditional Statements

Introduction

Think about the last time you went to a grocery store. If it's anything like my local supermarket, when you approach the sliding doors to enter they appear to open as if they were operated by invisible elves. If we're able to reasonably convince ourselves that it isn't magical elves, there must be something present allowing for that movement to occur. Further investigation would have us finding a sensor which is able to detect whether some body is in front of the door and *if* that's the case the mechanism to open the door should fire. Furthermore, there is something preventing the door from opening *if* there is nothing detectable. The internal logic of the automatic door is *conditional* on the *state* of its components.

Programs too will have special logic that will only be used in the event a certain condition is met. In C, these logical scaffolds are called *conditional statements*.

Control Flow

Conditional statements are used to direct the runtime behavior of the executing program. Depending on how they are evaluated, conditional statements will decide which set of instructions are processed next. In computer science we call this the *control flow* of a program—i.e. how each instruction *flows* into the next. C provides a few different statements to aid in the construction of a program's control flow.

if Statement

The `if` statement is the fundamental conditional statement. It's syntax is as follows:

Listing 1: if statement syntax

```
1:   if(conditional)
2:     statement; //Do this
```

As long as `conditional` evaluates to a *true* value then the `statement` will be executed. A filled in example should illustrate the point:

Listing 2: Conditional example

```
1:  #include <stdio.h>
2:  const int X = _; // Will set later...
3:  int main(){
4:    printf("Welcome to the even checker!\n");
5:    if(X % 2 == 0)
6:      printf("%d is even!\n", X);
7:  }
```

If I compile and run this with `X` set to 4, I get the following results:

Listing 3: Listing 2 with X = 4

```
josephraskind@stargazer:/tmp/cond$ gcc if.c -o if; ./if
Welcome to the even checker!
4 is even!
```

What about if I set `X` equal to 5?

Listing 4: Listing 2 with X = 5

```
josephraskind@stargazer:/tmp/cond$ gcc if.c -o if; ./if
Welcome to the even checker!
```

What happened to my printout? Well, since the conditional failed the corresponding call to `printf` was not executed. The `printf` statement is attached to the `if` statement evaluating a *true* conditional—it will **never** execute if a *false* is evaluated.

Taking a look at the assembly generated by `gcc -O0 -fno-omit-frame-pointer -fno-stack-protector -S if.c` should help us understand what's happening:

Listing 5: Listing 2 as x86-64 assembly code

```
1:          .file   "if.c"
2:          .text
3:          .globl  X
4:          .section        .rodata
5:          .align 4
6:          .type   X, @object
7:          .size   X, 4
8:  X:
9:          .long   4
10: .LC0:
11:         .string "Welcome to the even checker!"
12: .LC1:
13:         .string "%d is even!\n"
14:         .text
15:         .globl  main
16:         .type   main, @function
17: main:
18: .LFB0:
19:         .cfi_startproc
20:         endbr64
21:         pushq   %rbp
22:         .cfi_def_cfa_offset 16
23:         .cfi_offset 6, -16
24:         movq    %rsp, %rbp
25:         .cfi_def_cfa_register 6
26:         leaq    .LC0(%rip), %rdi
27:         call    puts@PLT
28:         movl    $4, %eax
29:         andl    $1, %eax
30:         testl   %eax, %eax
31:         jne     .L2
32:         movl    $4, %eax
33:         movl    %eax, %esi
34:         leaq    .LC1(%rip), %rdi
35:         movl    $0, %eax
36:         call    printf@PLT
37: .L2:
38:         movl    $0, %eax
39:         popq    %rbp
40:         .cfi_def_cfa 7, 8
41:         ret
42:         .cfi_endproc
43: .LFE0:
44:         .size   main, .-main
45:         .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
46:         .section        .note.GNU-stack,"",@progbits
47:         .section        .note.gnu.property,"a"
48:         .align 8
49:         .long    1f - 0f
50:         .long    4f - 1f
51:         .long    5
52: 0:
53:         .string  "GNU"
54: 1:
55:         .align 8
56:         .long    0xc0000002
57:         .long    3f - 2f
58: 2:
59:         .long    0x3
60: 3:
61:         .align 8
62: 4:
```

There's a lot going on here. First, `.LC0:` and `.LC1:` represent the locations in memory for the two string constants used in the print statements. Second, there are two labels contained within the `main` function, `.LFB0:` which represents the entry point, and `.L2:` which represents the conditional jump destination. Line 26 loads the `"Welcome..."` string into the first argument register, and Line 27 call `printf`[1]. Lines 28-30 represent the conditional expression `x % 2 == 0`: 4 is loaded into **%eax** which is then given a bitwise **and** with 1 and is then tested to see if the result equals 0[2]. If the zero flag has not been set then control will be redirected over to `.L2:`, otherwise every further instruction will be processed. Therefore when `X` is equal to 5 control will be transferred ober to `.L2:` and the second call to `printf` will be passed over.

`if` statements can also encompass more than one statement. For example:

Listing 6: if statement syntax with blocks

```
1:  if(conditional){ // Start block with {
2:    statement 1;
3:    statement 2;
4:    ....
5:    statement N;
6:   } // End block with }
```

Applied to a "real code" expmaple we would see:

Listing 7: Conditional example with block

```
1:  #include <stdio.h>
2:  const int X = _; // Will set later...
3:  int main(){
4:    printf("Welcome to the even checker!\n");
5:    if(X % 2 == 0){
6:      printf("%d is even!\n", X);
7:      printf("Isn't that neat!\n");
8:    }
9:  }
```

Running the above with `X` equal to 10 we get:

Listing 8: Listing 7 with X = 10

```
josephraskind@stargazer:/tmp/cond$ gcc if.c -o if; ./if
Welcome to the even checker!
4 is even!
Isn't that neat!
```

And with `X` equal to 13:

Listing 9: Listing 7 with X = 13

```
josephraskind@stargazer:/tmp/cond$ gcc if.c -o if; ./if
Welcome to the even checker!
```

Note: Be careful when omitting "{}" from your conditional statements! Without them, C will only match the first statement to the conditional's control flow behaviour. It's good practice to ere on the side of caution and always include brackets when possible.

else if

`if` statements can be further modified to allow for mutually exclusive behaviors. For example, say we didn't want users to enter negative numbers we could write:

Listing 10: Checking for negatives with three /if/s

```
 1:  #include <stdio.h>
 2:  const int X = _; // Will set later...
 3:  int main(){
 4:    printf("Welcome to the even checker!\n");
 5:    if(X < 0){
 6:      printf("Ew! %d is negative!\n", X);
 7:    }
 8:    if(X % 2 == 0){
 9:      printf("%d is even!\n", X);
10:    }
11:    if(X % 2 != 0){
12:      printf("%d is odd!\n", X);
13:    }
14:  }
```

But if I run the above code with `X` equal to -4 I get the following results:

Listing 11: Conditional example with two conditionals

```
josephraskind@stargazer:/tmp/cond$ gcc if.c -o if; ./if
Welcome to the even checker!
Ew! -4 is negative!
-4 is even!
```

But this wasn't what I wanted! I didn't want to print out the evenness of the negative number!

We can solve this with the use of an `else if` clause:

Listing 12: Using else if

```
 1:  #include <stdio.h>
 2:  const int X = _; // Will set later...
 3:  int main(){
 4:    printf("Welcome to the even checker!\n");
 5:    if(X < 0){
 6:      printf("Ew! %d is negative!\n", X);
 7:    }else if(X % 2 == 0){
 8:      printf("%d is even!\n", X);
 9:    }else if(X % 2 != 0){
10:      printf("%d is odd!\n", X);
11:    }
12:  }
```

Now if I run the above code with `X` equal to -4 we'll see something different:

Listing 13: Conditional example if else

```
josephraskind@stargazer:/tmp/cond$ gcc if.c -o if; ./if
Welcome to the even checker!
Ew! -4 is negative!
```

The `if else` clause makes each subsequent `if` block *mututally exclusive*. No two blocks can be entered for the same set of conditional checks.

else

Sometimes we want mutually exclusive behaviors without having to write an additional conditional check. For example,

Listing 14: Conditional example with two conditionals

```
1:  #include <stdio.h>
2:  const int X = _; // Will set later...
3:  int main(){
4:    printf("Welcome to the even checker!\n");
5:    if(X % 2 == 0){
6:      printf("%d is even!\n", X);
7:    }
8:    if(X % 2 != 0){
9:      printf("%d is odd!\n", X);
10:   }
11: }
```

gets the job done, but it looks a bit clunky. There's a lot more work our eyes have to do when scanning the code. Instead we can write,

Listing 15: if-else statement

```
1:  #include <stdio.h>
2:  const int X = _; // Will set later...
3:  int main(){
4:    printf("Welcome to the even checker!\n");
5:    if(X % 2 == 0){
6:      printf("%d is even!\n", X);
7:    }else{
8:      printf("%d is odd!\n", X);
9:    }
10: }
```

now `X % 2 != 0` is implied as it is the only other possibility if the original conditional check fails.

We can combine this with our negative check above (Listing 12):

Listing 16: Using else if with else

```
1:  #include <stdio.h>
2:  const int X = _; // Will set later...
3:  int main(){
4:    printf("Welcome to the even checker!\n");
5:    if(X < 0){
6:      printf("Ew! %d is negative!\n", X);
7:    }else if(X % 2 == 0){
8:      printf("%d is even!\n", X);
9:    }else{
10:     printf("%d is odd!\n", X);
11:   }
12: }
```

Note: You can only ever have one `else` in your if statement conditional chain.

switch Statement

The `switch` statement is a variation of the `if` statement. It provides slightly different functionality to using `if-else` statement chains. The syntax is as follows:

Listing 17: switch statement syntax

```
1:   switch(expr){
2:    case SOMETHING:
3:       statement 1; // Do statement
4:    case SOMETHING_ELSE:
5:      statement 2; // Do statement
6:      break; // Exit control flow
7:    default:
8:      statement 3; // Do statement
9:   }
```

The `switch` statement takes an expression that resolves to an integer type (`char`, `int`, `short int`, `long int`, etc). And will transfer control to the block of code underneath the corresponding label. The `default` label is used when no corresponding label for the expression's value can be found. The `break` statement transfers control out of the switch block—this is particularly useful because without it the control will "fall through" to the next case. Meaning in the above example if `expr` resolves to `SOMETHING` then both `statement 1` and `statement 2` will be executed (`statement 3` will not because of the `break` statement).

?: Operator

Although not strictly a conditional statement, the `?:` "ternary" operator is often bunched into the same group due to its conditional evaluation logic. The syntax is as follows

Listing 18: Ternary operator syntax

```
conditional ? eval_true : eval_false
```

Using the evenness example from earlier:

Listing 19: Using else if

```
1:  #include <stdio.h>
2:  const int X = _; // Will set later...
3:  int main(){
4:    printf("Welcome to the even checker!\n");
5:    printf(X % 2 == 0 ? "%d is even!\n" : "%d is odd!\n" , X);
6:  }
```

Compiling and running the above gives us:

Listing 20: Using the ternary expression

```
josephraskind@stargazer:/tmp/cond$ gcc if.c -o if; ./if
Welcome to the even checker!
4 is even!
```

Exercises

1. Write a C program using an `if` statement that checks if a `const int X` is greater than 100 and prints "X is large!" if so. Test it with at least three different values. What happens when the condition is false?
2. Modify Listing 2 to also print "X is not even!" when `X` is odd using an `else` clause. Test with both even and odd values. Why is this preferable to writing a second `if` statement with the condition `X % 2 != 0`?
3. Write a C program that uses `if`, `else if`, and `else` to classify a `const int X` into one of the following categories and prints the result:
    - Negative
    - Zero
    - Between 1 and 100 inclusive
    - Greater than 100
4. The chapter warns that omitting `{}` from an `if` statement only attaches the first statement to the conditional. Write a program that demonstrates this bug by writing an `if` without braces followed by two `printf` calls. Which `printf` is controlled by the `if`? Verify by testing with a condition that evaluates to false.
5. Write a C program using a `switch` statement that takes a `const int day` between 1 and 7 and prints the corresponding day of the week. Use a `default` case for any value outside that range.
6. The chapter states that without a `break` statement, control "falls through" to the next case in a `switch`. Write a program that demonstrates this by writing a `switch` with three cases and no `break` statements. What gets printed when the first case matches? When might fall-through behavior actually be useful?
7. Rewrite the following `if-else` chain as a `switch` statement:

```
1:  const int X = 2;
2:  if(X == 1) printf("one\n");
3:   else if(X == 2) printf("two\n");
4:   else if(X == 3) printf("three\n");
5:   else printf("other\n");
```

Are there any cases where a `switch` cannot replace an `if-else` chain?

1. Write a C program that uses nested `if` statements (an `if` inside another `if`) to check whether a `const int X` is both positive and even. Print an appropriate message for each of the four possible combinations: positive even, positive odd, negative even, negative odd.
2. The assembly output in Listing 5 shows that the compiler uses a `jne` (jump if not equal) instruction to implement the `if` statement. Looking at the assembly, identify which line performs the conditional jump and where control is transferred when the condition is false. What does `testl %eax, %eax` do and why is it needed?
3. Further modify the ternary example in Listing 19 to include a check for a negative number all in the same line.

Footnotes:

1 `gcc` will optimize a call to `printf` by replacing it with `puts` if there is only a single string provided as an argument.

2 Again, we see here that `gcc` has optimized the instructions. Checking evenness can be done by seeing if the least significant bit is set to 0.

Loops

Introduction

Oftentimes we run into situations where we would like to repeat the same computation multiple times. As an example, consider a program which computes the power of one number raised to another. We know that 3^8 is equivalent to $3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$. This would be easy enough to hard code[1]. We could write the following C program:

Listing 1: Naive 3^8 program

```
1:  #include <stdio.h>
2:  int main(){
3:    int val = 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3;
4:    printf("3^8 = %d\n", val);
5:  }
```

And it works well enough, printing the expected `3^8 = 6561`. But what if **3** and **8** weren't constant values, but variables? If I replace 3 with `x`:

Listing 2: Naive power with variables

```
1:  #include <stdio.h>
2:  int main(){
3:    int x = 3;
4:    int y = 10;
5:    int val = x * x * x * x * x * x * x * x;
6:    printf("%d^%d = %d\n", x, y, val);
7:  }
```

If we run the above, we don't get the expected `3^10 = 59049`, but `3^10 = 6561`! It's clear that because Line 5 does not adapt to the fact that the target power has changed there can be no alteration in the computational task. One way we can fix this is with *looping*.

while Loops

There are three looping constructs provided by the C programming language, two of which are slight variatios on the foundational `while` loop. The syntax of a while loop is as follows:

Listing 3: while loop syntax

```
1:  while(condition)
2:    block; // Do this
```

Pretty simple! While a certain condition is met, i.e. the expression resolves to *true*, the instructions in the block will be executed. Once the instructions in the block have been exhausted the condition is checked again. If it is met the instructions start over (hence the "loop"); if the condition is not met the next set of instructions outside of the loop will be executed.

Here's how we can modify the power example to work with a while loop:

Listing 4: power as a while loop

```
 1:   #include <stdio.h>
 2:   int main(){
 3:     int x = 3;        // Base
 4:     int y = 10;       // Target power
 5:     int pow = 1;      // Starting accumulator value
 6:     int i = 0;        // Iterator
 7:     while(i < y){     // Conditional check
 8:       pow = pow * x; // Computation
 9:       i = i + 1;      // Incrementing i to eventually satisfy looping
condition
10:     }
11:     printf("%d^%d = %d\n", x, y, pow);
12:   }
```

There's a lot going on here. First, you'll notice there are two more variables: `pow` and `i`. `pow` is an accumulator which stores the partial result of each computational step needed to get the final power value. `i` acts as a tally that keeps track of how many loops have been performed. Additionally, you can see that there's some new logic: a conditional operator `i < y` and a code block. The conditional is used to limit the number of loops performed (more on this later), and the code block is for defining a set of computations to be performed each loop. Here, the code block is multiplying the `pow` accumulator with the base `x` and incrementing the `i` loop tally. We can trace the value at the end of every loop for each variable in a table to keep track of what's happening:

Loop #	i	pow
1	1	3
2	2	9
3	3	27
4	4	81
5	5	243
6	6	729
7	7	2187
8	8	6561
9	9	19683
10	10	59049

After the 10th loop completes the conditional is checked on final time. Since `i` is now equal to 10 the expression `i < 10` resolves to *false* and control is transferred to the first block outside of the loop, which is the print function call.

Infinite Loops

It is possible to accidentally create loops that never terminate. These are often referred to as *infinite loops*. Here's an example of how one could occur through a little bug in our program:

Listing 5: power implementation that loops infinitely

```
1:  #include <stdio.h>
2:  int main(){
3:    int x = 3;        // Base
4:    int y = 10;       // Target power
5:    int pow = 1;      // Starting accumulator value
6:    int i = 0;        // Iterator
7:    while(i < y){     // Conditional check
8:      pow = pow * x; // Computation
9:                      // Something's missing...
10:   }
11:   printf("%d^%d = %d\n", x, y, pow);
12: }
```

We can see here that the body of the while loop has had the tallying statement removed. This will cause i to never be incremented, thus keeping it at 0, and therefore never allowing for the conditional check to change values.

Note: Try compiling the code from Listing 4 yourself. You'll find that the compiler does not warn you of the infinite loop!

Infinite loops are not always a bad thing. Sometimes it makes sense for a program to include an infinite loop. For example, a chat room program might want to constantly poll a collection of network socket connections to see whether or not a new message has come through. Or, maybe a video game program might want to have a thread dedicated to always painting the screen for graphics content. What's important is to recognize when an infinite loop is happening and how to stop it if it's *not what you intended*.

How Are Loops Implemented?

In order to get a better intuition for how loops work we can take a look at how the assembly code is generated for them. Here's what it looks like when I run `gcc -O0 -fno-omit-frame-pointer -fno-stack-protector -S power_while.c` on Listing 4:

Listing 6: Listing 4 in x86-64 assembly

```
1:          .file   "power_while.c"
2:          .text
3:          .section        .rodata
4:  .LC0:
5:          .string "%d^%d = %d\n"
6:          .text
7:          .globl  main
8:          .type   main, @function
9:  main:
10: .LFB0:
11:         .cfi_startproc
12:         endbr64
13:         pushq   %rbp
14:         .cfi_def_cfa_offset 16
15:         .cfi_offset 6, -16
16:         movq    %rsp, %rbp
17:         .cfi_def_cfa_register 6
18:         subq    $16, %rsp
19:         movl    $3, -12(%rbp)
20:         movl    $10, -16(%rbp)
21:         movl    $1, -4(%rbp)
22:         movl    $0, -8(%rbp)
23:         jmp     .L2
24: .L3:
25:         movl    -4(%rbp), %eax
26:         imull   -12(%rbp), %eax
27:         movl    %eax, -4(%rbp)
28:         addl    $1, -8(%rbp)
29: .L2:
30:         movl    -8(%rbp), %eax
31:         cmpl    -16(%rbp), %eax
32:         jl      .L3
33:         movl    -4(%rbp), %ecx
34:         movl    -16(%rbp), %edx
35:         movl    -12(%rbp), %eax
36:         movl    %eax, %esi
37:         leaq    .LC0(%rip), %rdi
38:         movl    $0, %eax
39:         call    printf@PLT
40:         movl    $0, %eax
41:         leave
42:         .cfi_def_cfa 7, 8
43:         ret
44:         .cfi_endproc
45: .LFE0:
46:         .size   main, .-main
47:         .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
48:         .section        .note.GNU-stack,"",@progbits
49:         .section        .note.gnu.property,"a"
50:         .align 8
51:         .long    1f - 0f
52:         .long    4f - 1f
53:         .long    5
54: 0:
55:         .string  "GNU"
56: 1:
57:         .align 8
58:         .long    0xc0000002
59:         .long    3f - 2f
60: 2:
61:         .long    0x3
62: 3:
63:         .align 8
64: 4:
```

Again, there's a lot going on here so we'll focus on the major points. There are three labels to be aware of here `.LFB0:`, the label for the start of `main`, `.L2:`, the label for the conditional check, and `.L3:`, the label for the code block inside of the loop. Here is a table representing the variables to stack locations:

Variable	Location
x	-12(%rbp)
y	-16(%rbp)
pow	-4(%rbp)
i	-8(%rbp)

Note: There's nothing special about the order in which the variables were assigned locations on the stack, this is just how `gcc` decided to do so.

Line 23 starts us on our loop with `jmp .L2` which says there should be an unconditional jump to the label `.L2`. The jump brings us to Lines 30 and 31

which set up and perform the `i < y` comparison. `i` is moved into **%eax** (Line 30), `y` is then compared with **%eax**—`y - i`—(Line 31), and if the result is less than 0 control is transferred to `.L3`. Lines 25-28 represent the body of the loop (multiplying by `pow` by `x` and incrementing `i`). We can see that once the `jl` instruction fails on line 32 the instructions making up the `printf` call are followed one by one.

do while Loops

C provides a slight variation on the `while` loop with the `do while` loop. The syntax is as follows:

Listing 7: while loop syntax

```
1:  do{
2:    block; // Do this
3:  }while(condition); // Needs a ; to terminate statement
```

The biggest difference here is that the block is processed first *before* the first conditional check. The `do while` construct does not add anything new, it just makes certain kinds of looping logic a bit easier to write. For example, let's say I wanted to write a program that prompted the user to enter a positive number. I could write:

Listing 8: while-based positive number prompt

```
 1:  #include <stdio.h>
 2:  int main(){
 3:      int x;
 4:      printf("Enter a positive number: ");
 5:      scanf("%d", &x);
 6:      while(x <= 0){
 7:        printf("Enter a positive number: ");
 8:        scanf("%d", &x);
 9:      }
10:      printf("You entered: %d\n", x);
11:  }
```

I could cut down on the repeat code with a `do while` statement instead:

Listing 9: do while-based positive number prompt

```
1:  #include <stdio.h>
2:  int main(){
3:      int x;
4:      do {
5:          printf("Enter a positive number: ");
6:          scanf("%d", &x);
7:      } while(x <= 0);
8:      printf("You entered: %d\n", x);
9:  }
```

for Loops

The last looping structure provided by C is likely also it's most famous: the `for` loop. The syntax for the `for` loop is as follows:

Listing 10: while loop syntax

```
1:   for(expr1 ; expr2 ; expr3)
2:     block; // Do something
```

`expr1` is the *declaration expression*. It is where you can define or set a variable at the entrance of the `for` loop. `expr2` is the *conditional expression*. It is where you can place the conditional check that will be used to determine if another loop will be completed. `expr3` is the *increment expression*. It is used for performing an additional operation at the end of the loop that is not included in the block body. `expr1`, `expr2`, and `expr3` are all optional.

Note: When `expr2` is not present it will be treated as a *true* value. This means that `for(;;)` is equivalent to a `while(1)` statement.

Here is an example of the power program with a `for` loop:

Listing 11: power as a for loop

```
 1:   #include <stdio.h>
 2:   int main(){
 3:     int x = 3;                        // Base
 4:     int y = 10;                       // Target power
 5:     int pow = 1;                      // Starting accumulator value
 6:     for(int i = 0; i < y; i = i + 1){ // Declaration, conditional check,
and increment expression
 7:       pow = pow * x;                  // Computation
 8:     }
 9:     printf("%d^%d = %d\n", x, y, pow);
10:   }
```

Both Listing 4 and Listing 11 are functionally equivalent[2]. The obvious change is that the loop tallying variable, **i**, has been declared in `expr1` and is incremented in `expr3`. Everything else is identical.

Many people prefer to write `for` loops over `while` loops due to convenience. From the compiler's perspective, it often does not matter.

Note: Additions like the `for` loop and the `do while` loop can be classified as *syntactic sugar*. They do not provide additional functionality for the language, but they provide stylisitic benefits.

Nested Loops

Loops can be nested within one another. For example, if I wanted to print out a 4x3 grid of asterixes I could write:

Listing 12: Nested for loops

```c
#include <stdio.h>
int main(){
  for(int i = 0; i < 4; i++){
    for(int j = 0; j < 3; j++){
      printf("*");
    }
    printf("\n");
  }
}
```

There are many situations where nested for loops are desirable, particularly with complex data structures.

break Keyword

We can use the `break` statement that we learned about with the switch statement. `break` will immediately transfer control out of the loop to the next block available as if the condition failed. If we did not want our power function to iterate anymore if pow was greater than 100:

Listing 13: power with break

```c
#include <stdio.h>
int main(){
  int x = 3;       // Base
  int y = 10;      // Target power
  int pow = 1;     // Starting accumulator value
  int i = 0;       // Iterator
  while(i < y){    // Conditional check
    pow = pow * x; // Computation
    if(pow > 100)
      break;       // Exists the loop
    i = i + 1;     // Incrementing i to eventually satisfy looping condition
  }
  printf("%d^%d = %d\n", x, y, pow);
}
```

I leave it as an exercise to you to see what happens when the above is ran.

continue Keyword

Similarly to the `break` keyword, the `continue` keyword alters the control flow of a loop. When a `continue` is reached control flow will be transferred

to the "end" of the loop. For a `while` loop that means the conditional will immediately be rechecked. For a `for` loop that means the *increment expression* will be performed then the conditional will be checked. Here is a `for` loop that sums up all even numbers from 1 to 100:

Listing 14: Even sum with continue

```
#include <stdio.h>
int main(){
  int sum = 0;
  for(int i = 0; i < 100; i++){
    if(i % 2 == 1)
      continue;
    sum = sum + i;
  }
  printf("sum = %d\n", pow);
}
```

Exercises

1. Rewrite the power program from Listing 4 using a `for` loop instead of a `while` loop. Then rewrite it again using a `do while` loop. Which version do you find most readable and why?

2. The chapter provides a trace table for Listing 4 showing the value of `i` and `pow` at the end of each loop. Construct a similar trace table by hand for the following program, then verify by running it:

   ```
   int x = 2;
   int y = 8;
   int pow = 1;
   int i = 0;
   while(i < y){
       pow = pow * x;
       i = i + 1;
   }
   ```

3. Write a `for` loop that computes the sum of all integers from 1 to 100. Then write the equivalent using a `while` loop. Verify both produce the same result.
4. The chapter states that `for(;;)` is equivalent to `while(1)`. Write a program using each form that prints "looping..." exactly 5 times using a `break` statement to exit. Verify both behave identically.
5. Modify the even sum program from Listing 14 to instead sum all odd numbers from 1 to 100 using `continue`. What change do you need to make to the conditional? Verify your result.
6. The chapter notes that the compiler does not warn about infinite loops. Write a program containing an infinite loop as shown in Listing 5, compile it with `gcc`, and confirm no warning is produced. Then use `Ctrl+C` in the terminal to kill it. What does `Ctrl+C` do at the OS level?

7. Write a `for` loop using the `break` keyword that finds the first integer greater than 1 whose square exceeds 500. Print both the integer and its square.
8. The chapter shows the assembly output for the `while` loop in Listing 6. Compile the `for` loop version of the power program from Listing 11 with `gcc -O0 -fno-omit-frame-pointer -fno-stack-protector -S` and compare the assembly output. Are they identical? What does this tell you about the relationship between `for` and `while` loops at the machine level?

9. Using nested loops as shown in Listing 12, write a program that prints the following multiplication table:

```
1  2  3  4  5
2  4  6  8  10
3  6  9  12 15
4  8  12 16 20
5  10 15 20 25
```

10. The chapter states that all three expressions in a `for` loop are optional, as shown in Listing 10. Write three separate programs demonstrating this: one with `expr1` omitted, one with `expr3` omitted, and one with both omitted. In each case, what change must you make elsewhere to keep the program correct?

Footnotes:

1 "Hard coding" refers to writing logic directly into source code with little to no abstractions.

2 Although, they are not *semantically* equivalent. In the while loop version, `i` is declared outside of the loop block and therefore is visible after the loop finishes. In the for loop version, `i` is declared in the declaration expression and therefore is only visible within the for loop. From the perspective of the program output, this is splitting hairs. From the perspective of understanding the language runtime, this is an important distinction.

Functions

Introduction

The most powerful tool a computer programmer has is *abstraction*. Crafting abstractions allows us the ability to offload cognitive effort and focus on more generalized problems. For example, when we use our `bash` shell we often don't worry about the inner workings of how the OS handles the bytes that constitute files or folders. We simply run commands like `cd`, `ls`, and `mv` while taking for granted that these files will be managed appropriately by the filesystem. When constructing large, and even small, programs a principal abstraction is that of the *function*, which, when implemented correctly, acts as a means of compartmentalizing behavior in the same way that files or their corresponding `bash` operations do.

Why Functions?

The usefulness of functions becomes readily apparent when applied to a real example. Take the power function we defined when discussing loops:

Listing 1: power loop

```c
#include <stdio.h>
int main(){
  int x = 3;
  int y = 5;
  int pow = 1;
  for(int i = 0; i < y; i = i + 1){
    pow = pow * x;
  }
  printf("%d^%d = %d\n", x, y, pow);
}
```

If I wanted to also find the values of 3^5 and 4^10, I would need to alter the program like so:

Listing 2: Two powers

```
#include <stdio.h>
int main(){
  int x = 3;
  int y = 5;
  int pow = 1;
  for(int i = 0; i < y; i = i + 1){
    pow = pow * x;
  }
  printf("%d^%d = %d\n", x, y, pow);
  x = 4;
  y = 10;
  for(int i = 0; i < y; i = i + 1){
    pow = pow * x;
  }
  printf("%d^%d = %d\n", x, y, pow);
}
```

In order to achieve the two outputs I had to copy and paste my power code a second time into my main function. But what if I also wanted to see the values of 7^3 and 16^6?

Listing 3: Four powers

```
#include <stdio.h>
int main(){
  int x = 3;
  int y = 5;
  int pow = 1;
  for(int i = 0; i < y; i = i + 1){
    pow = pow * x;
  }
  printf("%d^%d = %d\n", x, y, pow);
  x = 4;
  y = 10;
  for(int i = 0; i < y; i = i + 1){
    pow = pow * x;
  }
  printf("%d^%d = %d\n", x, y, pow);
  x = 7;
  y = 3;
  for(int i = 0; i < y; i = i + 1){
    pow = pow * x;
  }
  printf("%d^%d = %d\n", x, y, pow);
  x = 16;
  y = 6;
  for(int i = 0; i < y; i = i + 1){
    pow = pow * x;
  }
  printf("%d^%d = %d\n", x, y, pow);

}
```

It all gets to be a bit much, no? What if instead we had a power function to represent the computation, like so:

Listing 4: Four powers as four function calls

```c
#include <stdio.h>
int main(){
  printf("%d^%d = %d\n", 3, 5, pow(3,5));
  printf("%d^%d = %d\n", 4, 10, pow(4,10));
  printf("%d^%d = %d\n", 7, 3, pow(7,3));
  printf("%d^%d = %d\n", 16, 6, pow(16,6));
}
```

It looks much cleaner and is easier to follow. The only issue is now we need to define a function called `pow`...

Defining Functions

Functions in programming languages work similarly to functions in pure mathematics: they take input and they produce output. The syntax for defining a function in C is as follows:

Listing 5: function syntax

```c
type function_name(arg1, arg2, ...){
  statements; // Do this
  return val; //Return some data (output)
}
```

Like variables, every function must have a *type*—for functions, it is called a *return type*. By default, all functions in C have the `int` return type if not otherwise specified. Functions may or may not take arguments; however, if arguments are specified in the function definition that function **must** be passed arguments. Arguments must be sent in the order in which they were defined. Functions may or may not specify a return value; the return value **must** match the type of the defined return type[1].

In C, we can represent the identity function `f(x) = x` as:

Listing 6: Identity function in C

```c
int id(int x){
  return x;
}
```

Despite the small amount of syntax there are a few things we can notice. First, the function has a name, `id`, just like variables do. Knowing a function's name is essential to calling that function. Furthermore, we can see that the `id` function has a return type, `int`. Again, the return type indicates the type of data that will be output by that function. Additionally, we can see that there is an argument defined by the `id` function, `x` with type `int`. This means that `id` *takes* an integer value when it is called. Finally, we can see a `return` statement which sends back the value tied to the expression it is provided (here the value of `x`).

Let's now define the pow function we referenced in Listing 4:

Listing 7: pow function in C

```
1:  int pow(int x, int y){
2:    int acc = 1;
3:    for(int i = 0; i < y; i = i + 1){
4:      acc = acc * x;
5:    }
6:    return acc;
7:  }
```

The fundamentals we learned with `id` are still present in `pow`: there is a return type, arguments, a function body, and a return statement within the function body.

Returning Values

I find that students learning about programming for the first time often get tripped up on what it means to "return" a value. The syntax for a return statement is as follows:

Listing 8: return statement syntax

```
1:  return expr;
```

If you thick back to the chapter on expressions, you'll remember that an expression is something that evaluates to some value. It is the *value* that is returned by the return statement. For example, given the `pow` function as it's defined in Listing lst:pow_function, we can see that `acc` is returned. It is not the *variable* that is returned, but rather the *value stored in* the variable `acc`. When we call `pow` with `pow(3,2)`, the function will reserve space for a varable called `acc`, eventually store 9 in that memory location, and then ultimately return that 9 to the caller. (Refer to the section on calling conventions for further explanation.)

void Type

Some functions don't need to return a value. For example, we've often used the `printf` function to print characters to the screen. The function does not return anything, but rather produces *side effects*, i.e. it changes something about the state of the program which isn't captured in a return value. Functions which do not return anything are tagged with a special `void` type. For example, I could write a function which prints the results of a call to pow:

Listing 9: Printing a call to pow

```
1:  #include <stdio.h>
2:  void print_pow(int a, int b){
3:      printf("%d^%d = %d\n", a, b, pow(a,b));
4:    }
```

Here, I've created a function `print_pow` which I've assigned the `void` return type. Because I've assigned the function `void`, there is no expectation on the part of the compiler for a `return` statement to be present.

Variables in Functions

You might be wondering what is going on with the arguments in the function declarations. Why am I treating them like they're normal variables? Well, it's because they are. The only thing special about arguments is that the function caller provides the values in the function call, they are not set *a priori*. Once a function argument has been declared it can be used like any other variable in the function body.

I mentioned in the variables chapter that scope would eventually be relevant again. This is exactly that time! All variables, including arguments, defined within a function block are visible *only in that function block*. For example, take `pow` above (Listing 7). Variables `x`, `y`, `i`, and `acc` can only be referenced within that given function. If I were to define another function, say `is_x_negative`, I wouldn't be able to reference the `x` defined in `pow`:

Listing 10: pow function in C

```
1:  int pow(int x, int y){
2:    int acc = 1;
3:    for(int i = 0; i < y; i = i + 1){
4:      acc = acc * x;
5:    }
6:    return acc;
7:  }
8:
9:  int is_x_negative(){
10:   return x < 0 : 1 else 0; // x cannot be referenced here!
11: }
```

Another thing that trips up students is declaring variables with the same name in two different locations. For example, if I try to call `pow` with a variable `x` in `main`:

Listing 11: pow function in C

```
1:  int pow(int x, int y){
2:    int acc = 1;
3:    for(int i = 0; i < y; i = i + 1){
4:      acc = acc * x;
5:    }
6:    return acc;
7:  }
8:
9:  int main(){
10:   int x = 10;
11:   pow(x, 3);
12: }
```

The `x` in `main` and the `x` in `pow` are *two entirely different variables*. The `x` on Line 10 is only visible in the `main` function, whereas the `x` on Line 1 is

only visible in the `pow` function. This is because their lexical scopes are set at the time of declaration, within the scope in which they are declared. During compilation, variables with the same name are tagged[2] with information regarding the location in which they are defined to disambiguate their scope information (or they are assigned unique id's).

Forward Declarations

Functions can be declared without actually being implemented. For example, I could define the following logic in one file called `my_program.c`:

Listing 12: Forward declaration of pow

```
1:   #include <stdio.h>
2:   int pow(int,int); // Forward declaration of a pow function with a certain
type signature
3:   int main(){
4:     printf("3^10 = %d\n", pow(3,10));
5:   }
```

The above code will compile, but it will not build into an executable. You can see what I mean below:

Listing 13: Compiling my_program.c two ways

```
1:   josephraskind@stargazer:/tmp/func$ gcc -Wno-builtin-declaration-mismatch
my_program.c -o my_program
2:   /usr/bin/ld: /tmp/ccUIN1k5.o: in function `main':
3:   my_program.c:(.text+0x13): undefined reference to `pow'
4:   collect2: error: ld returned 1 exit status
5:   josephraskind@stargazer:/tmp/func$ ls
6:   my_program.c
7:   josephraskind@stargazer:/tmp/func$ gcc -c -Wno-builtin-declaration-
mismatch my_program.c -o my_program.o
8:   josephraskind@stargazer:/tmp/func$ ls
9:   my_program.c  my_program.o
```

The first call to `gcc` on Line 1 tries to compile *and link* the c files it was provided. The linking stage involves trying to match unknown symbols with known symbols. The forward declaration on Line 2 of Listing lst:forward_dec is essentially an IOU. We're telling the compiler, "Trust me, there will be a `pow` function coming your way that takes two ints and returns an int". The problem is that we never actually supply it. The second call to `gcc` uses the `-c` flag which tells the compiler to stop after the compilation stage—don't bother linking any of the compiled files. This is why we get a **.o** file, or an *object* file, without any issue. We can solve this one of three ways:

1. Define the pow function in `my_program.c`.
2. Write another C file to define `pow` in and compile it together.
3. Compile another `.o` file and link the two after each are compiled seperately.

(1) is trivial, and (3) I will leave as an exercise. To solve (2) we can write another C file called `my_pow.c` which contains the code found in Listing 7 and then compile them together in `gcc`:

Listing 14: Combined compilation and linkage between given C files

```
1:   josephraskind@stargazer:/tmp/func$ gcc -Wno-builtin-declaration-mismatch
my_program.c my_pow.c -o my_program
2:   josephraskind@stargazer:/tmp/func$ ./my_program
3:   3^10 = 59049
```

Here, we make good on the promise that `pow` will eventually be defined.

Calling Functions

The syntax for calling functions is rather simple, and I couldn't avoid showing it in prior examples so you have seen it before. All you need to call the function is:

- To be in a lexical scope that recognizes the function's name
- Use the name of the function
- Place a set of parentheses next to the name
- Insert the correct number and type of arguments corresponding to each parameter defined in the function header

For example, to call `pow` all I need to write is `pow(3,5)`. 3 corresponds to the first argument, `x`, and 5 corresponds to the second argument, `y`. If that function has a return type then the returned value will be evaluated in function call expression. Therefore, `int p = pow(3,5)` will store the value `243` into the variable `p`.

Because function call arguments are *expressions* we can nest function calls within one another. For example, if I wanted to evaluate the 2^(3^2), I could write `pow(2, pow(3, 2))`. First the expression `pow(3,2)` will be evaluated, then the expression `pow(2, 9)` will be evaluated.

Calling Convention

Functions at the high-level programming language view can look mighty abstract. One thing that helps break down what's involved in implementing functions is taking a look at the assembly code generated by the compiler. Below is the C code I will compile down to assembly:

Listing 15: calling.c file

```c
#include <stdio.h>
int pow(int x, int y){
  int acc = 1;
  for(int i = 0; i < y; i = i + 1){
    acc = acc * x;
  }
  return acc;
}

int main(){
  printf("result: %d\n", pow(5, 3));
}
```

I then use the command `gcc -O0 -Wno-builtin-declaration-mismatch -fno-omit-frame-pointer -fno-stack-protector -S calling.c` to produce the following assembly:

Listing 16: calling.s file

```
1:          .file   "calling.c"
2:          .text
3:          .globl  pow
4:          .type   pow, @function
5:  pow:
6:  .LFB0:
7:          .cfi_startproc
8:          endbr64
9:          pushq   %rbp
10:         .cfi_def_cfa_offset 16
11:         .cfi_offset 6, -16
12:         movq    %rsp, %rbp
13:         .cfi_def_cfa_register 6
14:         movl    %edi, -20(%rbp)
15:         movl    %esi, -24(%rbp)
16:         movl    $1, -4(%rbp)
17:         movl    $0, -8(%rbp)
18:         jmp     .L2
19: .L3:
20:         movl    -4(%rbp), %eax
21:         imull   -20(%rbp), %eax
22:         movl    %eax, -4(%rbp)
23:         addl    $1, -8(%rbp)
24: .L2:
25:         movl    -8(%rbp), %eax
26:         cmpl    -24(%rbp), %eax
27:         jl      .L3
28:         movl    -4(%rbp), %eax
29:         popq    %rbp
30:         .cfi_def_cfa 7, 8
31:         ret
32:         .cfi_endproc
33: .LFE0:
34:         .size   pow, .-pow
35:         .section        .rodata
36: .LC0:
37:         .string "result: %d\n"
38:         .text
39:         .globl  main
40:         .type   main, @function
41: main:
42: .LFB1:
43:         .cfi_startproc
44:         endbr64
45:         pushq   %rbp
46:         .cfi_def_cfa_offset 16
47:         .cfi_offset 6, -16
48:         movq    %rsp, %rbp
49:         .cfi_def_cfa_register 6
50:         movl    $3, %esi
51:         movl    $5, %edi
52:         call    pow
53:         movl    %eax, %esi
54:         leaq    .LC0(%rip), %rdi
55:         movl    $0, %eax
56:         call    printf@PLT
57:         movl    $0, %eax
58:         popq    %rbp
59:         .cfi_def_cfa 7, 8
60:         ret
61:         .cfi_endproc
62: .LFE1:
63:         .size   main, .-main
64:         .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
65:         .section        .note.GNU-stack,"",@progbits
66:         .section        .note.gnu.property,"a"
67:         .align 8
68:         .long    1f - 0f
69:         .long    4f - 1f
70:         .long    5
71: 0:
72:         .string  "GNU"
73: 1:
74:         .align 8
75:         .long    0xc0000002
76:         .long    3f - 2f
77: 2:
78:         .long    0x3
79: 3:
80:         .align 8
81: 4:
```

There is now quite **a lot** going on here! The first, most important thing to take note of is the fact that there are two labels `main:` and `pow:`. These refer to the two functions defined in the source code—this is how Line 52 can have the `call pow` instruction. However, there are a few key events that happen prior to that `call` instruction. First, Line 45 has the `main` function invoking the `pushq %rbp` instruction. **%rbp** is a register which stores the *base pointer* of the previous function's stack. Any startup functions called before main can still be reached with their initial stack contents. Next, `movq %rsp, %rbp` is processed, which moves the current stack pointer onto the base pointer. Then, Lines 50 and 51 involve setting up the arguments that will be used by the `pow` function. **3** is placed in **%esi**, the register used for the second

argument, and **5** is placed in **%edi** the register used for the first argument[3]. The register order for arguments is settled by the architecture's *calling convention*.

Once Line 52 has been executed, control is transferred to the `pow` function. The `pow` function, like the `main` function, makes some room for its own stack. Lines 14 and 15 have the argument registers **%esi** and **%edi** going into their target variables (i.e., locations on the stack). The function then operates normally until we reach Line 28. Line 28 places the `acc` variable, location **-4(%rbp)**, into the conventional return register **%eax**. The `pow` stack frame is then popped off (it will now be ignored in memory). Line 31 finally returns out of the `pow` function.

Line 53 has us returning back into the `main` function and storing the return value sent by `pow` through **%eax** into **%esi** (which should make sense as `pow(5,3)` is the second argument to our call to `printf`). `printf` is called and the `main` function returns after it is finished.

Note: If it's not clear how the `ret` call works, take another look at the `movq %rsp, %rbp` instruction on Line 48 again. When `ret` is invoked it will take whatever is at the top of the stack and jump to that memory location.

Function Uniqueness

All functions in C must be unique. This means I cannot define two functions with exactly the same name. For example:

Listing 17: Similar functions

```
1:  int my_func(int i); // Initial function
2:
3:  int my_func(int i); // Big problem!
4:                      // I already have a function named my_func
5:                      // that takes an int and returns an int
6:
7:  double my_func(int i); //Big problem, again!
8:                         // I already have a function named my_func
9:                         // that takes an int
10:
11: int my_func(int i, int j); // Big problem, yet again!
12:                            // I already have a function named my_func
13:
14: int my_other_func(int i); // No problem here, different name!
15:
```

Note: This is an example of where forward declaration could be useful. If there are two libraries with a function called `sort_data` then you can decide which file to link with during compilation when benchmarking.

Recursive Functions

Sometimes, complex problems are easily defined with recursive solutions. Take the fibonacci sequence for example. It is defined as:

- `f(0) = 0`
- `f(1) = 1`
- `f(n) = f(n-1) + f(n-2)`
- where `n > 1`

I leave it as an exercise for you to implement the fibonacci sequence as a recursive function in C.

Exercises

1. Given the `id` function defined in Listing 6, what would be returned by a call to `id(15.7)`? Would an error be thrown? Why or why not?
2. The chapter states that all functions in C default to the `int` return type if not otherwise specified. Write a function `add` that takes two `int` arguments and returns their sum, but omit the return type from the function header. Compile it with `gcc -Wall` and record any warnings. Does the program still run correctly? What does this tell you about the difference between a compiler warning and a compiler error?
3. Rewrite Listing 3 using the `pow` function defined in Listing 7. How many lines of code did you save? What would happen if you discovered a bug in the power logic — how many places would you need to fix it in Listing 3 versus the function-based version?
4. Write a `void` function called `print_pow` that takes two integers and prints the result of raising the first to the power of the second, as shown in Listing 9. Then rewrite Listing 3 using `print_pow`. What is the difference between a function that returns a value and one that produces a side effect?
5. The chapter states that function arguments are just variables. Write a program that declares `int x = 10` in `main` and passes it to `pow`. After the call to `pow`, print `x` in `main`. Does `x` change? What does this tell you about the relationship between the `x` in `main` and the `x` in `pow`?
6. Implement the fibonacci sequence as a recursive function in C as suggested at the end of the chapter. Test it with the values 0, 1, 2, 5, and 10. Then construct a trace table showing the recursive calls made to compute `fib(5)`.
7. Write a forward declaration for `pow` in a file called `my_program.c` and define `pow` in a separate file called `my_pow.c`. Compile them separately into object files using `gcc -c` and then link them together manually using `gcc my_program.o my_pow.o -o my_program`. Verify the output matches Listing 14.
8. The chapter shows that `pow(2, pow(3, 2))` can be used to compute 2^(3^2). Write a C program that evaluates this nested call and prints the result. In what order are the two calls to `pow` evaluated? Verify by adding a `printf` statement inside `pow` that prints each time it is called.

9. Compile Listing 15 with `gcc -O0 -Wno-builtin-declaration-mismatch -fno-omit-frame-pointer -fno-stack-protector -S calling.c` and examine the assembly output. Identify which lines correspond to: (a) saving the base pointer, (b) placing arguments into registers, (c) the function call, and (d) reading the return value. Compare your findings to the explanation given in the chapter.
10. The chapter states that the `ret` instruction takes whatever is at the top of the stack and jumps to that memory location. Looking at Listing 16, identify where the return address is placed on the stack and where it is consumed. What instruction places it there implicitly, and why is this necessary for the program to continue executing correctly after `pow` returns?

Footnotes:

1 If the types do not match up the C compiler will try to perform a conversion. If a conversion cannot be performed a type error will be raised.

2 Often called the "mangled" name.

3 This is done in reverse order which might be a bit confusing

Memory

Introduction

> “
>
> *Programmers never die*
>
> *They just lose their memory — Desk tchotchke I own*

We now know that data is a fundamental abstraction which is necessary for the completion of computing tasks. By definition, it would be absurd to conceive of performing computations if there was no data to perform computations on. Logically, considering our computers are physical devices which hold physical information, it would follow that if we need to perform computations with data it must be had *from somewhere*. We call that "somewhere" *memory*.

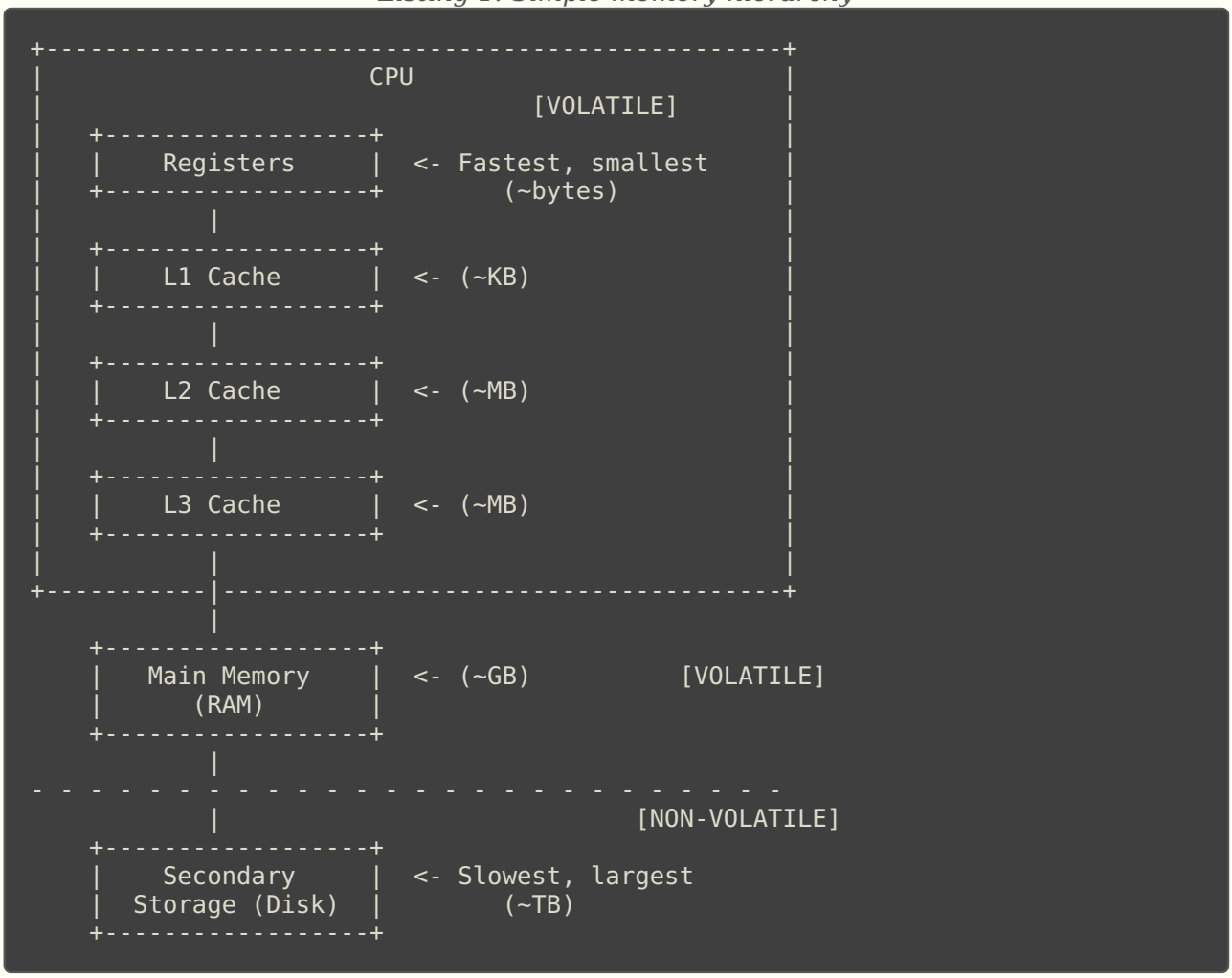
What is Memory?

One possible definition of memory, provided by David Patterson and John Hennessy in their classic introductory text *Computer Organization and Design*, goes as follows: "Memory is where the programs are kept when they are running; it also contains the data needed by the running programs." That is to say that memory is a *location*. It is some *space* within the components that make up a computer. The simple term "memory" itself is often ambiguous when devoid of context; as it turns out, there's many different kinds of memory and devices which act as computer memory. For example, there's volatile and non-volatile memory, main and secondary memory, virtual and physical memory, etc. Functionally, *memory is anything that cannot be stored purely in the CPU*—specifically within its registers. Although there are many registers on the average x86-64 chip, all of them combined could not store the entirety of `gcc`.

Memory Hierarchy

Again, when we refer to "memory" its often a shorthand for a wide variety of components inside of the computer. These components can be arranged to form a memory hierarchy:

Listing 1: Simple memory hierarchy



The closer a component is to the CPU, the easier it is to retrieve information from it. Additionally, read/write speeds exist in an inverse relationship to cost and storage size. Very fast components, like registers, are as speedy as they are *because* they cannot store much information. This has to do with the physical make-up of memory devices (something that is outside the scope of this class to get into).

Additionally, there is a split between volatile and non-volatile memory. Volatile memory refers to memory components which lose their charge (data retention) after they are powered down. Dynamic random access memory, or DRAM, is one such example[1]—CPU registers and caches are another. Non-volatile memory is constructed in such a way that cells are capable of storing information long after the device is powered down. Non-volatile memory components, like solid-state drives (SSDs) and hard disk drives (HDDs), are the reason we can shut off our computers and turn them on again without losing precious files and programs.

Memory Semantics

What matters most to us as C programmers right now, is not so much the innerworkings of memory devices, but rather how the C runtime interacts

with memory (how it is read, written, allocated, and deallocated). In the programming language community, we call that sort of behavior the *memory semantics* of a language. Luckily for us, C is about as "bare metal" as most high-level programming languages get, which means it doesn't take to much to describe its semantics. We can categorize data in our C runtime under two umbrellas: **statically allocated memory** and **dynamically allocated memory**.

Statically Allocated Memory

Statically allocated memory refers to memory that is available at the start of the program's runtime—meaning it has *already been allocated* by the time it has been compiled and linked into an executable.

Global Variables

Variables declared in the global scope must be allocated before the execution of a program. These variables must be reachable by any and all functions that reference them.

Listing 2: Global variable example

```
1:  int g = 100;
2:
3:  int foo(int a, int b){
4:    int x, y, val;
5:    //....
6:    return val;
7:  }
8:
9:  int goo(int c, int d){
10:   int t, u, val;
11:   //...
12:   return val + g;
13: }
14:
15: int main(){
16:   goo(1,2);
17: }
```

In Listing 2 we see a global variable `g` declared at the top. We also see three functions defined: `foo`, `goo`, and `main`. Neither `main` nor `foo` uses `g`, but *they could have*. Since `g` was defined in the global scope, any function is capable of referencing it. This necessitates the compiler reserving space *a priori* for any location in the code that may want to reference it.

String Constants

Think back to our first program in class, `hello_world.c`, represented below:

Listing 3: Classic "Hello, World!" program

```
1:  #include <stdio.h>
2:  int main(){
3:    printf("Hello, World!\n");
4:  }
```

I had mentioned that the characters contained within the quotation marks were called a *string*. The colloquial type refers to the "string" of characters contained within the phrase. It so happens that these characters are stored in what's called "read-only" memory within the executable file. Let's take a look at the assembly dump of the **.c** file (using `gcc -O0 -fno-omit-frame-pointer -fno-stack-protector -S hello_world.c`):

Listing 4: Classic "Hello, World!" program in x86-64 assembly

```
1:          .file   "hello_world.c"
2:          .text
3:          .section        .rodata
4:  .LC0:
5:          .string "Hello, World!"
6:          .text
7:          .globl  main
8:          .type   main, @function
9:  main:
10: .LFB0:
11:         .cfi_startproc
12:         endbr64
13:         pushq   %rbp
14:         .cfi_def_cfa_offset 16
15:         .cfi_offset 6, -16
16:         movq    %rsp, %rbp
17:         .cfi_def_cfa_register 6
18:         leaq    .LC0(%rip), %rdi
19:         call    puts@PLT
20:         movl    $0, %eax
21:         popq    %rbp
22:         .cfi_def_cfa 7, 8
23:         ret
24:         .cfi_endproc
25: .LFE0:
26:         .size   main, .-main
27:         .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
28:         .section        .note.GNU-stack,"",@progbits
29:         .section        .note.gnu.property,"a"
30:         .align 8
31:         .long    1f - 0f
32:         .long    4f - 1f
33:         .long    5
34: 0:
35:         .string  "GNU"
36: 1:
37:         .align 8
38:         .long    0xc0000002
39:         .long    3f - 2f
40: 2:
41:         .long    0x3
42: 3:
43:         .align 8
44: 4:
```

We can see on Line 3 the assembly code defines a section called `.rodata` which stands for "read only data". Lines 5 and 6 define the string constant "Hello, World!" which is later referenced on Line 18 for the call to `printf` (again, `printf` is optimized away and replaced by `puts`). If I were to add some code which tries to alter the string directly, by replacing "Hello, World!" with "Jello, World!", we'll find it doesn't work as expected:

Listing 5: A potentially new classic "Jello, World!" program

```
1:  #include <stdio.h>
2:  int main(){
3:    char * msg = "Hello, World!\n";
4:    msg[0] = 'J'; // Trying to reset the first character of the string
5:    printf("%s", msg);
6:  }
```

Now if I try to compile and run that:

Listing 6: Running my now world famous "Jello, World!" program

```
josephraskind@stargazer:/tmp/mem$ gcc -O0 -fno-omit-frame-pointer -fno-stack-
protector jello_world.c -o jello_world
josephraskind@stargazer:/tmp/mem$ ./jello_world
Segmentation fault (core dumped)
```

I get a strange error! This happened because I tried to edit, or write to, memory that was set to read only. We'll learn more about this when we discuss pointers.

static Keyword

Note: This one is a bit of a subtle point which I've added for completeness.

Variables tagged with the `static` keyword variables take on the same behavior as global variables do regarding memory allocation, although their lexical scope remains within the block they were defined in. Here is an example:

Listing 7: Using a static variable

```
 1:  #include <stdio.h>
 2:
 3:  int foo(){
 4:    static int x = 100;
 5:    x = x + 1;
 6:    return x;
 7:  }
 8:
 9:  int main(){
10:    printf("1st call to foo(): %d\n", foo());
11:    printf("2nd call to foo(): %d\n", foo());
12:  }
```

Notice I have prefixed the `static` tag to my int declaration on Line 4. Now we can compile and run the code ():

Listing 8: Running Listing 7

```
josephraskind@stargazer:/tmp/mem$ gcc static.c -o static; ./static
1st call to foo(): 101
2nd call to foo(): 102
```

How is this possible?! Again, turning to assembly will help us out (`gcc -O0 -fno-omit-frame-pointer -fno-stack-protector -S static.c`):

Listing 9: Classic "Hello, World!" program in x86-64 assembly

```
1:          .file   "static.c"
2:          .text
3:          .globl  foo
4:          .type   foo, @function
5:  foo:
6:  .LFB0:
7:          .cfi_startproc
8:          endbr64
9:          pushq   %rbp
10:         .cfi_def_cfa_offset 16
11:         .cfi_offset 6, -16
12:         movq    %rsp, %rbp
13:         .cfi_def_cfa_register 6
14:         movl    x.2315(%rip), %eax
15:         addl    $1, %eax
16:         movl    %eax, x.2315(%rip)
17:         movl    x.2315(%rip), %eax
18:         popq    %rbp
19:         .cfi_def_cfa 7, 8
20:         ret
21:         .cfi_endproc
22: .LFE0:
23:         .size   foo, .-foo
24:         .section        .rodata
25: .LC0:
26:         .string "1st call to foo(): %d\n"
27: .LC1:
28:         .string "2nd call to foo(): %d\n"
29:         .text
30:         .globl  main
31:         .type   main, @function
32: main:
33: .LFB1:
34:         .cfi_startproc
35:         endbr64
36:         pushq   %rbp
37:         .cfi_def_cfa_offset 16
38:         .cfi_offset 6, -16
39:         movq    %rsp, %rbp
40:         .cfi_def_cfa_register 6
41:         movl    $0, %eax
42:         call    foo
43:         movl    %eax, %esi
44:         leaq    .LC0(%rip), %rdi
45:         movl    $0, %eax
46:         call    printf@PLT
47:         movl    $0, %eax
48:         call    foo
49:         movl    %eax, %esi
50:         leaq    .LC1(%rip), %rdi
51:         movl    $0, %eax
52:         call    printf@PLT
53:         movl    $0, %eax
54:         popq    %rbp
55:         .cfi_def_cfa 7, 8
56:         ret
57:         .cfi_endproc
58: .LFE1:
59:         .size   main, .-main
60:         .data
61:         .align 4
62:         .type   x.2315, @object
63:         .size   x.2315, 4
64: x.2315:
65:         .long   100
66:         .ident  "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
67:         .section        .note.GNU-stack,"",@progbits
68:         .section        .note.gnu.property,"a"
69:         .align 8
70:         .long    1f - 0f
71:         .long    4f - 1f
72:         .long    5
73: 0:
74:         .string  "GNU"
75: 1:
76:         .align 8
77:         .long    0xc0000002
78:         .long    3f - 2f
79: 2:
80:         .long    0x3
81: 3:
82:         .align 8
83: 4:
```

We can see on Line 60 a new section called `.data` is demarcated. `.data` refers to read/write memory. Lines 64 and 65 provide the label and starting value associated with the `static int x` defined in Listing 7 (the label name is a "mangled" version of the original variable name to separate it from other potential static variables named `x` in other lexical scopes). Lines 14-16 show the increment and store operation broken up into three instructions (load, add, and store). Even though the variable was defined within the local function scope it has a program-long shelf life due to the `static` keyword.

Dynamically Allocated Memory

Dynamically allocated memory refers to memory that is not available at the start of the program's runtime—meaning it must be allocated and set *during the program's execution*. There are two runtime memory data structures used to handle dynamic memory: the **stack** and the **heap**.

Stack

The stack is the primary method of dynamic allocation that we have interacted with so far. Almost all of our variables have been placed and manipulated on the stack without our involvement. This is because the C compiler is responsible for manageing the stack—i.e. reserving space, pushing the frame pointer, popping unneeded values, etc. A short example to illustrate the point:

Listing 10: is_negative.c

```c
#include <stdio.h>

void print_bool(int b){
  b? printf("True\n") : printf("False\n");
}

int is_negative(int n){
  return n < 0;
}

int main(){
  int x = 1;
  int y = -1;
  print_bool(is_negative(x));
  print_bool(is_negative(y));
}
```

Above we have a program which checks to see if two variables, `x` and `y`, are negative and then prints "True" or "False" depending on the outcome. We can take a look at the assembly genrated here (`gcc -O0 -fno-omit-frame-pointer -fno-stack-protector -S is_negative.c`):

Listing 11: is_negative.c in x86-64 assembly

```
        .file   "is_negative.c"
        .text
        .section        .rodata
.LC0:
        .string "True\n"
.LC1:
        .string "False\n"
        .text
        .globl  print_bool
        .type   print_bool, @function
print_bool:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        cmpl    $0, -4(%rbp)
        je      .L2
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        jmp     .L4
.L2:
        leaq    .LC1(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
.L4:
        nop
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   print_bool, .-print_bool
        .globl  is_negative
        .type   is_negative, @function
is_negative:
.LFB1:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        shrl    $31, %eax
        movzbl  %al, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE1:
        .size   is_negative, .-is_negative
        .globl  main
        .type   main, @function
main:
.LFB2:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movl    $1, -4(%rbp)
        movl    $-1, -8(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %edi
        call    is_negative
        movl    %eax, %edi
        call    print_bool
        movl    -8(%rbp), %eax
        movl    %eax, %edi
        call    is_negative
        movl    %eax, %edi
        call    print_bool
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

We detailed the calling convention in the chapter on functions, so I'll spare the overly detailed process here. Suffice it to say that it can be seen that **%rbp**, or the base/frame pointer register, is often being manipulated at the beginning of functions and anytime there should be an instruction where a local variable is being manipulated.

Heap

We'll learn more about the heap later when we discuss pointers. You can think of the heap as "the other stack", which it more or less is. The only difference is the heap is *persistent* between function calls, unlike the stack which is often riddled with pushes and pops that allow for old memory to be overwritten without programmer input.

Default Values

We know by now that variables do not need to be initialized with a value. By what are the values associated with an uninitialized variable? It turns out in C the behavior is *undefined*. If we declare a variable, say, `int x;`, we cannot guarantee the value that will contained in `x`. We can see that represented by compiling a simple C program,

Listing 12: Program with uninitialized int

```
1:  int main(){
2:    int x;
3:    int y = 10;
4:    return x + y;
5:  }
```

Which can be compiled down into assembly (`gcc -O0 -fno-omit-frame-pointer -fno-stack-protector -S uninit.c`):

Listing 13: Program with uninitialized int in x86-64 assembly (snapshot)

```
1:  main:
2:  .LFB0:
3:          .cfi_startproc
4:          endbr64
5:          pushq   %rbp
6:          .cfi_def_cfa_offset 16
7:          .cfi_offset 6, -16
8:          movq    %rsp, %rbp
9:          .cfi_def_cfa_register 6
10:         movl    $10, -4(%rbp)
11:         movl    -8(%rbp), %edx
12:         movl    -4(%rbp), %eax
13:         addl    %edx, %eax
14:         popq    %rbp
15:         .cfi_def_cfa 7, 8
16:         ret
```

We can see that on Line 10 a constant value of 10 was moved into `-4(%rbp)`, the stack location for `y`. However, there's no equivalent movement for `-8(%rbp)` (the stack location for `x`). This means that `x` will store whatever value happens to be in that memory location at startup!

Exercises

1. The chapter shows that trying to modify a string constant causes a segmentation fault as in Listing 6. Compile and run Listing 5 yourself and confirm the segfault. Then look at the assembly from Listing 9 and identify which section directive indicates that the string is stored in read-only memory. Why does the OS enforce this protection?

2. Compile Listing 7 and run it. Then remove the `static` keyword from the declaration of `x` and recompile. What changes in the output? Explain why in terms of storage duration.
3. Looking at the assembly in Listing 9, identify the label and section used to store the string constant "Hello, World!". Now write a C program with two string constants and compile it to assembly. How many `.rodata` entries are produced?
4. The assembly for the `static` variable in Listing 9 uses a mangled name `x.2315` instead of just `x`. Why does the compiler do this? What problem would arise if it used the plain name `x` instead?
5. Write a C program with two functions where the first function declares a local variable, sets it to a known value, and returns. The second function then declares a local variable without initializing it and prints it. Call them in sequence from `main`. Based on what you know about how the stack works from Listing 11, predict what value the uninitialized variable might hold and explain why. Does the output match your prediction? What does this tell you about relying on uninitialized variables?
6. The chapter states that the heap is "persistent between function calls" unlike the stack. Based on what you know about how the stack works from the assembly in Listing 11, explain in your own words why local variables do not persist between function calls.
7. Write a C program with a `static` local variable inside a function that counts how many times that function has been called, printing the count each time. Call it five times from `main`. Compile to assembly and identify where in the assembly the static variable is stored compared to the local variables.
8. The chapter states that global variables must be reachable by any function that references them. Write a program based on Listing 2 that has two functions both modifying the same global variable and print it after each modification. What does this demonstrate about the risks of using global variables?
9. Looking at Listing 13, the assembly shows no instruction initializing `-8(%rbp)` for the uninitialized variable `x`. Compile the same program with `gcc -Wall` and record the warning produced. Then initialize `x` to 0 and recompile to assembly — what new instruction appears?
10. The chapter distinguishes between statically and dynamically allocated memory. Categorize each of the following as static or dynamic and justify your answer:
    - A `const int` declared at the top of a file
    - A loop counter `int i` declared inside `main`
    - A `static int` declared inside a function
    - The string `"CS211"` passed directly to `printf`
    - A function argument `int x`

Footnotes:

1 Back in the day, giant tubes filled with mercury called "mercury delay lines" were used as volatile memory.

Pointers

Introduction

> “
>
> *The finger pointing at the moon is not the moon — Buddhist saying*[1]

The great benefit C provides over other programming languages is the fact that its runtime system is about as close to the hardware as any high-level programming language can conceivably get. To the novice programmer, this fact reveals itself to be a double-edged sword. C, unlike Java or other meticulously managed runtime languages, requires the programmer to be not just an expert in understanding programming abstractions, but to *also* be an expert in understanding how memory is managed during a program's execution. The average Python programmer likely has little to no idea about how the data they are manipulating is actually being handled by their machine. Conversely, the average C programmer *must* have some understanding as to what is happening under the hood during their computations. This reality flows forth from the most maligned and feared part of the language: *pointers*[2].

Note: This is often the most difficult topic of the semester. Do not be discouraged if it does not make sense right away!

Addresses

Recall that all variables in C represent locations in memory. Again, every variable that we declare in C code is nothing more than a named, symbolic stand-in for the physical location within the machine that holds data. When we declare a variable, say `int x = 5`, `x` is a useful symbol for the programmer because it is easier to reason about a symbol `x` than to reason about the relative address `-4(%rbp)`.

All memory in C is "byte addressable". This means that the smallest memory unit that a C program can target is a byte[3]. Every single byte of memory then must have an address associated with it. The structure that maintains the addresses for a program's memory is called the *address space*. Below is a diagram which represents one in the abstract:

Listing 1: Address Space

```
High Address (0xFFFFFFFFFFFFFFFF)
+------------------+
|      Stack       |  <- Local variables, function calls
|                  |     grows downward
|        |         |
|        v         |
|                  |
|                  |
|                  |
|        ^         |
|        |         |
|                  |     grows upward
|       Heap       |  <- Dynamically allocated memory
+------------------+
|       BSS        |  <- Uninitialized global/static variables
+------------------+
|      Data        |  <- Initialized global/static variables
+------------------+
|      Text        |  <- Program code (read-only)
+------------------+
Low Address (0x0000000000000000)
```

There are a few things to note. One, the two dynamic memory regions, the stack and the heap, "grow" towards one another. This is what makes them *dynamic*—the size of each is not settled beforehand. Two, we can see that program code itself is contained within the address space in the text segment. Three, that the addresses are bounded between two hexadecimal numners, 0x0 and 0xFFFFFFFFFFFFFFFF, the latter of which is equivalent to 2^64-1. This is due to the fact that x86-64 machines will run 64-bit Operating Systems which provide processes with 64-bit addresses[4].

What is a Pointer

A pointer is a variable type which stores a *memory location*. Despite all of the difficulties involved with dealing with pointers in code, the definition is rather simple. For example, if I declare a pointer `int* p`, then p can be assigned the memory location of another variable:

Listing 2: One integer pointer p

```
1:  int main(){
2:    int x = 10;  // Declaring an integer x
3:                 // with a value of 10
4:
5:    int *p = &x; // Declaring an integer pointer p
6:                 // with a value of the memory location of x
7:  }
```

(All of the operators will be dealt with in more detail below.) Here, `p` does not store 10, but rather the memory location that the variable `x` represents. If `x` is at the relative address **-4(%rbp)** then `p` stores **-4(%rbp)**, not the value contained at that address. Again, `p` is storing a *reference* to `x`. Before we get into more examples, I think it's best to describe all of the operators associated with pointers.

Defining a Pointer Type

Listing 2 defines an integer pointer `p` on Line 5. It does so using the type *declarator* *. Adding a * prior to the variable name indicates that it is a pointer type variable. For example, `char *cp` is a `char` pointer, `double *dp` is a `double` pointer, `int *ip` is an `int` pointer, and `float *fp` is a `float` pointer.

Pointer types can have multiple levels of *indirection*. This means that I can further alter some of the above definitions to include something like `char **cpp` which defines `cpp` as a pointer to a `char` pointer. This can be done arbitrarily[5].

Pointer Operators

One of the holdovers of C development ocurring in the 1970s is that the symbol set to represent a multitude of operators was rather small. That paired with a desire to cut down on the potential visual noise of many bespoke operators led the language designers to reuse operators in different contexts. Many of these symbols we've seen in different contexts, but here they are used for pointer-specific purposes.

* operator

*** is a unary operator which is used for dereferencing pointers.

Given an integer pointer `p` which holds a reference to a memory location storing the integer 10:

- `*p` → `10`

& operator

& is a unary operator which returns the memory location of the variable in its operand.

Given an `int`, `x`, which is stored at memory location `0x77ffff`:

- `&x` → `0x77ffff`

+ operator

+ is the same binary addition operator we encountered before. The difference here is that only integers can be used when a pointer is involved. The expression resolves the sum of the pointer in the left operand with the integer offset in the right operand. It will evaluate (left operand location) + (sizeof(type) * offset):

Given an integer pointer, `ip`, which is pointing to memory location `0x0`:

- `ip + 4` → `0x10` (on a standard x86-64 machine with a 4 byte `int`)

Given a character pointer, `cp`, which is pointing to memory location `0x0`:

- `ip + 4` → `0x4` (on a standard x86-64 machine with a 1 byte `char`)

Dereferencing a Pointer

One of the trickiest parts when working with pointers is understanding what it means to *dereference* them. Dereferencing just means that you are telling the C runtime to "look into" the reference that is stored inside of the pointer. Again, the pointer *points to* something. Dereferencing has the program trying to get the thing that is being pointed to. Here's a simple example which takes Listing 2 and modifies it slightly:

Listing 3: Dereferencing p

```
1:   #include <stdio.h>
2:   int main(){
3:       int x = 10;
4:       int *p = &x;        // Here, * is used as a declarator
5:       printf("%d\n", *p); // Here, * is used as a dereference operator
6:     }
```

Line 3 declares an `int`, `x`, and sets its value equal to 10. Line 4 declares a pointer to an `int`, `p`, and sets its value equal to the memory location of `x`. Line 5 *dereferences* the `p` by jumping to the memory location stored in the variable `p` and returning the value contained therein. We can compile and run the program like so:

Listing 4: Dereferencing p results

```
josephraskind@stargazer:/tmp/pointers$ gcc deref.c -o deref; ./deref
10
```

Why is 10 returned and not the memory location of `x`? Again, its because the C runtime has looked inside of the memory location stored in `p` and retrieved the value that it was storing.

Let's expand Listing 3 somewhat:

Listing 5: More with p

```
 1:   #include <stdio.h>
 2:   int main(){
 3:       int x  = 10;
 4:       int *p = &x;
 5:       printf("Value of &x:  0x%p\n", &x);  // Memory location of x
 6:       printf("Value of p:   0x%p\n", p);   // Value stored in p
 7:       printf("Value of &p:  0x%p\n", &p);  // Memory location of p
 8:       printf("Value of *p:  %d\n", *p);    // Dereferencing p
 9:       printf("Value of x:   %d\n", x);     // Value stored in x
10:     }
```

Let's compile and run the above:

Listing 6: Listing 5 results

```
1:   josephraskind@stargazer:/tmp/pointers$ gcc deref_exp.c -o deref_exp; ./
deref_exp
2:   Value of &x:  0x0x7ffe140349ac
3:   Value of p:   0x0x7ffe140349ac
4:   Value of &p:  0x0x7ffe140349b0
5:   Value of *p:  10
6:   Value of x:   10
```

Let's take the output of Listing 6 one line at a time. Line 2 shows the output of `&x` which is the memory address that `x` was assigned. Line 3 shows that the `p` variable was storing *exactly that* memory address—this should make sense as `p` is a pointer to `x`. Line 4 shows the memory address of `p`; `p`'s memory address is distinct from `x`'s because they are *two different variables*.

Line 5 shows the result from dereferencing `p`; the C runtime looks inside of `0x0x7ffe140349ac` and finds the value `10`. Line 6 shows the value of `x`, which we would expect to be 10.

Let's expand this one level further by introducing a double pointer `pp`:

Listing 7: p now with pp

```
 1:   #include <stdio.h>
 2:   int main(){
 3:       int x   = 10;
 4:       int *p  = &x;
 5:       int **p = &p;
 6:       printf("Value of &x:   0x%p\n", &x);   // Memory location of x
 7:       printf("Value of p:    0x%p\n", p);    // Value stored in p
 8:       printf("Value of &p:   0x%p\n", &p);   // Memory location of p
 9:       printf("Value of *p:   %d\n", *p);     // Dereferencing p
10:       printf("Value of x:    %d\n", x);      // Value stored in x
11:       printf("\n");
12:       printf("Value of pp:   0x%p\n", pp);   // Value stored in pp
13:       printf("Value of &pp:  0x%p\n", &pp);  // Memory location of pp
14:       printf("Value of *pp:  0x%p\n", *pp);  // Dereferencing pp
15:       printf("Value of **pp: 0x%p\n", **pp); // Dereferencing pp, twice
16:     }
```

And then we compile and run it:

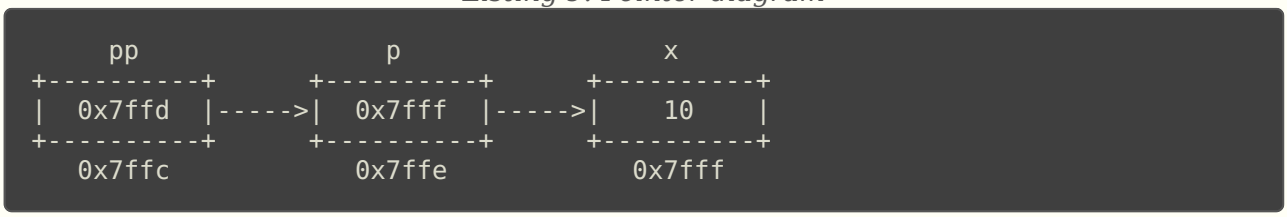
Listing 8: Listing 7 results

```
 1:   josephraskind@stargazer:/tmp/pointers$ gcc deref_exp_pp.c -o
deref_exp_pp; ./deref_exp_pp
 2:   Value of &x:   0x0x7fffb54bd364
 3:   Value of p:    0x0x7fffb54bd364
 4:   Value of &p:   0x0x7fffb54bd368
 5:   Value of *p:   10
 6:   Value of x:    10
 7:
 8:   Value of pp:   0x0x7fffb54bd368
 9:   Value of &pp:  0x0x7fffb54bd370
10:   Value of *pp:  0x0x7fffb54bd364
11:   Value of **pp: 10
```

The same line by line analysis should be helpful. Line 9 outputs the address stored by `pp` which is equivalent to the address of `p` on Line 4. Line 10 outputs the address of `pp`. Line 11 outputs the value stored at the location pointed to by `0x0x7fffb54bd368` which happens to be the memory address of `x`, or what is stored in `p`. Line 12 shows that we can stack dereferences: *(1)* We dereference to look inside of the memory location of `p` to grab the memory location of `x`; *(2)* We dereference the memory location of `x` to get the value of `x` which is 10.

This works because a pointer is nothing more than a variable that stores a *memory location*:

Listing 9: Pointer diagram



Dereferencing for Assignment

We can also use pointers to store values in memory locations. If we take our simple *p* example from Listing 2 and alter it slightly:

*Listing 10: Assigning p with **

```
1:   #include <stdio.h>
2:   int main(){
3:       int x = 10;
4:       int *p = &x;
5:       printf("(Before) Value of x: %d\n", x);
6:       *p = 5; // Dereference with assignment
7:       printf("(After)  Value of x: %d\n", x);
8:     }
```

If we compile and run the above:

*Listing 11: Assigning p with * results*

```
1:   josephraskind@stargazer:/tmp/pointers$ gcc assignment.c -o assignment; ./
assignment
2:   (Before) Value of x: 10
3:   (After)  Value of x: 5
```

What happened?! It turns out that the dereference operator returns both an **lvalue** *and* an **rvalue**, much the same as a simple named variable expression might. Recall that `x` is nothing more than a memory location playing dressup. That means when you have a memory location in the form of a pointer, `p`, then it can act in the same way that `x` does. The * on the left side of the assignment statement tells the compiler to treat whatever is stored in the pointer as a real, valid memory address just like `x`.

Dereferences That Go Wrong

Dereferences are not always guaranteed to work out. As per the C Standard spec, "If an invalid value has been assigned to the pointer, the behavior of the unary * operator is undefined." In most situations, this will likely lead to a *segmentation fault*. Here's some code to set this up:

Listing 12: Creating a bad dereference

```
1:  #include <stdio.h>
2:  int main(){
3:      int *p = 0;
4:      printf("Value of *p: %d\n", *p);
5:    }
```

If we compile and run it we'll see:

Listing 13: Creating a bad dereference result

```
josephraskind@stargazer:/tmp/pointers$ gcc bad_deref.c -o bad; ./bad
Segmentation fault (core dumped)
```

The code compiles just fine, but fails during the runtime execution of the program. This is because `gcc` has to respect the fact that `p` *could* point to somewhere meaningful and, as such, will generate code that tries to chase the reference. Of course, since it points to 0, or the *null* pointer, the address is invalid and does not store any date that can be interpreted as an `int`.

Dereferencing with Mixed Types

During our discussion on types I made mention of the fact that C is a *weakly* typed language. The biggest case for that comes in the ability to explicitly cast pointer types freely. For example, look at the code below:

Listing 14: Explicitly casting pointers

```
 1:  #include <stdio.h>
 2:  int main(){
 3:    unsigned int x = 0xFFFFFF41;
 4:    int          *ip = &x;             // no casting, x is an int
 5:    char         *cp = (char *) &x;    // casting to a char*
 6:    float        *fp = (float *) &x;   // casting to a float*
 7:    double       *dp = (double *) &x;  // casting to a double*
 8:
 9:    printf("Value of *ip: 0x%x\n", *ip); // Printing hex of x
10:    printf("Value of *cp: %c\n", *cp);   // Printing x as if it was a char
11:    printf("Value of *cp: %s\n", cp);    // Printing x as if it was a
string
12:    printf("Value of *fp: %f\n", *fp);   // Printing x as if it was a float
13:    printf("Value of *dp: %f\n", *dp);   // Printing x as if it was a
double
14:  }
```

After compiling and running it we get:

Listing 15: Explicitly casting pointers results

```
josephraskind@stargazer:/tmp/pointers$ gcc mixed.c -o mixed; ./mixed
Value of *ip: 0xffffff41
Value of *cp: A
Value of *cp: A\377\377\377□□"$\377□
Value of *fp: -nan
Value of *dp: 0.000000
```

C lets us freely cast the `int *` returned by `&x` and, when we do so, we lose the original, and valuable, type information we started with! Other, more strongly typed languages like Java would have stopped the sort of manipulation shown on Line 4—it makes no semantic sense to treat an `int` like a string, and yet C allows that to happen as long as we pass through pointers first. Again, this comes from the language's implicit assumption that the programmer knows what she is doing and that she wants a language which is willing to forgo a strong type system for something that allows this sort of low-level manipulation.

void Pointers

As per the C language spec, "The void type comprises an empty set of values; it is an incomplete object type that cannot be completed." This does not preclude the creation and use of a `void` pointer type. In fact, C programmers often use `void` pointers in their day to day tasks. Many standard library functions, like `malloc`, return `void` pointers as the absence of a type is necessary to preserving the abstraction associated with the function call.

Dereferencing void Pointers

It is impossible to capture the value of a `void` pointer using the dereference operator. This is because `void` does not map to a data type, but rather represents the absence of a type. `void` pointers can be cast to any other pointers and dereferenced afterwards (see the above section).

Arrays

Arrays refer to a special sort of type which represent contiguously allocated nonempty values. Arrays have a base, or *element*, type associated with them. For example, we can define an integer array, `arr`, with ten elements using the following syntax: `int arr[10]`. Arrays are not guaranteed to have set default values. We can use an initializer to guarantee zeros with `int arr[10] = {0}`. Let's have a look:

Listing 16: Uninitialized array

```
1:  #include <stdio.h>
2:  int main(){
3:    int arr[5];                  // Declaring an int array of 5 elements
4:    for(int i = 0; i < 5; i++)
5:      printf("%d\n", arr[i]);    // The [] are used for selecting an element
6:    int arr_z[5] = {0};          // Declaring a new array initialized to 0
7:    for(int i = 0; i < 5; i++)
8:      printf("%d\n", arr_z[i]);
9:  }
```

And after we compile and run the above:

Listing 17: Uninitialized array results

```
1:  josephraskind@stargazer:/tmp/pointers$ gcc arr_init.c -o arr; ./arr
2:  1894002662
3:  32766
4:  1254343277
5:  22066
6:  -525053208
7:  0
8:  0
9:  0
10: 0
11: 0
```

Lines 2-6 print out the elements of the unitialized array. Lines 7-11 print out the elements of the initialized array.

Arrays are *contiguous* in memory. This means that there is an entire block of memory reserved for the elements of that array side-by-side. When I write `int x[5]`, that means that on my machine 20 bytes are reserved for the array `x` (5 4 byte ints). We can see this in code:

Listing 18: Contiguous array

```
1:  #include <stdio.h>
2:  int main(){
3:    int arr[5] = {1,2,3,4,5};
4:    for(int i = 0; i < 5; i++)
5:      printf("0x%p: %d\n", &arr[i], arr[i]); // Print the memory location
of each element and its value
6:    printf("%ld\n", sizeof(arr));           // Print the size of the array
7:  }
```

After compiling and running the above:

Listing 19: Contiguous array results

```
josephraskind@stargazer:/tmp/pointers$ gcc contiguous.c -o arr; ./arr
0x0x7ffd1aa80530: 1
0x0x7ffd1aa80534: 2
0x0x7ffd1aa80538: 3
0x0x7ffd1aa8053c: 4
0x0x7ffd1aa80540: 5
20
```

You'll notice that every address is 4 spots higher than the last. Again, this is because each element is an `int`, and in x86-64 machines an `int` is usually 4 bytes.

[] operator

[] is an operator used for extracting values in an array. Given an integer array, `n`, of the first 10 natural numbers:

- `n[0]` → `1`
- `n[1]` → `2`
- etc.

Note: Arrays are indexed by 0. The first element matches to the index 0, the second element matches to the index 1, and so on.

[] also returns *lvalues*, so they can be used for assignment.

- if `n[0] = 10` then `n[0]` → `1`

Multidimensional Arrays

Arrays can take on more than 1 dimmension. For example, I could construct a 3x4 matrix using the following syntax: `int matrix[3][4]`. This creates an array of 12 integer elements which can be indexed using two sets of brackets. Conceptually we can think of the matrix looking like:

Listing 20: Two dimensional array

```
        [0]        [1]        [2]        [3]
     +----------+----------+----------+----------+
[0]  |    1     |    2     |    3     |    4     |
     +----------+----------+----------+----------+
[1]  |    5     |    6     |    7     |    8     |
     +----------+----------+----------+----------+
[2]  |    9     |    10    |    11    |    12    |
     +----------+----------+----------+----------+
```

But in memory it will look like:

Listing 21: Two dimensional array in memory

```
+--------+--------+--------+--------+--------+--------+--------+--------
+--------+--------+--------+--------+
|   1    |   2    |   3    |   4    |   5    |   6    |   7    |   8    |
9    |   10   |   11   |   12   |
+--------+--------+--------+--------+--------+--------+--------+--------
+--------+--------+--------+--------+
  [0][0]   [0][1]   [0][2]   [0][3]   [1][0]   [1][1]   [1][2]   [1][3]   [2]
[0]   [2][1]   [2][2]   [2][3]
```

Arrays vs Pointers

Functionally, there's not an awful lot separating arrays and pointers. The main difference is that arrays define a contiguous block of memory. Pointers do not. Below we can see similarities in their syntax/behavior:

Listing 22: Pointers and arrays

```c
#include <stdio.h>
int main(){
  int arr[5] = {1,2,3,4,5};  // Initializing int array
  int *p = &(arr[0]);        // Getting the pointer to the first elem

  printf("Value of arr[1]:   %d\n", arr[1]);
  printf("Value of p[1]:     %d\n", p[1]);

  printf("Value of *(arr+3): %d\n", *(arr+3));
  printf("Value of *(p+3):   %d\n", *(p+3));
}
```

Compiling and running the above:

Listing 23: Pointers and arrays results

```
josephraskind@stargazer:/tmp/pointers$ gcc arr_and_p.c -o arr; ./arr
Value of arr[1]:   2
Value of p[1]:     2
Value of *(arr+3): 4
Value of *(p+3):   4
```

We can see that pointers and arrays are treated almost identically from the perspective of the **[]**, *, and **+** operators.

Dynamic Allocation

Until now, memory allocation has been the compiler's job. We've never had to worry much about reserving space on the stack to make sure there was a place to store our data because the compiler would guarantee there was always enough room. Sometimes, however, it's advantageous to be more particular about memory management. The C standard library provides functions like `malloc` and `calloc` to do exactly that.

Why Dynamic Allocation?

The best way to explain the advantages of dynamic allocation is to illustrate what happens when we don't bother with it:

Listing 24: Greeting function with static allocation

```
 1:   #include <stdio.h>
 2:   #include <stdlib.h>
 3:   #include <string.h>
 4:
 5:   char *make_greeting(char *name){
 6:     char greeting[50];             // Reserving 50 bytes statically
 7:     strcpy(greeting, "Hello, ");   // Copying string constant "Hello, "
into greeting
 8:     strcat(greeting, name);        // Concatenating name with greeting.
 9:     return greeting;
10:   }
11:
12:   int main(){
13:     char *g = make_greeting("Joe");
14:     printf("%s\n", g);
15:     free(g);
16:   }
```

In the above code I'm sending a string constant to the function `make_greeting` which is meant to prepend `"Hello, "` to it and return the resulting string. Let's see what happens when I try to compile and run it:

Listing 25: Greeting function with static allocation results

```
1:   josephraskind@stargazer:/tmp/pointers$ gcc greeting_static.c -o
greeting_static; ./greeting_static
2:   greeting_static.c: In function 'make_greeting':
3:   greeting_static.c:9:10: warning: function returns address of local
variable [-Wreturn-local-addr]
4:       9 |   return greeting;
5:         |          ^~~~~~~~
6:   Segmentation fault (core dumped)
```

We get a segmentation fault! We can already see that the compiler is giving us a hint about what will, and did, happen. The char array `greeting` is a *local variable* whose existence is tied only to the function call. The moment the `make_greeting` function is exited the compiler is free to forget about any local variables and their corresponding memory locations. What's returned by `make_greeting` is a pointer to a variable that *no longer exists*. As such, when `printf` tries to interpret the bits that `g` points to there's an error whic causes a halt to the program execution.

Now let's try it with dynamic allocation using `malloc`:

Listing 26: Greeting function with dynamic allocation

```
 1:   #include <stdio.h>
 2:   #include <stdlib.h>
 3:   #include <string.h>
 4:
 5:   char *make_greeting(char *name){
 6:     char *greeting = malloc(50);   // Reserving 50 bytes dynamically
 7:     strcpy(greeting, "Hello, ");   // Copying string constant "Hello, "
into greeting
 8:     strcat(greeting, name);        // Concatenating name with greeting.
 9:     return greeting;
10:   }
11:
12:   int main(){
13:     char *g = make_greeting("CS211");
14:     printf("%s\n", g);
15:     free(g);
16:   }
```

After we compile and run it:

Listing 27: Greeting function with dynamic allocation results

```
1:   josephraskind@stargazer:/tmp/pointers$ gcc greeting_dynamic.c -o
greeting_dynamic; ./greeting_dynamic
2:   Hello, CS211
```

And with that little change we no longer experience a segmentation fault! The `greeting` variable is still local, and the variable memory itself is forgotton, yet the program runs without any problems. This happened because the standard library function `malloc` reserved memory *on the heap*, outside of the local stack. The memory address where those contiguous bytes were reserved was returned by `malloc` and placed within the `greeting` variable. This is also why `g` is able to be referenced without error—it was passed back the memory location returned by `malloc`.

Because the heap is distinct from the stack there is no danger of the reserved memory on the heap being overwritten from other function calls or other local variable declarations.

How to Explicitly Allocate Memory?

Although it may be difficult to grasp conceptually, it is actually very easy to use the C standard library functions dedicated to dynamic allocation. For this class, we'll only focus on `malloc` and `calloc`. They are both in the **stdlib.h** header file.

malloc

`malloc` stands for "memory allocation". It is a function that takes an integer and returns a `void` pointer to a memory location corresponding to the beginning of the reserved space. Below shows an example of allocating a single `int` as well as an `int` array:

Listing 28: Using malloc

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:
4:  int main(){
5:    int *i     =  (int *) malloc(sizeof(int));      // Allocating a single
int
6:    int *i_arr =  (int *) malloc(sizeof(int) * 5);  // Allocating int[5]
7:  }
```

Just like when defining local variables, there is no guarantee that the integers contained therein are zeros. We can guarantee that with `calloc`:

Listing 29: Using calloc

```
1:  #include <stdio.h>
2:  #include <stdlib.h>
3:
4:  int main(){
5:    int *i     =  (int *) calloc(1, sizeof(int));      // Allocating and
zeroing a single int
6:    int *i_arr =  (int *) calloc(5, sizeof(int));  // Allocating and
zeroing int[5]
7:  }
```

`calloc` takes the number of elements desired and the size of each one and sets all bits in the contiguous space to 0.

Note: Since `malloc` returns a `void *` it's a best practice to explicitly cast it to the desired pointer type.

What is a Memory Leak?

You may have notice on Line 15 in Listing 26 there is a reference to a function called `free`. `free` *de*-allocates memory that has been allocated with members of the `malloc` family. Without the use of `free`, there is nothing to "clean up" unused memory. This is what makes dynamic memory allocation explicit on the part of the programmer---*you must always clean up your mess*. Memory leaks are not easy to spot during program execution, so tools like `valgrind` can be useful for checking for leaks. Here is an example of me using valgrind on the code from Listing 29:

Listing 30: Using valgrind

```
josephraskind@stargazer:/tmp/pointers$ gcc m.c -o m; valgrind ./m
==7175== Memcheck, a memory error detector
==7175== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7175== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
info
==7175== Command: ./m
==7175==
==7175==
==7175== HEAP SUMMARY:
==7175==     in use at exit: 24 bytes in 2 blocks
==7175==   total heap usage: 2 allocs, 0 frees, 24 bytes allocated
==7175==
==7175== LEAK SUMMARY:
==7175==    definitely lost: 24 bytes in 2 blocks
==7175==    indirectly lost: 0 bytes in 0 blocks
==7175==      possibly lost: 0 bytes in 0 blocks
==7175==    still reachable: 0 bytes in 0 blocks
==7175==         suppressed: 0 bytes in 0 blocks
==7175== Rerun with --leak-check=full to see details of leaked memory
==7175==
==7175== For lists of detected and suppressed errors, rerun with: -s
==7175== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Valgrind runs the code in a virtual environment which uses reference counting to ensure an accurate sweep for memory leaks. The two calls to malloc return addresses which are never subsequently cleaned up and, as a result, lead to 24 bytes unaccounted for. If we add two `free` calls to the end of the code we'll find valgrind will not detect any leaks:

Listing 31: Using free on Listing 28

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
  int *i     =  (int *) malloc(sizeof(int));      // Allocating a single
int
  int *i_arr =  (int *) malloc(sizeof(int) * 5);  // Allocating int[5]
  free(i);
  free(i_arr);
}
```

After compiling and running:

Listing 32: Using free on Listing 28 results

```
josephraskind@stargazer:/tmp/pointers$ gcc m.c -o m; valgrind ./m
==7221== Memcheck, a memory error detector
==7221== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7221== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
info
==7221== Command: ./m
==7221==
==7221==
==7221== HEAP SUMMARY:
==7221==     in use at exit: 0 bytes in 0 blocks
==7221==   total heap usage: 2 allocs, 2 frees, 24 bytes allocated
==7221==
==7221== All heap blocks were freed -- no leaks are possible
==7221==
==7221== For lists of detected and suppressed errors, rerun with: -s
==7221== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Memory leaks are less of an observable problem for our class projects and labs, but it is important you are aware of them including how they can happen and how they can be fixed.

Function Pointers

Our last for foray into pointers leads us into some really unintuitive territory with *function pointers*. It turns out that not only can we have pointers that point to data and other pointers, but we can also have pointers that point to functions. How is that possible? Recall that pointers store locations in memory. If we combine that with the knowledge that functions are also stored in memory then it's only logical that pointers should also be able to point to functions. Here's an example:

Listing 33: Using function pointers

```
 1:  #include <stdio.h>
 2:
 3:  int cube(int c){
 4:    return c * c * c;
 5:  }
 6:
 7:  int square(int s){
 8:    return s * s;
 9:  }
10:
11:  int add_one(int a){
12:    return a + 1;
13:  }
14:
15:  int apply(int x, int (*fn)(int)){ // The second argument is a function
pointer which takes an int and returns an int
16:    return fn(x);                   // Calling a function using the pointer
17:  }
18:
19:  int main(){
20:    int (*funcs[3])(int) = {add_one, square, cube}; // An array of
function pointers
21:    int v = 10;
22:    for(int i = 0; i < 3; i++){
23:      int res = apply(v, funcs[i]);               // Providing function
pointers as arguments to the apply function
24:      printf("%d\n", res);
25:    }
26:  }
```

Wow…what a nightmare! If we restrict our view to the `main` method we can see a declaration of a variable `funcs` which is an array of function pointers. The pointers are to functions which take an `int` (shown to the right of the parentheses) and return an `int` (shown to the left of the parentheses). I then send each function pointer into a function called `apply` which also takes a regular int. Line 16 shows that `apply` calls the function pointer's function reference with the argument provided by the call to `apply`. We can see the the output below:

Listing 34: Using function pointers results

```
1:  josephraskind@stargazer:/tmp/pointers$ gcc func_p.c -o func_p; ./func_p
2:  11
3:  100
4:  1000
```

First `add_one` is called, then `square`, then `cube`.

Function pointers are a very powerful abstraction that allows C to operate similarly to higher-level functional programming languages like *Lisp* and *Haskell*.

Exercises

1. Compile and run Listing 5 on your own machine. Do the memory addresses match those shown in the chapter? Why or why not? What stays the same between runs on the same machine?

2. Draw a pointer diagram in the style of Listing 9 for the following code before running it, then verify by compiling and printing the addresses:

```
int x = 42;
int *p = &x;
int **pp = &p;
```

3. Write a C program that declares an `int x = 5` and a pointer `int *p = &x`. Using only `*p` (no direct reference to `x`), change the value of `x` to 100 and print it. Verify that printing `x` directly also shows 100.
4. The chapter shows that dereferencing a null pointer causes a segmentation fault as in Listing 12. Compile and run it yourself. Then run it under `valgrind` and record the output. What additional information does `valgrind` provide compared to just running the program?
5. Write a C program that declares `int x = 0xFFFFFF41` and prints it as an `int`, a `char`, and a `float` by casting the pointer, as in Listing 14. Before running, predict what each output will be based on what you know about how each type interprets bits.
6. The chapter states that arrays and pointers behave similarly. Write a program based on Listing 22 that declares an `int` array of 5 elements and a pointer to its first element. Access the third element using four different methods: `arr[2]`, `*(arr+2)`, `p[2]`, and `*(p+2)`. Verify all four produce the same result.
7. Write a C program that declares an uninitialized `int` array of 5 elements as in Listing 16 and prints each element alongside its memory address. What is the distance in bytes between each address? Does this match what you expect given the size of an `int`?
8. The chapter states that the `[]` operator returns an lvalue. Write a program that uses `[]` on the left side of an assignment to modify elements of an array, then prints the modified array to verify.
9. Rewrite the static allocation version of `make_greeting` from Listing 24 to use `malloc` instead as shown in Listing 26. Compile both versions and compare the compiler warnings. Explain in your own words why the `malloc` version works and the static version does not.
10. Write a program that uses `malloc` to allocate an `int` array of 10 elements, sets each element to its index multiplied by 2, prints all elements, and then frees the memory. Run it under `valgrind` to confirm there are no leaks.
11. Write the same program as above using `calloc` instead of `malloc`. Print the array before setting any values. What do you observe and why does it differ from `malloc`?

12. The chapter shows that `valgrind` reports 24 bytes leaked for two `malloc` calls in Listing 30. Explain why the total is 24 bytes given that only 4 bytes were requested for the single `int` and 20 bytes for the `int` array. Does this match what you'd expect?
13. Write a program that intentionally creates a memory leak by calling `malloc` without a corresponding `free`. Run it under `valgrind` and record the output. Then fix the leak and run `valgrind` again to confirm it reports no leaks.
14. Declare a 3x4 `int` matrix as in Listing 20 and initialize it with values 1 through 12. Print each element alongside its memory address using nested `for` loops. Verify that the addresses are contiguous and separated by 4 bytes.
15. The chapter shows that the `+` operator on pointers accounts for the size of the type. Write a program that declares both an `int` array and a `char` array and prints the addresses of each element. What is the difference in byte offsets between elements of each array? Does it match what the chapter predicts?
16. Write a program using function pointers based on Listing 33 that defines three functions: one that doubles its input, one that negates it, and one that returns its absolute value. Store them in a function pointer array and apply each to the value -5 using an `apply` function.
17. The chapter states that `void` pointers cannot be dereferenced directly. Write a program that allocates memory with `malloc`, stores the result in a `void *`, then casts it to an `int *` and stores a value through it. Print the value by dereferencing the cast pointer.
18. Write a program that declares `int x = 10`, `int *p = &x`, and `int **pp = &p`. Using only `**pp`, change the value of `x` to 99. Print `x`, `*p`, and `**pp` to verify all three reflect the change.
19. The chapter mentions that the text segment stores program code in read-only memory. Based on what you know from the memory chapter about the `.rodata` section, explain why attempting to modify a string constant causes a segmentation fault but modifying a `char` array initialized from a string constant does not. Write a program demonstrating the difference between the two.
20. Write a program that uses a function pointer to implement a simple calculator. Define four functions for addition, subtraction, multiplication, and integer division, each taking two `int` arguments and returning an `int`. Store them in a function pointer array and loop through all four, applying each to the values 10 and 3 and printing the result.

Footnotes:

1 As recorded in *Types and Programming Languages* by Benjamin Pierce.

2 Dun DUn DUN!!!

3 Recall that a "byte" in C just means "addressable unit of data storage large enough to hold any member of the basic character set of the execution environment". In our x86-64 machines, that maps to a real byte in the binary sense (8-bits), but it doesn't necessarily have to be the case.

[4] This is not the course to delve too deep on what's happening behind the scenes. Any operating systems course worth its salt *will* do so.

[5] Oftentimes, the level of indirection does not go beyond five pointers deep. Theoretically, it can and God help you if it does.

Struct

Introduction

C provides only a select few predefined types: `char`, `int`, `float`, `double`. Some of these types can be further modified to represent `long` and `short` versions, as well as be made `signed` or `unsigned`. Every element type can be represented as a pointer using the `*` delcarator. There's even a type that represents the "absence" of a type with `void`. For many programmers, this simply isn't enough. C also furnishes space for those wish to go a bit further with the addition of *structs*.

What is a struct?

A `struct` is an aggregate data type defined by the programmer which combines other predefined data types into one larger grouping. From the C specification:

> *A structure type describes a sequentially allocated nonempty set of member objects (and, in certain circumstances, an incomplete array), each of which has an optionally specified name and possibly distinct type.*

The above technical definition sounds a bit scarier than it is. Here is an example of a struct declaration:

Listing 1: Defining a struct

```
1:    struct frac{
2:    int n;   // Numerator
3:    int d;   // Denominator
4:  };
```

Here, I declare a new data type called `struct frac` which contains two **member variables**, `n` and `d`. I can declare a variable of type `struct frac` like this (assuming the struct has already been declared above):

Listing 2: Declaring a variable of type struc frac

```
1:  int main(){
2:    struct frac f; // Declaring a variable "f" of type "struct frac"
3:    f.n = 1;       // Using "." to reference member "n"
4:    f.d = 2;       // Using "." to reference member "d"
5:  }
6:
```

The code above shows the declaration of a variable `f` of type "struct frac" and the setting of both of its member variables.

Much like arrays, initialization lists can be used to set the members:

Listing 3: Using an initialization list

```
1:  int main(){
2:    struct frac f = {1, 2};
3:  }
```

Above, `n` and `d` are set using the initilization list. Initilization lists will set the values in the same order in which they were defined (`{1,2}` will match `{n,d}`).

Using typedef with structs

Using the definition shown in Listing 3, I would be forced to always write `struct frac` when referencing my bespoke fraction type. This leads to visual clutter in the code that can often be tiresome to the eye. A way to avoid that is through the use of the `typedef` keyword which will map a given symbol to a type definition. For example:

Listing 4: Declaring a variable of type struc frac

```
 1:  typedef struct frac{
 2:    int n;
 3:    int d;
 4:  }frac; // Matching a symbol "frac" to the entire type definition
 5:
 6:  int main(){
 7:    frac f;        // Declaring a variable "f" of type "frac"
 8:    f.n = 1;       // Using "." to reference member "n"
 9:    f.d = 2;       // Using "." to reference member "d"
10:  }
```

Although it only eliminated a little bit of extra keyword usage in the above example, this can be rather useful when applied to longer stretches of code.

Dereferencing struct pointers

It is entirely possible to create pointers to structs. For example, I might want to place one of my `frac`s on the heap. We know that using `malloc` to do so will return a pointer. We can access members of struct references in one of two ways, both shown below:

Listing 5: Dereferencing members of a struct pointer

```
1:  #include<stdio.h>
2:
3:  typedef struct frac{
4:    int n;
5:    int d;
6:  } frac;
7:
8:  int main(){
9:    frac* fp = malloc(sizeof(frac));
10:   (*fp).n  = 1;       // Using "*()." to reference member "n"
11:   fp->d    = 2;       // Using "->" to reference member "d"
12: }
```

Line 10 shows that we can dereference `fp` like any other pointer type. Once that's done we should get the `frac` returned by the dereference and then use the `.` notation to grab the `n` member. Line 11 condenses both with the `->` notation—`f` is dereferenced and then member `d` is selected.

Why use structs?

Structures are incredibly useful when programming in C. They allow for programmers to define type-specifc logic to functions that would not be aware of it otherwise. For example, we could utilize an array as if it were a derived type like frac as follows:

Listing 6: Using an array as a struct stand-in

```
1:  void print_frac(void * arr){
2:    int * f = (int *) arr;
3:    printf("%d/%d\n",  f[0], f[1]);
4:  }
5:
6:  int main(){
7:    int frac1[2] = {1,2};
8:    char* frac2[2] = {"hello", "world"};
9:
10:   print_frac(frac1);
11:   print_frac(frac2);
12: }
```

The above will not throw warnings or provide any information to the user that there is any attempt to break the underlying logic of the `print_frac` function. The C compiler lacks additional type information to see that:

Listing 7: Using an array as a struct stand-in results

```
1:  josephraskind@stargazer:/tmp/struct$ gcc arr_struct.c -o arr_struct; ./
arr_struct
2:  1/2
3:  2032603147/21931
```

If we alter the code in Listing 6 to now be:

Listing 8: Using a struct

```
 1:   #include <stdio.h>
 2:
 3:   typedef struct frac{
 4:     int n;
 5:     int d;
 6:   } frac;
 7:
 8:   void print_frac(frac f){
 9:     printf("%d/%d\n",  f.n, f.d);
10:   }
11:
12:   int main(){
13:     frac frac1 = {1,2};
14:     frac frac2 = {"hello", "world"};
15:
16:     print_frac(frac1);
17:     print_frac(frac2);
18:   }
```

We can compile and run it:

Listing 9: Using an array as a struct stand-in results

```
 1:   josephraskind@stargazer:/tmp/struct$ gcc strict_struct.c -o
strict_struct; ./strict_struct
 2:   strict_struct.c: In function ‘main’:
 3:   strict_struct.c:14:17: warning: initialization of ‘int’ from ‘char *’
makes integer from pointer without a cast [-Wint-conversion]
 4:      14 |   frac frac2 = {"hello", "world"};
 5:         |                 ^~~~~~~
 6:   strict_struct.c:14:17: note: (near initialization for ‘frac2.n’)
 7:   strict_struct.c:14:26: warning: initialization of ‘int’ from ‘char *’
makes integer from pointer without a cast [-Wint-conversion]
 8:      14 |   frac frac2 = {"hello", "world"};
 9:         |                          ^~~~~~~
10:   strict_struct.c:14:26: note: (near initialization for ‘frac2.d’)
11:   1/2
12:   1715884043/1715884049
```

Although C allows the logic to be broke, the compiler does provide useful type mismatch information in the form of warnings which could be used to debug any apparent problems.

How are structs Stored?

In order for structs to function as data types in C there size must be known at compile time. This can be seen in Line 9 of 5 which shows that the built-in `gcc` macro `sizeof` is capable of being applied to the user-defined struct symbol. The size of structs are not always obvious *a priori*. For example, take the following struct definitions and the results of `sizeof` on each of them:

Listing 10: Struct padding

```c
#include <stdio.h>
struct padded {
    char  a;  // 1 byte  -- offset 0
              // 3 bytes padding
    int   b;  // 4 bytes -- offset 4
    char  c;  // 1 byte  -- offset 8
              // 7 bytes padding
    double d; // 8 bytes -- offset 16
};            // total: 24 bytes

struct packed {
    double d; // 8 bytes -- offset 0
    int    b; // 4 bytes -- offset 8
    char   a; // 1 byte  -- offset 12
    char   c; // 1 byte  -- offset 13
              // 2 bytes padding
};            // total: 16 bytes

int main(){
  printf("%lu\n", sizeof(struct padded));  // 24
  printf("%lu\n", sizeof(struct packed));  // 16
}
```

Listing 11: Struct padding results

```
josephraskind@stargazer:/tmp/struct$ gcc struct_size.c -o size; ./size
24
16
```

These results are for my x86-64 machine—it could be different depending on the architecture. The padding is a result of the natural word size of x86-64 machines being 8 bytes long. The architecture expects each variable to be aligned 4/8 byte boundaries, so the C compiler which targets x86-64 will comply by stuffing "padding" bits in the slots in between.

Note: There are ways of overriding bit padding, but they are well outside the scope of this class.

Exercises

1. Implement the following functions for the `frac` struct defined in Listing 3:

   - `print frac` — prints the fraction in the form `n/d`
   - `add frac` — adds two fractions and returns the result
   - `mul frac` — multiplies two fractions and returns the result
   - `simplify` — divides numerator and denominator by their GCD

   Test each with at least two examples and verify the results by hand.

2. Using `typedef` as shown in Listing 4, rewrite your solutions from exercise 1 so that `struct frac` is replaced with the alias `frac` throughout. How does this change the readability of your code?

3. Extend the `frac` struct with two more functions:

   - `sub_frac` — subtracts one fraction from another
   - `div_frac` — divides one fraction by another by multiplying by the reciprocal

   Verify that `div_frac(add_frac(f1, f2), f2)` returns the same value as `f1` for a few test cases.

4. Using `malloc` and the `->` operator as shown in Listing 5, allocate a `frac` on the heap, set its members, pass it to `print_frac`, and free it. Run the program under `valgrind` to confirm there are no memory leaks.
5. Write a program that declares an array of five `frac` structs, initializes each using an initialization list, and prints them all using `print_frac`. Then write a function `sum_all` that takes an array of `frac` structs and its length and returns their sum as a single `frac`.
6. Compile Listing 10 on your machine and verify the `sizeof` outputs match those shown in Listing 11. Then use `offsetof` from `<stddef.h>` to print the byte offset of each member in both `struct padded` and `struct packed`. Do the offsets match the comments in the listing?
7. Design a new struct `struct packed_frac` that reorders the members of `struct frac` to minimize padding. Is it possible to reduce the size below what the default layout produces? Use `sizeof` and `offsetof` to verify your answer.
8. The chapter shows in Listing 6 that using a plain array instead of a struct loses type information. Reproduce this example and compile it. Then alter it to use the `frac` struct as in Listing 8 and compare the compiler output. In your own words, explain what the struct provides that the array does not.

9. Consider the following attempted self-referencing struct:

```
struct frac {
    int n;
    int d;
    struct frac next;
};
```

   Try to compile it and record the error. Explain why C cannot allow a struct to contain a member of its own type directly. Then fix it by replacing `struct frac next` with `struct frac *next` and explain why the pointer version works where the direct member does not.

10. Implement a simple family tree using a self-referencing struct:

```
struct person {
    char name[50];
    int  age;
    struct person *parent;
};
```

Create at least three generations, link them together using the `parent` pointer, and write a function `print_ancestors` that walks up the chain printing each person's name and age until it reaches a `NULL` parent pointer.

Vectors

Introduction

Computer programming, in the form we're familiar with it, has been around for over eighty years. All of that time has come with advancements not just in technology, but also in the way that we think about and write compute programs. The main aspect we'll be concerned with throughout the rest of this course is in *data structures*. A data structure is a way to organize and orient data within a computer program. There have been many data structures theorized, formalized, and utilized throughout the long history of computing, so we can only focus on a few in a reasonable amount of time. We'll begin with a simple, intuitive data structure which acts as a useful supplement to a basic array.

Note: Many of the remaining exercises will be having you implement the data structures in C yourself. This means the only code I will provide for the remainder of this course is *pseudocode*.

Where Do Arrays Fail?

Arrays are incredibly useful for storing and retrieving a fixed number of elements. For example, if I'm writing software that catalogues all of the titles in a library, and I know the number of books contained within that library, it's trivial to represent every item as an element in an array, albeit a bit tedious. But what happens the moment a new book comes out and the library adds it to its physical collection? Already the old program is out of sync with reality and must be reworked to allow for the additional item. What we need is a data structure that can grow on demand without the programmer having to manage the resizing manually.

What is a Vector?

A **vector** is the exact response to the above problem. The idea is simple: combine the simplicity and contiguous behavior of arrays with automatic resizing logic. Vectors are often also called **resize-able arrays**.

Vector API

Every data structure we will be discussing has historically been defined and implemented in one form or another. Regardless of particular historical machines/libraries, there are always key functions and behaviours the remain

constant across all implementations. We call the relevant functions and their behaviors tied to a data structure its application programmer interface (API). We list the API for a vector below:

- `init`
  - Initializes a vector
- `add`
  - Appends an element to the vector
- `get`
  - Gets an element at a given index
- `set`
  - Sets an element at a given index
- `size`
  - Returns the number of elements in the vector
- `capacity`
  - Returns the underlying size of the vector
- `is_empty`
  - Indicates whether the vector is empty or not
- `free`
  - Frees the vector

Implementing a Vector

There is no one, true way of implementing any data structure—there will always be differences depending on goals, target language quirks, and programming styles. When implementing a vector it is always advisable to use the target language's array data type as the underlying array that be resized. The essential logic is quite simple:

```
1:  add(elem):
2:    if size = capacity:
3:      resize()
4:    array[size] ← elem
5:    size ← size + 1
6:    return
```

When adding elements to the array you must first check to see if there's any space. If so, the element can be added; if not, the array must be grown. The way the array is resized is down to implementation. What matters is that the original elements must be copied over into the new array.

I leave it as an exercise to the reader to further flesh out the Vector implementation in C.

Note: Remember that if arrays are going to be passed around multiple functions you should make sure to allocate them on the heap!

Vector: Pros & Cons

The choice to use one data structure over another is always an *engineering* decision. It is up to the programmer to weight the pros and cons of all of the possible data structures that could be used for a given task. Here, we enumerate the pros and cons for vectors:

Pros:

+ Random access

Cons:

- Must copy everything over on size up
- Wasted space if anomalous growth occurs

Common Vector Implementations

Vectors are very common throughout the standard libraries of programming languages, although they are often given different names. Java calls its vector an **ArrayList**, Python a **list**, Rust a **Vec**, Go a **slice**, and so on. C++ sticks to **vector** which is why I used that terminology here. Each of these language implement their vectors differently, although the core functionality remains. The major difference will come in the form of the default array size and the algorithm which determines the new resized array length.

Exercises

1. Design a `struct` to represent a vector in C. It should contain at minimum an integer array pointer, a `size` field, and a `capacity` field. What type should the array pointer be and why? What initial values should `size` and `capacity` have after `init`?
2. Implement `init` and `free_vec` for your vector struct. `init` should allocate an initial array on the heap with a starting capacity of your choice and set `size` to 0. Run your `init` and `free_vec` under `valgrind` to confirm there are no memory leaks.
3. Implement `size`, `capacity`, and `is_empty` for your vector. These should be straightforward one-liners. Write a small `main` that calls `init`, then prints the result of all three functions before any elements are added.
4. Implement `get` and `set` for your vector. What should happen if the index provided is out of bounds? Add a bounds check and print an error message if the index is invalid.
5. Implement the `resize` helper function referenced in the pseudocode. It should allocate a new array of larger capacity, copy all existing elements over, free the old array, and update the capacity field. What factor should you grow by and why? Look up how Python and Java grow their dynamic arrays and compare.

6. Implement `add` using the pseudocode given in the chapter as a guide. Test it by adding 20 integers to a vector with an initial capacity of 4 and printing the `size` and `capacity` after each insertion. How many times does `resize` get called?
7. The chapter states that "the way the array is resized is down to implementation." Implement two versions of `resize`: one that doubles the capacity and one that grows by a fixed amount of 4. Add 100 elements to each and count the total number of copy operations performed. Which strategy results in fewer copies overall?
8. The chapter lists random access as a pro of vectors. Write a program that uses `get` to access the middle element of a vector containing 1000 integers. Then think about how you would find the middle element of a linked list — what would that require?
9. The chapter notes that wasted space is a con of vectors. Write a program that adds 5 elements to a vector with a doubling resize strategy and then prints `size` and `capacity`. How much space is wasted? Under what circumstances would this wasted space become a practical problem?
10. The chapter distinguishes between `size` and `capacity` as two separate concepts. Write a program that creates a vector with an initial capacity of 4 and adds elements one at a time, printing both `size` and `capacity` after each insertion until the vector has been resized at least three times. In your own words, explain the difference between what `size` and `capacity` represent and why a vector needs to track both.

Stacks & Queues

Introduction

Imagine a scenario where you've put off doing the dishes for far too long and you need a plate to eat your daily bread. You think, "Oh God, I can't keep living like this." Before it's time to begin running the water you pull your sleeves up to your forearms, wearing out the last bit of plasticity remaining in your elastic cuffs. The joyful mundanity of scrubbing a week's worth of solidifying crust feels like a secular baptism. Every dish you polish and dry is sat atop one another such that the most recently cleaned is at the peak. Once you're done, you take the plate resting at the summit, the one you just finished drying off. You have just implemented and utilized a **stack**.

Stacks

Stacks are one of the simplest and most foundational data structures in computing. In fact, I've already used the word "stack" many times already throughout this course when referring to dynamic memory allocated by the compiler. A stack guarantees that the order in which elements are retrieved, or *popped* off, are in the reverse order in which they were placed, or *pushed*, on the stack.

Stack API

Like vectors, stacks can be implemented a variety of ways and may have different features depending on the library being used. The basic functionality is as follows:

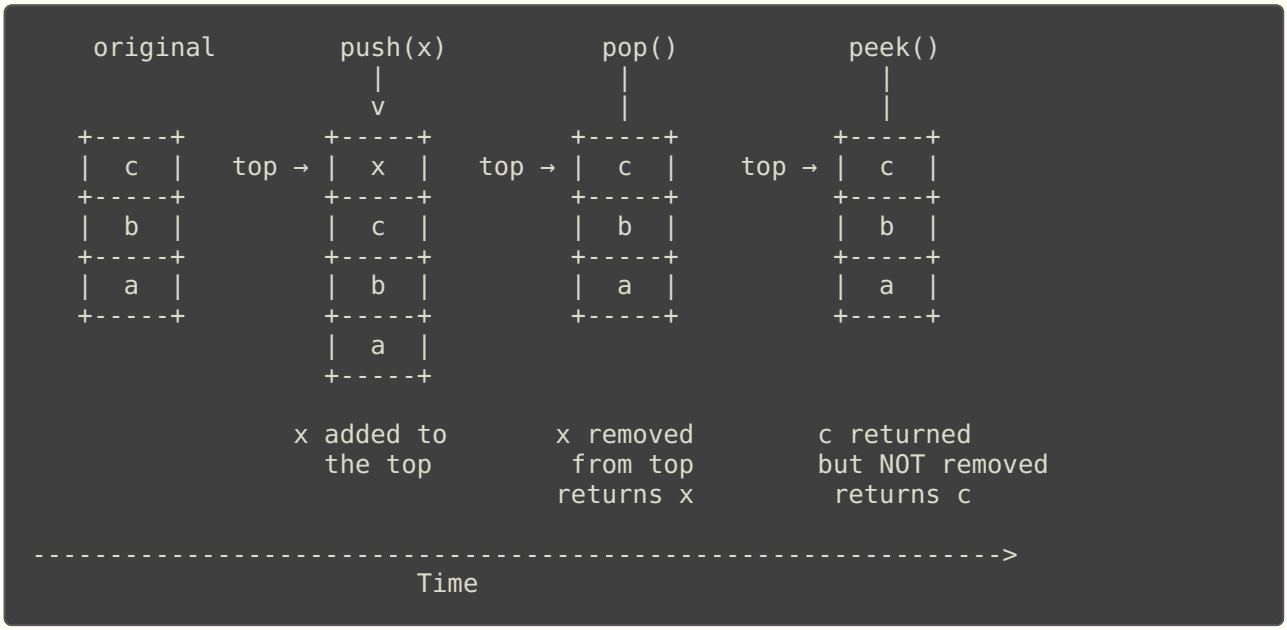
- `init`
  - Initializes a stack
- `push`
  - Pushes an element on the top of the stack
- `peek`
  - Retrieves the top element on the stack but does not remove it
- `pop`
  - Removes and returns the top element on the stack
- `is_empty`
  - Indicates whether the stack is empty or not
- `free`
  - Frees the stack

Implementing a Stack

All a stack must do is maintain last-in-first-out (LIFO) order. This gives a wide variety of possible backend implementations for the stack in question. Stacks may have a fixed or a dynamic capacity—again, this is decided by need; traditionally, stacks in standard libraries will have dynamic capacity logic so as to best apply to all general cases. It is left as an exercise for the reader to implement the stack logic.

Using a Stack

The diagram below shows the behavior of the main three stack functions (push, pop, and peek):



We begin with a stack that has the elements `(c, b, a)` where `c` is at the top of the stack (the elements were pushed in alphabetical order). A call to `push(x)` places `x` at the top of the stack. A call to `pop()` removes the element at the top of the stack, now `x`, and returns it. A call to `peek()` takes a look at the new element at the top of the stack, now `c` again, and returns that element *without removing it from the stack*.

Case Study: Function Calls, Parsing and Reverse Polish Notation

When I first started teaching data structures years ago, I found that stacks came across as far too abstract for students to connect with as something genuinely useful. Why would I ever need anything in the *reverse* order in which I added them? It's not immediately intuitive. First, think back to how function calls work in C:

Listing 1: Function call order

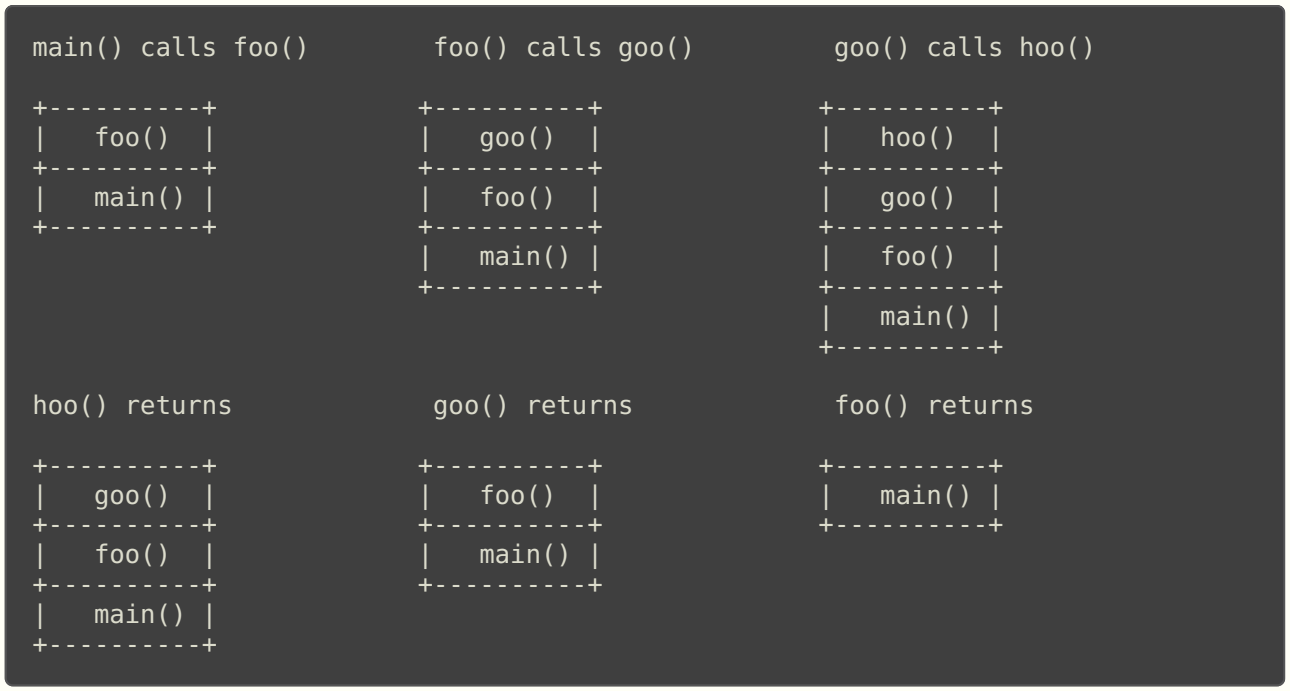
```
 1:  #include <stdio.h>
 2:  int hoo(){
 3:    printf("entering hoo()\n");
 4:    printf("leaving hoo()\n");
 5:  }
 6:
 7:  int goo(){
 8:    printf("entering goo()\n");
 9:    hoo();
10:    printf("leaving goo()\n");
11:  }
12:
13:  int foo(){
14:    printf("entering foo()\n");
15:    goo();
16:    printf("leaving foo()\n");
17:  }
18:
19:  int main(){
20:    printf("entering main()\n");
21:    foo();
22:    printf("leaving main()\n");
23:  }
```

You should know the order in which Listing 1 prints, but I'll print out the results anyway:

Listing 2: Function call order results

```
1:  josephraskind@stargazer:/tmp/stack$ gcc print_stack.c -o print_stack; ./
print_stack
2:  entering main()
3:  entering foo()
4:  entering goo()
5:  entering hoo()
6:  leaving hoo()
7:  leaving goo()
8:  leaving foo()
9:  leaving main()
```

We get a nice symmatry in the "entering" and "leaving" print outs. Why? Well, it's because the compiler maintains a *function call* ***stack***. We can see a visualization of this below:



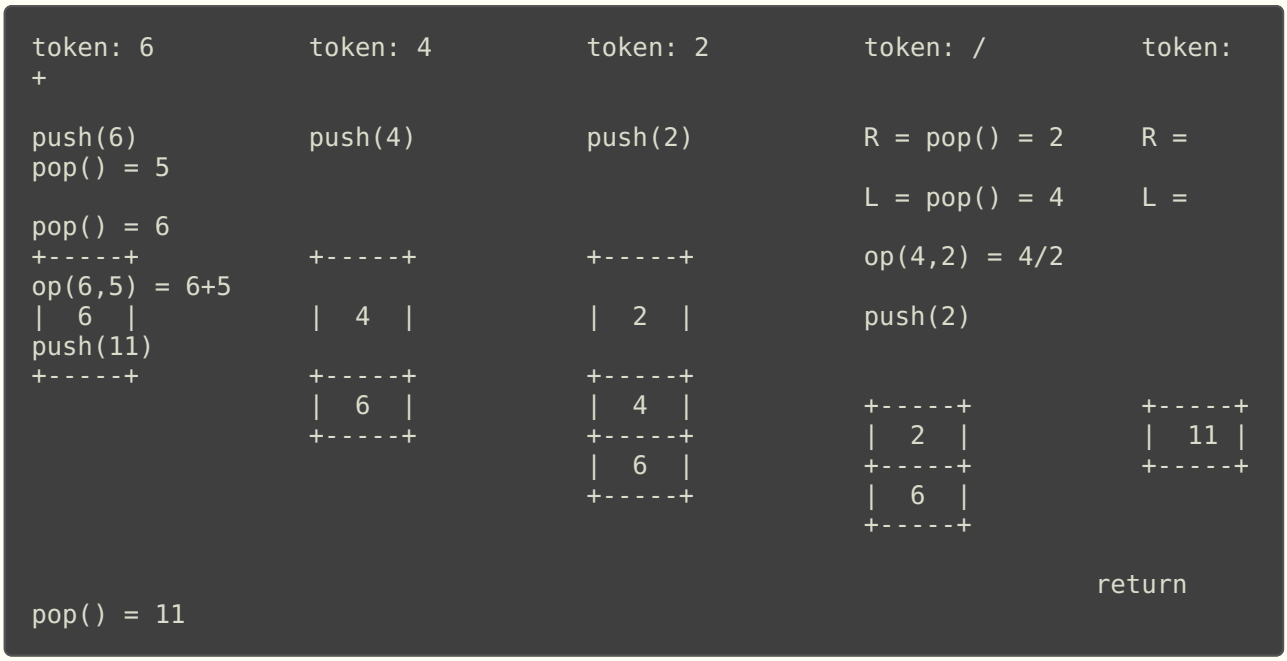
Each subsequent call *pushes* the previous function's context onto the stack. Every return *pops* the recent call off of the stack. The same is done with local variables which is why they go out of scope within the lifetime memory semantics.

Stacks can also be used for parsing tasks. If you remember back to when we discussed operators, I mentioned there were three ways we can represent the syntax for binary operators: prefix, infix, and postfix. We'll focus on postfix notation, or *Reverse Polish Notation* (RPN), as it is perfectly suited for being handled by a stack.

Take the string `6 + 4 / 2`. Without being aware of operator precedence, which a computer is not natively aware of, it is impossible to discern whether or not the user wanted to add 6 and 4 and then divide by 2, or to divide 4 by 2 then add 6. In RPN, there is no ambiguity. If we take classic precedence as a given then we can rewrite that string `6 4 2 / +`. The algorithm for RPN is simple:

```
 1:  evaluate(tokens):
 2:    for token in tokens:
 3:      if token is VALUE:
 4:        stack.push(token)
 5:      if token is OPERATOR:
 6:        op ← token
 7:        operand_R ← stack.pop()
 8:        operand_L ← stack.pop()
 9:        push(op(operand_L, operand_R))
10:    return stack.pop()
```

We can see this evaluation play out visually:

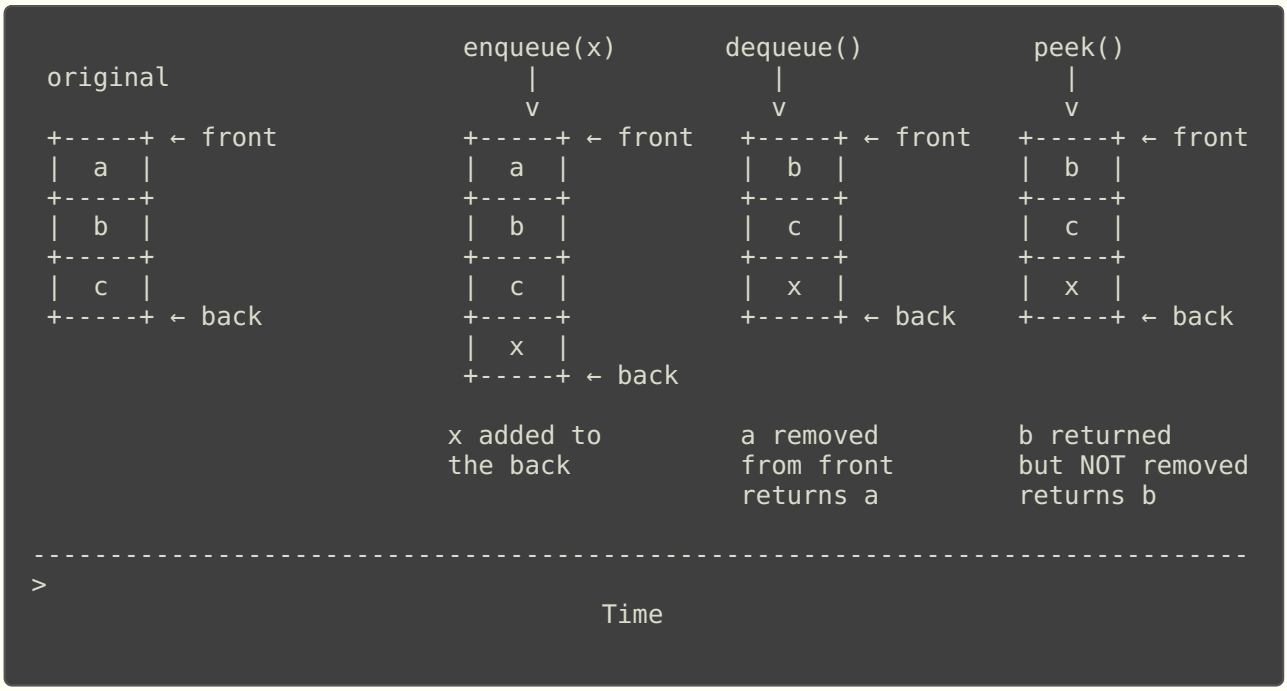


I leave it as an exercise to implement RPN in C.

Note: RPN was used in one of the first computers, Konrad Zuse's Z3!

Queues

Queues are nearly identical to stacks with one key difference: instead of maintaining LIFO order they maintain FIFO order, or *first*-in-*first*-out. The use-case for a queue are a bit more intuitive. For example, if there are many users who wish to run their programs on a remote server each request could be entered into a queue such that the requests are processed in the order in which they were received.



It does not take much turn a stack into a queue. I leave that as an exercise for the reader.

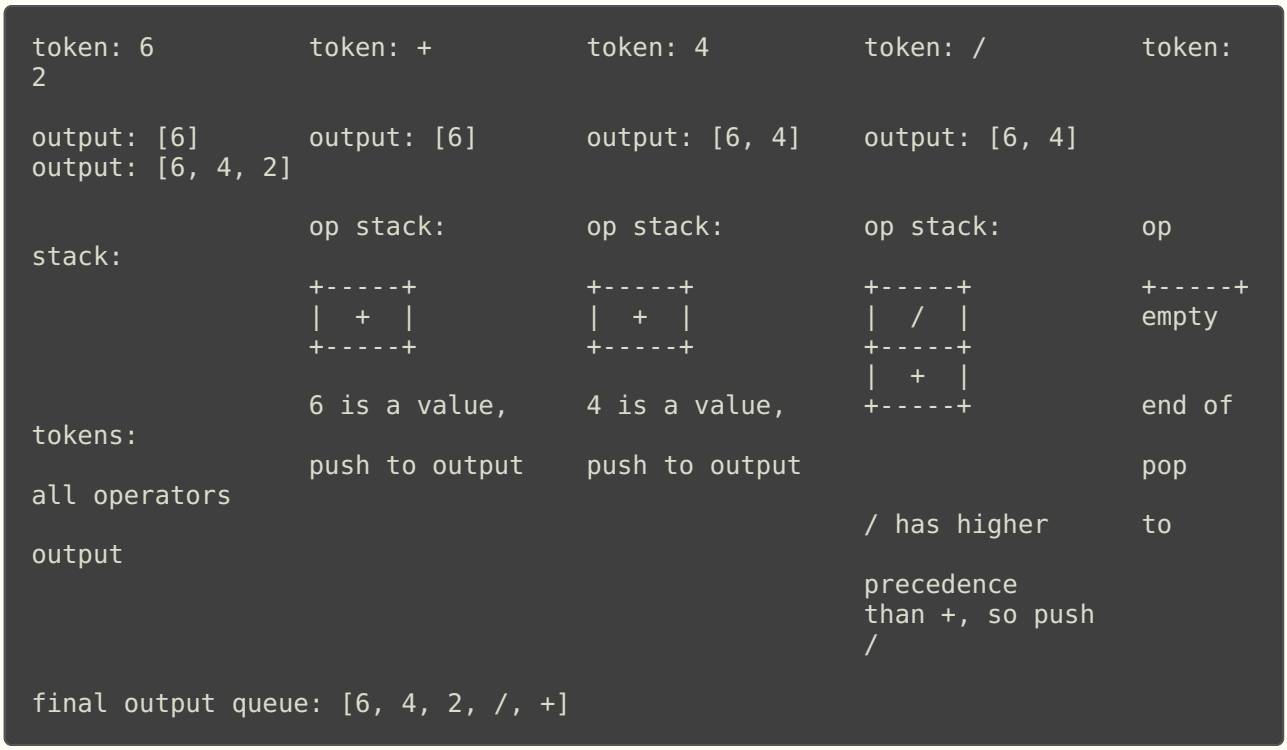
Case Study: The Shunting Yard Algorithm

Following up on the case study for parsing postfix operators with a stack, we can use a stack in tandem with a queue to convert infix notation to postfix notation. The following algorithm was developed by Edsger Dijkstra in the early 1960s[1]:

Listing 3: Shunting Yard Algorithm

```
 1:   while there are tokens to be read:
 2:       read a token
 3:       if the token is:
 4:       - a number:
 5:           put it into the output queue
 6:       - a function:
 7:           push it onto the operator stack
 8:       - an operator o1:
 9:           while (
10:               there is an operator o2 at the top of the operator stack
which is not a left parenthesis,
11:               and (o2 has greater precedence than o1 or (o1 and o2 have
the same precedence and o1 is left-associative))
12:           ):
13:               pop o2 from the operator stack into the output queue
14:           push o1 onto the operator stack
15:       - a ",":
16:           while the operator at the top of the operator stack is not a
left parenthesis:
17:                pop the operator from the operator stack into the output
queue
18:       - a left parenthesis (i.e. "("):
19:           push it onto the operator stack
20:       - a right parenthesis (i.e. ")"):
21:           while the operator at the top of the operator stack is not a
left parenthesis:
22:               {assert the operator stack is not empty}
23:               /* If the stack runs out without finding a left parenthesis,
then there are mismatched parentheses. */
24:               pop the operator from the operator stack into the output
queue
25:           {assert there is a left parenthesis at the top of the operator
stack}
26:           pop the left parenthesis from the operator stack and discard it
27:           if there is a function token at the top of the operator stack,
then:
28:               pop the function from the operator stack into the output
queue
29:   /* After the while loop, pop the remaining items from the operator stack
into the output queue. */
30:   while there are tokens on the operator stack:
31:       /* If the operator token on the top of the stack is a parenthesis,
then there are mismatched parentheses. */
32:       {assert the operator on top of the stack is not a (left) parenthesis}
33:       pop the operator from the operator stack onto the output queue
```

Below represents the transformation of the above mentioned `6 + 4 / 2` into RPN:



I leave it as an exercise for the reader to implement this in C.

Stack/Queue: Pros & Cons

Pros:

+ Keeps well-defined order

Cons:

- Elements cannot be easily retrieved without breaking order

Exercises

1. Implement a stack in C using the vector you implemented in the previous chapter as the underlying data structure. Your stack should support all of the operations listed in the Stack API: `init`, `push`, `peek`, `pop`, `is_empty`, and `free`. Since your vector already handles dynamic resizing, your stack implementation should require very little additional logic.
2. Using your stack implementation, write a program that replicates the function call order behavior shown in Listing 1. Each time a function is "entered" push its name onto the stack and print the stack contents. Each time a function "leaves" pop it off and print the stack contents. Verify the output matches the diagram in the chapter.
3. Implement the RPN evaluator from the pseudocode in the chapter using your stack. Test it with the following expressions and verify each result:
    - `6 4 2 / +` → 8

    - `3 4 * 2 +` → 14
    - `5 1 2 + 4 * + 3 -` → 14
4. Implement a queue in C. Consider what changes need to be made relative to your stack implementation to enforce FIFO rather than LIFO order. Your queue should support `init`, `enqueue`, `dequeue`, `peek`, `is_empty`, and `free`.
5. The chapter states that it "does not take much to turn a stack into a queue." Explain in your own words what the key structural difference is between the two. What changes in terms of where elements are added and removed?
6. Using your queue implementation, simulate a print queue. Write a program that enqueues five print jobs (represented as strings), then dequeues and prints them one at a time. Verify that they are processed in the order they were submitted.
7. Implement the Shunting Yard Algorithm from Listing 3 in C using your stack and queue implementations. Test it on `6 + 4 / 2` and verify the output queue matches `[6, 4, 2, /, +]`. Then pipe the result into your RPN evaluator from exercise 3 and verify the final result is 8.
8. Run your stack and queue implementations under `valgrind` to check for memory leaks. If any are found, identify which function is responsible and fix it. What is the most common source of memory leaks in data structure implementations?
9. The chapter lists "elements cannot be easily retrieved without breaking order" as a con of stacks and queues. Write a program that demonstrates this limitation by attempting to find a specific element in a stack without permanently destroying its contents. What strategy can you use to restore the stack after searching it?
10. The chapter presents two case studies showing stacks used for real computational tasks: function call management and RPN evaluation. Write a third use case of your own by implementing a program that uses your stack to check whether a string of parentheses is balanced. For example `(((())))` and `(()())` are balanced while `(()` and `)(` are not. The algorithm is simple: push every opening parenthesis onto the stack and pop when a closing parenthesis is encountered — if the stack is empty when you try to pop, or non-empty when the string ends, the parentheses are unbalanced.
11. Implement a circular buffer in C with a fixed capacity of your choosing. Your implementation should support `enqueue`, `dequeue`, `peek`, `is_empty`, and `is_full` operations. Unlike your vector and stack implementations, the circular buffer should **not** resize — when it is full, attempting to enqueue should fail gracefully. Use two integer indices, a read pointer and a write pointer, to track the front and back of the buffer, and use the modulo operator to wrap them around the array. Test your implementation by filling the buffer to capacity, dequeuing a few elements, and then enqueuing new ones into the freed slots to verify the wrap-around behavior works correctly.

Footnotes:

1 This psuedocode was taken from the Wikipedia article on the topic.

Linked Lists

Introduction

Every data structure we've encountered so far has shared one serious limitation: they have all needed to be defined with contiguous stretches of memory. This can be a major issue when dealing with variable sized data—remember back to one of the major cons of vectors. That then raises the question: "Can we create a data structure which can grow arbitrarily large, but does not require large swathes of connected strips of memory?" The answer: *linked lists*.

What is a Linked List?

Linked lists were first formalized in a work published by Allen Newell, Cliff Shaw, and Herbert A. Simon in 1957 titled *Programming the Logic Theory Machine*. Newell, Shaw, and Simon had previously worked on establishing a heuristic search-based algorithm which would mimic human-like proof discovery with symbolic symbol sets. The biggest barrier they encountered when trying to practically implement the Logic Theory Machine was that, by definition, the search would take an amount of time and space not known *a priori*—human beings never know how long it might take to discover proofs, so the algorithmic representation was equally haunted by that reality. This meant that storing all of the different possible branches and twists the search algorithm went on was beyond the capabilities of known data structures and programming language constructs. Their solution was to craft a new programming language, IPL, which was focused around the maintenance of a revolutionary data structure they called a "list".

The IPL list had to satisfy the following constraints:

1. "There should be no restriction on the number of different lists of items of information to be stored. This number should not have to be decided in advance; that is, it should be possible to create new lists at will during the course of computation."
2. "There should be no restriction on the nature of the items in a list. These might range from a single symbol or number to an arbitrary list. Thus, it should be possible to make lists, lists of lists, lists of lists of lists, etc."
3. "It should be possible to add, delete, insert, and rearrange items of information in a list at any time and in any way. Thus, for example, one should be able to add to the front of a list as well as to the end."
4. "It should be possible for the same item to appear on any number of lists simultaneously."

And thus, what we call a "linked list" was born!

Anatomy of a Linked List

I've found that, for many students, linked lists mark a major point where data structures enter a level of what can feel like incomprehensible abstraction. The most important thing to keep in mind when dealing with linked lists is that they are really only constituted of two components: **data** and a *pointer* to the **next** element in the list. The "elements" that make up linked lists are called **nodes**. A node looks like the following:

```
+--------+--------+
|  data  |  next  |
+--------+--------+
```

The node stores two values: the key **data** associated with that node and the pointer that will be used to connect it to the **next** node in the chain.

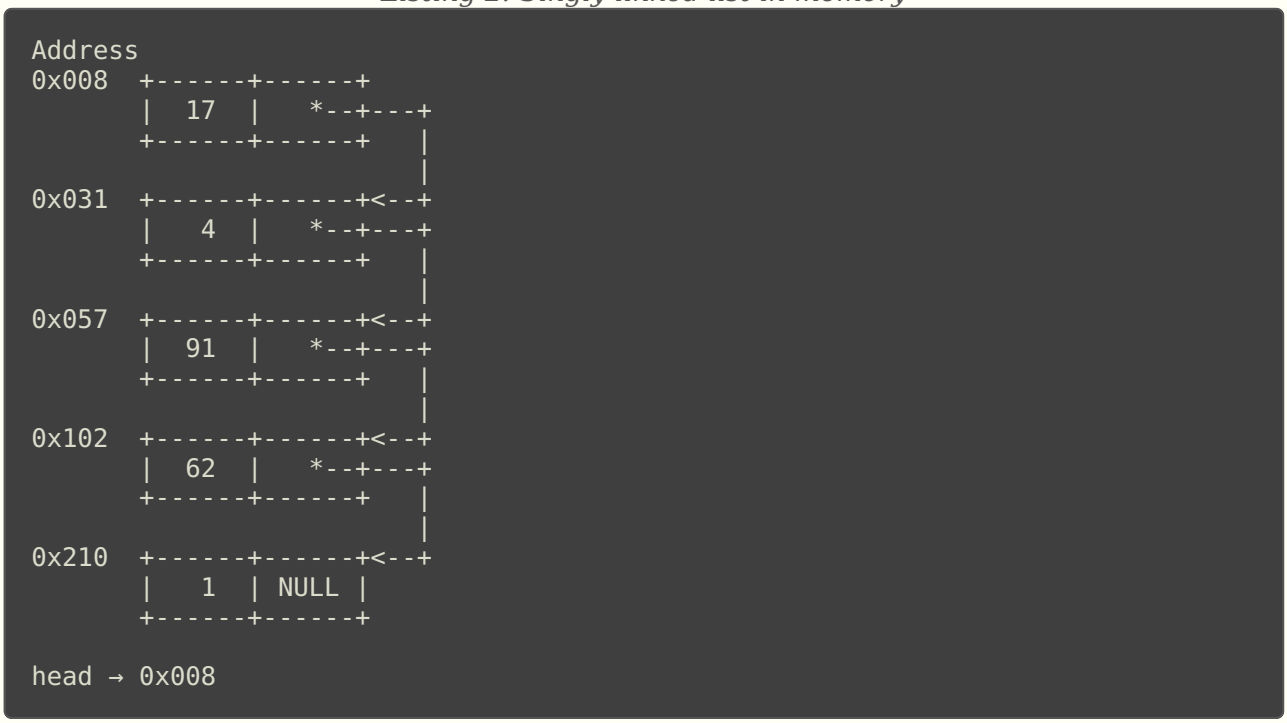
Now that we know what a node looks like, we can visualize what a full linked list looks like:

Listing 1: Singly linked list diagram

```
+------+------+    +------+------+    +------+------+    +------+------+
+------+------+
|  17  |   *--+--->|   4  |   *--+--->|  91  |   *--+--->|  62  |   *--+---
>|   1  | NULL |
+------+------+    +------+------+    +------+------+    +------+------+
+------+------+
 head
```

Here we see a linked list which is storing integer values. This list is comprised of five nodes all of which have up to a single connection to another node. We call this sort of linked list a *singly*-linked list for that very reason. The advantage of the linked list is that, since it is entirely pointer, or *reference*, based none of the nodes must be connected contiguously in memory. For example, Listing 1 could be represented in memory like this:

Listing 2: Singly linked list in memory



Again, in a linked list nodes can be connected between disparate locations in the address space.

One thing that you will have noticed is the use of the phrase **head**. The head pointer refers to the start of the linked list. Without a head pointer there would be no way to "enter" into the linked list.

Linked List API

We can define a simple linked list API as follows:

- `init`
  - Initializes a linked list
- `append`
  - Adds an element to the end of the linked list
- `insert`
  - Inserts an element at the desired index of the linked list
- `search`
  - Retrieves the desired element from the linked list
- `get`
  - Retrieves the element at the desired index of the linked list
- `remove`
  - Removes the element at the desired index of the linked list
- `is_empty`
  - Indicates whether the linked list is empty or not
- `free`
  - Frees the linked list

Adding Elements to a Linked List

Linked list implementations often allow for two ways of adding elements: appending and inserting.

Append

Appending elements to a linked list is very simple. The following pseudocode describes the process:

Listing 3: append pseudocode

```
1:  append(elem):
2:    current_node ← head
3:    while current_node.next not = null:
4:      current_node ← current_node.next
5:    current_node.next ← elem
```

We begin at the head, or the entrace to the linked list, work our way to the last node, and add the desired element to the chain. I leave it as an exercise to the reader to implement the function in C.

Note: The pseudocode in the append function is missing an edge case check. Can you figure out what it is?

Insert

Appending elements to the list is a special case of `insert` where the insertion index is the beyond the end of the list. Here I will define the pseudocode for insertion in the middle of a linked list:

Listing 4: insert pseudocode — here we assume the head is at index 0

```
 1:  insert(index, elem):
 2:    current_node ← head
 3:    prev_node ← null
 4:    c_i ← 0
 5:    while c_i not = index:
 6:      prev_node ← current_node
 7:      current_node ← current_node.next
 8:      c_i ← c_i + 1
 9:    prev_node.next = elem
10:    elem.next = current_node
```

I leave it as an exercise to the reader to implement this in C.

Note: Like the append function, the insert function is missing two distinct case checks. Can you figure out what they are?

Searching a Linked List

Searching a linked list follows the exact same algorithm as appending/inserting with the exception of checking equivalency at each data item.

Removing Elements from a Linked List

Removing elements from a linked list follow a similar algorithm to insertion. The only difference being is you want to break the next node chain and free up the node that is marked for deletion.

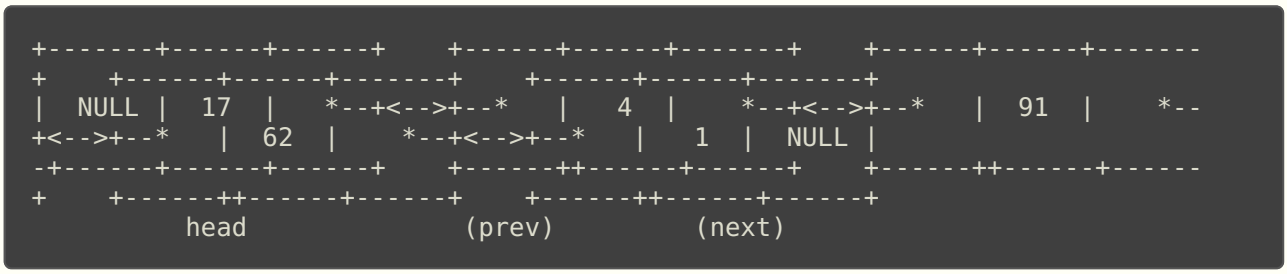
Listing 5: delete pseudocode — here we assume the head is at index 0

```
 1:   delete(index):
 2:     current_node ← head
 3:     prev_node ← null
 4:     c_i ← 0
 5:     while c_i not = index:
 6:       prev_node ← current_node
 7:       current_node ← current_node.next
 8:       c_i ← c_i + 1
 9:     prev_node.next = current_node.next
10:     free(current_node)
```

Note: Again, the delete function is missing two distinct case checks. Can you figure out what they are?

Doubly Linked Lists

The references to linked lists above were all to *singly*-linked lists as each node had only one next pointer. We can increase the complexity of our linked lists by adding a second pointer to each node which points to the previous node in the chain. The diagram below shows a *doubly*-linked list variant of the singly-linked list diagram used above:



The introduction of a second pointer requires slightly more upkeep and overhead from the algorithm perspective. Again, I leave it as an exercise to the reader to implement a doubly-linked list using the singly-linked list as a guide.

Linked List: Pros & Cons

Pros:

+ Can grow arbitrarily large
+ Appending can be made a constant time operation

Cons:

- Can be costly to search through

Exercises

1. Implement a singly linked list node as a `struct` in C. It should contain an integer data field and a `next` pointer. Then implement `init` which initializes an empty linked list with a `head` pointer set to `NULL`.
2. Implement `append` from Listing 3 in C. The chapter notes that the pseudocode is missing an edge case check — identify what it is and handle it in your implementation. Test by appending the values `(17, 4, 91, 62, 1)` and printing each node.
3. The chapter notes that Listing 4 is missing two distinct case checks. Identify both before implementing `insert` in C. Hint: think about what happens when inserting at index 0 and when inserting beyond the end of the list.
4. Implement `delete` from Listing 5 in C. The chapter notes it is also missing two case checks — identify and handle them. Test by deleting the head, the tail, and a middle element from your list.
5. Implement `search` for your linked list. It should traverse the list and return a pointer to the first node whose data matches the search value, or `NULL` if not found. What is the worst case number of nodes that must be visited to find an element?
6. The chapter lists "appending can be made a constant time operation" as a pro of linked lists. Currently, `append` from Listing 3 traverses the entire list to find the last node, making it O(n). Add a `tail` pointer to your linked list struct that always points to the last node and update `append`, `insert`, and `delete` to maintain it correctly. Verify that `append` no longer needs to traverse the list.
7. The chapter shows in Listing 2 that linked list nodes can be stored at non-contiguous memory addresses. Write a program that allocates five nodes using `malloc` and prints the memory address of each. Are they contiguous? What does this confirm about how linked lists use memory compared to arrays?

8. Implement `free` for your linked list. It should traverse the entire list and free each node individually. Run your full linked list implementation under `valgrind` to confirm there are no memory leaks.
9. Implement a doubly linked list in C using the singly linked list as a guide. Each node should have both a `next` and a `prev` pointer. Update `append`, `insert`, and `delete` to maintain both pointers correctly. What additional steps are required compared to the singly linked list versions?
10. The chapter lists "can be costly to search through" as a con of linked lists. Write a program that creates a linked list of 1000 integers and searches for a value stored at the very end. Count the number of nodes visited during the search and print it. Then repeat the same experiment with a value stored at the front. What does the difference tell you about the relationship between where an element is stored and how long it takes to find it? How does this compare to retrieving an element from a vector using `get` at any index?

Binary Trees

Introduction

If you went to kindergarten/grade school in the United States there's a good chance you had to complete a school project where you sketched out a family tree. The top of the tree would consist of distant, or not *so* distant, ancestors and it would eventually pass through your parents and end up at you. Each person's name in the tree would have two branches going up, representing parents[1], but could have many branches going down, representing children. If we prune the nodes that are not strictly ancestral by only keeping parents and no siblings, what you drew was a **binary tree** and it is one of the most used, fundamental data structures in computing.

What is a Binary Tree?

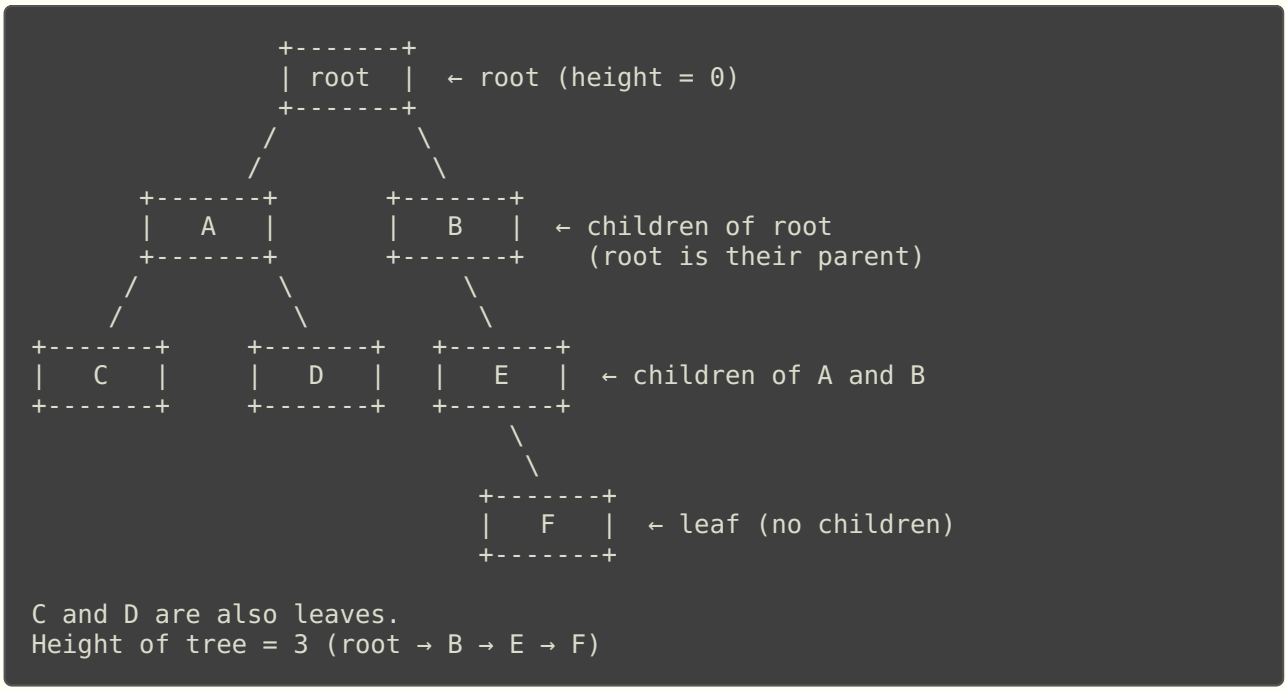
A binary tree is a data structure which, like linked lists, must satisfy a handful of logical constraints:

1. All binary trees must have a starting node called the root node
2. All nodes in the tree can have at most two children.
3. All nodes may have at most one parent.

Any tree-like structure which breaks these rules are by-definition not *binary* trees. For example, the Linux filesystem can easily be represented as a *tree*, but it is not a *binary* tree because a single directory (node) may have more than two files (children).

Binary Tree Terminology

The **root** node is the starting node in a tree ("root" for the roots, or anchoring point, of a tree). Nodes can connect to other nodes. Nodes that are connected to a given node that go up the tree are called **parents**. Nodes that are connected to a given node that go down the tree are called **children**. Nodes that have no children are called **leaves**. The **height** of a tree indicates how many jumps from the root need to be made to reach the farthest leaf. The following diagram visualizes this terminology:



We call a binary tree **full** if every node has either two children or none. We call a binary tree **complete** if every level is filled with nodes, except the last level where nodes do not have to be filled in completely, but must be packed to the left. Finally, we call a binary tree **perfect** if all interior nodes have two children and all leaves have the same depth or same level. We can visualize the different types of trees below:

Binary Tree Traversal

There are three standard ways of traversing a Binary Tree: **preorder**, **inorder**, and **postorder**. I'll outline the pseudocode for **preorder**:

Listing 1: preorder binary tree traversal

```
1:   preorder(node):
2:     print(node.data)     # Perform computation
3:     preorder(node.left)
4:     preorder(node.right)
```

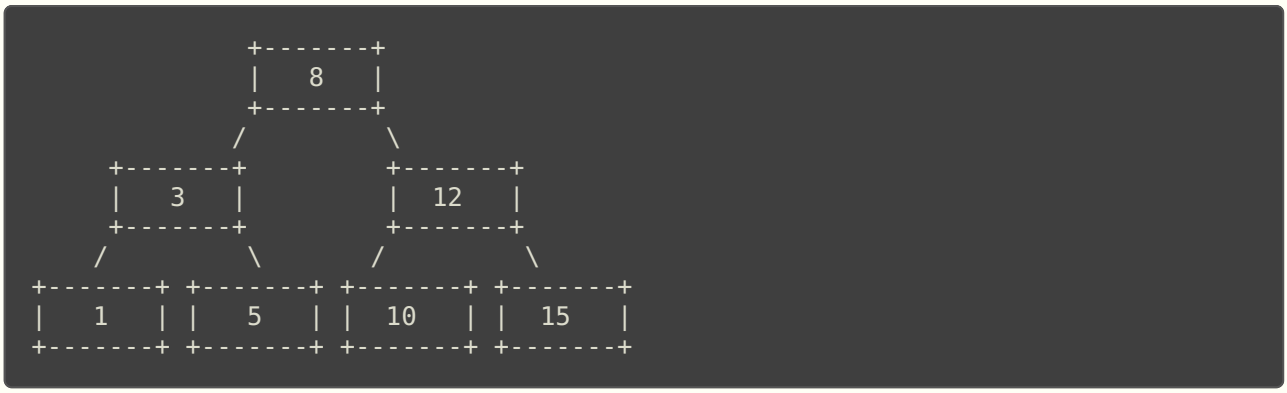
Here, we have a recursive function which looks at a node in a binary tree, performs a computation, and then moves to the left node, recursively, and then the right node. We can see it visually below:

```
                    +-------+
                    |   1   |  ← visit first (root)
                    +-------+
                   /         \
            +-------+       +-------+
            |   2   |       |   5   |  ← visit 2 before 5
            +-------+       +-------+
           /         \     /         \
        +-------+ +-------+ +-------+ +-------+
        |   3   | |   4   | |   6   | |   7   |
        +-------+ +-------+ +-------+ +-------+

Order visited: 1 → 2 → 3 → 4 → 5 → 6 → 7

Rule: visit the current node BEFORE visiting children
      always go left before right

Steps:
  visit 1 (root)
  go left  → visit 2
  go left  → visit 3 (leaf, go back up)
  go right → visit 4 (leaf, go back up)
  go right → visit 5
  go left  → visit 6 (leaf, go back up)
  go right → visit 7 (leaf)
```

I'll leave it as an exerise to the reader to try out inorder traversal (where the computation is between moving left and right) and postorder traversal (where the computation is after moving left and right).

Binary Search Tree

Binary trees in and of themselves are often not useful in the real world. They become more interesting when further constraints are added to them. One such binary tree variant is the **binary search tree**. Binary search trees are binary trees with one additional wrinkle: they maintain a strict order of nodes. For any given node, all values in its left subtree must be less than that node's value, and all values in its right subtree must be greater. Here is an example of a binary search tree:



It can be seen that for every node, all of the values to the right of it are greater than its value, and all of the values to the left are smaller.

Binary Search Algorithm

If you've ever tried looking through the index of a book to find a specific word, you've likely already employed the basic logic of *binary search*. Let's say you're in a C textbook and you're trying to find the word *function*. First, you open up the index to the A's and grab a hunk of pages and flip right to left. Next, you find yourself in the T's, which you know comes after F, so you grab a hunk of pages and flip left to right. Now, you're in the D's which you know comes before F, so you grab a hunk of pages and flip right to left again. Finally, we've made it to the F's and function should be smack dab in the middle of the page.

This more or less describes the algorithm for performing a binary search on a binary search tree. The following pseudocode implements it more formally:

```
1:  search(node, elem):
2:    if node.data = elem:
3:      return true
4:    else:
5:      if elem > node.data:
6:        return search(node.right, elem)
7:      else:
8:        return search(node.left, elem)
```

We can visualize the binary search below:


50
25
75
12
37
62
87
6
18
81
93
Searching for 81:
step 1: visit 50 candidates: whole tree (15 nodes)
81 > 50 → go right, left subtree eliminated
step 2: visit 75 candidates: 7 nodes
81 > 75 → go right, left subtree (62) eliminated
step 3: visit 87 candidates: 3 nodes
81 < 87 → go left, right subtree (93) eliminated

```
step 4: visit 81 ✓        candidates: 1 node
        found!
```

What you'll notice is that every single time a decision is made to move left or right half of the binary tree is ignored further down the search. The order of the binary search is what allows for this to occur. Intuitively, the smaller the portion of the tree we have to search, the faster the search will be. It turns out that binary search in a "well-balanced" binary search tree should take only about $\log_2(n)$ steps where n is the number of nodes in the tree. If we take n to be 15, corresponding to the number of nodes in the above tree, we'll find that $\log_2(15) \approx 4$ which is equivalent to the number of steps taken to get the search results!

Not all binary search trees are guaranteed to be "well-balanced", i.e. an even distribution of nodes across all levels.

Binary Search Tree API

We can define a BST API as follows:

- `init`
  - Initializes a BST
- `insert`
  - Inserts an element into the BST
- `search`
  - Searches for an element in the BST
- `remove`
  - Removes the desired element in the BST
- `is_empty`
  - Indicates whether the BST is empty or not
- `free`
  - Frees the BST

Inserting Elements into a Binary Search Tree

Binary search trees only work if their constraints are maintained. We can guarantee that elements are slotted into their proper place by utilizing the same traversal algorithm as the binary search (we'll assume here that elements must be unique):

```
1:  insert(node, elem):
2:    if node.data = elem:
3:      ERROR
4:    else:
5:      if elem > node.data:
6:          if node.right = null:
7:            node.right ← elem
8:          else:
9:            insert(node.right, elem)
10:     else:
11:       if node.left = null:
12:         node.left ← elem
13:       else:
14:         insert(node.left, elem)
```

We can visualize this algorithm by adding `70` to the previous BST:

Listing 2: Inserting an element

Deleting Elements from a Binary Search Tree

Removing elements from a BST presents a greater challenge. For example, consider the tree in Listing 2. If we want to delete `81`, then the process is trivial. But what if we want to delete `50`? How could that be accomplished? If we naively replace it with `75` then something must be done about `62` and `87`. This is why BST's will have a well-defined method for finding a successor node. A common solution is to find the smallest node that is greater than the node being removed. For example, if `50` is marked for deletion, then `62` will be chosen:

```
successor(n):
  if n.right not = null:
    return minimum(n.right)
  s ← n.parent
  while s not = null and n = s.right:
    n ← s
    s ← s.parent
  return s
```

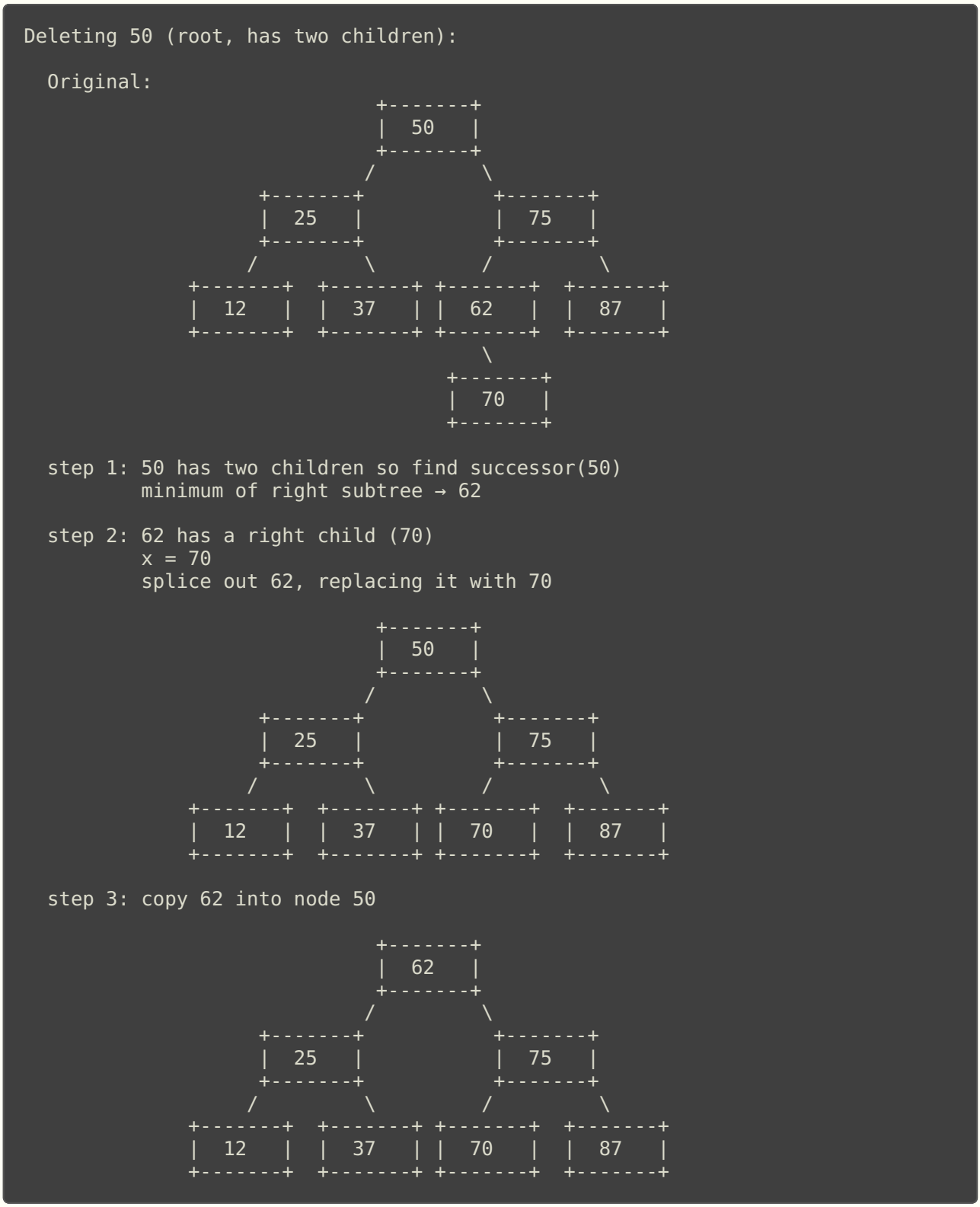
where `minimum` spans the leftmost children. This will always grab either the smallest element greater than `n` from its right subtree, or the lowest ancestor for which `n` lies in the left subtree—both of which are the next largest value in the tree relative to `n`.

We can then define deletion as follows:

Listing 3: deletion for BSTs

```
delete(n):
  if n.left = null or n.right = null:  # Find a node without two children
    y ← n                              # and set it equal to y
  else:
    y ← successor(n)
  if y.left not = null:                # x is set to the left or right
child
    x ← y.left                         # or nill if neither exist
  else:
    x ← y.right
  if x not = null:                     # We then splice out y
    x.parent ← y.parent
  if y.parent = null:
    tree.root ← x
  else if y = y.parent.left:
    y.parent.left ← x
  else:
    y.parent.right ← x
  if y not = n:                        # If the succesor of n was
spliced out
    n.data = y.data                    # swap the data and return y for
removal
  return y

```

We can see the algorithm in action with the above BST:



We can see that after the deletion the assumptions about the binary search tree are maintained.

Binary Seach Tree: Pros & Cons

Pros:

+ Fastest possible search if balanced
+ Dynamic size
+ Naturally recursive structure

Cons:

- Loses search speed if unbalanced
- Memory overhead (storing pointers *and* data)
- No constant time operations

Exercises

1. Implement a BST node as a `struct` in C. It should contain an integer data field, a `left` pointer, a `right` pointer, and a `parent` pointer. Then implement `init` which initializes an empty BST with a `root` pointer set to `NULL`.
2. Implement `insert` from the pseudocode in the chapter in C. Test it by inserting the values `(50, 25, 75, 12, 37, 62, 87)` in that order and verifying the resulting tree matches the diagram shown in the chapter.
3. Implement `search` from the pseudocode in the chapter in C. Test it on the tree built in exercise 2 by searching for both values that exist and values that do not. What is the maximum number of nodes visited when searching a tree of height 3?
4. The chapter leaves inorder and postorder traversal as an exercise. Write the pseudocode for both and then implement all three traversals (preorder, inorder, postorder) in C. Run inorder traversal on the BST from exercise 2 and observe the output. What do you notice about the order of the values printed?
5. Write a recursive function `height` in C that computes the height of a binary tree. Recall from the chapter that height is defined as the number of jumps from the root needed to reach the farthest leaf. Test it on the BST from exercise 2 and verify the result matches what you would expect from inspecting the tree diagram. Then test it on a tree with only a root node — what should the height be in that case?
6. The chapter notes that a BST "loses search speed if unbalanced." Starting from an empty BST, insert the values `(1, 2, 3, 4, 5, 6, 7)` in that order. Draw the resulting tree. What does it look like? Use your `height` function from exercise 5 to measure its height and compare it to a balanced BST containing the same values. What does this tell you about the importance of insertion order?
7. Implement `delete` from Listing 3 in C. Handle all three cases: deleting a leaf, deleting a node with one child, and deleting a node with two children. Test each case on the tree from exercise 2 and verify the BST property is maintained after each deletion.

8. Implement `successor` from the pseudocode in the chapter in C. Test it by finding the successor of every node in the tree from exercise 2 and verify each result by inspection of the tree diagram.
9. Implement `free` for your BST. It should recursively free every node in the tree. Consider which traversal order is most appropriate for freeing nodes safely — preorder, inorder, or postorder — and explain your choice. Run your full BST implementation under `valgrind` to confirm there are no memory leaks.
10. A perfect binary tree of height *h* contains 2^(h+1) - 1 nodes. Verify this formula for heights 1, 2, and 3 by drawing the trees and counting nodes. Then write a function `is_perfect` that returns 1 if a binary tree is perfect and 0 otherwise, using your `height` function from exercise 5 to help.

Footnotes:

1 This, of course, implies a purely geneological/biological family tree. Those with step parents might break that assumption. Similarly, descendents of the Hapsburgs may also quickly run into serious trouble.

Hash Tables

Introduction

Sometimes when programming, it can be convenient to think of collections of data in the form of *pairs*. For example, if you're implementing a benchmarking suite which will run multiple different applications and store their behavioral data it's natural to want some sort of data structure which attaches an application's name with that information. It would be convenient to have a singular source that can be quickly indexed by sending in the target application's name. Python provides exactly that natively with *dictionaries*:

Listing 1: Pythonic dictionary

```
1:  benchmarks = ["sunflow", "jython", "h2"]
2:  behavior_stats = dict()
3:  for b in benchmarks:
4:      behavior_stats[b] = run(b)
5:  # ...
6:  print(f"sunflow took {behavior_stats['sunflow'].wall_time}s to run!")
```

Python's *dictionary*, or *dict*, class is implemented under the hood as a **hash table**.

What is a Hash Table?

A hash table "is a generalization of the simpler notion of an ordinary array"[1]. The idea is to be able to approximate the immediacy of an array while also abstracting the indexing mechanism to fit for any datatype the user defines. This is what the example in Listing 1 is showing: we can index into the hash table, `behavior_stats`, with a string! This is a major upgrade for thinking through certain programming challenges where sets of data need to sync up with a common non-integer identifier. A hash table is essentially just an array with an extra step for determining the desired index —the extra step is called a *hash function*.

Hash Table API

We can define a simple hash table API as follows:

- `init`
  - Initializes a hash table
- `insert`
  - Inserts an element with the desired key into the hash table

- `get`
  - Retrieves the element with the desired key from the hash table
- `delete`
  - Removes the element with the desired key from the hash table
- `free`
  - Frees the hash table

Hash Functions

A hash function is any function which takes some "key" input and maps it to an index.

Collisions

Ideally, a hash function should create a perfect one-to-one mapping from input to index for all possible inputs. If all inputs are known beforehand then this is a trivial task. For example, if we take the hash function `f(x) = x % 5`, the input `[10, 72, 13, 66]`, and apply it to an array of five elements, then we have a perfect one-to-one mapping:

```
input       f(x) = x % 5      index     array
                                       +-------+
   10         10 % 5 = 0    →   [0]   |  10   |
                                       +-------+
   66         66 % 5 = 1    →   [1]   |  66   |
                                       +-------+
   72         72 % 5 = 2    →   [2]   |  72   |
                                       +-------+
   13         13 % 5 = 3    →   [3]   |  13   |
                                       +-------+
                                       |  ___  |
                                       +-------+
```

What happens when a new, unforseen input value, `22`, comes in? Based on the formula, `22 % 5 = 2`, but `[2]` is already reserved for `72`. This is what's called a **collision** and it is the reason that implementing a hash table is a non-trivial task.

Note: What isn't shown in the diagram is the full key-value pair of the element stored in the hash table. It would be pointless to store `66` as both the key and the value—likely `66` would be a unique id mapped to some other information, say a student's name for a gradebook.

How to Handle Collisions: Chaining

One way to handle collisions is through a strategy called *chaining*. The "chain" in chaining refers to using a linked list for every index in the hash

table array. When an element maps to an index, then that element is stored within the linked list associated with that index. We can visualize this below:

```
index    array

       +-------+   +-------+    +-------+
 [0]   |   A   | → |   B   | →  |   C   | → NULL
       +-------+   +-------+    +-------+
 [1]   |   D   | → NULL
       +-------+
 [2]   |  NULL |
       +-------+   +-------+
 [3]   |   E   | → |   F   | → NULL
       +-------+   +-------+
 [4]   |  NULL |
       +-------+
```

Although the above works well to handle collisions, it can detract from the initial incentive of the quick retrieval of elements when using hash tables.

How to Handle Collisions: Hashing Methods

There are some heuristic hashing function methods that are often employed in real-world hash maps such as the *division method*, *multiplication method*, *mid square method*, and the *digit folding method*

- Division method
  - h(k) = k mod m
  - m should not be a power of 2
  - a prime not too close to a power of 2 yields good results
- Multiplication method
  - h(k) = ⌊m (k A mod 1)⌋ where A in range 0 < A < 1
  - m is often a power of 2
  - Donald Knuth suggests A = (5^(1/2) - 1)/2
- Mid square method
  - h(k) = k^(2) and extract middle r digits
- Digit folding method
  - Subdivide k into groups of digits
  - Add those digits together

I have included a Python script, hash_benchmark.py, which can be used to see how these collision functions work in practice with different distributions of data.

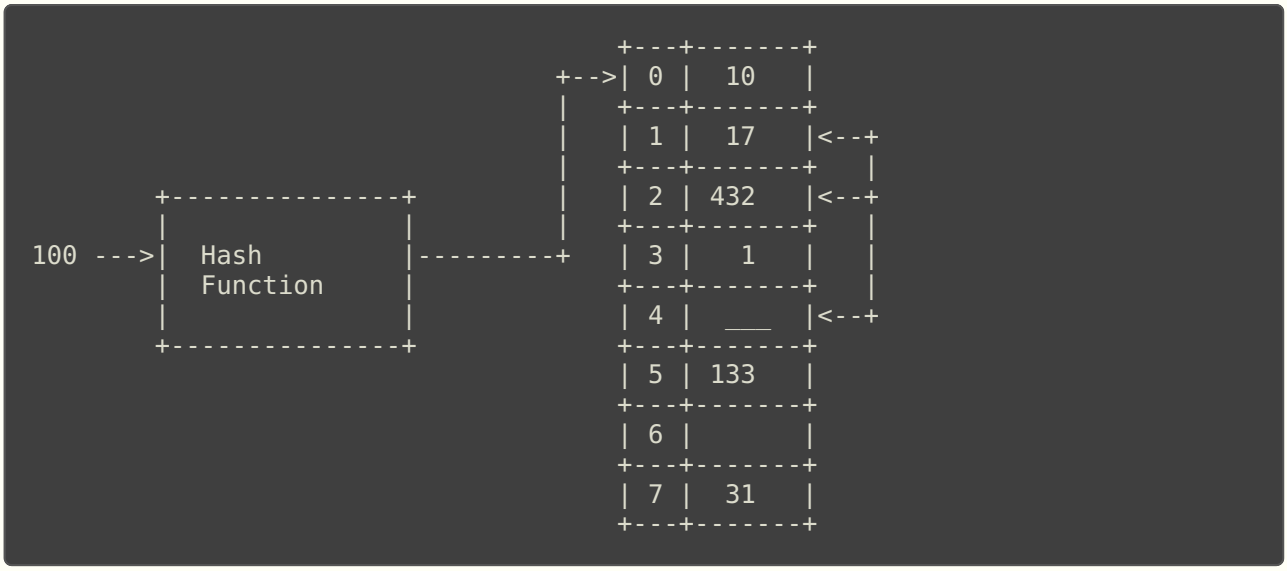
How to Handle Collisions: Closed Hashing

Closed hashing, sometimes called *open addressing*, refers to a hash table strategy where chaining is eschewed for other clever workarounds to collisions. Instead of having each index mapped to a linked list, an empty

index will be found to map the element to. We'll look at three types of closed hashing strategies: *linear probing*, *quadratic probing*, and *double hashing*.

Linear Probing

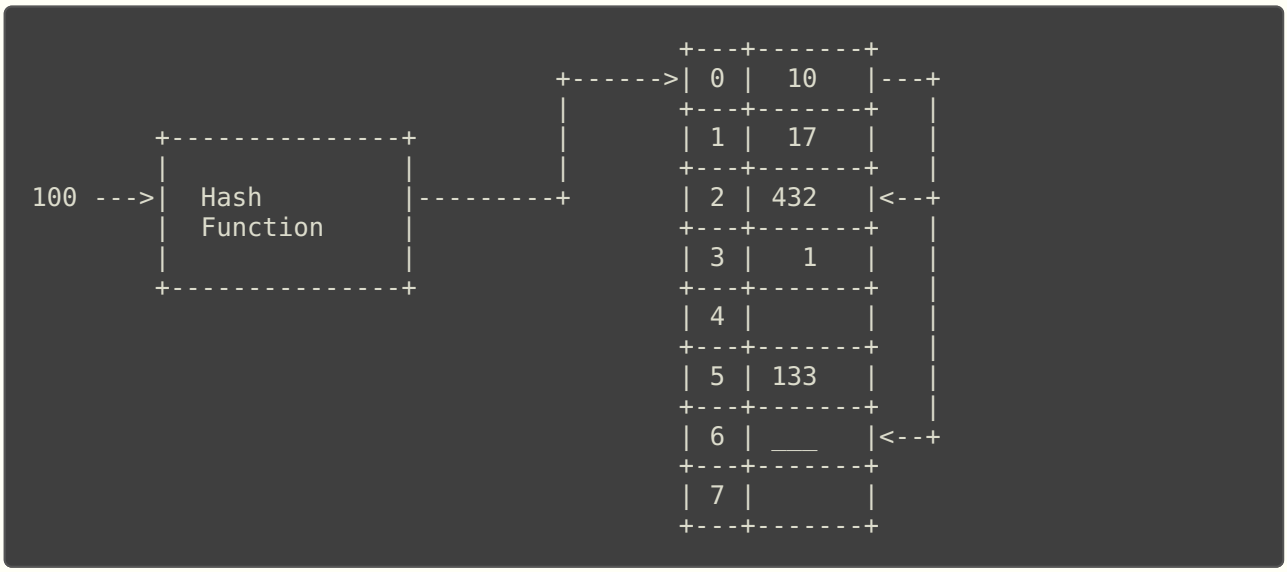
Linear probing simply searches for an empty slot when one cannot immediately be found:



The hash function maps `100` to `0`, but `0` is already taken. We then flip through the remaining indices until we find an empty spot to slot `100`.

Quadratic Probing

Quadratic probing uses the same idea as linear probing, but combines it with a quadratic polynomial expression for determining the next index. A potential formula would be $h(k, i) = (h(k) + (a \times i) + (b \times i\text{^}2)) \bmod m$. If we let a = b = 1, then:



First, we plug 100 into h(x) and receive 0 (h(100) is given by the diagram). We then see that it is already taken, which means we need to deal with the collision. Second, we plug our values into the above formula, where the constants `a` and `b` equal 0 and i is the initial index, and get:

- (h(k) + (a × i) + (b × i^2)) mod m
- (0 + (1 × 0) + (1 × 0)) mod m
- 0

0 is still a collision, so we increment i:

- (h(k) + (a × i) + (b × i^2)) mod m
- (0 + (1 × 1) + (1 × 1)) mod m
- 2

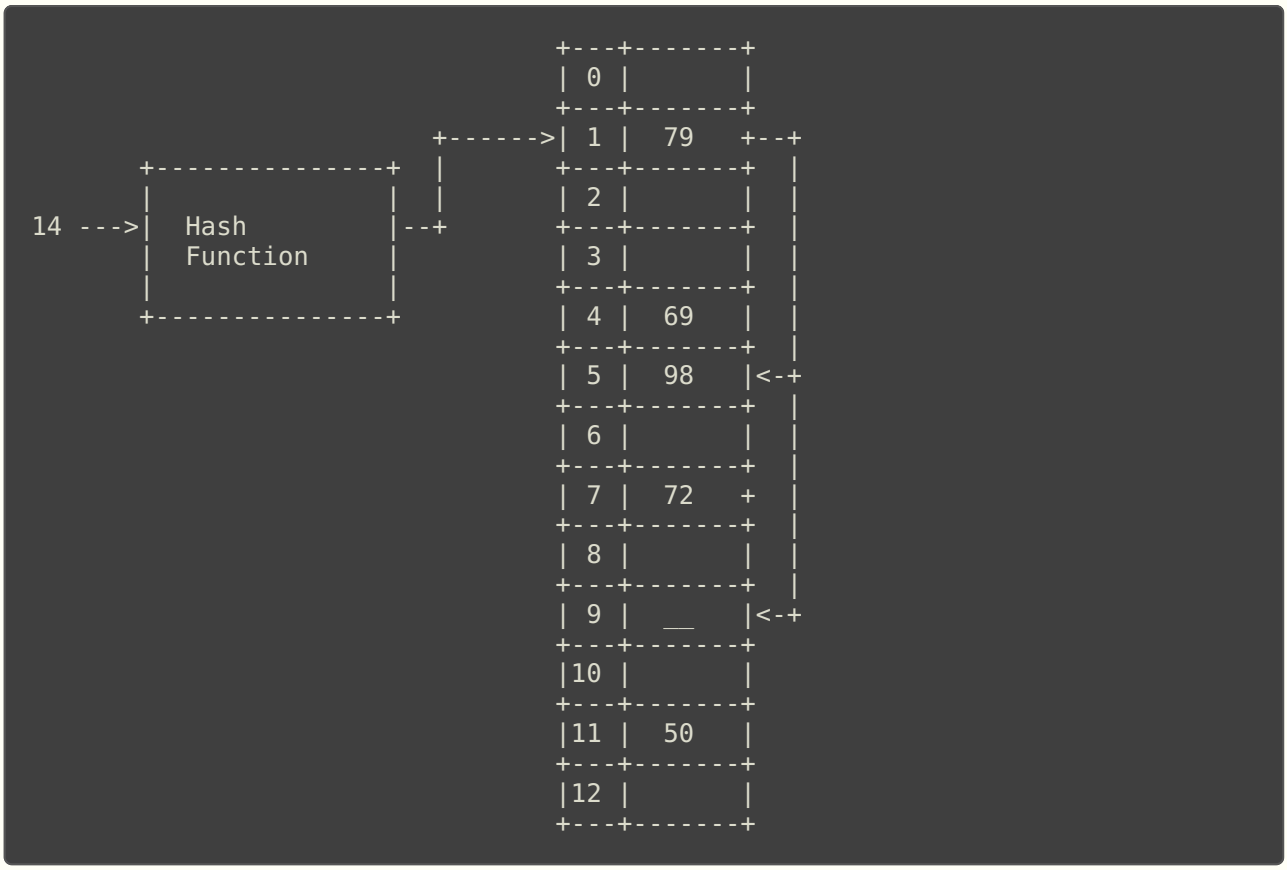
2 is still a collision, so we increment i again:

- (h(k) + (a × i) + (b × i^2)) mod m
- (0 + (1 × 2) + (1 × 4)) mod m
- 6

6 gives us an empty slot to place our element into!

Double Hashing

Double hashing takes the philosophy of "when in doubt, hash again." This time we have two predefined hash functions, `h` and `g`. In the case of a collision we refer to the function $h(k,i) = (h(k) + (i \times g(k))) \bmod m$. If we take h(k) to be $k \bmod 13$ and g(k) to be $1 + (k \bmod 11)$ then we can see what it would look like to input `14`:



First, we try 1 which is taken. Second, we try 5 which is (1 + 1 × (1 + 3)). Finally, we try 9 which is (1 + 2 × (1 + 3)). 9 is empty, so we can slot it in.

Rehashing

Recall our discussion of vectors. When a vector got too full, a categorization dependent on the implementation, it would need to be "stretched out" with the contents copied over to another, larger array. Rehashing works in the exact same way, although the major difference is that elements can no longer be naively copied over---*they must be hashed again*.

Exercises

1. Using the hash function `f(x) = x % 7` and the input `[14, 27, 3, 56, 91, 42]`, compute the index for each element by hand and draw the resulting hash table array. Does any collision occur? If so, identify which elements collide and at which index.
2. Run `hash_benchmark.py --summary` with the default distributions and record the collision counts for each method. Then run it with `--graph` and compare the growth curves. Which method performs best on each distribution and why? Write a short paragraph summarizing your findings.
3. Implement a hash table in C using the chaining strategy shown in the chapter. Your implementation should use an array of linked list heads as the underlying structure. Implement `insert`, `get`, and `delete`

operations. Test it by inserting `[10, 22, 31, 4, 15, 28, 17, 88, 59]` with the hash function `h(k) = k % 7` and printing the contents of each bucket.

4. The chapter states that chaining "can detract from the initial incentive of quick retrieval." Explain in your own words why this is the case. What is the worst case number of comparisons needed to find an element in a hash table using chaining when all elements hash to the same index?
5. Implement linear probing in C. Using the hash function `h(k) = k % 8` and the sequence `[10, 17, 432, 1, 100, 133, 31]`, insert each element and print the final state of the array. How many probing steps were required in total across all insertions?
6. Using the quadratic probing formula `h(k, i) = (h(k) + (a × i) + (b × i²)) mod m` with `a = b = 1` and `m = 8`, manually compute the insertion index for `100` given that `h(100) = 0` and indices 0 and 2 are already occupied. Show each step of the calculation. Then implement quadratic probing in C and verify your result.
7. The double hashing example in the chapter uses `h(k) = k mod 13` and `g(k) = 1 + (k mod 11)` to insert `14`. Verify each step of the calculation by hand: confirm that index 1 collides, that index 5 collides, and that index 9 is empty. Then use the same two hash functions to insert the value `27` into the same table and show each probing step.
8. The chapter compares rehashing to the resizing of vectors. Implement a hash table in C that tracks its load factor (number of elements divided by table size) and automatically rehashes into a table twice the size when the load factor exceeds 0.7. Verify by inserting enough elements to trigger a rehash and printing the table before and after.

Footnotes:

1 *Introduction to Algorithms - Second Edition*, Cormen et al.

(Appendix) Using printf

Introduction

One of the easiest ways for beginngers to get output from a C program is to use the standard library `printf` function. The major advantage is that it is relatively easy to use and comes for free with an include of `stdio.h`. The major disadvantage is new programmers often don't understand functions yet and find the formatted string argument a bit arcane. This appendix exists to try to minimize confusion.

The Anatomy of printf

```
int printf(char *format, arg1, arg2, ...);
```

`printf` is a variable argument function which prints out a string of characters to standard out—which for this class is pretty much always linked to the terminal screen. The first argument is always a string of characters, and every subsequent argument is a value that corresponds to a format string variable. For example, `printf("%d - %d = %d", 3, 2, 1)` will print out "3 - 2 = 1" because the first `%d` maps to `3`, the second `%d` mapes to `2`, and the final `%d` maps to `1`.

Formatted String

The first argument taken in `printf` is a formatted string. The formatting comes in the form of special characters and symbols peppered throughout the character sequence. For example, above you saw the special characters `%d` which is programmed to represent an integer value. Here is a table of *conversion characters*:

Character	Argument Type	Printed As
d, i	int	Decimal number
o	int	Unsigned octal number (without leading zero)
x, X	int	Unsigned hexadecimal number (without leading 0x or 0X), using abcdef or ABCDEF for 10, ..., 15
u	int	Unsigned decimal number
c	int	Single character
s	char *	Print characters from the string until \0 or the number of characters given by the precision

Character	Argument Type	Printed As
f	double	[-]m.dddddd, where the number of d's is given by the precision (default 6)
e, E	double	[-]m.dddddde+/-xx or [-]m.ddddddE+/-xx, where the number of d's is given by the precision (default 6)
g, G	double	Use %e or %E if the exponent is less than -4 or greater than or equal to the precision; otherwise use %f. Trailing zeros and a trailing decimal point are not printed
p	void *	Pointer (implementation-dependent)
%	no argument	Print a %

printf Examples

Here are a few examples using `printf` that you can learn from and try out yourself:

```c
#include <stdio.h>
int main(){

    /* 1. Basic integer and string */
    printf("Name: %s, Age: %d\n", "Alice", 30);

    /* 2. Padded integer right-aligned in field of width 10 */
    printf("Score: %10d\n", 42);

    /* 3. Padded integer left-aligned in field of width 10 */
    printf("Score: %-10d|\n", 42);

    /* 4. Zero-padded integer */
    printf("ID: %08d\n", 1234);

    /* 5. Hexadecimal upper and lower case */
    printf("Hex: %x  HEX: %X\n", 255, 255);

    /* 6. Hexadecimal with 0x prefix */
    printf("Hex with prefix: %#x\n", 255);

    /* 7. Float with default precision */
    printf("Pi: %f\n", 3.14159265);

    /* 8. Float with custom precision */
    printf("Pi (2dp): %.2f\n", 3.14159265);

    /* 9. Float in scientific notation */
    printf("Scientific: %e\n", 123456789.0);

    /* 10. Float using %g (auto selects %e or %f) */
    printf("Auto: %g  Auto: %g\n", 0.00001, 123456.0);

    /* 11. Character and its decimal value */
    printf("Char: %c  Dec: %d\n", 'A', 'A');

    /* 12. String with precision (truncate after 5 chars) */
    printf("Truncated: %.5s\n", "Hello, World!");

    /* 13. String padded to width 20, left-aligned */
    printf("%-20s|\n", "left");

    /* 14. Pointer address */
    int x = 42;
    printf("Address of x: %p\n", (void *)&x);

    /* 15. Mixed: octal, unsigned, padded float, and percent sign */
    printf("Oct: %o  Uint: %u  Float: %8.3f  Pct: 100%%\n",
           255, 4294967295u, 3.14159);

    return 0;
}
```

(Appendix) #include and Macros

Introduction

If you're reading this appendix chapter you've likely already encountered the use of the `#include` phrase in coding examples. You'd be hard pressed to find any C code that doesn't make use of that or other lines using the `#` symbol. This chapter aims to demystify what's going on when it and similar phrases are invoked.

#include Command

Every single time we've used `printf` we've included the line `#include <stdio.h>`. For example, even in a simple "hello world" program we use it—here's what happens if we don't:

```
1:   int main(){
2:     printf("Hello, World!\n");
3:   }
```

```
 1:   josephraskind@stargazer:/tmp/macros$ gcc hello_world.c
 2:   hello_world.c: In function ‘main’:
 3:   hello_world.c:2:3: warning: implicit declaration of function ‘printf’ [-
Wimplicit-function-declaration]
 4:       2 |   printf("Hello, World!\n");
 5:         |   ^~~~~~
 6:   hello_world.c:2:3: warning: incompatible implicit declaration of built-
in function ‘printf’
 7:   hello_world.c:1:1: note: include ‘<stdio.h>’ or provide a declaration of
‘printf’
 8:     +++ |+#include <stdio.h>
 9:       1 | int main(){
10:   josephraskind@stargazer:/tmp/macros$ ./a.out
11:   Hello, World!
```

Why the warnings? Well, `gcc` actually has no issue compiling when a non-existant function is called—it essentially defers to the programmer that there really is a function called `printf` that will eventually be resolved at the linking stage. We can see what happens if I replace `printf` with the non-existant function `printNOTREAL`:

```
josephraskind@stargazer:/tmp/macros$ gcc hello_world.c
hello_world.c: In function ‘main’:
hello_world.c:2:3: warning: implicit declaration of function
‘printNOTREAL’ [-Wimplicit-function-declaration]
      2 |   printNOTREAL("Hello, World!\n");
        |   ^~~~~~~~~~~
/usr/bin/ld: /tmp/ccnimtjV.o: in function `main':
hello_world.c:(.text+0x15): undefined reference to `printNOTREAL'
collect2: error: ld returned 1 exit status
```

`printNOTREAL` isn't resolved, but `printf` is? What gives?

Well, let's take a look at what is actually called by `gcc`:

```
 1:  josephraskind@stargazer:/tmp/macros$ gcc -v
 2:  Using built-in specs.
 3:  COLLECT_GCC=gcc
 4:  COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
 5:  OFFLOAD_TARGET_NAMES=nvptx-none:hsa
 6:  OFFLOAD_TARGET_DEFAULT=1
 7:  Target: x86_64-linux-gnu
 8:  Configured with: ../src/configure -v --with-pkgversion='Ubuntu
9.4.0-1ubuntu1~20.04.2' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs
--enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr
--with-gcc-major-version-only --program-suffix=-9 --program-prefix=x86_64-
linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --
without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls
--enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --
with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-
verify --enable-plugin --enable-default-pie --with-system-zlib --with-target-
system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --
with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-
multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/
gcc-9-9QDOt0/gcc-9-9.4.0/debian/tmp-nvptx/usr,hsa --without-cuda-driver --
enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --
target=x86_64-linux-gnu
 9:  Thread model: posix
10:  gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)
```

That's a lot of hidden configurations! The most instructive for us is Line 4 which shows the COLLECT_LTO_WRAPPER environment variable for Link Time Optimization. If we peek inside the binary file:

```
00000350: 2f6c 6962 3634 2f6c 642d 6c69 6e75 782d  /lib64/ld-linux-
00000360: 7838 362d 3634 2e73 6f2e 3200 0000 0000  x86-64.so.2.....
```

we see a reference to `ld-linux-x86-64.so.2` which is the dynamic linker and loader for my Linux environment. Within the loader is a reference to `printf`, but there isn't an actual implementation. For that we can check the executable returned by `gcc hello_world.c -o hello`:

```
josephraskind@stargazer:/tmp/macros$ ldd ./hello
        linux-vdso.so.1 (0x00007fff485c8000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe54e8f2000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe54eb07000)
```

`ldd` prints shared object dependencies, and we can see that the major dependency in our `hello` executable is `libc.so.6` which *does* store a precompiled implementation of `printf`! Why all of this jumping around? Well, it turns out that by default `gcc` does something called *dynamic linking* where certain bits of code are only linked to in the executable instead of being baked into the executable itself. Why? Well lets dynamically link and statically link `hello_world.c`:

```
josephraskind@stargazer:/tmp/macros$ gcc hello_world.c -o hello_dynamic
josephraskind@stargazer:/tmp/macros$ gcc hello_world.c --static -o hello_static
josephraskind@stargazer:/tmp/macros$ ./hello_dynamic
Hello, World!
josephraskind@stargazer:/tmp/macros$ ./hello_static
Hello, World!
```

They both work exactly the same…so what's the big deal? If we take a look at their sizes:

```
josephraskind@stargazer:/tmp/macros$ du -h ./hello_dynamic ./hello_static
20K     ./hello_dynamic
852K    ./hello_static
```

We can see that the statically linked binary is 40x greater than the dynamically linked binary!

Header Files

What exactly is `stdio.h`? It is a **header file** which is just a regular C file that is specifically written to be *included* by different C files. There is nothing *semantically* special about header files. For example, I can define my own header file called `cool_print.h`:

```
 1:  #include <stdio.h>
 2:  #include <stdarg.h>
 3:
 4:  void cool_printf(const char *fmt, ...){
 5:    va_list args;         // Using variable argument list type
 6:    va_start(args, fmt);
 7:    printf("cool: ");     // Preprinting "cool:" before calling real printf
 8:    vprintf(fmt, args);
 9:    va_end(args);         // Unwinding argument list
10:  }
```

Then, in the same directory, I can define a new C file called `using_cool_printf.c`:

```
1:  #include "cool_print.h"
2:
3:  int main(){
4:    cool_printf("Hello, World!\n");
5:  }
```

Notice that I put double quotes around `cool_print.h` instead of angle brackets. Angle brackets are for header files that are stored in the standard library (see the macros section below for how that gets expanded). Double quotes are for relative or absolute file paths. We can compile and run the program:

```
josephraskind@stargazer:/tmp/macros$ gcc using_cool_print.c -o cool; ./cool
cool: Hello, World!
```

Again, if I don't use the include statement:

```
josephraskind@stargazer:/tmp/macros$ gcc using_cool_print.c -o cool
using_cool_print.c: In function ‘main’:
using_cool_print.c:4:3: warning: implicit declaration of function
‘cool_printf’ [-Wimplicit-function-declaration]
    4 |   cool_printf("Hello, World!\n");
      |   ^~~~~~~~~~~
/usr/bin/ld: /tmp/ccnBuSd7.o: in function `main':
using_cool_print.c:(.text+0x15): undefined reference to `cool_printf'
collect2: error: ld returned 1 exit status
```

Because `cool_printf` is not in the standard, dynamic linked binaries it cannot be linked and thus an executable cannot be created.

What happens if I include my header twice? Let's see:

```
1:  #include "cool_print.h"
2:  #include "cool_print.h"
3:
4:  int main(){
5:    cool_printf("Hello, World!\n");
6:  }
```

```
josephraskind@stargazer:/tmp/macros$ gcc using_cool_print.c -o cool
In file included from using_cool_print.c:2:
cool_print.h:6:6: error: redefinition of ‘cool_printf’
    6 | void cool_printf(const char *fmt, ...){
      |      ^~~~~~~~~~~
In file included from using_cool_print.c:1:
cool_print.h:6:6: note: previous definition of ‘cool_printf’ was here
    6 | void cool_printf(const char *fmt, ...){
      |      ^~~~~~~~~~~
```

As we'll see later, include is essentially a simple copy-paste job of the header "into" the **.c** file. As detailed in the chapter on functions, we cannot define a function twice in C. This can be solved with a header guard (which you'll find on every standard library header file):

```
1:  #ifndef CS211_COOL_PRINT
2:  #define CS211_COOL_PRINT
3:  #include <stdio.h>
4:  #include <stdarg.h>
5:
6:  void cool_printf(const char *fmt, ...){
7:    va_list args;         // Using variable argument list type
8:    va_start(args, fmt);
9:    printf("cool: ");     // Preprinting "cool:" before calling real printf
10:   vprintf(fmt, args);
11:   va_end(args);         // Unwinding argument list
12: }
13: #endif
```

The guards check to see if the symbol `CS211_COOL_PRINT` exists. If it does, the macro will not be expanded; if it doesn't, the macro will be expaned. If we try the double include again:

```
josephraskind@stargazer:/tmp/macros$ gcc using_cool_print.c -o cool; ./cool
cool: Hello, World!
```

Now it works perfect!

Macros

Anytime you see the `#` symbol at the start of a C statement you're actually seeing something that exists outside the C Programming Language proper. You are looking at a **macro**. There are many macro *directives*, but in this class we're only concerned with `define`, `include`, `ifdef`, and `endif`:

- `define`
  - Defines a macro symbol with content
  - `#define SIZE 5` : Anytime `SIZE` is used it will be replaced with `5`.
- `include`
  - Will paste header file contents into the source code
  - `#include <stdio.h>` : Pastes all of the standard library `stdio.h` file into the host file
- `ifdef`
  - Checks to see if a macro symbol has been defined already and will perform the next macro directive if true
- `endif`
  - Terminates the matching `ifdef` directive

Let's look at a simple example:

```
1:  #define VAL 10
2:  int main(){
3:    return VAL;
4:  }
```

We see here a macro definition for the symbol `VAL` which is tied to 10. If I run this code and capture the return value of the executable:

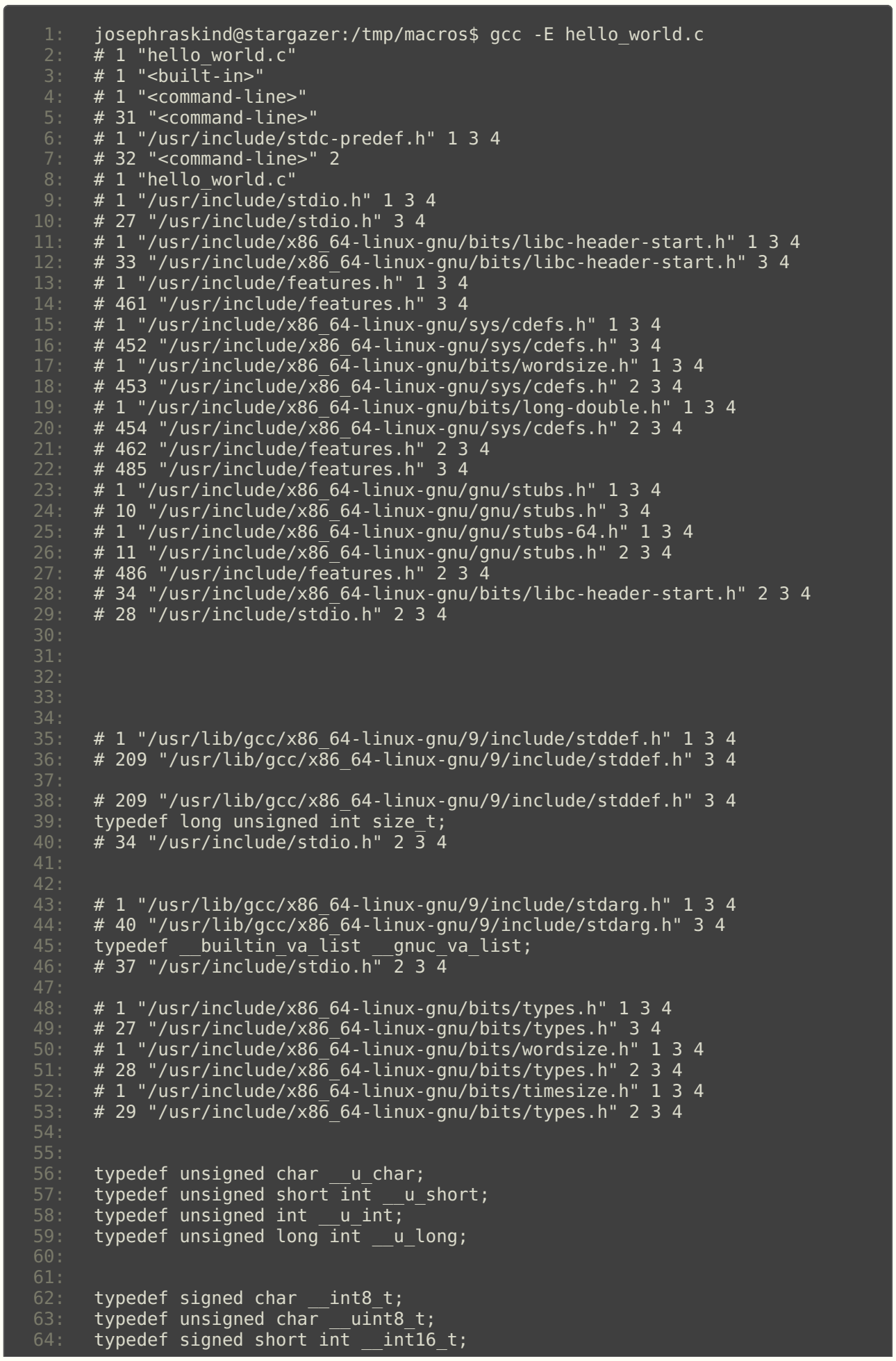
```
josephraskind@stargazer:/tmp/macros$ gcc val.c -o val; ./val; echo $?
10
```

We can see that 10 was returned, just like what we defined. How does that work? Well, in `gcc` we can stop compilation at the macro expansion stage with `gcc -E`:

```
josephraskind@stargazer:/tmp/macros$ gcc -E val.c
# 1 "val.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "val.c"

int main(){
  return 10;
}
```

We can see that our `VAL` is gone completely and has been replaced by the number it was initially defined with! Let's see what happens with our basic `hello_world.c`:

```
josephraskind@stargazer:/tmp/macros$ gcc -E hello_world.c
# 1 "hello_world.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "hello_world.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4
# 33 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 461 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 452 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 453 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/long-double.h" 1 3 4
# 454 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 462 "/usr/include/features.h" 2 3 4
# 485 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 486 "/usr/include/features.h" 2 3 4
# 34 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4
```



```
# 1 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 1 3 4
# 209 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 3 4

# 209 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stddef.h" 3 4
typedef long unsigned int size_t;
# 34 "/usr/include/stdio.h" 2 3 4


# 1 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stdarg.h" 1 3 4
# 40 "/usr/lib/gcc/x86_64-linux-gnu/9/include/stdarg.h" 3 4
typedef __builtin_va_list __gnuc_va_list;
# 37 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/x86_64-linux-gnu/bits/types.h" 1 3 4
# 27 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 28 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/timesize.h" 1 3 4
# 29 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4


typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;


typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
```

```
  65:   typedef unsigned short int __uint16_t;
  66:   typedef signed int __int32_t;
  67:   typedef unsigned int __uint32_t;
  68:
  69:   typedef signed long int __int64_t;
  70:   typedef unsigned long int __uint64_t;
  71:
  72:
  73:
  74:
  75:
  76:
  77:   typedef __int8_t __int_least8_t;
  78:   typedef __uint8_t __uint_least8_t;
  79:   typedef __int16_t __int_least16_t;
  80:   typedef __uint16_t __uint_least16_t;
  81:   typedef __int32_t __int_least32_t;
  82:   typedef __uint32_t __uint_least32_t;
  83:   typedef __int64_t __int_least64_t;
  84:   typedef __uint64_t __uint_least64_t;
  85:
  86:
  87:
  88:   typedef long int __quad_t;
  89:   typedef unsigned long int __u_quad_t;
  90:
  91:
  92:
  93:
  94:
  95:
  96:
  97:   typedef long int __intmax_t;
  98:   typedef unsigned long int __uintmax_t;
  99:   # 141 "/usr/include/x86_64-linux-gnu/bits/types.h" 3 4
 100:   # 1 "/usr/include/x86_64-linux-gnu/bits/typesizes.h" 1 3 4
 101:   # 142 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4
 102:   # 1 "/usr/include/x86_64-linux-gnu/bits/time64.h" 1 3 4
 103:   # 143 "/usr/include/x86_64-linux-gnu/bits/types.h" 2 3 4
 104:
 105:
 106:   typedef unsigned long int __dev_t;
 107:   typedef unsigned int __uid_t;
 108:   typedef unsigned int __gid_t;
 109:   typedef unsigned long int __ino_t;
 110:   typedef unsigned long int __ino64_t;
 111:   typedef unsigned int __mode_t;
 112:   typedef unsigned long int __nlink_t;
 113:   typedef long int __off_t;
 114:   typedef long int __off64_t;
 115:   typedef int __pid_t;
 116:   typedef struct { int __val[2]; } __fsid_t;
 117:   typedef long int __clock_t;
 118:   typedef unsigned long int __rlim_t;
 119:   typedef unsigned long int __rlim64_t;
 120:   typedef unsigned int __id_t;
 121:   typedef long int __time_t;
 122:   typedef unsigned int __useconds_t;
 123:   typedef long int __suseconds_t;
 124:
 125:   typedef int __daddr_t;
 126:   typedef int __key_t;
 127:
 128:
 129:   typedef int __clockid_t;
```

```
130:
131:
132:   typedef void * __timer_t;
133:
134:
135:   typedef long int __blksize_t;
136:
137:
138:
139:
140:   typedef long int __blkcnt_t;
141:   typedef long int __blkcnt64_t;
142:
143:
144:   typedef unsigned long int __fsblkcnt_t;
145:   typedef unsigned long int __fsblkcnt64_t;
146:
147:
148:   typedef unsigned long int __fsfilcnt_t;
149:   typedef unsigned long int __fsfilcnt64_t;
150:
151:
152:   typedef long int __fsword_t;
153:
154:   typedef long int __ssize_t;
155:
156:
157:   typedef long int __syscall_slong_t;
158:
159:   typedef unsigned long int __syscall_ulong_t;
160:
161:
162:
163:   typedef __off64_t __loff_t;
164:   typedef char *__caddr_t;
165:
166:
167:   typedef long int __intptr_t;
168:
169:
170:   typedef unsigned int __socklen_t;
171:
172:
173:
174:
175:   typedef int __sig_atomic_t;
176:   # 39 "/usr/include/stdio.h" 2 3 4
177:   # 1 "/usr/include/x86_64-linux-gnu/bits/types/__fpos_t.h" 1 3 4
178:
179:
180:
181:
182:   # 1 "/usr/include/x86_64-linux-gnu/bits/types/__mbstate_t.h" 1 3 4
183:   # 13 "/usr/include/x86_64-linux-gnu/bits/types/__mbstate_t.h" 3 4
184:   typedef struct
185:   {
186:     int __count;
187:     union
188:     {
189:       unsigned int __wch;
190:       char __wchb[4];
191:     } __value;
192:   } __mbstate_t;
193:   # 6 "/usr/include/x86_64-linux-gnu/bits/types/__fpos_t.h" 2 3 4
194:
```

```
195:
196:
197:
198:   typedef struct _G_fpos_t
199:   {
200:     __off_t __pos;
201:     __mbstate_t __state;
202:   } __fpos_t;
203:   # 40 "/usr/include/stdio.h" 2 3 4
204:   # 1 "/usr/include/x86_64-linux-gnu/bits/types/__fpos64_t.h" 1 3 4
205:   # 10 "/usr/include/x86_64-linux-gnu/bits/types/__fpos64_t.h" 3 4
206:   typedef struct _G_fpos64_t
207:   {
208:     __off64_t __pos;
209:     __mbstate_t __state;
210:   } __fpos64_t;
211:   # 41 "/usr/include/stdio.h" 2 3 4
212:   # 1 "/usr/include/x86_64-linux-gnu/bits/types/__FILE.h" 1 3 4
213:
214:
215:
216:   struct _IO_FILE;
217:   typedef struct _IO_FILE __FILE;
218:   # 42 "/usr/include/stdio.h" 2 3 4
219:   # 1 "/usr/include/x86_64-linux-gnu/bits/types/FILE.h" 1 3 4
220:
221:
222:
223:   struct _IO_FILE;
224:
225:
226:   typedef struct _IO_FILE FILE;
227:   # 43 "/usr/include/stdio.h" 2 3 4
228:   # 1 "/usr/include/x86_64-linux-gnu/bits/types/struct_FILE.h" 1 3 4
229:   # 35 "/usr/include/x86_64-linux-gnu/bits/types/struct_FILE.h" 3 4
230:   struct _IO_FILE;
231:   struct _IO_marker;
232:   struct _IO_codecvt;
233:   struct _IO_wide_data;
234:
235:
236:
237:
238:   typedef void _IO_lock_t;
239:
240:
241:
242:
243:
244:   struct _IO_FILE
245:   {
246:     int _flags;
247:
248:
249:     char *_IO_read_ptr;
250:     char *_IO_read_end;
251:     char *_IO_read_base;
252:     char *_IO_write_base;
253:     char *_IO_write_ptr;
254:     char *_IO_write_end;
255:     char *_IO_buf_base;
256:     char *_IO_buf_end;
257:
258:
259:     char *_IO_save_base;
```

```
260:     char *_IO_backup_base;
261:     char *_IO_save_end;
262:
263:     struct _IO_marker *_markers;
264:
265:     struct _IO_FILE *_chain;
266:
267:     int _fileno;
268:     int _flags2;
269:     __off_t _old_offset;
270:
271:
272:     unsigned short _cur_column;
273:     signed char _vtable_offset;
274:     char _shortbuf[1];
275:
276:     _IO_lock_t *_lock;
277:
278:
279:
280:
281:
282:
283:
284:     __off64_t _offset;
285:
286:     struct _IO_codecvt *_codecvt;
287:     struct _IO_wide_data *_wide_data;
288:     struct _IO_FILE *_freeres_list;
289:     void *_freeres_buf;
290:     size_t __pad5;
291:     int _mode;
292:
293:     char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof
(size_t)];
294:   };
295:   # 44 "/usr/include/stdio.h" 2 3 4
296:   # 52 "/usr/include/stdio.h" 3 4
297:   typedef __gnuc_va_list va_list;
298:   # 63 "/usr/include/stdio.h" 3 4
299:   typedef __off_t off_t;
300:   # 77 "/usr/include/stdio.h" 3 4
301:   typedef __ssize_t ssize_t;
302:
303:
304:
305:
306:
307:
308:   typedef __fpos_t fpos_t;
309:   # 133 "/usr/include/stdio.h" 3 4
310:   # 1 "/usr/include/x86_64-linux-gnu/bits/stdio_lim.h" 1 3 4
311:   # 134 "/usr/include/stdio.h" 2 3 4
312:
313:
314:
315:   extern FILE *stdin;
316:   extern FILE *stdout;
317:   extern FILE *stderr;
318:
319:
320:
321:
322:
323:
```

```
324:   extern int remove (const char *__filename) __attribute__
((__nothrow__ , __leaf__));
325:
326:   extern int rename (const char *__old, const char *__new) __attribute__
((__nothrow__ , __leaf__));
327:
328:
329:
330:   extern int renameat (int __oldfd, const char *__old, int __newfd,
331:          const char *__new) __attribute__ ((__nothrow__ , __leaf__));
332:   # 173 "/usr/include/stdio.h" 3 4
333:   extern FILE *tmpfile (void) ;
334:   # 187 "/usr/include/stdio.h" 3 4
335:   extern char *tmpnam (char *__s) __attribute__ ((__nothrow__ ,
__leaf__)) ;
336:
337:
338:
339:
340:   extern char *tmpnam_r (char *__s) __attribute__ ((__nothrow__ ,
__leaf__)) ;
341:   # 204 "/usr/include/stdio.h" 3 4
342:   extern char *tempnam (const char *__dir, const char *__pfx)
343:          __attribute__ ((__nothrow__ , __leaf__)) __attribute__
((__malloc__)) ;
344:
345:
346:
347:
348:
349:
350:
351:   extern int fclose (FILE *__stream);
352:
353:
354:
355:
356:   extern int fflush (FILE *__stream);
357:   # 227 "/usr/include/stdio.h" 3 4
358:   extern int fflush_unlocked (FILE *__stream);
359:   # 246 "/usr/include/stdio.h" 3 4
360:   extern FILE *fopen (const char *__restrict __filename,
361:         const char *__restrict __modes) ;
362:
363:
364:
365:
366:   extern FILE *freopen (const char *__restrict __filename,
367:           const char *__restrict __modes,
368:           FILE *__restrict __stream) ;
369:   # 279 "/usr/include/stdio.h" 3 4
370:   extern FILE *fdopen (int __fd, const char *__modes) __attribute__
((__nothrow__ , __leaf__)) ;
371:   # 292 "/usr/include/stdio.h" 3 4
372:   extern FILE *fmemopen (void *__s, size_t __len, const char *__modes)
373:     __attribute__ ((__nothrow__ , __leaf__)) ;
374:
375:
376:
377:
378:   extern FILE *open_memstream (char **__bufloc, size_t *__sizeloc)
__attribute__ ((__nothrow__ , __leaf__)) ;
379:
380:
381:
```

```
382:
383:
384:   extern void setbuf (FILE *__restrict __stream, char *__restrict __buf)
__attribute__ ((__nothrow__ , __leaf__));
385:
386:
387:
388:   extern int setvbuf (FILE *__restrict __stream, char *__restrict __buf,
389:         int __modes, size_t __n) __attribute__ ((__nothrow__ , __leaf__));
390:
391:
392:
393:
394:   extern void setbuffer (FILE *__restrict __stream, char *__restrict
__buf,
395:           size_t __size) __attribute__ ((__nothrow__ , __leaf__));
396:
397:
398:   extern void setlinebuf (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__));
399:
400:
401:
402:
403:
404:
405:
406:   extern int fprintf (FILE *__restrict __stream,
407:         const char *__restrict __format, ...);
408:
409:
410:
411:
412:   extern int printf (const char *__restrict __format, ...);
413:
414:   extern int sprintf (char *__restrict __s,
415:         const char *__restrict __format, ...) __attribute__
((__nothrow__));
416:
417:
418:
419:
420:
421:   extern int vfprintf (FILE *__restrict __s, const char *__restrict
__format,
422:          __gnuc_va_list __arg);
423:
424:
425:
426:
427:   extern int vprintf (const char *__restrict __format, __gnuc_va_list
__arg);
428:
429:   extern int vsprintf (char *__restrict __s, const char *__restrict
__format,
430:          __gnuc_va_list __arg) __attribute__ ((__nothrow__));
431:
432:
433:
434:   extern int snprintf (char *__restrict __s, size_t __maxlen,
435:          const char *__restrict __format, ...)
436:         __attribute__ ((__nothrow__)) __attribute__ ((__format__
(__printf__, 3, 4)));
437:
438:   extern int vsnprintf (char *__restrict __s, size_t __maxlen,
```

```
439:           const char *__restrict __format, __gnuc_va_list __arg)
440:        __attribute__ ((__nothrow__)) __attribute__ ((__format__
(__printf__, 3, 0)));
441:   # 379 "/usr/include/stdio.h" 3 4
442:   extern int vdprintf (int __fd, const char *__restrict __fmt,
443:          __gnuc_va_list __arg)
444:        __attribute__ ((__format__ (__printf__, 2, 0)));
445:   extern int dprintf (int __fd, const char *__restrict __fmt, ...)
446:        __attribute__ ((__format__ (__printf__, 2, 3)));
447:
448:
449:
450:
451:
452:
453:
454:   extern int fscanf (FILE *__restrict __stream,
455:        const char *__restrict __format, ...) ;
456:
457:
458:
459:
460:   extern int scanf (const char *__restrict __format, ...) ;
461:
462:   extern int sscanf (const char *__restrict __s,
463:        const char *__restrict __format, ...) __attribute__
((__nothrow__ , __leaf__));
464:
465:
466:
467:
468:
469:
470:   extern int fscanf (FILE *__restrict __stream, const char *__restrict
__format, ...) __asm__ ("" "__isoc99_fscanf")
471:
472:                                    ;
473:   extern int scanf (const char *__restrict __format, ...) __asm__ ("" 
"__isoc99_scanf")
474:                                    ;
475:   extern int sscanf (const char *__restrict __s, const char *__restrict
__format, ...) __asm__ ("" "__isoc99_sscanf") __attribute__ ((__nothrow__ ,
__leaf__))
476:
477:                           ;
478:   # 432 "/usr/include/stdio.h" 3 4
479:   extern int vfscanf (FILE *__restrict __s, const char *__restrict
__format,
480:          __gnuc_va_list __arg)
481:        __attribute__ ((__format__ (__scanf__, 2, 0))) ;
482:
483:
484:
485:
486:
487:   extern int vscanf (const char *__restrict __format, __gnuc_va_list
__arg)
488:        __attribute__ ((__format__ (__scanf__, 1, 0))) ;
489:
490:
491:   extern int vsscanf (const char *__restrict __s,
492:         const char *__restrict __format, __gnuc_va_list __arg)
493:        __attribute__ ((__nothrow__ , __leaf__)) __attribute__
((__format__ (__scanf__, 2, 0)));
494:
```

```
495:
496:
497:
498:   extern int vfscanf (FILE *__restrict __s, const char *__restrict
__format, __gnuc_va_list __arg) __asm__ ("" "__isoc99_vfscanf")
499:
500:
501:
502:         __attribute__ ((__format__ (__scanf__, 2, 0))) ;
503:   extern int vscanf (const char *__restrict __format, __gnuc_va_list
__arg) __asm__ ("" "__isoc99_vscanf")
504:
505:         __attribute__ ((__format__ (__scanf__, 1, 0))) ;
506:   extern int vsscanf (const char *__restrict __s, const char *__restrict
__format, __gnuc_va_list __arg) __asm__ ("" "__isoc99_vsscanf") __attribute__
((__nothrow__ , __leaf__))
507:
508:
509:
510:         __attribute__ ((__format__ (__scanf__, 2, 0)));
511:   # 485 "/usr/include/stdio.h" 3 4
512:   extern int fgetc (FILE *__stream);
513:   extern int getc (FILE *__stream);
514:
515:
516:
517:
518:
519:   extern int getchar (void);
520:
521:
522:
523:
524:
525:
526:   extern int getc_unlocked (FILE *__stream);
527:   extern int getchar_unlocked (void);
528:   # 510 "/usr/include/stdio.h" 3 4
529:   extern int fgetc_unlocked (FILE *__stream);
530:   # 521 "/usr/include/stdio.h" 3 4
531:   extern int fputc (int __c, FILE *__stream);
532:   extern int putc (int __c, FILE *__stream);
533:
534:
535:
536:
537:
538:   extern int putchar (int __c);
539:   # 537 "/usr/include/stdio.h" 3 4
540:   extern int fputc_unlocked (int __c, FILE *__stream);
541:
542:
543:
544:
545:
546:
547:
548:   extern int putc_unlocked (int __c, FILE *__stream);
549:   extern int putchar_unlocked (int __c);
550:
551:
552:
553:
554:
555:
```

```
556:   extern int getw (FILE *__stream);
557:
558:
559:   extern int putw (int __w, FILE *__stream);
560:
561:
562:
563:
564:
565:
566:
567:   extern char *fgets (char *__restrict __s, int __n, FILE *__restrict
__stream)
568:        ;
569:   # 603 "/usr/include/stdio.h" 3 4
570:   extern __ssize_t __getdelim (char **__restrict __lineptr,
571:                                size_t *__restrict __n, int __delimiter,
572:                                FILE *__restrict __stream) ;
573:   extern __ssize_t getdelim (char **__restrict __lineptr,
574:                              size_t *__restrict __n, int __delimiter,
575:                              FILE *__restrict __stream) ;
576:
577:
578:
579:
580:
581:
582:
583:   extern __ssize_t getline (char **__restrict __lineptr,
584:                             size_t *__restrict __n,
585:                             FILE *__restrict __stream) ;
586:
587:
588:
589:
590:
591:
592:
593:   extern int fputs (const char *__restrict __s, FILE *__restrict
__stream);
594:
595:
596:
597:
598:
599:   extern int puts (const char *__s);
600:
601:
602:
603:
604:
605:
606:   extern int ungetc (int __c, FILE *__stream);
607:
608:
609:
610:
611:
612:
613:   extern size_t fread (void *__restrict __ptr, size_t __size,
614:          size_t __n, FILE *__restrict __stream) ;
615:
616:
617:
618:
```

```
619:   extern size_t fwrite (const void *__restrict __ptr, size_t __size,
620:           size_t __n, FILE *__restrict __s);
621:   # 673 "/usr/include/stdio.h" 3 4
622:   extern size_t fread_unlocked (void *__restrict __ptr, size_t __size,
623:            size_t __n, FILE *__restrict __stream) ;
624:   extern size_t fwrite_unlocked (const void *__restrict __ptr, size_t
__size,
625:            size_t __n, FILE *__restrict __stream);
626:
627:
628:
629:
630:
631:
632:
633:   extern int fseek (FILE *__stream, long int __off, int __whence);
634:
635:
636:
637:
638:   extern long int ftell (FILE *__stream) ;
639:
640:
641:
642:
643:   extern void rewind (FILE *__stream);
644:   # 707 "/usr/include/stdio.h" 3 4
645:   extern int fseeko (FILE *__stream, __off_t __off, int __whence);
646:
647:
648:
649:
650:   extern __off_t ftello (FILE *__stream) ;
651:   # 731 "/usr/include/stdio.h" 3 4
652:   extern int fgetpos (FILE *__restrict __stream, fpos_t *__restrict
__pos);
653:
654:
655:
656:
657:   extern int fsetpos (FILE *__stream, const fpos_t *__pos);
658:   # 757 "/usr/include/stdio.h" 3 4
659:   extern void clearerr (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__));
660:
661:   extern int feof (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__)) ;
662:
663:   extern int ferror (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__)) ;
664:
665:
666:
667:   extern void clearerr_unlocked (FILE *__stream) __attribute__
((__nothrow__ , __leaf__));
668:   extern int feof_unlocked (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__)) ;
669:   extern int ferror_unlocked (FILE *__stream) __attribute__
((__nothrow__ , __leaf__)) ;
670:
671:
672:
673:
674:
675:
```

```
676:
677:   extern void perror (const char *__s);
678:
679:
680:
681:
682:
683:   # 1 "/usr/include/x86_64-linux-gnu/bits/sys_errlist.h" 1 3 4
684:   # 26 "/usr/include/x86_64-linux-gnu/bits/sys_errlist.h" 3 4
685:   extern int sys_nerr;
686:   extern const char *const sys_errlist[];
687:   # 782 "/usr/include/stdio.h" 2 3 4
688:
689:
690:
691:
692:   extern int fileno (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__)) ;
693:
694:
695:
696:
697:   extern int fileno_unlocked (FILE *__stream) __attribute__
((__nothrow__ , __leaf__)) ;
698:   # 800 "/usr/include/stdio.h" 3 4
699:   extern FILE *popen (const char *__command, const char *__modes) ;
700:
701:
702:
703:
704:
705:   extern int pclose (FILE *__stream);
706:
707:
708:
709:
710:
711:   extern char *ctermid (char *__s) __attribute__ ((__nothrow__ ,
__leaf__));
712:   # 840 "/usr/include/stdio.h" 3 4
713:   extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__));
714:
715:
716:
717:   extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__)) ;
718:
719:
720:   extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__));
721:   # 858 "/usr/include/stdio.h" 3 4
722:   extern int __uflow (FILE *);
723:   extern int __overflow (FILE *, int);
724:   # 873 "/usr/include/stdio.h" 3 4
725:
726:   # 2 "hello_world.c" 2
727:
728:   # 2 "hello_world.c"
729:   int main(){
730:     printf("Hello, World!\n");
731:   }
```

Wow! That's a lot! What happened is that the entirety of the `stdio.h` header was pasted into `hello world.c` which includes a forward declaration of `printf` that satisfies `gcc` enough not to give a warning.

CS211: (Appendix) Command-line Arguments

Introduction

Oftentimes you'll want to write programs that take input directly from the user. One way to accomplish that is to take in arguments from the *command line*—i.e. directly from the terminal itself. It's actually rather simple to do so.

Taking Command-line Arguments

As part of the hosted environment specification the C programming language provides rules for taking in command line arguments. These come in the form of standard arguments for the `main` function. The syntax is as follows:

```
int main(int num_of_args, char* string_of_args[])
```

For example, if I want to define a function called `hello.c` which will say "Hello" to every name given in the command line, I could write the following:

```
1:   #include <stdio.h>
2:   int main(int argc, char* argv[]){ // argc and argv are conventional
names, but not required
3:     for(int i = 1; i < argc; i++){
4:       printf("Hello, %s!\n", argv[i]);
5:     }
6:   }
```

Then I can compile and run it with three arguments:

```
josephraskind@stargazer:/tmp/cli$ gcc args.c -o ./hello_args; ./hello_args
Marx Engels Lenin
Hello, Marx!
Hello, Engels!
Hello, Lenin!
```

One thing you'll have noticed is I started `i` with 1 instead of 0. Why? Well, that's because `argv[0]` is always set to the name of the executable as it is the first string on the command line.

Converting Command-line Arguments

Command line arguments are always strings (`char *` in C). This means you *must* convert them if you want to take in numbers to perform calculations. For example, if I want to write a program that calculates standard deviation:

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]){
  int size = argc - 1;
  double mean;
  double sum;
  double elements[size];
  for(int i = 1; i < argc; i++){
    elements[i-1] = atof(argv[i]);  //  Converting the string to a double
    sum += elements[i-1];
  }
  mean = sum / size;
  sum = 0;
  for(int i = 0; i < size; i++){
    sum += pow(elements[i] - mean, 2);
  }
  double std = sqrt(sum/size);
  printf("mean: %.2f\n std: %.2f\n", mean, std);
}

```

```
josephraskind@stargazer:/tmp/cli$ gcc std.c -lm -o std; ./std 1 2 3
mean: 2.00
 std: 0.82
```

Common functions for string conversions are listed below for reference (all are declared in `stdlib.h`):

- `atoi(str)`
  - String to int
- `atol(str)`
  - String to long
- `atof(str)`
  - String to double

(Appendix) Debugging with gdb

Introduction

A lot of beginner programmers do what's called "print debugging"—using print statements to debug their programs. Although it tends to be fine for tiny programs, it is often not enough to get at the heart of catastrophic errors in complicated programs. This is where the use of programs specifically made for debugging shines.

gdb

`gdb` is the GNU debugger which was made to pair with `gcc` in order to debug C programs. Whenever using `gdb` it's best to make sure your C programs are compiled with the `-g` flag which sets special debugging symbols in the binary that `gdb` can pick up on.

Below is an example C program which has a simple bug in it:

```c
#include <stdio.h>
#include <stdlib.h>

int sum_array(int *arr, int n){
    int total = 0;
    for(int i = 0; i <= n; i++){   // bug: should be i < n
        total += arr[i];
    }
    return total;
}

int main(){
    int arr[5] = {1, 2, 3, 4, 5};
    int result = sum_array(arr, 5);
    printf("Sum: %d\n", result);
    return 0;
}
```

Compiling and running it I get:

```
josephraskind@stargazer:/tmp/gdb$ gcc arr_bug.c -g -o ./arr_bug; ./arr_bug
Sum: 32779
```

Which is clearly not the `15` I would have expected. This tells me I should fire up the debugger. I can do so with a call to `gdb ./arr_bug`:

```
josephraskind@stargazer:/tmp/gdb$ gdb ./arr_bug
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./arr_bug...
(gdb)
```

I've now entered `gdb`. There are a handful of core `gdb` instructions that are often always helpful:

- `run`
  - Execute the program
- `break`
  - `fn break` breaks at the beginning of the `fn` function
- `next`
  - Advances one line at a time (bypasses functions)
- `step`
  - Advances one line at a time (enters functions)
- `info locals`
  - Prints out all local variables and their values

`gdb` let's you freeze a program and look at it step by step:

```
(gdb) b sum_array
Breakpoint 1 at 0x1169: file arr_bug.c, line 4.
(gdb) run
Starting program: /tmp/gdb/arr_bug

Breakpoint 1, sum_array (arr=0xf0b5ff, n=21845) at arr_bug.c:4
4       int sum_array(int *arr, int n){
(gdb)
```

Here I set a breakpoint at the entrance to the function `sum_array` and run the program. Once program execution hits `sum_array` it is paused. I can then step one line (after the function arguments are initialized) and look at the local variables and arguments:

```
Breakpoint 1, sum_array (arr=0xf0b5ff, n=21845) at arr_bug.c:4
4       int sum_array(int *arr, int n){
(gdb) step
5           int total = 0;
(gdb) info local
total = -8729
(gdb) info args
arr = 0x7fffffffdde0
n = 5
```

We see that `gdb` is now paused at the `int total = 0` statement. I use `info locals` to check the value of all local variables (here `total` has yet to be initialized). I then use `info args` to check that the arguments look right—everything seems to check out.

I can then keep using `next` to go from one line to the next. Eventually, we'd be able to see that `i` goes beyond the boundaries of `arr`:

```
6           for(int i = 0; i <= n; i++){   /* bug: should be i < n */
(gdb) info locals
i = 5
(gdb) print arr[i]
$1 = 32767
(gdb) print arr[5]
$2 = 32767
```

I can use the `print` function to evaluate expressions using the available symbols at that point in the runtime. We can see that the access out of the array boundaries is what is returning that large value. Now it is a simple process of exiting `gdb` with `quit` and fixing the code!

Note: Using `gdb` can get extremely complicated. For this class, most major issues can be solved with setting a breakpoint and stepping through a function line by line while examining local variables/global data structures. It is a great resource and should not be ignored when you are in trouble.

(Appendix) Hosted Environment

Introduction

At all times when we are compiling and running C programs in the class we are doing so within the confines of a Linux Operating System. This is not something that can be ignored! Within the context of the C runtime specification this is what is called the **hosted environment**, referring to the fact that the C code is not being run independant of an OS.

The Hosted Environment

The C specification demands that "the function called at program startup is named main…It shall be defined with a return type of int and with no parameters…or with two parameters…" (§ 5.1.2.3.2) This gives us the reason why we've kept writing `int main` over and over and over again! Similarly when exiting the function:

> *If the return type of the main function is a type compatible with int, a return from the initial call to the main function is equivalent to calling the exit function with the value returned by the main function as its argument; reaching the } that terminates the main function returns a value of 0. If the return type is not compatible with int, the termination status returned to the host environmentis unspecified.*