



C Programming

The Bare Minimum

Ron Wellman

License: CC BY-NC-SA 4.0

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

June 2020



Contents

	The Ugly Truth	i
0.1	About this Book	i
0.2	Why C	i
0.3	Following Along	ii
0.4	Who am I	ii
0.5	Images	iii

I

Part One

1	Getting Started	3
1.1	Basic Structure of a C Program	3
1.1.1	Basics 1	3
1.1.2	Basics 2	4

1.1.3	Basics 3	5
1.2	Compilation	7
1.2.1	gcc	8
1.2.2	Make	10
2	Data Types	15
2.1	Basic Types	15
2.1.1	Integers	16
2.1.2	Characters	20
2.1.3	Floats	22
2.2	Enum	24
2.3	Void	24
2.4	Operators	25
2.5	Arrays	28
2.6	Pointers	30
2.7	Dynamic Memory	33
2.7.1	Stack	33
2.7.2	Heap	36
3	Control Flow	41
3.1	If/Else	41
3.2	Switch	42
3.3	While	43
3.3.1	Do/While	43
3.4	For	43

3.5	Goto	44
4	Command Line Arguments	45
4.1	Manual Parsing	45
4.2	getopt	46

II

Part Two

5	Structures	53
6	Working with Files	61
6.1	Opening a File	61
6.2	Reading Text	62
6.3	Reading and Writing Binary Data	64
6.4	File Stats	68
7	Concurrency	73
7.1	Forking	73
7.2	Multithreading	76
7.2.1	Threads.h	78
7.2.2	Mutex	79
7.2.3	Condition Variables	80
8	Networking	87
8.1	getaddrinfo	91
8.2	sigaction	95

9	Data Structures	99
9.1	Linked-List	99
9.1.1	createList	103
9.1.2	destroyList	103
9.1.3	addNode	104
9.1.4	removeNode	104
9.1.5	printList	104
9.1.6	Testing the Linked-List	104
9.2	Queue	107
9.2.1	createQueue	110
9.2.2	destroyQueue	110
9.2.3	enqueue	110
9.2.4	dequeue	111
9.2.5	Testing the Queue	111
9.3	Stack	114
9.4	Binary Search Tree	118
9.5	HashTable	118
	Bibliography	119
	Websites	119
	Books	119
	Index	121



The Ugly Truth

0.1 About this Book

When I set out to write this book, we as a development section were struggling to certify our developers. For one reason or another, we watched multiple failed attempts at a certification exam that we are required to pass. It is our hope that providing extra materials, guidance, and hands-on exercises, we'll be able to reverse this trend. I don't expect this book to be all encompassing; you won't go from "Zero to Hero". In some cases, at least some prior programming knowledge will be essential. However, where I can, I'll attempt to be thorough without being overly verbose and guide you toward a deeper level of knowledge within the C programming language.

I completely expect this book to have errors. Writing it also serves as a forcing function for me to read and experiment more. If you do find errors, please be kind and send me a message or submit your fixes in a pull request. Some of the code will frankly look a bit odd. I tried to make sure it would all fit on the screen nicely.

0.2 Why C

C is not my favorite language nor do I expect it to be yours. However, C has been around for a long time and many languages take their roots from it. Learning C will not only help you understand existing code bases, but will give you insights into and maybe even an appreciation for newer languages. At the end of the day though, C is a major portion of our required exam and we need to know it in order to pass.

0.3 Following Along

I have attempted to include sample programs wherever appropriate. I think its important for developers to read other developers code and to talk about it. These samples are included within the *src* directory of the repository. They get compiled every time I build this document to ensure what you see in the book is at least syntactically correct enough to compile.

The machine I have been building this on is a 64-bit machine running *PoP!_OS 20.04* which is based on *Ubuntu 20.04* and running *gcc 9.3.0*. The following compile time flags were used: `-std=c11 -Wall -Wextra -Wpedantic -Wwrite-strings -Wfloat-equal -Waggregate-return -Winline -Wvla`. During an early chapter, I compiled one of the sample programs as a 32 bit binary. In order to do this, I needed 32-bit libraries and therefore ran: `sudo apt install gcc-multilib`. As long as this happens, nearly any modern Linux based system should have no issues compiling and running all of the sample code.

If you are compiling this document as well, it is written using \LaTeX . There are a few extra packages you may require. Hopefully I've fully captured them but I used *apt* to install:

- `texstudio`
- `texworks`
- `texlive-latex-extra`
- `texlive-bibtex-extra`
- `biber`

Where appropriate, I'll also try and point out programs such as *valgrind* to look for memory leaks in your programs and *gdb* to assist you in troubleshooting your code. I highly recommend installing and using these programs as we go along. These should both be available via *apt*.

- `gdb`
- `valgrind`

Also, to ensure I had access to documentation on the various functions used throughout this book, I installed the following packages via *apt*:

- `glibc-doc`
- `glibc-doc-reference`

0.4 Who am I

I am not an expert in C and I don't claim to be. I first learned it in college roughly 15+ years ago and only recently picked it back up about two years ago. I knew just enough to pass our certification exam and as indicated earlier, I'm also writing this book to serve as a forcing function to progress my own knowledge. Additionally, I am far removed from the academic side of development so

expect me to slip up in my terminology from time to time. Attempts will be made to not totally screw you up either but I am only human. Again, **I am not an authority on the C language** and when conflict arrives from something else you've read, the other author is probably correct!

I'm in no way an accomplished author. I can barely speak English even though I've been doing so my whole life. Prepare yourself mentally for the worst grammar you've ever seen or layout for a book. When all else fails, focus on the points I'm attempting to make and not on how I'm presenting it. If all else fails, put in a pull request to help fix my deficiencies.

If you're reading this book after 2020, we are going through a Pandemic. Yay us! However, what this has given me is extra time to work on things like this. Hopefully at least something good comes from it.

ron.wellman01@gmail.com

https://github.com/ronwellman/bare_min_c.

0.5 Images


The pictures used throughout this book were primarily borrowed from the European Space Agency's and the US Department of State's Flickr pages.

<https://www.flickr.com/photos/europeanspaceagency/>

<https://www.flickr.com/photos/iip-photo-archive/>

Part One

1	Getting Started	3
1.1	Basic Structure of a C Program	
1.2	Compilation	
2	Data Types	15
2.1	Basic Types	
2.2	Enum	
2.3	Void	
2.4	Operators	
2.5	Arrays	
2.6	Pointers	
2.7	Dynamic Memory	
3	Control Flow	41
3.1	If/Else	
3.2	Switch	
3.3	While	
3.4	For	
3.5	Goto	
4	Command Line Arguments	45
4.1	Manual Parsing	
4.2	getopt	



1. Getting Started

© ESA/DLR/FU Berlin (G. Neukum)

1.1 Basic Structure of a C Program

1.1.1 Basics 1

C is a general purpose language and a rather simple language by today's standards. This does not mean it's easy to write good C code but rather that the language itself is small. C is also a typed language, meaning you as the programmer have to tell the compiler how to interpret the data flowing through your program. We do this by assigning types to the data. We'll talk all about types in due time. Additionally, C is a compiled language. This means, we write our source code in a human readable syntax and then a compiler converts that into machine-code ones and zeros our computer can understand. Lets take a minute or two to discuss how a typical C program is laid out. Don't worry too much about each and every command. The point here is to see how to construct our program so that we can later compile and run it. A C source file ends in the ".c" file extension.

```
1 #include <stdio.h>
2
3 int
4 main() {
5     printf("I can output to the screen!\n");
6     return 0;
7 }
```

Listing 1.1: src/01-basics1.c

Line 1: We import the *stdio.h* **header** file. The *stdio.h* header file is part of the *standard library*

which are essentially pre-written functions that are available for our use. This header file in particular informs the compiler about the existence of the *printf* function.

Line 3: We declare the *main* function returns an integer.

Line 4: The *main* function serves as the entry point into our program and must exist in a C program. In this example, nothing has been placed between the parentheses which means this function takes no arguments. The function body begins at the opening curly brace "{" on line 4 and ends at the closing curly brace "}" on line 7.

Line 6: A *return* of 0 indicates successful completion of our program. A *return* of anything else, indicates there was an error.

1.1.2 Basics 2

In the first example, we used the *printf* function which was part the *standard library*. In the next example, we're going to write our own function. Lets take a look.

```
1 #include <stdio.h>
2
3 static void
4 anotherFunc(const char *);
5
6 int
7 main() {
8     char msg[16] = {"Say something"};
9     anotherFunc(msg);
10    return 0;
11 }
12
13 static void
14 anotherFunc(const char *saying) {
15     printf("Received: %s\n", saying);
16 }
```

Listing 1.2: src/01-basics2.c

Line 3: Here we declare that *anotherFunc* does not have a return value by using the **void** keyword. Additionally, we use the **static** keyword. The K&R manual [4] indicates declaring a function *static* limits the scope of the function. In other words, it doesn't exist outside of this source file and can therefore not be imported by another program. It is good programming practice to limit the scope of your functions and variables to just where they are needed.

Line 4: This is a function **declaration**. Notice it ends with a semicolon. It exists here because of line 9. On line 9, *anotherFunc* is called but at this point, since the compiler is reading the file from top to bottom, it doesn't know about the *anotherFunc* function to include what arguments it requires and what it returns. We therefore provide a function *declaration* to make the compiler

aware of this function, its parameters (arguments), and its return type. Even if it doesn't know where the function exists, it now knows enough information to know how the function should be called. We could have written lines 13 thru 16 above our *main* function. This means that we would not have required the function *declaration*. However, this example follows the convention of trying to keep the *main* function as close to the top as possible.

Line 4: Also included in this line is the keyword **const**. I know we haven't discussed variables yet but essentially the *const* keyword indicates the value of the *msg* variable is "constant" and cannot be changed. This gives the programmer some assurance that the value will not be inadvertently changed inside of *anotherFunc*.

Line 14: Earlier on line 4 we saw a function *declaration*. This line is a function **definition**. It must be laid out in the same way as the *declaration*. This is where the function body is defined and really comes to life.

1.1.3 Basics 3

So we've seen how we can *include* code from the *standard library* and we've seen how we can write our own functions. Now we'll take a look at building our own header. In this way, we can define variables and functions written elsewhere and *include* them in our program. Why would we want to do this? One reason may be to keep our files as concise as possible. In order to do this, we may logically group some of our functions together rather than have them scattered across one very large file.

While we're at it, we also want to ensure we are following the **DRY** principle. **Don't Repeat Yourself** means that there should ideally be a single function that performs a specific action. In this way, we are not replicating the same capability multiple times throughout our code. If we've been adhering to the *DRY* principle and a change is required to that capability, we only need to make the change in one place rather than in multiple places.

A header file assists us in keeping our files concise as well as adhering to the *DRY* principle by allowing us to share functions across our program. C header files end with the ".h" file extension.

The following example consists of three files:

1. *01-helper.h* - Contains various *definitions* and *declarations*.
2. *01-basics3.c* - Contains our main function.
3. *01-helper.c* - Contains a single function *definition*.

```
1 #ifndef HELPER_H
2 #define HELPER_H
3
4 typedef enum {
5     FRED,
6     DAPHNE,
```

```

7   VELMA,
8   SHAGGY,
9   SCOOBY
10  } mysteryMember_t;
11
12  int distributeSnacks(mysteryMember_t);
13
14  #endif

```

Listing 1.3: src/01-helper.h

01-helper.h - Lines 1, 2, and 13: These lines form a **header guard**. A *header guard* is a *macro* that ensures that a header file is only pulled in a single time even if *included* into multiple source files. Otherwise, you may see multiple *definitions* and *declarations* for the same functions and variables strewn across a single program. By convention, the name of the *header guard* will be the all caps version of the filename with an underscore instead of a period. In my case, *macro* names must start with letters and therefore *01-helper.h* was shortened to *HELPER_H*.

01-helper.h - Lines 4 - 10: This provides the *declaration* and the *definition* for a new type called *mysteryMember_t*[1].

01-helper.h - Line 12: This provides the *declaration* for the *distributeSnacks* function.

```

1  #include <stdio.h>
2  #include "01-helper.h"
3
4  int
5  main() {
6      mysteryMember_t member = SHAGGY;
7
8      printf("I get %d scooby snacks!\n", distributeSnacks(member));
9      return 0;
10 }

```

Listing 1.4: src/01-basics3.c

01-basics3.c - Line 2: On this line, notice the import uses quotation marks around the header file. This instructs the compiler to look locally for *01-helper.h* rather than in the standard library (typically found in */usr/include/*). Without this file, lines 6 and 8 would cause compiler errors.

01-basics3.c - Line 6: The type *mysteryMember_t* *definition* is in the *01-helper.h* header file. Additionally, the *enum* value assigned to *SHAGGY* comes from *01-helper.h* as well.

01-basics3.c - Line 8: The function *distributeSnacks* *declaration* is also in the *01-helper.h* header file.

```

1  #include "01-helper.h"
2
3  int

```

```
4 distributeSnacks(mysteryMember_t member) {  
5     int num = 0;  
6  
7     switch (member) {  
8         case SHAGGY:  
9             num = 20;  
10            break;  
11            case SCOOPY:  
12                num = 40;  
13                break;  
14            default:  
15                num = 0;  
16        }  
17  
18        return num;  
19    }
```

Listing 1.5: src/01-helper.c

01-helper.c - Line 1: Again, we see the local *01-helper.h* header file being imported. Without this, *01-helper.c* would not know about the *mysteryMember_t* type that is on line 4 or the enum values on lines 8 and 11.

01-helper.c - Line 4: This is the function *definition* for *distributeSnacks* that defines what this function does.

We have three files that will all get compiled together into a single executable. The *01-helper.h* header file is what makes all of this possible. In the next section, we'll talk about what the compiler is, what it does, and how to use it.

1.2 Compilation

At a very high level, **compiling** a program is the act that takes raw source code into machine code that can be executed. A common compiler, and the one I'll be using in this book, is the *GCC* compiler. It is available on nearly every UNIX based system. You can invoke it directly and in the following examples we will do so. Later on we'll see how we can leverage the *make* utility and *Makefiles* to make this easier and repeatable.

There are distinct steps that a compiler goes through in order to produce an executable. These steps generally fall into four phases: *preprocessing*, *compiling*, *assembling*, and *linking*[\[5\]](#).

Preprocessing

During this phase, the source code is cleaned up with consistent new lines and line continuations (the slash) are removed. Comments are also removed as they are not needed. The source code

is further modified to expand out *macros* and *preprocessor directives*. For example, the *header guard* we saw earlier must be processed accordingly. The *preprocessor* also pulls in any header files specified with the *#include* directive. The output from this phase is just modified source code. This source code can be output by passing the *-E* option to *gcc*.

Compiling

At this point, the source code is ready to be translated into something closer to what the machine can understand. In this phase the source code is translated into assembly code. Assembly code is still human readable but is architecture specific. This assembly code can be output by *gcc* by passing the *-S* option.

Assembling

Since assembly code is architecture specific, the assembly code is now passed to an *assembler*. The results of this phase is an object file ending with a ".o" file extension. This object file contains the machine code of the program that was just compiled minus any external dependencies that were included.

Linking

Now that all source files have been compiled into object files, it is the job of the *linker* to link them together. For example, because we used the *printf* function from the *standard library*, the *libc* static library will be automatically linked in. However, for other libraries, this process is not automatic. For example, as we'll see later in this book, when we begin using *pthread*s, we have to tell the linker to link to the *libpthread* library by passing *gcc* the *-pthread* option or by adding *-pthread* to the *LDLIBS* environment variable.

1.2.1 gcc

```
$ ls
01-basics1.c

$ gcc 01-basics1.c

$ ls -laF
...
-rw-rw-r-- 1 ron ron    88 Jun 26 10:30 01-basics1.c
-rwxrwxr-x 1 ron ron 16696 Jun 26 12:22 a.out*

$ ./a.out
I can output to the screen!
```

On our Linux terminal, we run *gcc* giving it our source code and it generates an executable called

a.out. When we run it, we get the output "I can output to the screen!" to our terminal. *a.out* is the default name given to the executable because we didn't tell the gcc compiler what name we wanted. To do this, we use the *-o* option.

```
$ gcc -o 01-basics1 01-basics1.c

$ ls -laF
...
-rwxrwxr-x 1 ron ron 16696 Jun 26 12:34 01-basics1*
-rw-rw-r-- 1 ron ron   88 Jun 26 10:30 01-basics1.c

$ ./01-basics1
I can output to the screen!
```

Up until now we've seen a single file being compiled with *gcc* but in our Basic3 example, there were three files. To compile that, we just need to specify the additional files.

```
$ gcc -o 01-basics3 01-basics3.c 01-helper.h 01-helper.c

$ ls -laF
...
-rwxrwxr-x 1 ron ron 16776 Jun 26 13:08 01-basics3*
-rw-rw-r-- 1 ron ron  166 Jun 26 10:30 01-basics3.c
-rw-rw-r-- 1 ron ron  221 Jun 26 10:30 01-helper.c
-rw-rw-r-- 1 ron ron  160 Jun 26 10:30 01-helper.h
...

$ ./01-basics3
I get 20 scooby snacks!
```

So now we can compile our source code into an executable with the name of our choosing but still we're missing out on a bunch of checking we could have the compiler perform.

```
$ gcc -std=c11 -Wall -Wextra -Wpedantic -Wwrite-strings \
> -Wfloat-equal -Waggregate-return -Winline -Wvla \
> -o 01-basics1 01-basics1.c

$ ls -laF
...
-rwxrwxr-x 1 ron ron 16696 Jun 26 12:40 01-basics1*
-rw-rw-r-- 1 ron ron   88 Jun 26 10:30 01-basics1.c

$ ./01-basics1
I can output to the screen!
```

In this example, we compiled with the following flags:

- *-std=c11* - The 2011 ISO C++ standard plus amendments.
- *-Wall* - This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
- *-Wextra* - This enables some extra warning flags that are not enabled by *-Wall*.
- *-Wpedantic* - Issue all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any *-std* option used.
- *-Wwrite-strings* - When compiling C, give string constants the type "const char[length]" so that copying the address of one into a non-"const" "char *" pointer produces a warning. These warnings help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using "const" in declarations and prototypes.
- *-Wfloat-equal* - Warn if floating-point values are used in equality comparisons.
- *-Waggregate-return* - Warn if any functions that return structures or unions are defined or called.
- *-Winline* - Warn if a function that is declared as inline cannot be inlined.
- *-Wvla* - Warn if a variable-length array is used in the code.

The above options or flags are a small sample of what can be chosen. However, they represent common issues with compiling C code and therefore are the flags I used in compiling the examples for this book. The descriptions were pulled from referencing the *gcc* manual page by running "man gcc".

Additional information can be found on the GNU website at <https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html#Option-Summary>.

1.2.2 Make

When you're compiling often or compiling a large number of files, typing in the *gcc* command manually can be cumbersome and error prone. Luckily GNU has a utility called *make*. It makes compiling much easier.

```
$ make 01-basics3
gcc      01-basics3.c  -o 01-basics3

$ ls -laF
...
-rwxrwxr-x 1 ron ron 16696 Jun 26 14:28 01-basics1*
-rw-rw-r-- 1 ron ron   88 Jun 26 10:30 01-basics1.c
...

$ ./01-basics1
```


I can output to the screen!

This slightly shortens how much we need to type but to see the real power behind the *make* utility, you need to create a file called *Makefile*. Inside of that file, you can specify all sorts of things that will happen when you type *make* in the same directory as the *Makefile*. *Makefiles* are highly customizable. Below is a part of the example I use for compiling the code for this book./

```

1 # https://www.gnu.org/software/make/manual/make.html
2 # https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
3 CC= gcc
4 CFLAGS= -g -std=c11 -Wall -Wextra -Wpedantic -Wwrite-strings
5 CFLAGS+= -Wfloat-equal -Waggregate-return -Winline -Wvla
6 LDLIBS= -pthread
7
8 # grab all files ending in c
9 SRCS := $(wildcard *.c)
10
11 # substitute all files ending with c with just their filename
12 BINS := $(SRCS:%.c=%)
13
14 # all and clean do not refer to actual files
15 .PHONY: all clean
16
17 # compile all to binaries
18 all: 01-basics1 01-basics2 01-basics3 02-integer-32bit \
19     02-integer-64bit 02-character1 02-float1 \
20     02-xor 02-array1 02-array2 02-pointers1 \
21     02-dynamic1 03-if 04-cli1 04-cli2 \
22     05-struct1 06-file1 06-file2 \
23     06-file3 07-concurrency1 07-concurrency2 \
24     07-concurrency3 08-network1_server \
25     09-queue 09-11 09-stack
26
27
28 01-basics3: 01-helper.c 01-helper.h
29
30 05-struct1: 05-player.c 05-player.h
31
32 02-integer-32bit:
33     ${CC} ${CFLAGS} -m32 -o 02-integer-32bit 02-integer-32bit.c
34
35 07-concurrency2: 07-concurrency2.c 09-queue.h 09-queue.c
36
37 07-concurrency3: 07-concurrency3.c 09-queue.h 09-queue.c
38
39 09-queue: 09-queue_test.c 09-queue.c 09-queue.h
40
41 09-11: 09-11_test.c 09-11.c 09-11.h
42

```

```

43 09-stack: 09-stack_test.c 09-stack.c 09-stack.h
44
45 # remove all binaries verbosely
46 clean:
47     rm -rvf *.o ${BINS}

```

Listing 1.6: src/Makefile

Line 3: This line indicates we want to use the *gcc* compiler.

Lines 4 and 5: These lines set our compile time flags. Every time we compile, these are the flags that will be used to reduce mistakes on forgetting to add one or mistyping another.

Lines 8 and 11: These wildcards identify all is all of our ".c" files and *BINS* is the name of the resulting binaries.

Line 14: Identifies the targets below that do not refer to actual filenames.

all: By convention, the first target is *all*. Since its the first, it will build each target specified. By listing my three programs there, the matching ".c" files will be build by *make* as you see below.

01-basics3: Since *01-basics3* required two additional files (prerequisites) to build, those files are specified. This takes the form of:

```

target: prerequisites
<TAB> recipe

```

clean: Here we define a target called *clean* and specify the recipe is to run *rm -rvf* (remove verbosely) against all local object files (files ending in ".o") as well as the compiled binaries.

```

$ make
gcc -std=c11 -Wall -Wextra -Wpedantic -Wwrite-strings \
-Wfloat-equal -Waggregate-return -Winline -Wvla \
01-basics1.c -o 01-basics1

gcc -std=c11 -Wall -Wextra -Wpedantic -Wwrite-strings \
-Wfloat-equal -Waggregate-return -Winline -Wvla \
01-basics2.c -o 01-basics2

gcc -std=c11 -Wall -Wextra -Wpedantic -Wwrite-strings \
-Wfloat-equal -Waggregate-return -Winline -Wvla \
01-basics3.c 01-helper.c 01-helper.h -o 01-basics3

$ ls -laF
...
-rwxrwxr-x 1 ron ron 16696 Jun 26 15:08 01-basics1*
-rw-rw-r-- 1 ron ron    88 Jun 26 10:30 01-basics1.c
-rwxrwxr-x 1 ron ron 16784 Jun 26 15:08 01-basics2*
-rw-rw-r-- 1 ron ron   224 Jun 26 10:30 01-basics2.c
-rwxrwxr-x 1 ron ron 16776 Jun 26 15:08 01-basics3*
-rw-rw-r-- 1 ron ron   166 Jun 26 10:30 01-basics3.c

```

```
-rw-rw-r-- 1 ron ron 221 Jun 26 10:30 01-helper.c
-rw-rw-r-- 1 ron ron 160 Jun 26 10:30 01-helper.h
-rw-rw-r-- 1 ron ron 595 Jun 26 14:59 Makefile

$ make
make: Nothing to be done for 'all'.

$ make clean
rm -rvf *.o 01-basics1 01-basics2 01-helper 01-basics3
removed '01-basics1'
removed '01-basics2'
removed '01-basics3'
```

You can see by running *make* a single time, it compiles all of our files with the appropriate compile time flags thereby eliminating a lot of typing and the possibility of missing a flag. Additionally, if we run *make* again, it already knows the binaries exist, and the source files have not changed, so there is no reason to recompile. Lastly we run *make clean* which automatically removes our binaries. There is quite a bit of other functionality that you can build into your *Makefiles*. I invite you to check out: <https://www.gnu.org/software/make/manual/make.html>



2. Data Types

2.1 Basic Types

Much of programming is about the receiving, manipulating, and sending of data. Since C is a typed language, it needs to know how to interpret the data that you are working with. In many cases, the underlying bits that are actually being stored are the same. However, based on the type assigned to that data, the C compiler will interpret it differently. C can't really tell the difference between 41 and the letter 'A' (which has the ASCII value of 41). It's up to you as the programmer to specify if the data is an integer or if it's a character.

Below are a few of the basic data types you'll encounter in C.

- **int** - Represents whole numbers.
- **char** - Represents a single character.
- **float** - Represents numbers with decimal points (single precision).
- **double** - Represents numbers with decimal points (double precision).

The platform that a program is compiled for can affect the number of bytes that a data type requires. While most of today's platforms are 64-bit, there are still a few 32-bit systems out there. If the max or min size of a data type is a concern to you, you'll want to be sure you know the size of that type on the platform your code will be compiled for.

2.1.1 Integers

By default, an *int* utilizes 4 bytes on both the 32-bit and 64-bit platforms. However, there are a couple of qualifiers that can affect how the data is interpreted as well as how many bytes are used.

- **short** - Utilizes 2 bytes (16 bits).
- **long** - Utilizes 4 bytes on a 32 bit platform and 8 bytes on a 64 bit platform.
- **long long**- Utilizes 8 bytes on a 32 bit platform and 8 bytes on a 64 bit platform.
- **signed** - Integers are *signed* by default meaning they utilize two's complement to represent positive and negative numbers. You do not need to specify *signed* but can for greater clarity for someone reading your code.
- **unsigned** - As an *unsigned* integer, there is no sign bit. Therefore you can store larger positive numbers but you cannot store negative numbers.

```

1 #include <stdio.h>
2
3 int
4 main() {
5     short int si;
6     int i;
7     unsigned int ui;
8     long int li;
9     long long int lli;
10
11     si = -275;
12
13     printf("Number of Bytes:\nshort int: %d\nint: %d\n",
14         sizeof(si), sizeof(i));
15
16     printf("long int: %d\nlong long int: %d\n",
17         sizeof(li), sizeof(lli));
18
19     printf("\n");
20     printf("Value of signed short: %d\n", si);
21     printf("Value of unsigned short: %u\n", (unsigned short)si);
22
23     printf("\n");
24     i = 0x80000000; // 0b10000000000000000000000000000000
25     printf("Min signed int: %d\n", i);
26     i = 0x7fffffff; // 0b01111111111111111111111111111111
27     printf("Max signed int: %d\n", i);
28
29     printf("\n");
30     ui = 0; // 0b00000000000000000000000000000000
31     printf("Min unsigned int: %u\n", ui);
32     ui = 0xffffffff; // 0b11111111111111111111111111111111
33     printf("Max unsigned int: %u\n", ui);
34

```



```

35     return 0;
36 }

```

Listing 2.1: src/02-integer-32bit.c

```

$ file 02-integer-32bit
02-integer-32bit: ELF 32-bit LSB shared object, Intel 80386

$ ./02-integer-32bit
Number of Bytes:
short int: 2
int: 4
long int: 4
long long int: 8

Value of signed short: -275
Value of unsigned short: 65261

Min signed int: -2147483648
Max signed int: 2147483647

Min unsigned int: 0
Max unsigned int: 4294967295

```

As you can see with the *file* command, 02-integer-32bit has been compiled as a 32-bit binary. Notice that an *int* and *long int* are both 4 bytes but a *long long int* is 8 bytes.

Also notice on line 11 that I assign the value of -275 to *si* which is a *signed short int* and it prints out just fine on line 14. However, when I print it out a second time on line 15 I cast it as an *unsigned short int*. Under the hood, the bits haven't changed; how they were interpreted did. -275 is actually 111111011101101 in binary. The left-most bit (most significant bit) is the *sign* bit. When cast as an *unsigned short int*, this bit is not longer interpreted as a signed bit which is why we now get a positive number. I point this out only because it is important to note that you as the programmer need to understand the data you are working with because C only pays attention to whether or not you declared it as *signed* or *unsigned*. To further demonstrate this, I've printed out the max and min value that a signed integer and an unsigned integer can hold.

```

1  #include <stdio.h>
2
3  int
4  main() {
5      short int si;
6      int i;
7      long int li;
8      long long int lli;
9
10     printf("Number of Bytes:\nshort int: %ld\nint: %ld\n",
11           sizeof(si), sizeof(i));

```

```

12  printf("long int: %ld\nlong long int: %ld\n",
13         sizeof(li), sizeof(lli));
14
15  return 0;
16  }

```

Listing 2.2: src/02-integer-64bit.c

```

$ file 02-integer-64bit
02-integer-64bit: ELF 64-bit LSB shared object, x86-64

$ ./02-integer-64bit
Number of Bytes:
short int: 2
int: 4
long int: 8
long long int: 8

```

Notice this 02-integer-64bit is compiled as a 64-bit executable. In this case, an *int* is still 4 bytes but a *long int* is now 8 bytes allowing you to store much larger numbers.

stdint.h

The *stdint.h* header imports other header files such as *stdint-intn.h* and *stdint-uintn.h* that specify a number of macros and type definitions (*typedef*) to make working with integers more consistent across 32-bit and 64-bit platforms. By using it, you can guarantee the size of the resulting integer.

- **uint8_t, uint16_t, uint32_t, uint64_t** - All unsigned integers that utilize 8-bit, 16-bit, 32-bit, and 64-bit accordingly.
- **int8_t, int16_t, int32_t, int64_t** - All signed integers that utilize 8-bit, 16-bit, 32-bit, and 64-bit accordingly.

Due to the inconsistent sizes for integers we identified between 32-bit and 64-bit platforms, you'll often see code that uses the *stdint.h* header file and specifies the actual size that is needed.

Additionally, *stdint.h* also specifies macros for the MAX and MIN sizes of various integers. You may decide to use these in your programs to test for an overflow of an integer.

```

/* Minimum of signed integral types. */
# define INT8_MIN          (-128)
# define INT16_MIN         (-32767-1)
# define INT32_MIN         (-2147483647-1)
# define INT64_MIN         (-__INT64_C(9223372036854775807)-1)
/* Maximum of signed integral types. */
# define INT8_MAX          (127)

```

```
# define INT16_MAX          (32767)
# define INT32_MAX          (2147483647)
# define INT64_MAX          (__INT64_C(9223372036854775807))

/* Maximum of unsigned integral types.  */
# define UINT8_MAX          (255)
# define UINT16_MAX         (65535)
# define UINT32_MAX         (4294967295U)
# define UINT64_MAX         (__UINT64_C(18446744073709551615))
```

stddef.h

This header file defines a number of additional types as well as macros.

One type that you will see quite frequently is *size_t*. This is an *unsigned integer* that is often returned when the result is the number of bytes of something. For example, the manpage for *strlen* function shows:

SYNOPSIS

```
#include <string.h>
```

```
size_t strlen(const char *s);
```

DESCRIPTION

The `strlen()` function calculates the length of the string pointed to by `s`, excluding the terminating null byte (`'\0'`).

RETURN VALUE

The `strlen()` function returns the number of bytes in the string pointed to by `s`.

So the *strlen* function returns the number of bytes in the string as type *size_t* which under the hood, is really just an *unsigned int*.

stdbool.h

Languages other than C, often have a type for boolean variables. These are variables that can only hold the value of *true* or *false*. C accomplishes this with a set of macros which under the hood are really just integer constants.

- **true** - constant 1
- **false** - constant 0

You may see these returned from functions to indicate whether something was successful or not. As a programmer using type *bool* as a return may make it easy to ascertain the reason behind what a function is returning. However, be cautious about using them to test the return from standard library functions. In many cases, a success is returned as a zero but a failure is returned as a non-zero number. It is always a good idea to read the manual page of a function to understand what it uses for return types and what the return values actually mean.

2.1.2 Characters

A *char* within C typically requires just a single byte to store a single character. This is based upon the fact that the *char* was meant to store the ASCII value which only required 7 bits. Depending upon the type of project you are working on, this may or may not be sufficient. Below is the ASCII chart 2.1 as printed out by running the *ascii* command on the terminal. Notice the values only go upto 127.

Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec	Hex								
0	00	NUL	16	10	DLE	32	20		48	30	0	64	40	@	80	50	P	96	60	~	112	70	p
1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
4	04	EOT	20	14	DC4	36	24	\$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
7	07	BEL	23	17	ETB	39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
8	08	BS	24	18	CAN	40	28	(56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
9	09	HT	25	19	EM	41	29)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[107	6B	k	123	7B	{
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D]	109	6D	m	125	7D	}
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

Figure 2.1: ASCII Chart

Obviously the ASCII chart was built for the English language. When not sufficient, there are ways of using an *unsigned int* and *UTF-8* encoding instead. Additionally, there are wider character data types such as *wchar_t* that utilize more than 1 byte in order to store character values by their Unicode value rather than an ASCII value. I will not be going into these methods and alternate data types but if it interests you, I suggest checking out articles such as <https://www.cprogramming.com/tutorial/unicode.html>.

```

1 #include <stdio.h>
2 #include <ctype.h>
3
4 int
5 main() {
6
7     char c1 = 'a';
8     char c2 = toupper(c1);

```



```

9
10 printf("The character '%c' has the value %d\n", c1, c1);
11 printf("The character '%c' has the value %d\n", c2, c2);
12
13 printf("\n");
14 if (0 != isalpha(c1)) {
15     printf("%c - alphabetic character.\n", c1);
16 }
17 else {
18     printf("%c - not an alphabetic character.\n", c1);
19 }
20
21 printf("%c - %s character.\n", c1,
22        isdigit(c1) ? "numeric" : "not a numeric");
23
24 return 0;
25 }

```

Listing 2.3: src/02-character1.c

Line 2: Here I import *ctype.h*. Doing so gives me access to a number of functions and macros pertaining to the use of characters.

Line 7: Here I declare a *char* and give it the value of *a*. Notice it holds a single character and I use single quotes.

Line 8: Here I use the function *toupper* which has a function *declaration* of **int toupper(int c);**. This is our first indication that under the hood, a *char* is treated very much like an *int*.

Line 14: Here I use the function *isalpha* to test to see if the character is alphabetic.

Line 22: Here I use the function *isdigit* to test to see if the character is a digit.

By typing 'man 3 toupper' on the terminal, I am presented with the manual page describing the C *toupper* function:

```

TOUPPER(3)
NAME
    toupper, tolower, toupper_l, tolower_l - convert uppercase or lowercase
SYNOPSIS
    #include <ctype.h>

    int toupper(int c);
    int tolower(int c);

```

Figure 2.2: toupper

Likewise, I can view the manpage for *isdigit* by typing 'man 3 isdigit'.

```

SYNOPSIS
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);

int isascii(int c);
int isblank(int c);

```

Figure 2.3: isalpha, isdigit

Character Constants

A character constant is any character surrounded by single quotes. For example, when we assigned 'a' to *c1* in the previous example, that was an example of a character constant. However, if we were using wider characters other than the standard *char*, we may want to pass in the Unicode value. To do this we may specify *u'*`\x3b3'`[\[5\]](#).

The `\` we just saw with `\x` known as an escape sequence. There are a number of other escape sequences you may encounter. You've already seen me use `'\n'` in nearly every *printf*. This is a character constant for the newline character. Some of the other character constants you may encounter are:

- `\0` - The NULL character. You'll often see this as terminating a character array.
- `\n` - New line.
- `\r` - Carriage return.
- `\t` - Horizontal tab.
- `\v` - Vertical tab.
- `\o` - Octal values. This will look something like `\o456` where we are referring to octal value 456 which is 302 in decimal.
- `\x` - Hexidecimal values. This will look something like `\x456` where we are referring to hexadecimal 456 which is 1110 in decimal.
- `\u` - Unicode values.

2.1.3 Floats

Floating point numbers are numbers with decimal points. Just as with integers, the data type affects the number of bytes used to store values. With integers, concern was with max and min values as well as overflows. With floats, its about precision. A floating point value is an approximation and

the cost of being more precise in that approximation is in the number of bytes that are required to store it.

Below is a chart directly taken from the book "C in a Nutshell"[5].

Type	Storage Size	Value Range	Smallest positive Value	Precision
float	4 bytes	$\pm 3.4\epsilon+38$	$1.2\epsilon-38$	6 digits
double	8 bytes	$\pm 1.7\epsilon+308$	$2.3\epsilon-308$	15 digits
long double	10 bytes	$\pm 1.1\epsilon+4932$	$3.4\epsilon-4932$	19 digits

Table 2.1: Real floating-point types

```

1 #include <stdio.h>
2
3 int
4 main() {
5     float x;
6
7     x = 1 / 3;
8     printf("1 / 3 = %f\n", x);
9
10    x = (float)1 / 3;
11    printf("1 / 3 = %f\n", x);
12
13    return 0;
14 }
```

Listing 2.4: src/02–float1.c

```

$ ./02 – float1
1 / 3 = 0.000000
1 / 3 = 0.333333
```

Line 7: On this line, 1 is divided by 3 and the results are stored in x . However, when x is printed out on line 8, we see that what actually got stored is 0.0. This is because we divided one integer by another integer. In C, this will result in another integer. Since integers cannot have decimal points, 0 gets stored in x .

Line 10: Again we do the same division but instead of dividing two integers, we first cast 1 as a *float*. Under the hood, this operation should result in a *double* being generated but because we are storing it in a *float*, it is rounded to 6 digits of precision. We can see this when we print x on line 11 in that there are only 6 digits (leading and trailing zeros are ignored).

There is another header file called *complex.h* that allows for the use of complex floating point types. I will not be discussing them here.

2.2 Enum

An *enum* is a data type that consists of named values that correspond to integer constants. Often times it makes sense to use an *enum* rather than a constant to make it clearer. In the previous chapter, we built an *enum* in our header file to represent Mystery Inc members.

```
1 #ifndef HELPER_H
2 #define HELPER_H
3
4 typedef enum {
5     FRED,
6     DAPHNE,
7     VELMA,
8     SHAGGY,
9     SCOOPY
10 } mysteryMember_t;
11
12 int distributeSnacks (mysteryMember_t);
13
14 #endif
```

Listing 2.5: src/01-helper.h

By default, FRED would correspond to the value of 0, DAPHNE to 1, VELMA TO 2, SHAGGY to 3, and SCOOPY to 4. However, I could have specified different values.

```
enum temperature {
    FAHRENHEIT = 1,
    CELCIUS = 2,
}
```

Depending upon the coding standard you are following, it is often customary to pair up an *enum* definition with a type declaration *typedef*[1] and to name the resulting type with a trailing "_t" to signify it is a custom type.

2.3 Void

In chapter 1, I mentioned a function can indicate it returns no values by using the *void* keyword. However, *void* can be a type as well. C uses them with pointers all the time. We'll discuss pointers shortly but for now, a *void* pointer holds a memory address but does not indicate the data type that resides at the address. This can be useful in making certain types of functions or data types generic to the data that they store or manipulate at the cost of additional code complexity.

2.4 Operators

The following operators are some of the most common you'll see when writing C. I won't go into depth with them as you'll see most of them used in further examples.

Operator	Purpose	example
*	Multiplication	a * b
/	Division	a / b
+	Addition	a + b
-	Subtraction	a - b
%	Modulus (remainder)	a % b
++	Increment	++a or a++
--	Decrement	--a or a--
&	BITWISE AND	a & b
	BITWISE OR	a b
^	BITWISE XOR	a ^ b
~	BITWISE NOT	~a
«	Left Shift	a « 2
»	Right Shift	a » 2
=	Assignment	c = a + b
==	Equality	a == b
<	Less than	a < b
>	Greater than	a > b
<=	Less than or equal to	a <= b
>=	Greater than or equal to	a >= b
!	Negation (NOT)	!a
&&	Logical AND	a && b
	Logical OR	a b
&	Address of	&a
*	Indirection	*a
[]	Subscript	a[b]
.	Struct or Union member designator	a.b
->	Struct or Union member designator by Reference	a->b
sizeof	Storage size of object	sizeof(a)
?:	Conditional Evaluation	a ? b : c

Table 2.2: Operators

The Assignment '=' operator is often combined with other operators such as '+', '-', '/', '*' to shorten the amount of code the programmer has to type. For example, `a += 5` is equivalent to `a = a + 5`.

Just as in mathematical equations, certain operations are given precedence over other operations.

The same is true for C, certain operators have precedence over other operators. "C in a Nutshell"[5] has a good chart listing them out in Chapter 5. A small sampling of the chart shows that the postfix operators such as `[]`, `.`, and `->` have precedence over the unary operators `*`, `&`, `!`, and `~` which have precedence over multiplication operators `*`, `/`, and `%`. This list goes on and on for the remainder of the operators. It is good to generally understand the precedence but I don't suggest you rely on them. Instead, I suggest taking following the guidance given in the "Embedded C Coding Standard"[1]:

- a. Do not rely on C's operator precedence rules, as they may not be obvious to those who maintain the code. To aid clarity, use parentheses (and / or break long statements into multiple lines of code) to ensure proper execution order within a sequence of operations.
- b. Unless it is a single identifier or constant, each operand of the logical AND (`&&`) and logical OR (`||`) operators shall be surrounded by parentheses.

Lets take a look at an example that uses a few of these operators. Don't worry if it doesn't all make sense yet.

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdbool.h>
4
5 bool
6 xor(char *original, size_t oLen, char *key, size_t kLen);
7
8 int
9 main() {
10     // some character array and its length
11     char clear[] = "A sentence is made of words. " \
12         "Words are made of characters.";
13     size_t clearLen = strlen(clear);
14
15     // some key and its length
16     char key[] = "X>P.X3/tMK{P\\:sr";
17     size_t keyLen = strlen(key);
18
19     printf("Original: %s\n", clear);
20
21     xor(clear, clearLen, key, keyLen);
22
23     printf("XOR'd: %s\n", clear);
24
25     xor(clear, clearLen, key, keyLen);
26
27     printf("Original: %s\n", clear);
28     return 0;
29 }
```

```
30
31 bool
32 xor(char *original, size_t oLen, char *key, size_t kLen) {
33
34     // quick check to see that we received a non-null array
35     if ((NULL == original) || (NULL == key) || (0 == oLen) || (0 == kLen)) {
36         return false;
37     }
38
39     // loop through each character xoring it with the key
40     for (size_t i = 0; i < oLen; i++) {
41         original[i] ^= key[i % kLen];
42     }
43
44     return true;
45 }
```

Listing 2.6: src/02-xor.c

Line 11: Here we assign a string constant to a *char* array. We utilize the subscript operators [] to designate that the variable *clear* is a *char* array. We'll talk about *arrays* in future sections. We also use the assignment operator = to assign the address of the string constant to the *char* array.

line 13: We run the function *stringlen* to get the length of the string stored in the *character array* and use the assignment operator = to store the result in *keyLen*.

Line 35: *Original* and *key* are pointers (which we haven't discussed thus far). We know this because the arguments use the indirection operator *. Pointers hold addresses so when we use the indirection operator, what we are getting back is the thing that the pointer points to. In this case, they should point to the first character of the string constants that got assigned to *clear* and *key* from earlier. However, reaching into a pointer without first checking to make sure they actually hold an address is dangerous. This is why we use the equality operator == to see if they point to *NULL*; a macro set to 0. In other words, if our program has been handling pointers correctly up to this point, they should contain either a valid address or 0. By testing for *NULL* at the beginning of our function, we can be assured that moving forward, the pointers actually point to something.

Line 40: This may be our first example utilizing a *for* loop. Don't worry, we'll talk about control flow in coming sections. For now, we use the assignment control operator = to set an initial value for variable *i*. We also utilize the less than operator < to make sure the value of *i* doesn't exceed the value of *oLen*. This would indicate we've passed the end of the *char* array. Additionally we use the increment operator ++ to increment the value of *i* after each loop.

Line 41: In this line we use the subscript operators [] to operate on a single *char* within the *char* array. In this case, *i* is an index value that starts at 0 (corresponding to the first character in the *char* array) and by incrementing by 1 after each loop, we'll XOR against each character. We then combine the XOR operator ^ with the assignment operator = to form ^=. This means that we will XOR the *char* on the left side with the *char* on the right side of this operator, and assign the result back to the same location as the original *char* on the left. So what character are we using on the right? Here we use the subscript operators [] once again to pull out a single *char* from

key. We utilize the modulus operator % with index *i* and *kLen* which is the length of the key. Since a modulus does remainder division, what we essentially get is a number between 0 and *kLen* - 1. This allows us to loop back over each character within *key* until we've gone all the way through each character in *original*.

2.5 Arrays

Arrays are objects that can contain multiple elements of the same type. All elements are stored in a contiguous block of memory. The number of elements is determined when the array is created and cannot be adjusted during its lifetime. Individual elements of the array can be accessed via an index value that starts at 0. This is why in the previous example, we started *i* with a value of 0 so that it would start at the beginning of the *char* array.

Most arrays sizes are determined by the programmer at compile time by passing in integer constants. You can however, have variable sized arrays. I tend not to use these and have compiler flags that will complain if the size of an array can't be determined at compile time because it is the result of an expression or user input. Instead I would use dynamic memory by building my array on the heap. We'll discuss dynamic memory in a later section.

In the XOR example we saw earlier, we saw a *char* array for *clear* and *key*. Those were single dimension arrays. However, you can add additional dimensions by adding additional subscript operators [].

It is almost a good idea to initialize a variable with a value. This is especially true for arrays since not every element may be used right away. When the memory is allocated for the array, whatever happens to be at the memory address will show up in the array. This can cause problems down the road as you have no way of knowing how that value arrived. For example:

```

1  #include <stdio.h>
2
3  #define STUDENTCNT 2
4  #define TESTCNT 3
5
6  int
7  main () {
8
9      float grades[STUDENTCNT][TESTCNT];
10     printf("Grades consumes %lu bytes of memory.\n", sizeof(grades));
11
12     for(unsigned int student = 0; student < STUDENTCNT; student++) {
13         for (unsigned int test = 0; test < TESTCNT; test++) {
14             printf("Student %u\tTest: %u\tGrade: %f\n",
15                 student, test, grades[student][test]);
16         }
17     }
18 }
```

```

17     }
18     return 0;
19 }

```

Listing 2.7: src/02-array1.c

Line 9: Here I hardcode the size of the array through the use of a *#define* macro. Doing this makes it obvious what the dimensions of the array are for but won't trigger the compiler into thinking I'm creating a variable sized array. The use of *#define* should be used in very limited cases[1].

Line 9: A two-dimensional array is created to track students and their grades. All of the grades will be of type float. Notice that this creates space for the array but does not assign it any values.

Line 10: The sizeof operator determines how many bytes were allocated for the array.

Line 15: This line runs inside of a nested for loop where the inner loop increments the *test* index and the outer loop increments the *student* index. The end result is that we get all the grades for the first student and then each of the grades for each remaining student.

```

$ ./02 - array1
Grades consumes 24 bytes of memory.
Student 0 Test: 0 Grade: 0.000000
Student 0 Test: 1 Grade: 0.000000
Student 0 Test: 2 Grade: 0.000000
Student 1 Test: 0 Grade: 0.000000
Student 1 Test: 1 Grade: 1158749014972294498713827247390720.000000
Student 1 Test: 2 Grade: 0.000000

```

Notice because I didn't initialize the contents of the array, I pick up a stray value that happened to already be in memory when the array space was allocated there.

```

1 #include <stdio.h>
2
3 #define STUDENTCNT 2
4 #define TESTCNT 3
5
6 int
7 main () {
8
9     char students[STUDENTCNT][5] = { "Jenny", "Jimmy" };
10    float grades[STUDENTCNT][TESTCNT] = { 0 };
11    printf("Grades consumes %lu bytes of memory.\n", sizeof(grades));
12
13    for(unsigned int student = 0; student < STUDENTCNT; student++) {
14        for (unsigned int test = 0; test < TESTCNT; test++) {
15            printf("Student %s\tTest: %u\tGrade: %f\n",
16                students[student], test, grades[student][test]);
17        }
18    }

```



```
19     return 0;  
20 }
```

Listing 2.8: src/02-array2.c

Line 10: I now initialize my *grades* array by placing a zero in curly braces. This zeros out each of the grades.

Line 9: Here I under-allocate the size of the strings which are nothing more than *char* arrays. This causes no compile time warnings or errors. However, as you'll see in the following output, this causes an issue in printing the first name in the array. The way that C knows it has reached the end of a string is that it reaches a null termination `\0` character. Since I under-allocated for the array, the first *char* of the next string overwrites the `\0`. This is why they print out together when trying to print just the first string. Just as disturbing and potentially destructive is the final `\0` for the last name technically writes into memory that is not allocated to the *students* array.

```
$ ./02-array2  
Grades consumes 24 bytes of memory.  
Student JennyJimmy Test: 0 Grade: 0.000000  
Student JennyJimmy Test: 1 Grade: 0.000000  
Student JennyJimmy Test: 2 Grade: 0.000000  
Student Jimmy Test: 0 Grade: 0.000000  
Student Jimmy Test: 1 Grade: 0.000000  
Student Jimmy Test: 2 Grade: 0.000000
```

2.6 Pointers

Pointers are used frequently in C so being proficient in their use is a must. We've seen previously in the XOR example that pointers do as their name suggests; they point to things. They contain the memory addresses. Passing around memory address rather than entire objects is often times much more efficient than moving and copying values. However, misuse of pointers can lead to buggy and insecure programs. A common theme you may see is the **ABC** principle. **A**lways **B**e **C**hecking. In other words, its good practice to always test your pointers before reaching into them. I believe I mentioned this already but its worth repeating. Reaching into a pointer that is set to *NULL* will cause a *Segmentation fault* and cause your program to terminate.

You'll often use the `&` operator when you need the memory address of something in order to pass it to a pointer. You will also see the dereference `*` operator used on a pointer to work with the data that resides at the memory address of the pointer. In other words, the data that the pointer points to. Using subscript operators `[]` with pointers is also quite common. This can be used to dereference into a portion of the memory address based on index value and the data type of the pointer. For example, lets assume we have a integer pointer and on a particular platform, an *int* uses 4 bytes. If we dereference into the 5th index position, we're going 20 bytes ($4 * 5$) passed the address held by

the pointer. In the following example, we'll see an example of this using a non-optimized version of a sorting algorithm.

```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4 void
5 sort(int *, size_t);
6
7 void
8 swap(int *, int *);
9
10 void
11 printArray(int *, size_t);
12
13 int
14 main() {
15     int numbers[] = { 1, 9, 7, 15, 42, 18, 12 };
16
17     const size_t numElements = sizeof(numbers) / sizeof(int);
18
19     printArray(numbers, numElements);
20
21     sort(numbers, numElements);
22
23     printArray(numbers, numElements);
24
25     return 0;
26 }
27
28 void
29 sort(int *pArray, size_t numElements) {
30     if ((NULL == pArray) || (0 == numElements)) {
31         return;
32     }
33
34     bool complete = false;
35     while (false == complete) {
36         complete = true;
37         for (size_t i = 0; i < (numElements - 1); i++) {
38             if (pArray[i] > pArray[i+1]) {
39                 complete = false;
40                 swap(pArray + i, pArray + i + 1);
41             }
42         }
43     }
44 }
45
46 void
47 swap(int *p1, int *p2) {
```

```

48     if ((NULL == p1) || (NULL == p2)) {
49         return;
50     }
51
52     int temp;
53     temp = *p1;
54     *p1 = *p2;
55     *p2 = temp;
56 }
57
58 void
59 printArray(int *pArray, size_t numElements) {
60     if (NULL == pArray) {
61         return;
62     }
63
64     for (size_t i = 0; i < numElements; i++) {
65         printf("%d ", pArray[i]);
66     }
67     printf("\b\n");
68 }

```

Listing 2.9: src/02-pointers1.c

```

$ ./02-pointers1
1 9 7 15 42 18 12
1 7 9 12 15 18 42

```

Line 15: An array is declared to hold various integer values. The size of the array is based on the number of elements I initialize it with.

Line 17: We calculate the number of elements in the array. This works fine with arrays but as soon as I pass it to the *sort* function and it becomes a pointer, it no longer works[1]. Instead, if I use the *sizeof* on a pointer, what I'll get is the number of bytes required to store the address. This will equate to 4 bytes on a 32 bit system and 8 bytes on a 64 bit system.

Line 21: We pass the *numbers* array to our *sort* function. I don't have to use the *&* operator since *numbers* being an array can automatically be interpreted in the called function as an *int ** (integer pointer). We also pass in the number of elements in our array since the *sort* function will not be able to determine this.

Line 29: We receive the *numbers* array address as *int *pArray*. I have the option to name it anything I want. Depending upon the coding standard you are following, often times giving the pointer a name that also indicates its a pointer can help in the readability later on[1]. In this case, by starting the name with a 'p' its more obvious that its a pointer.

Line 30: We check to make sure we received an address and not a *NULL* pointer. This is something you must do each time you are about to use a pointer. Not doing this is asking for your program to crash.

Line 38: The subscript operators [] can still be used to access values within the array.

Line 40: This line passes two different pointer addresses to our *swap* function using pointer arithmetic. Under the hood, the pointer *pArray* is an integer pointer. An *int* takes up 4 bytes so what we're actually doing is $ADDRESS + (i * 4)$ to calculate the adjusted address that we actually send to the *swap* function. This is the exact same math that gets applied when we use the subscript operators `[]`. However, I didn't want to dereference into the array and access the value at a particular address. I wanted the address itself. I could have also used `&pArray[i]` and gotten the same results. Care must be taken as C won't stop you from calculating addresses passed the original array address space.

Line 48: I once again check to ensure I received pointers that contain addresses and not *NULL*.

Lines 53 - 55: Here we move the values between the pointers by dereferencing them via the `*` operator. In essence, we copy the value at the memory address pointed to by *p1* into temp. We then copy the value at the memory address pointed to by *p2* into the memory address pointed to by *p1*. Finally, we copy the value of temp into the memory address pointed to by *p2*.

2.7 Dynamic Memory

2.7.1 Stack

Up until this point, the variables we've created have a lifetime based on the control block they were created in. For example, in *src/01-helper.c* (1.5), the *num* variable is created at the top of the *distributeSnacks* function and is automatically cleaned up at the closing bracket of the function. Lets take a look at the *01-basics3* program we looked at in chapter 1. In order to produce the below code, I decompiled the executable by running: `objdump -M "intel" -d 01-basics3`. Below is a copy of the decompiled *main* and *distributeSnacks* functions.

```

1  0000000000001149 <main>:
2  1149: f3 0f 1e fa          endbr64
3  114d: 55                  push    rbp
4  114e: 48 89 e5            mov     rbp, rsp
5  1151: 48 83 ec 10         sub     rsp, 0x10
6  1155: c7 45 fc 03 00 00 00 mov     DWORD PTR [rbp-0x4], 0x3
7  115c: 8b 45 fc            mov     eax, DWORD PTR [rbp-0x4]
8  115f: 89 c7              mov     edi, eax
9  1161: e8 1a 00 00 00     call   1180 <distributeSnacks>
10 1166: 89 c6              mov     esi, eax
11 1168: 48 8d 3d 95 0e 00 00 lea     rdi, [rip+0xe95]
12 116f: b8 00 00 00 00     mov     eax, 0x0
13 1174: e8 d7 fe ff ff     call   1050 <printf@plt>
14 1179: b8 00 00 00 00     mov     eax, 0x0
15 117e: c9                leave
16 117f: c3                ret
17
18 0000000000001180 <distributeSnacks>:

```

```

19  1180: f3 0f 1e fa      endbr64
20  1184: 55                  push    rbp
21  1185: 48 89 e5            mov     rbp, rsp
22  1188: 89 7d ec            mov     DWORD PTR [rbp-0x14], edi
23  118b: c7 45 fc 00 00 00 00 mov     DWORD PTR [rbp-0x4], 0x0
24  1192: 83 7d ec 03         cmp     DWORD PTR [rbp-0x14], 0x3
25  1196: 74 08              je      11a0 <distributedSnacks+0x20>
26  1198: 83 7d ec 04         cmp     DWORD PTR [rbp-0x14], 0x4
27  119c: 74 0b              je      11a9 <distributedSnacks+0x29>
28  119e: eb 12              jmp     11b2 <distributedSnacks+0x32>
29  11a0: c7 45 fc 14 00 00 00 mov     DWORD PTR [rbp-0x4], 0x14
30  11a7: eb 10              jmp     11b9 <distributedSnacks+0x39>
31  11a9: c7 45 fc 28 00 00 00 mov     DWORD PTR [rbp-0x4], 0x28
32  11ab: eb 07              jmp     11b9 <distributedSnacks+0x39>
33  11b2: c7 45 fc 00 00 00 00 mov     DWORD PTR [rbp-0x4], 0x0
34  11b9: 8b 45 fc            mov     eax, DWORD PTR [rbp-0x4]
35  11bc: 5d                  pop     rbp
36  11bd: c3                  ret
37  11be: 66 90              xchg    ax, ax

```

If you've never looked at assembly code before, this can look a bit crazy. Without going into the nitty gritty details of assembly language syntax, I wanted to take a second and point out a few things. There is something called a **stack frame**. *Stack frames* typically hold the variables declared during compilation time. What do I mean about during compilation time? Essentially, the compiler has enough information during compilation to determine how much space to allocate for a variable and what its lifetime will be. Each function gets its own stack frame. This is something worth considering especially for recursive functions (functions that call themselves) because for each call, a new stack is provisioned. There is also something called a **call stack** that is composed of various stack frames and the return addresses associated with them. This is demonstrated in figure 2.4.

In essence, as one function calls another, a new *stack frame* is built so that the called function has a place to store its variables and the program knows where to go back to once that called function is complete. There are two registers within the processor that keep track of the start and end of the *stack frame*. The **rbp** (base pointer) register marks the bottom and the **rsp** (stack pointer) register marks the top of the stack frame. In the context of the assembly code above, you can see on lines 3-5, the main function is establishing its stack by saving the address of the current base pointer (*push rbp*), establishing a new location for the new base pointer (*mov rbp, rsp*), and then subtracting 16 bytes (*sub rsp, 0x10*) from the stack pointer (*rsp*) to establish 16 bytes worth of space on the new stack for storing variables. Again, you don't need to understand assembly code but I wanted you to see this in action in order for you to understand the purpose of the stack. Of note, notice that we subtract in order to allocate space. This means that the stack actually grows down in memory addresses. We can then see on line 7 that the value 3 is being copied onto the stack at location *rbp - 0x4*. This correlates to where we created the variable *member* and assigned to it the value of *SHAGGY* back in *01-basics3.c*(1.4).

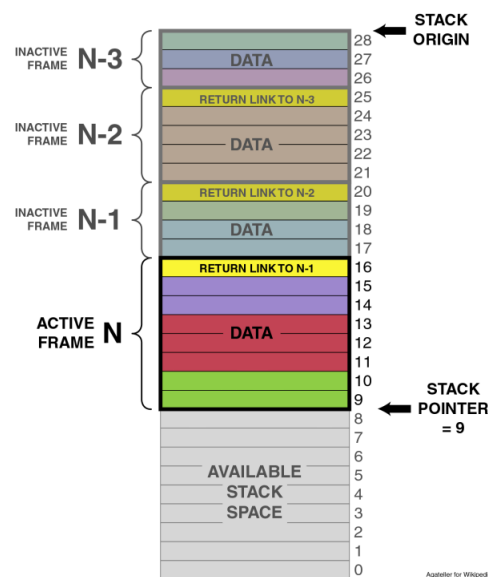


Figure 2.4: Call Stack

On line 9, we also see a *call* to the *distributeSnacks* function. There is another register that keeps track of the current line or instruction that is being executed (**rip** - instruction pointer). The *call* first pushes *rip + 1* onto the stack. This is so that when we return from *distributeStacks*, execution will resume on line 10. Then *rip* is set to begin executing at the beginning of the *distributeSnacks* function. The *return* on line 36 performs a *pop rip* which pulls the return address off of the stack and places it in *rip*. This is why execution returns to line 10.

For whatever reason, when the gcc compiler built this program, it didn't create a new stack for function *distributeSnacks* large enough to store the *num* variable. This could be because the program is so small and this was a small optimization that the compiler made. Whatever the reason, we can see that on lines 20 and 21, we save the address of our current base pointer address by pushing it on the stack and then set it equal to the current stack pointer address. Although this is a whole new stack frame, it doesn't allocate space to save any variables by subtracting from the stack pointer. Instead, we can see that on line 23, it's actually storing its values in the previous stack frame. Again, this is probably an optimization that saves one or two CPU cycles.

Moving things on and off the stack is very common and fairly quick. However, there are advantages and disadvantages for doing so. One disadvantage is that because the return addresses for the calling function lives on the stack, it's a place that a malicious user may want to attack in order to control what the program does upon returning from the called function. A lot of work has been put into protecting the stack and detecting *stack smashing* but nothing is full proof. Additionally, because a *stack frame* gets torn down after a function returns, the variables that lived there are no longer reachable. Normally this is the functionality that you want but not always. Instead, you may have reasons for wanting to allocate a larger amount of memory away from the rest of your stack variables or maybe you want it to outlive the end of the called function. Additionally, you may

not know how much memory you're going to require at the time of compilation. Therefore, the compiler won't know how much space to allocate on the stack.

2.7.2 Heap

The **heap** is a section of memory that allows for the dynamic allocation of memory. Dynamic memory does not get automatically cleaned up when a function returns. Instead, the programmer has to keep track of it and clean it up manually as needed. If not, memory will be lost (leaked). The four most common functions you'll see when working with dynamic memory are:

1. **malloc** - void *malloc(size_t size);

The *malloc()* function allocates *size* bytes and returns a pointer to the allocated memory. The memory is **not** initialized.

2. **calloc** - void *calloc(size_t nmemb, size_t size);

The *calloc()* function allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory **is** set to zero.

3. **realloc** - void *realloc(void *ptr, size_t size);

The *realloc()* function changes the size of the memory block pointed to by *ptr* to *size* bytes.

4. **free** - void free(void *ptr);

The *free()* function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to *malloc()*, *calloc()*, or *realloc()*.

Lets take a look at some of these functions in action:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <string.h>
5
6
7 bool
8 getInput(char **, FILE *);
9
10 int
11 main() {
12     char *buffer = NULL;
13     bool result;
14     result = getInput(&buffer, stdin);
15     printf("result: %d\n", result);
16     printf("buffer: %s\n", buffer);
17
18     free(buffer);
19     return 0;
20 }
21
```

```

22 bool
23 getInput(char **buffer, FILE *fPtr) {
24     if ((NULL == buffer) || (NULL != *buffer) || (NULL == fPtr)) {
25         return false;
26     }
27
28     size_t blockSize = 16;
29     size_t maxSize = blockSize;
30     size_t actualSize = 0;
31
32     // initially allocate a buffer to read into
33     *buffer = (char *)malloc(maxSize * sizeof(char));
34     if (NULL == *buffer) {
35         fprintf(stderr, "Unable to initially allocate for buffer. %s:%d.\n",
36             __FILE__, __LINE__);
37         return false;
38     }
39
40     // null terminate initial buffer
41     *buffer[0] = '\0';
42     char * temp;
43
44     bool keepLooping = true;
45     while (keepLooping) {
46
47         // read into the buffer avoiding writing over existing characters
48         temp = fgets(*buffer + actualSize, maxSize - actualSize, fPtr);
49         if (NULL == temp) {
50             fprintf(stderr, "Unable to read from file pointer. %s:%d.\n",
51                 __FILE__, __LINE__);
52             return false;
53         }
54
55         // determine if the current buffer is full
56         actualSize = strlen(*buffer);
57         if (((maxSize - 1) == actualSize) && ((*buffer)[actualSize - 1] != '\n' )) {
58             maxSize += blockSize;
59             printf("Realloc maxSize: %ld\n", maxSize);
60
61             // perform a realloc based on the new maxSize
62             temp = (char *)realloc(*buffer, maxSize * sizeof(char));
63             if (NULL == temp) {
64                 fprintf(stderr, "Unable to realloc buffer to %ld bytes. %s:%d.\n",
65                     maxSize, __FILE__, __LINE__);
66                 return false;
67             }
68             *buffer = temp;
69         }
70         else {
71             keepLooping = false;
72         }

```

```
73     }  
74  
75     return true;  
76 }
```

Listing 2.10: src/02-dynamic1.c

The point of the *getInput* function is to prompt the user for input and take whatever the user types and store it into a buffer. It's not particularly exciting code but will continually allocate blocks of 16 bytes until the user hits ENTER or sends an End of File (EOF). This code was meant to demonstrate the use of *malloc*, *realloc*, and *free* and was not meant to be efficient or even practical for that matter.

Notice on line 14, the *main* function calls the *getInput* function and passes it the address of a *char* pointer that currently doesn't point anywhere because it's set to *NULL*. This is why on line 23, *getInput* receives a pointer to a pointer (*char ***). On line 24 we apply the ABC principle by making sure the address we're given for the buffer is not *NULL*. This ensures we don't hit a *SEGMENTATION* fault when we dereference the first pointer of the buffer (**buffer*).

On line 33 we see *malloc* used to allocate space on the *heap* and assign the address to **buffer*. Because there is no guarantee that *malloc* will return successfully, on line 34 we test to make sure that an actual address was stored. On line 41 we see a *NULL* byte being written to the first byte of the new buffer. This is because strings should be *NULL* terminated and *malloc* only allocates memory; which means the new buffer is currently filled with garbage (whatever was already in memory where the buffer was allocated).

Lines 48 - 53 reads user input but only enough to fill up the existing buffer that hasn't already been written to. Lines 56 and 57 attempt to determine if the buffer is full and if so, line 62 performs a *realloc* to increase the size of the buffer. The actual man page for *realloc* indicates:

The *realloc()* function returns a pointer to the newly allocated memory, which is suitably aligned for any built-in type, or *NULL* if the request failed. The returned pointer may be the same as *ptr* if the allocation was not moved (e.g., there was room to expand the allocation in-place), or different from *ptr* if the allocation was moved to a new address. If *size* was equal to 0, either *NULL* or a pointer suitable to be passed to *free()* is returned. If *realloc()* fails, the original block is left untouched; it is not freed or moved.

This is why you see the address returned from *realloc* being assigned to a temporary variable. If it fails, it will not make any changes to the existing buffer it was given and will return *NULL*. If successful, it will either extend the *heap* allocation or allocate a new buffer, copy the existing data into it, and free the old buffer. In the latter case of having to allocate a new buffer, that address will show up in the temporary variable and will need to be assigned back to the original buffer.

On line 18 we see the use of *free*. This releases the memory that was allocated for the buffer. Without doing so, our program would leak memory. Lets check to make sure this actually worked with the *valgrind* program.

```
$ valgrind ./02-dynamic1
==28239== Memcheck, a memory error detector
==28239== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==28239== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==28239== Command: ./02-dynamic1
==28239==
This input will exceed 16 characters and require a realloc.
Realloc maxSize: 32
Realloc maxSize: 48
Realloc maxSize: 64
result: 1
buffer: This input will exceed 16 characters and require a realloc.

==28239==
==28239== HEAP SUMMARY:
==28239==      in use at exit: 0 bytes in 0 blocks
==28239==    total heap usage: 6 allocs, 6 frees, 2,208 bytes allocated
==28239==
==28239== All heap blocks were freed -- no leaks are possible
==28239==
==28239== For lists of detected and suppressed errors, rerun with: -s
==28239== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Notice that we call *valgrind* and as an argument we call our program. Line 7 is the input that I typed into the program and then hit ENTER. This resulted in having to call *realloc* three times. In the summary, *valgrind* indicates there are 0 bytes in use on exit. Had we missed freeing memory, the number of bytes would indicate that amount of memory. Also notice, *valgrind* registered six allocations, six frees, and that leaks are not possible. Four of these are the result of running *malloc* and *realloc*. One is the result of calling *fgets* and one is most likely due to a default allocation for command line inputs (CLI) into our *main* function. We'll talk about using the CLI in Chapter 4.

3. Control Flow

3.1 If/Else

Up to this point, we've seen a few examples of using an if statement. We've even seen an example of the `?:` ternary conditional evaluation operator in `src/02-character1.c` [2.3]. Also, as shown in `src/02-xor.c` (2.6) when we are evaluating multiple things, it is common practice to surround each evaluation with parantheses^[1] to eliminate issues with operator precedence.

```
1 #include <stdio.h>
2
3 int
4 main() {
5     unsigned proto = 17;
6
7     if (6 == proto) {
8         printf("TCP traffic detected...\n");
9     }
10    else if (17 == proto) {
11        printf("UDP traffic detected...\n");
12    }
13    else {
14        printf("Unknown protocol number detected...\n");
15    }
16
17    return 0;
18 }
```

Listing 3.1: src/03-if.c

Line 7: The *if* statement is an evaluation that either evaluates to true (non-zero) or false (zero). The order I've chosen, `6 == proto`, may look odd but is in fact strategic. If you were to place *proto* first in the evaluation and mistyped the `==` as `=`, this becomes an assignment instead of an evaluation. This is an easy typo to miss as I have personally done it many times. By placing the constant first, attempting to assign a value to a constant is automatically going to fail compilation and alert you to the issue.

Line 10: If the first evaluation is false, we evaluate to see if the *proto* is equal to 17.

Line 13: If the first and second evaluation are both false, the last *else* will be run. An *else* block is not required.

3.2 Switch

The *switch* statement is similar to *if/else* but is restricted to integer types. We've already seen the switch used in src/01-helper.c [1.5]. Lets take a look at it again.

```
1 #include "01-helper.h"
2
3 int
4 distributeSnacks (mysteryMember_t member) {
5     int num = 0;
6
7     switch (member) {
8         case SHAGGY:
9             num = 20;
10            break;
11         case SCOOBY:
12             num = 40;
13            break;
14         default:
15             num = 0;
16     }
17
18     return num;
19 }
```

Listing 3.2: src/01-helper.c

Line 7: The *member* variable is being evaluated.

Line 8: The *case* statement is a label and if the value of *member* is equal to the value *SHAGGY*, control of the program will jump here and begin processing each line that follows it.

Line 10: The `break` statement causes control to jump out of the *switch* block. Without it, control would keep going line by line through the next label. If this is intentional, it should be noted with a comment [1].

Line 14: If none of the other evaluations end up being true, the *default* label is processed. The "Embedded C Coding Standard" dictates that every *switch* statement should have a *default* label [1].

3.3 While

We've seen a while loop while discussing our pointers in `src/02-pointers1.c` [2.9]. A *while* loop will continuously loop until its evaluation criterion evaluates to false or zero.

It is quite common to see an "infinite" loop in a program implemented in a *while* loop. These are loops that will intentionally essentially loop forever. It is quite common to see them implemented as:

```
while (1) {  
    ...  
}
```

The "Embedded C Coding Standard" [1] advises against this and instead recommends using a *for* loop like this:

```
for (;;) {  
    ...  
}
```

3.3.1 Do/While

In C, there is also a *do/while* loop. In this case, the evaluation criteria is not until the end of the loop instead of at the beginning like with a traditional *while* loop. A *while* loop will be executed zero or more times but a *do/while* loop will be executed one or more times.

3.4 For

We've already seen a number of *for* loops such as in `src/02-xor.c` 2.6. They allow us to concisely control when a loop starts and ends. A *for* loop has three main parts separated by semi-colons:

1. Variable initialization - This value may be loop counter or index value but is typically part of the evaluation criterion that determines when the loop should exit.

2. Evaluation criteria - This is some type of conditional that will determine when the *for* loop will exit.
3. Increment - Traditionally this is where a counter or index value will be incremented after the completion of each loop.

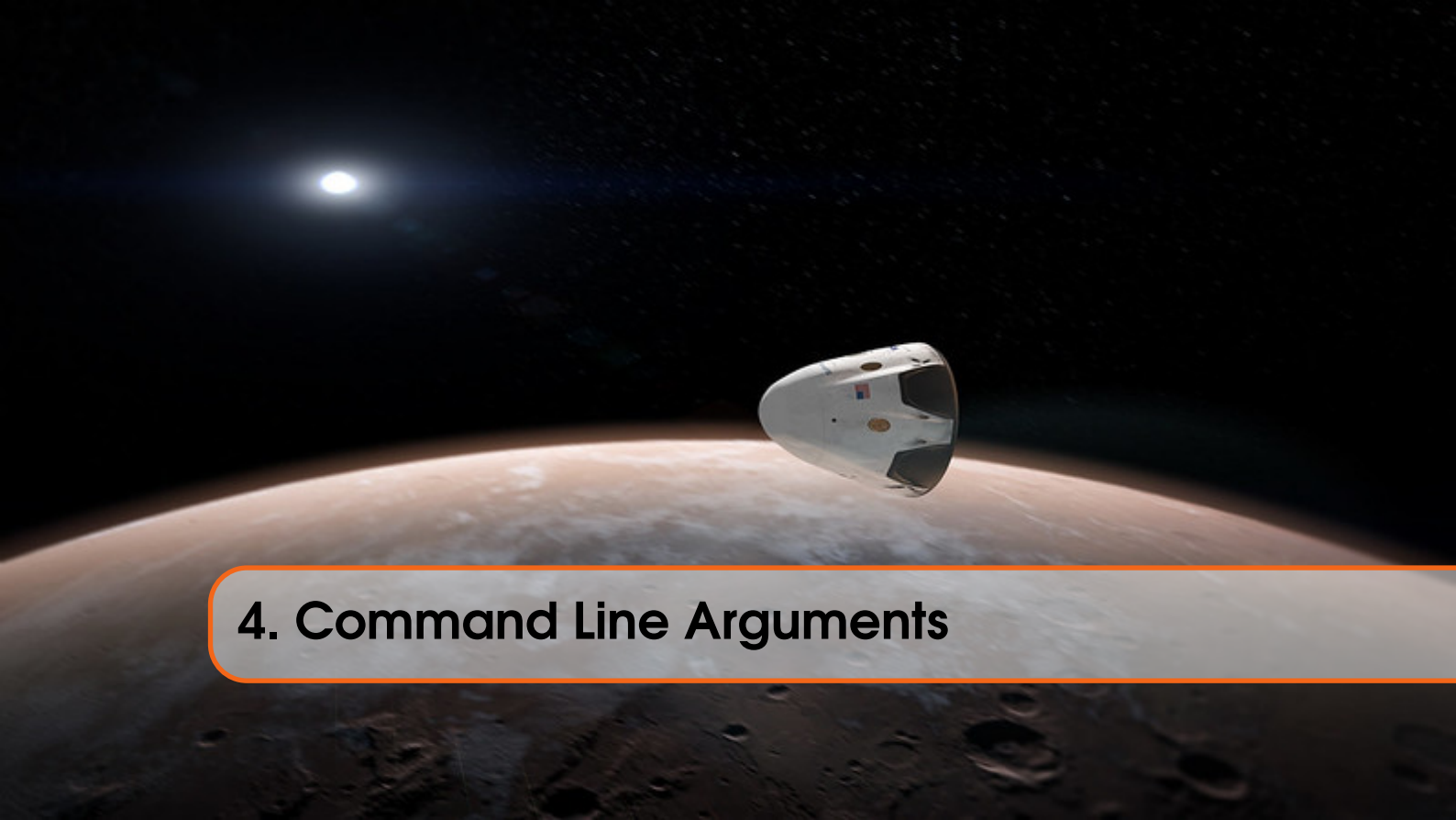
```
for (COUNTER_INITIALIZATION; LOOP_EVALUATION; COUNTER_INCREMENT) {  
    ...  
}
```

3.5 Goto

A *goto* is a way to perform an unconditional jump to a portion of your code. A quote from Michael Barr in "Embedded C Coding Standard" [1]:

- c. It is a preferred practice to avoid all use of the *goto* keyword. If *goto* is used it shall only jump to a label declared later in the same or an enclosing block.

I agree with Mr. Barr's assessment of the *goto*. There is such a thing as the "lifetime" of a variable. Using a *goto* incorrectly can mess with this and so the advice to either avoid its use entirely or only jump to a label later in your code that is in the same or enclosed block helps to minimize issues with this. The only place I have found a good use for them is when I've encountered some type of fail condition that I want to return from a function immediately but not before I've had a chance to clean up memory that I may have allocated. In those cases, I may use a *goto* to jump near the bottom of my function to a point right before I *free* up some memory and then exit the function.



4. Command Line Arguments

We've talked a bit about how to write very basic programs and compile them. Now we'll talk about how to pass arguments into our program.

4.1 Manual Parsing

Up until now, our main function hasn't had any arguments. However, it can actually receive input when our program is called and should look like `main(int argc, char **argv)`. You may see this also written as `main(int argc, char *argv[])`.

```
1 #include <stdio.h>
2
3 int
4 main(int argc, char **argv) {
5
6     printf("Number of args: %d\n", argc);
7
8     for(int i = 0; i < argc; i++) {
9         printf("%d: %s\n", i+1, argv[i]);
10    }
11
12    return 0;
13 }
```

Listing 4.1: src/04-cli1.c

```
$ ./04-CLI1
Number of args: 1
1: ./04-CLI1

$ ./04-CLI1 these are the arguments
Number of args: 5
1: ./04-CLI1
2: these
3: are
4: the
5: arguments
```

Line 6: As we can see in the output, the *argc* variable corresponds to the number of arguments that were passed including the name of the program as it was called.

Line 9: Here we can iterate through the array of character arrays.

4.2 getopt

C provides the *getopt* as well as the *getopt_long* functions to aid in parsing the various arguments you'll receive as well as enforce various rules around them. I'll present a few of the options you'll see frequently but I suggest you check out the manual page by running `man 3 getopt` from a terminal. There is an excellent example of its usage at the bottom of the manual page. The example I'll show you uses *getopt* to allow for options such as `-h`. Using *getopt_long* you can also specify long options such as `--help`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <getopt.h>
5 #include <limits.h>
6 #include <errno.h>
7
8 static void
9 usage(FILE *, char *);
10
11 typedef enum {
12     ORIGINALCASE,
13     LOWERCASE,
14     UPPERCASE
15 } case_t;
16
17 int
18 main(int argc, char **argv) {
19
20     case_t ul = ORIGINALCASE;
```



```

21  char *endptr;
22  long repeat = 1;
23
24  int opt;
25  while ((opt = getopt(argc, argv, "hulr:")) != -1) {
26      switch (opt) {
27          case 'u':
28              ul = UPPERCASE;
29              break;
30          case 'l':
31              ul = LOWERCASE;
32              break;
33          case 'r':
34              repeat = strtol(optarg, &endptr, 10);
35              if ((errno == ERANGE && (repeat == LONG_MAX || repeat == LONG_MIN))
36                  || (errno != 0 && repeat == 0) || (*endptr != '\0')
37                  || (repeat <= 0)) {
38                  fprintf(stderr, "Invalid numeric value for '-r'.\n");
39                  usage(stderr, argv[0]);
40                  exit(EXIT_FAILURE);
41              }
42              break;
43          case 'h':
44              usage(stdout, argv[0]);
45              exit(EXIT_SUCCESS);
46          default: /* '?' */
47              usage(stderr, argv[0]);
48              exit(EXIT_FAILURE);
49      }
50  }
51
52  // Test to make sure only one string was passed in
53  if ((argc == optind) || (argc > (optind + 1))) {
54      fprintf(stderr, "Invalid number of args.\n");
55      usage(stderr, argv[0]);
56      exit(EXIT_FAILURE);
57  }
58
59  printf("Name: %s argc:%d optind:%d case:%d repeat:%ld\n",
60         argv[optind],
61         argc,
62         optind,
63         ul,
64         repeat);
65
66  return 0;
67  }
68
69  static void
70  usage(FILE *fptr, char *program) {
71      fprintf(fptr, "Usage: %s [-r CNT] [-u] [-l] name\n",

```

```

72     program );
73     fprintf( fptr , "\t-r\tRepeat Count\n" );
74     fprintf( fptr , "\t-u\tUppercase\n" );
75     fprintf( fptr , "\t-l\tLowercase\n" );
76 }

```

Listing 4.2: src/04-cli2.c

Here we have a snippet from an application that takes a word from the command line and allows the user to specify if they want that word in uppercase, lowercase, and how many times they want it. While not all of that logic has been written out, there is enough that you can see a mix of *getopt* enforcing constraints on how the program can be called and logic that the programmer enforces as well. As we'll see, *getopt* is restricting what options can be used but does not specify if they have to be used or how many times they can be used. That latter logic is up to the programmer.

Line 25: This line is nearly a copy and paste straight from the manual page for *getopt*. It is composed of a *while* loop that utilizes *getopt* to process each command line option one by one. "hulr:" indicates there is a -h, -u, -l, and -r options that can be supplied to the program. Additionally, the -r option requires an argument due to the : that follows it.

Lines 27, 33, 43: Here we process valid options.

Line 46: Here we process invalid options.

Line 34: Since the -r options requires an argument, here we use *strtol* to convert it to a *long int*. We also see the *optarg* variable used. This variable isn't something we created. It was created in *getopt.h* and contains the option argument for the -r option.

Line 53: Here we see another variable, *optind*, that *getopt* created for us. This is the index value of the option within *argv* that *getopt* is processing. At this point, all valid options should have been processed meaning that if *argc* and *optind* are equal, then the user forgot to supply a word. *argc* ideally should be one value higher than *optind*. If they are equal, the user didn't supply a name. If the difference between *argc* and *optind* is greater than one, then the user supplied more than one word. This is part of the logic enforcement that the programmer has decided to make.

Line 59: Here we merely print out the results of parsing the options and arguments. What we should see is the last -u or -l will determine if it prints in uppercase or lowercase. If neither are provided, then the original case is used. If the -r option is used and a valid numeric argument is given, *repeat* will hold that value.

```

$ ./04-cli2 pizza
Name: pizza argc:2 optind:1 case:0 repeat:1

$ ./04-cli2 -h
Usage: ./04-cli2 [-r CNT] [-u] [-l] name
    -r Repeat Count
    -u Uppercase
    -l Lowercase

```

```
$ echo $?
0

$ ./04-cli2 pizza the hut
Invalid number of args.
Usage: ./04-cli2 [-r CNT] [-u] [-l] name
    -r Repeat Count
    -u Uppercase
    -l Lowercase

$ echo $?
1

$ ./04-cli2 pizza -p
./04-cli2: invalid option -- 'p'
Usage: ./04-cli2 [-r CNT] [-u] [-l] name
    -r Repeat Count
    -u Uppercase
    -l Lowercase

$ echo $?
1

$ ./04-cli2 -l pizza -r3
Name: pizza argc:4 optind:3 case:1 repeat:3

$ ./04-cli2 -l pizza -r3po
Invalid numeric value for '-r'.
Usage: ./04-cli2 [-r CNT] [-u] [-l] name
    -r Repeat Count
    -u Uppercase
    -l Lowercase

$ ./04-cli2 -l -l -u pizza -r3
Name: pizza argc:6 optind:5 case:2 repeat:3
```

As you can see in the first example, no options are required other than specifying a name or word. We can also see using valid options results in 0 (SUCCESS) being returned to our terminal but invalid options results in a 1 (FAILURE). Lastly we can see that multiple options can be specified but in the case of -u and -l, the last one to be specified wins.



Part Two

5	Structures	53
6	Working with Files	61
6.1	Opening a File	
6.2	Reading Text	
6.3	Reading and Writing Binary Data	
6.4	File Stats	
7	Concurrency	73
7.1	Forking	
7.2	Multithreading	
8	Networking	87
8.1	getaddrinfo	
8.2	sigaction	

5. Structures

We've now seen enough of the mechanics to talk about building a **structure**; better known as a **struct**. These allow us to build more complex or custom data types. Unlike arrays, *structs* allow for multiple different data types to be placed inside of a single object. Often times, instead of passing around the struct itself, you'll see pointers to *structs* being passed around instead. This reduces the amount of data that has to be added to the stack or passed between functions.

In the following example, we'll use multiple *structs* to represent a player capable of carrying items.

```
1  #ifndef PLAYER_H
2  #define PLAYER_H
3
4  #include <stdio.h>
5  #include <stdbool.h>
6
7  struct item {
8      char *name;
9  };
10
11 struct inventory {
12     size_t maxItems; // max items
13     size_t totalItems; // current items
14     struct item *items; // array of pointers to items
15 };
16
17 typedef struct player {
18     struct inventory inv; // player's inventory
19     char *name;
```

```

20 } player_t;
21
22 player_t *
23 createPlayer();
24
25 bool
26 addItem(player_t *);
27
28 void
29 listItems(const player_t *);
30
31 bool
32 destroyPlayer(player_t **);
33
34 #endif

```

Listing 5.1: src/05-player.h

There are three different *structs* here: *player* (line 17), *inventory* (line 11), and *item* (line 7). Rather than have a single *struct*, they are broken up by purpose to keep our player organized as we continue to build it out. Notice that a *player struct* contains a single *inventory struct*. Also notice that an *inventory struct* contains a pointer to an *item struct*. As we'll see in the next file, we'll use this pointer to actually allocate for an array of *item structs*. By doing this, a single player can have one inventory that contains multiple items.

As should be easy to see, a *struct definition* begins with the *struct* statement followed by the name of the *struct*. Inside the brackets we specify each member of the struct.

On line 17, we can also see we're using the keyword *typedef* and then on line 20 we're giving it the name *player_t*. This allows us to refer to a *player struct* as *player_t* rather than *struct player*. It's customary to provide a *typedef* when the user will be interacting with it.

```

1 #define _GNU_SOURCE
2 #include <stdio.h>
3 #include <stdbool.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <errno.h>
7 #include "05-player.h"
8
9 #define INV_SIZE 5
10
11 player_t *
12 createPlayer() {
13     // allocate for new player
14     player_t *player = calloc(1, sizeof(*player));
15     if (NULL == player) {
16         perror("Unable to allocate for new player.\n");

```

```

17     return NULL;
18 }
19
20 // get player's name
21 printf("Enter a name for the player: ");
22 size_t len = 0;
23 size_t nameLen = getline(&(player->name), &len, stdin);
24 if ('\n' == player->name[nameLen - 1]) {
25     player->name[nameLen - 1] = '\0';
26 }
27
28 // allocate for items
29 player->inv.maxItems = INV_SIZE;
30 player->inv.items = calloc(player->inv.maxItems,
31                             sizeof(*(player->inv.items)));
32
33 return player;
34 }
35
36 bool
37 addItem(player_t *player) {
38     if (NULL == player->inv.items) {
39         perror("Player items is NULL.\n");
40         return false;
41     }
42
43     // is inventory full
44     if (player->inv.maxItems == player->inv.totalItems) {
45         return false;
46     }
47
48     // get new item
49     printf("Enter an item name: ");
50     char *name = NULL;
51     size_t len = 0;
52     size_t nameLen = getline(&name, &len, stdin);
53     if ('\n' == name[nameLen - 1]) {
54         name[nameLen - 1] = '\0';
55     }
56
57     // store new item
58     if (strlen(name) > 0) {
59         player->inv.items[player->inv.totalItems].name = name;
60         player->inv.totalItems += 1;
61     } else {
62         free(name);
63         return false;
64     }
65
66     return true;
67 }

```

```

68
69 void
70 listItems(const player_t *player) {
71     if ((NULL == player) || (NULL == player->name)) {
72         perror("Can't list NULL player.\n");
73         return;
74     }
75
76     printf("Player: %s\n", player->name);
77
78     // empty inventory
79     if ((0 == player->inv.totalItems) || (NULL == player->inv.items)) {
80         printf("\tempty inventory\n");
81         return;
82     }
83
84     // loop through items
85     for (size_t num = 0; num < player->inv.totalItems; num++) {
86         printf("\tItem %ld: %s\n", num + 1, player->inv.items[num].name);
87     }
88 }
89
90 bool
91 destroyPlayer(player_t **player) {
92     if ((NULL == player) || (NULL == *player)) {
93         perror("Player is already NULL.\n");
94         return false;
95     }
96
97     if (NULL != (*player)->inv.items) {
98         for (size_t num = 0; num < (*player)->inv.totalItems; num++) {
99             // free individual items
100             free((*player)->inv.items[num].name);
101             (*player)->inv.items[num].name = NULL;
102         }
103         // free array of pointers to items
104         free((*player)->inv.items);
105         (*player)->inv.items = NULL;
106         (*player)->inv.totalItems = 0;
107     }
108
109     // free player
110     free((*player)->name);
111     (*player)->name = NULL;
112     free(*player);
113     *player = NULL;
114
115     return true;
116 }

```

Listing 5.2: src/05-player.c

I won't go over all of the above code. I'll leave that as an exercise for the reader. However, notice on line 14 that we use *calloc* to allocate for a *player struct*. I'm using *calloc* because all values will start at zero and all pointers will start as *NULL*. This is convenient when initially setting up a *struct* where you may not be setting initial values for each member. On lines 30 and 31 we also use *calloc* to allocate for our array of pointers to *item structs*. By doing this it sets the initial *char* array for *item struct names* to *NULL*. The number of pointers in this array is based on the value specified in *maxItems*.

Notice on line 29 that in order to access the *inv* member from our *player* we use *"->"*. *Player* is actually a pointer to a *player struct* and the *"->"* operator first dereferences the pointer before accessing the *inv* member. However, the *inv* member is not a pointer and so it's members can be directly accessed via the *"."* operator. Also notice that I'm checking for *NULL* quite a bit throughout this code. Anytime you are working with pointers and dynamically created memory, you have to be careful not to reach into a *NULL* pointer and cause a *SEGMENTATION* fault.

The last thing I would like to point out is the *destroyPlayer* function. While this function is a bit overkill, I wanted to point out a few things. First off, because we allocated for most of the memory used, we must run *free* on each piece that we allocated. Additionally, the order is nearly in the opposite order in which it was first allocated. By taking a bottom to top approach we ensure that we don't lose access to memory that we haven't gotten a chance to *free* yet. So, in this example, we first free each *name* of an item, then we *free* the array of pointers to items, we then *free* the name of the player, and then finally the *player* itself.

Also worth pointing out is that I set each pointer to *NULL* after I *free* it. I do this only to point out that the use of *free* doesn't actually change the value of the pointer. If you were to re-use the pointer later in a program and test for *NULL* before attempting to access it, you'll get a false sense of security as the underlying memory no longer belongs to you and will result in a *SEGMENTATION* fault. If there is the slightest chance the pointer will get used again, set it to *NULL* after you *free* it. Again, this isn't entirely required in this scenario and is only here for demonstration purposes. The one line that is arguably required is line 113. Because the *destroyPlayer* is taking the address of a pointer to a *player struct*, line 113 will ensure that the original pointer ends up as *NULL* in the calling function.

```

1  #include <stdio.h>
2  #include "05-player.h"
3
4  int
5  main() {
6
7      player_t *player1 = createPlayer();
8
9      bool result = addItem(player1);
10     if (true == result) {
11         printf("Successfully added.\n");
12     }

```

```

13
14     result = addItem(player1);
15     if (true == result) {
16         printf("Successfully added.\n");
17     }
18
19     listItems(player1);
20
21     result = destroyPlayer(&player1);
22     if ((true == result) && (NULL == player1)) {
23         printf("Successfully destroyed.\n");
24     }
25     return 0;
26 }

```

Listing 5.3: src/05-struct1.c

There isn't much here to discuss. The *cratePlayer* function returns a pointer to a newly created *player struct*. Here we use the *typedef* name *player_t* rather than *struct player* because its more convenient. When we call *destroyPlayer* we pass in the address of our *player1* pointer.

Let run our program and make sure all of the memory is properly released.

```

$ valgrind ./05-struct1
==65023== Memcheck, a memory error detector
==65023== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==65023== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==65023== Command: ./05-struct1
==65023==
Enter a name for the player: Ron
Enter an item name: Sword
Successfully added.
Enter an item name: Dental Floss
Successfully added.
Player: Ron
    Item 1: Sword
    Item 2: Dental Floss
Successfully destroyed.
==65023==
==65023== HEAP SUMMARY:
==65023==      in use at exit: 0 bytes in 0 blocks
==65023==    total heap usage: 7 allocs, 7 frees, 2,480 bytes allocated
==65023==
==65023== All heap blocks were freed -- no leaks are possible


```

```
==65023==
```

```
==65023== For lists of detected and suppressed errors, rerun with: -s
```

```
==65023== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

As we can see, there is no memory in use on exit and there are no errors.



6. Working with Files

At some point, we either have to read from a file or write to a file. You'll typically see this done in one of two ways, with *fopen* or with *open*. *fopen* is part of the C standard library where *open* is a system call to the operating system. This means that while *open* may allow for more advanced functionality, it comes with the cost of your code not being as portable between operating systems. Unless you have good reason, it is usually best to stick with the standard C library. For this reason, we will mainly focus on *fopen* and associated functions.

6.1 Opening a File

There are multiple modes that you can use to open a file. The man page for *fopen* specifies the following modes:

- r** Open text file for reading. The stream is positioned at the beginning of the file.
- r+** Open for reading and writing. The stream is positioned at the beginning of the file.
- w** Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file. **w+** Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- a** Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- a+** Open for reading and appending (writing at end of file). The file is created if it

does not exist. Output is always appended to the end of the file. POSIX is silent on what the initial read position is when using this mode. For glibc, the initial file position for reading is at the beginning of the file, but for Android/BSD/MacOS, the initial file position for reading is at the end of the file.

The mode string can also include the letter 'b' either as a last character or as a character between the characters in any of the two-character strings described above. This is strictly for compatibility with C89 and has no effect; the 'b' is ignored on all POSIX conforming systems, including Linux. (Other systems may treat text files and binary files differently, and adding the 'b' may be a good idea if you do I/O to a binary file and expect that your program may be ported to non-UNIX environments.

6.2 Reading Text

Lets take a look at opening a text file and reading from it.

```
1 #include <stdio.h>
2 #define BUFSIZE 256
3
4 int
5 main() {
6     // open the file
7     char pathname[] = "../sample/file1";
8     FILE *fp = fopen(pathname, "r");
9     if (NULL == fp) {
10         fprintf(stderr, "Error opening file: %s\n", pathname);
11         return 1;
12     }
13
14     // establish a buffer
15     char buffer[BUFSIZE] = { 0 };
16     char *temp = NULL;
17
18     // read from the file
19     while ((temp = fgets(buffer, BUFSIZE, fp)) != NULL) {
20         printf("%s", buffer);
21     }
22
23     // close the file
24     int result = fclose(fp);
25     if (0 != result) {
26         fprintf(stderr, "Error closing file: %s\n", pathname);
27         return 1;
28     }
29
30     return 0;
31 }
```

Listing 6.1: src/06-file1.c

On Line 8 we see *fopen* used to open the file `../sample/file1` in read mode. This function returns the file pointer *fp*. We then test on line 9 to ensure that we actually received a file pointer. If *fp* is *NULL* this indicates we were unable to open the file. This could be because of permissions applied to the file, the file doesn't exist, or a myriad of other issues. Later in this chapter, we'll talk about using *fstat* to get information about the file before attempting to open it.

On line 19 we use *fgets* to read text from the file. The man page for *fgets* indicates:

```
fgets() reads in at most one less than size characters from stream and stores them
into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline
is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.
```

So *fgets* automatically makes sure we leave room for the terminating *NULL* byte for our string. Additionally, *fgets* returns *NULL* when it encounters an error or it is unable to read any more characters from the stream. This makes for an easy to construct loop that will read until no more characters remain.

Had we been writing text instead of reading text, we could have used *fputs*. This would have allowed us to construct a buffer and write that to the file.

On line 30 I close my file using *fclose*. It is important to do so especially when writing to files. Because the streams we're writing to are buffered by default, its important that all remaining writes get flushed from the buffer to the file. Additionally, closing the file releases memory back to the operating system. Had I not closed the above file, *valgrind* would have showed the following:

```
$ valgrind --leak-check=full --show-leak-kinds=all ./06-file1
==29442== Memcheck, a memory error detector
==29442== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==29442== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==29442== Command: ./06-file1
==29442==
This is a sample text file.
Not much to see here.
Move along.
==29442==
==29442== HEAP SUMMARY:
==29442==      in use at exit: 472 bytes in 1 blocks
```

```

==29442==    total heap usage: 3 allocs, 2 frees, 5,592 bytes allocated
==29442==
==29442== 472 bytes in 1 blocks are still reachable in loss record 1 of 1
==29442==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/\
vgpreload_memcheck-amd64-linux.so)
==29442==    by 0x48EEAAD: __fopen_internal (iofopen.c:65)
==29442==    by 0x48EEAAD: fopen@@GLIBC_2.2.5 (iofopen.c:86)
==29442==    by 0x109252: main (06-file1.c:8)
==29442==
==29442== LEAK SUMMARY:
==29442==    definitely lost: 0 bytes in 0 blocks
==29442==    indirectly lost: 0 bytes in 0 blocks
==29442==    possibly lost: 0 bytes in 0 blocks
==29442==    still reachable: 472 bytes in 1 blocks
==29442==    suppressed: 0 bytes in 0 blocks
==29442==
==29442== For lists of detected and suppressed errors, rerun with: -s
==29442== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Notice it is pointing to line 8 (*main (06-file1.c:8)*) of our file as the source of the memory leak. This is where we ran *fopen*.

6.3 Reading and Writing Binary Data

Reading and binary data to a file isn't much different than text data. However, a little bit of extra thought should go into this to ensure you don't shoot yourself in the foot. For example, in chapter 5 we talked about using a *struct* to represent a player. Wouldn't it be nice to save that player's state to file so that we can maintain state between games? Well actually you can. However, if you attempt to write the player *structs* as they are currently written, you may find that reading them back in doesn't work so well. This could be for a number of reasons. First off, certain members of the *struct* could be pointers. Without dereferencing to the actual data, you'll end up writing an address rather than the data itself. Additionally, a compiler may align on certain byte boundaries to make reading from and writing to the *struct* more efficient. Additionally it may also pad certain members in the process of that alignment. This will result in gaps between members. If these things are not taken into consideration, you can find yourself with corrupted data.

There are a number of ways around this. You could write each member individually ensuring you do so in a manner that allows you to determine the number of bytes written for members that may variable in size much like our items array was in 5.1. This is one of the safest options but requires additional work when reading and writing.

Additionally, you can instruct the compiler not to align structs via the `-fpack-struct` [5] compile time flag. You can also use `__attribute__((packed))` [2] in your *struct* definition to instruct the compiler to pack it. Let see an example of this:

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define STUDENTIDSIZE 16
6
7  typedef struct __attribute__((packed)) student {
8      float grade;
9      char studentID [STUDENTIDSIZE];
10 } student_t;
11
12 typedef struct __attribute__((packed)) class {
13     unsigned int numStudents;
14     student_t *students;
15 } class_t;
16
17 int
18 main() {
19     int returnCode = 0;
20
21     // create student grades
22     class_t myClass = { 0 };
23     myClass.numStudents = 2;
24     myClass.students = malloc(sizeof(*(myClass.students)) * \
25                             myClass.numStudents);
26     myClass.students[0].grade = 90.5;
27     strncpy(myClass.students[0].studentID, "rwell123456", STUDENTIDSIZE);
28     myClass.students[1].grade = 87.3;
29     strncpy(myClass.students[1].studentID, "rdash246810", STUDENTIDSIZE);
30
31     // open the file
32     char pathname[] = "../sample/file2";
33     FILE *fp = fopen(pathname, "w+b");
34     if (NULL == fp) {
35         fprintf(stderr, "Error opening file: %s\n", pathname);
36         returnCode = 1;
37         goto exitNow;
38     }
39
40     // save student grades to file
41     size_t items = fwrite(&myClass, sizeof(myClass) - sizeof(myClass.students),
42                          1, fp);
43     if (1 != items) {
44         fprintf(stderr, "Something went wrong when writing to: %s\n", pathname);
45         returnCode = 1;
46         goto exitNow;

```

```

47     }
48     for (unsigned int i = 0; i < myClass.numStudents; i++) {
49         items = fwrite(&(myClass.students[i]), sizeof(student_t), 1, fp);
50         if (1 != items) {
51             fprintf(stderr, "Something went wrong when writing to: %s\n",
52                 pathname);
53             returnCode = 1;
54             goto exitNow;
55         }
56     }
57
58     // clear out current class and students
59     free(myClass.students);
60     memset(&myClass, 0, sizeof(myClass));
61
62     // flush the output buffer
63     int result = fflush(fp);
64     if (0 != result) {
65         fprintf(stderr, "Unable to flush output buffer to: %s\n", pathname);
66         returnCode = 1;
67         goto exitNow;
68     }
69
70     // move to beginning of file
71     result = fseek(fp, 0, SEEK_SET);
72     if (0 != result) {
73         fprintf(stderr, "Unable to move to beginning of file: %s\n", pathname);
74         returnCode = 1;
75         goto exitNow;
76     }
77
78     // read in the class
79     items = fread(&myClass, sizeof(myClass) - sizeof(myClass.students), 1, fp);
80     if (1 != items) {
81         fprintf(stderr, "Something went wrong reading myClass from: %s\n",
82             pathname);
83         returnCode = 1;
84         goto exitNow;
85     }
86
87     // allocate for number of students indicated in file
88     myClass.students = malloc(sizeof(*(myClass.students)) * \
89         myClass.numStudents);
90
91     // read in the students
92     items = fread(&(myClass.students[0]), sizeof(student_t),
93         myClass.numStudents, fp);
94     if (myClass.numStudents != items) {
95         fprintf(stderr, "Something went wrong reading students from: "\
96             "%s\n", pathname);
97         returnCode = 1;

```



```
98     goto exitNow;
99 }
100
101 printf("Number of Students: %d\n", myClass.numStudents);
102 for (unsigned int i = 0; i < myClass.numStudents; i++) {
103     printf("\tStudent: %-18s Grade: %f\n", myClass.students[i].studentID,
104         myClass.students[i].grade);
105 }
106
107 exitNow:
108     // close the file
109     if (NULL != fp) {
110         result = fclose(fp);
111         if (0 != result) {
112             fprintf(stderr, "Error closing file: %s\n", pathname);
113             returnCode = 1;
114         }
115     }
116
117     if (NULL != myClass.students) {
118         free(myClass.students);
119     }
120
121     return returnCode;
122 }
```

Listing 6.2: src/06-file2.c

On line 33 I open a file with *fopen*. The mode I have chosen to use is "w+b". This means the file is opened in read and write mode. The "b" indicated binary mode even though its not required on POSIX systems.

As you can see on lines 7 and 12, the two *structs* will be packed. This means we shouldn't have an issue writing them directly to file. However, notice on line 14 that the *class struct* contain a pointer that could point to one or more *student structs*. To get around this, we'll first store all the members up to the *student struct* pointer. We can see this on lines 41 and 42. Notice how we use *fwrite* and pass it the address of *myClass*. The next parameter that *fwrite* is expecting is the size of the item we are trying to write. I subtract the size of the pointer from the size of the overall *struct* so that we write everything up to the pointer. This works fine because the pointer is the last item in the *struct*. The next parameter is the number of items to write. Since I'm only writing a single *struct* I enter 1. The last parameter is the file stream that I want to write to. For this, I enter the file pointer created on line 33.

On line 48 I begin a *for* loop based on the number of students I made entries for. I then use *fwrite* on line 49 to write each *student structs* to the file. I write them one at a time but could have just as easily written them all at once as well see later.

Now that all of the *fwrites* have been completed, in order to prove I can read the data back out, I free the *student structs* and wipe the *class struct* on lines 59 and 60. At this point, the data should only exist on disk. To be sure none of the data is still sitting in a buffer, I use *fflush* to instruct the system to flush everything to disk. I then use *fseek* on line 71 to move back to the beginning of the file.

I now begin reading from the file with *fread*. Notice again that I am only reading data up to but not including the pointer to the *student structs*. Once complete, I should know how many students I need to allocate for which I do on lines 88 and 89. Once I've allocated for my *student structs*, I read them back out of the file on lines 92 and 93. Notice this time around, instead of building a *for* loop and reading in each one at a time, I'm able to take advantage of the fact that the *malloc* that places the memory addresses right next to each other. I could have also written them this way.

Lets quickly validate that we haven't done anything too unusual that *valgrind* throws a fit.

```
$ valgrind --leak-check=full --show-leak-kinds=all ./06-file2
==48126== Memcheck, a memory error detector
==48126== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==48126== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==48126== Command: ./06-file2
==48126==
Number of Students: 2
Student: rwell123456      Grade: 90.500000
Student: rdash246810     Grade: 87.300003
==48126==
==48126== HEAP SUMMARY:
==48126==      in use at exit: 0 bytes in 0 blocks
==48126==    total heap usage: 5 allocs, 5 frees, 5,672 bytes allocated
==48126==
==48126== All heap blocks were freed -- no leaks are possible
==48126==
==48126== For lists of detected and suppressed errors, rerun with: -s
==48126== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

6.4 File Stats

When working with files, you may want to know certain things about a file before attempting to open it. You may also want to know things about a file after you've opened it. For example, it would be nice to know that a file actually exists, that the file is a regular file, the file size, who owns the file, or when it was last accessed. This information is return through a *stat struct* from either the *stat*, *lstat*, or *fstat* functions. These are system calls available on POSIX systems. Additionally, for

larger files, there is *stat64*, *fstat64*, and *lstat64*.

int **stat**(const char *pathname, struct stat *statbuf);

int **fstat**(int fd, struct stat *statbuf);

int **lstat**(const char *pathname, struct stat *statbuf);

The *stat struct* has the following members:

The *stat structure*

All of these system calls **return** a *stat structure*, which contains the following fields:

```

struct stat {
    dev_t      st_dev;          /* ID of device containing file */
    ino_t      st_ino;         /* Inode number */
    mode_t     st_mode;        /* File type and mode */
    nlink_t    st_nlink;       /* Number of hard links */
    uid_t      st_uid;         /* User ID of owner */
    gid_t      st_gid;         /* Group ID of owner */
    dev_t      st_rdev;        /* Device ID (if special file) */
    off_t      st_size;        /* Total size, in bytes */
    blksize_t  st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t   st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim; /* Time of last access */
    struct timespec st_mtim; /* Time of last modification */
    struct timespec st_ctim; /* Time of last status change */

    #define st_atime st_atim.tv_sec      /* Backward compatibility */
    #define st_mtime st_mtim.tv_sec
    #define st_ctime st_ctim.tv_sec

};

```

Listing 6.3: chapters/ch6–stat.txt

Lets look at an example:

```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/stat.h>
4 #include <unistd.h>
5 #include <time.h>
6
7 int
8 main() {

```

```

9  const char *pathname = "../sample/file1";
10 int result;
11 struct stat statbuf = { 0 };
12
13 // fill stat struct
14 result = stat(pathname, &statbuf);
15 if (-1 == result) {
16     fprintf(stderr, "Could not run stat on: %s\n", pathname);
17     return 1;
18 }
19
20 printf("Pathname:                %s\n", pathname);
21 printf("Is the file a regular file: %s\n",
22 S_ISREG(statbuf.st_mode) ? "true" : "false");
23 printf("Is the file a directory:    %s\n",
24 S_ISDIR(statbuf.st_mode) ? "true" : "false");
25 printf("User ID of owner:           %d\n", statbuf.st_uid);
26 printf("Group ID of group owner:    %d\n", statbuf.st_gid);
27 printf("Size of file:               %ld bytes\n", statbuf.st_size);
28 printf("Last status change:         %s", ctime(&statbuf.st_ctime));
29 printf("Last file access:           %s", ctime(&statbuf.st_atime));
30 printf("Last file modification:     %s", ctime(&statbuf.st_mtime));
31
32 return 0;
33 }

```

Listing 6.4: src/06-file3.c

On line 11 I create a *stat struct* and call it *statbuf*. I then call *stat* on line 14 to fill it. The struct now contains information about the pathname that I passed to it.

The *stat struct* contains information about the mode. This information is stored in *st_mode* which is a bit field. Although we haven't talked about bit fields, they are essentially a data structure that stores information by individually setting bits. In the context of the *stat struct* each of the bits stands for something like "is this a regular file", "is this a directory", "is this a character device", "is this a named pipe", "is this a symbolic link", or "is this a socket". Knowing which bit answers which question can be difficult. Fortunately, there are a number of macros that help. On line 22 I use the *S_ISREG* macro to determine if the file is a regular file. I then use *S_ISDIR* on line 24 to determine if this is a directory. To lookup additional macros specific to the *stat struct* type man 7 *inode* into the terminal. Additionally there is more information about each of the different fields within the *stat struct*.

The *stat struct* has *timespec structs* that indicate access, modify, and last change times. In order to print them out nicely, I am utilizing the *ctime* function on lines 28 - 30.

Let take a look at the output:

```
$ ./06-file3
Pathname:                ../sample/file1
Is the file a regular file: true
Is the file a directory:  false
User ID of owner:        1000
Group ID of group owner:  1000
Size of file:            62 bytes
Last status change:      Fri Jul 24 18:06:13 2020
Last file access:        Fri Jul 24 18:06:13 2020
Last file modification:  Fri Jul 24 18:06:13 2020
```




7. Concurrency

Up until this point the programs that we've seen have only been able to execute a single task at any given time. However, there are times when you may want your program to handle multiple things at the same time. This is often the case with networked applications. You may need to listen for incoming connections while processing existing connections. There are a number of ways to do this and in this chapter we'll talk about a two. One way in which you can do this is to **fork** a process. When a process is *forked*, a child process with duplicate memory of the parent process is spun up. These two processes can work independently of one another. Another way in which a program may execute multiple tasks at a time is to create another **thread**. Threads are independent streams of instructions. In other words, a single process can have multiple threads where each thread is executing different instructions at the same time.

7.1 Forking

In our first example of handling multiple tasks at once we'll use the UNIX system call *fork*. Its fairly simple and straightforward to use and as a result has been incorporated into many different projects. For example, in certain versions of the Apache Web Server, they use a pre-fork model where they essentially have multiple *forked* processes waiting for a request to come in. As mentioned earlier, each process (parent and child) has a copy of the memory as it was when the *fork* occurred. This means there is an initial cost involved when the *fork* occurs to copy memory from the parent process into the child process. So by *forking* before a request comes in, they incur this cost early on. Additionally, once a request does come in and its handed to a child process, if something goes wrong with that request, it only affects that process instead of affecting the entire server. Of course

this also comes at the cost of higher resource utilization since each process has a separate copy of the server's memory at the time they were *forked*.

Lets take a look at a simple example of using *fork*:

```

1  #define _POSIX_C_SOURCE 199309L
2  #include <stdio.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5  #include <time.h>
6
7  void
8  echo(const char *, const char *);
9
10 int
11 main() {
12     const char *shared = "I am a shared string!";
13
14     pid_t myPid;
15     pid_t parentPid;
16     pid_t newPid;
17
18     myPid = getpid();
19     parentPid = getppid();
20     printf("Program begins with:\n\tPID: %d\n\tParentPid: %d\n",
21           myPid, parentPid);
22
23     newPid = fork();
24
25     if (-1 == newPid) {
26         /* error */
27         perror("Unable to fork.");
28     }
29     else if (0 == newPid) {
30         /* child */
31         parentPid = getppid();
32         printf("Child Process:\n\tPID: %d\n\tParentPid: %d\n",
33               newPid, parentPid);
34         echo("CHILD", shared);
35     }
36     else {
37         /* parent */
38         printf("My Child's PID: %d\n", newPid);
39         echo("PARENT", shared);
40     }
41 }
42
43 void
44 echo(const char *name, const char *saying) {
45     /* set timer */

```

```
46  struct timespec tsReq, tsRem;
47  tsReq.tv_sec = 0;
48  tsReq.tv_nsec = 200000000;
49
50  for (int i = 0; i < 5; i++) {
51      printf("%s: %s\n", name, saying);
52      nanosleep(&tsReq, &tsRem);
53  }
54 }
```

Listing 7.1: src/07-concurrency1.c

On lines 14 - 16 we can see that we are creating variables to store the various process IDs (PID) that we'll see. We then fill two of those PIDs on lines 18 and 19.

On line 23, we perform a *fork*. According to the man page:

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and *errno* is set appropriately.

This is why we follow up the *fork* with an if statement to detect which process we are working with. As you can see on lines 31 - 34, the child is merely looking up who its parent is and then calling the *echo* function that starts on line 43. On line 39 the parent process calls the same *echo* function so that we can see them executing simultaneously.

Since the execution of the *echo* function happens so quickly, I've added a *nanosleep* to slow it down. Without this slow down, the parent process often finishes all of its prints before memory has been replicated into the child process and it begins printing as well.

```
$ ./07-concurrency1
Program begins with:
PID: 29803
ParentPid: 3866
My Child's PID: 29804
PARENT: I am a shared string!
Child Process:
PID: 0
ParentPid: 29803
CHILD: I am a shared string!
PARENT: I am a shared string!
CHILD: I am a shared string!
PARENT: I am a shared string!
```

```
CHILD: I am a shared string!
PARENT: I am a shared string!
CHILD: I am a shared string!
PARENT: I am a shared string!
CHILD: I am a shared string!
```

Notice, that our program starts with a PID of 29803 and once *fork* has taken place, the parent reports that its child has PID 29804. Also notice that the child indicates it's parent is PID 29803. Lastly notice that both processes go about their business printing to the screen. Although this looks like a back and forth movement, this is really artificially created based on the sleeps that were added and the low CPU utilization on my computer when I ran it.

7.2 Multithreading

As mentioned, using multiple **thread** is one way in which a program can execute different tasks concurrently. *Threads* are lighter weight than using *fork* because each thread does not get its own memory. This makes communicating between threads easier but can also lead to *race* conditions where values are updated at the same time by two different threads. Extra care must be taken to protect shared memory and resources since its easy to mess up. To the beginner, writing *multithreaded* code doesn't sound overly difficult. However, doing it correctly can be difficult to get working correctly. Without extensive testing, its easy to write code that will *deadlock* under certain conditions. Whatever this condition is, it leads to a state where at least one thread gets stuck waiting for a resource that will never be available and as a result, it cannot perform any work.

There are two different libraries you'll often see used for multi-threading in C; `pthread.h` and `threads.h`. `threads.h` was made a part of the C standard library in C11 and for portability reasons, should be what you use going forward. However, `pthread.h` has been around for a long time for POSIX systems. Due to this, you'll still see it taught and used. In fact, even for current POSIX systems, `threads.h` is in many ways just a wrapper around `pthread.h` and will require you to link in `pthread.h` during compilation. Although `threads.h` was released as a part of the C11 standard, my Ubuntu 20.04 machine did not come with manpages for it. It does however, have extensive documentation for the various functions in use within `pthread.h`. If portability is a concern for you, I suggest you use `threads.h`. If you know you'll be writing programs only for POSIX systems, feel free to use `pthread.h` as well. I will attempt to expose you to both.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <threads.h>
4 #include "09-queue.h"
5
6 #define ITEMS 10
7 #define THREADCNT 5
```

```

8  #define UNUSED(x) (void)(x)
9
10 queue_t *q;
11 mtx_t qMtx;
12
13 void
14 destroyJob(void *);
15
16 static int
17 doWork(void *);
18
19 int
20 main() {
21     int returnCode = 0;
22     int result;
23     void(*dj)(void *) = destroyJob;
24     thrd_t *threads = NULL;
25     char work[ITEMS][2] = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J"};
26
27     /* create and initialize mutex */
28     mtx_t qMtx = { 0 };
29     if (thrd_success != mtx_init(&qMtx, mtx_plain )) {
30         fprintf(stderr, "Unable to initialize queue mutex.");
31         returnCode = 1;
32         goto exitNow;
33     }
34
35     /* add work to a queue */
36     q = createQueue();
37     printf("Enqueue work.\n");
38     for (int i = 0; i < ITEMS; i++) {
39         result = mtx_lock(&qMtx);
40         if (thrd_success == result) {
41             enqueue(q, (void *)work[i]);
42             mtx_unlock(&qMtx);
43         }
44     }
45
46     /* create threads to begin work */
47     threads = malloc(THREADCNT * sizeof(*threads));
48     for (int i = 0; i < THREADCNT; i++) {
49         printf("Creating Threads.\n");
50         if (thrd_success != thrd_create(&threads[i], doWork, NULL)) {
51             fprintf(stderr, "Unable to create thread.\n");
52             returnCode = 1;
53             goto exitNow;
54         }
55     }
56
57     /* wait for threads to exit */
58     for (int i = 0; i < THREADCNT; i++) {

```

```

59     thrd_join(threads[i], &result);
60     printf("Thread %d joined with result: %d\n", i, result);
61 }
62
63 exitNow:
64     destroyQueue(&q, dj);
65     mtx_destroy(&qMtx);
66     free(threads);
67     return returnCode;
68 }
69
70 static int
71 doWork(void *nothing) {
72     UNUSED(nothing);
73     int result;
74     void *job;
75
76     /* ID thread */
77     thrd_t thisThread = thrd_current();
78     for(;;) {
79         result = mtx_lock(&qMtx);
80         if (thrd_success == result) {
81             job = dequeue(q);
82             mtx_unlock(&qMtx);
83             if (NULL != job) {
84                 /* perform work */
85                 printf("%lu: dequeue: %s\n", thisThread, (char *)job);
86             } else {
87                 printf("%lu: No work.\n", thisThread);
88                 break;
89             }
90             thrd_yield();
91         }
92     }
93     /* return 0 */
94     thrd_exit(0);
95 }
96
97 void
98 destroyJob(void *job) {
99     UNUSED(job);
100 }

```

Listing 7.2: src/07-concurrency2.c

7.2.1 Threads.h

While this example isn't too involved, there are enough elements that you can see multi-threading in action. The first thing to notice is that on line 3 we are using *threads.h*. This means we are using

the C standard threading library. Lets follow the life of the actual threads. On line 24, we see a *thrd_t struct* pointer declared. On line 47, space for 5 threads is allocated and then on line 50 they are put to work. During the *thrd_create* the individual threads are given a function to perform. In this case, each thread will run the *doWork* function. The last argument in the *thrd_create* function is the argument that will be passed to the *doWork* function. In this case, we are not sending anything so it is set to NULL. The function *doWork* has a prescribed function prototype. It must be of the form: *int (*)(void)*. This means the function takes in a *void ** as an argument and returns an integer. This is different than the function prototype you'll pass to *pthread_create*. It takes the form: *void (*)(*)(void *)*. This means that it takes in a *void ** and returns a *void **.

So execution for each threads now starts at the *doWork* function on line 70. On line 77 we look up the thread ID. This is only needed as it was nice to use during printing to see which thread it was that was performing work. On line 86 we see the *thrd_yield* function used. This does little in the context of this program but is a way for the thread to indicate to the OS that its ok to schedule it at a lower priority. This could be due to the thread waiting for an action that may take awhile. On line 90 we see *thrd_exit* being called. This is how the thread returns values from the function. So now that the thread is done with all of its work, its up to the main thread to call *thrd_join* on line 59. This is a blocking action and the main thread will wait here for the thread it is joining to complete. One by one the threads finish their work, exit, and are joined by the main thread.

7.2.2 Mutex

Another aspect of this program is the use of a **mutex** (Mutual Exclusion). In multithreaded applications, its important to protect shared memory that multiple threads could potentially access or change at the same time. In the context of this example, there is a *queue* that has work being enqueued by the main thread and dequeued by the worker threads. Since it could cause issues if multiple threads tried to dequeue a job at the same time, we use a *mutex* to protect it. In order for a thread to access the *queue* they have to first lock the *mutex*. Only one thread can lock it at a time so if the *mutex* is already locked, other threads will wait. Once it is unlocked, another thread will attempt to lock it.

Since each thread must have access to the *mutex* it is created globally on line 11. On line 28 we zero out the *mutex* and then initialize it on line 29. We declare the *mutex* to be a plain type (non-recursive - can only be locked once). For the purpose of this program, the main thread doesn't need to lock the *mutex* when enqueueing work since the other threads don't exist yet. However, its left as a reminder that any thread, to include the main thread, needs to use a *mutex* when accessing a shared resource. On lines 79 and 82 we see the threads also utilize the *mutex* before accessing the *queue*. Lastly, on line 65 we destroy the *mutex* releasing any memory it is holding from when it was initialized.

Let's see what running this example looks like.

Enqueue work.

```
Creating Threads.
Creating Threads.
Creating Threads.
140045949937408: dequeue: A
Creating Threads.
140045941544704: dequeue: B
Creating Threads.
140045864400640: dequeue: C
140045864400640: dequeue: E
140045949937408: dequeue: F
140045856007936: dequeue: D
140045949937408: dequeue: I
140045941544704: dequeue: H
140045941544704: No work.
140045847615232: dequeue: J
140045847615232: No work.
140045856007936: No work.
140045949937408: No work.
140045864400640: dequeue: G
140045864400640: No work.
Thread 0 joined with result: 0
Thread 1 joined with result: 0
Thread 2 joined with result: 0
Thread 3 joined with result: 0
Thread 4 joined with result: 0
```

As we can see, although the *queue* releases the jobs in the same order they were enqueued, the individual threads printed them in a different order. This is because each individual thread is operating independently of the others and may get scheduled differently by the OS. These little scheduling differences result in a few of the jobs being printed out in a different order.

7.2.3 Condition Variables

In the previous example, the threads continued to check if there was work to be done and as soon as there wasn't, they closed down. It demonstrated the use of *threads* and using a *mutex* to protect a shared resource (the queue). However, this may not be very practical. In the next example, we'll not only switch to using the *pthread* library but we'll also use **condition variables** to signal the threads when they need to "wake" up and perform some work. There are number of functions that we could use to interact with our *condition variable*. The following are taken directly from the man pages:

pthread_cond_init - initializes the condition variable cond.

pthread_cond_destroy - destroys a condition variable, freeing the resources it might hold.

pthread_cond_wait - atomically unlocks the mutex (as per `pthread_unlock_mutex`) and waits for the condition variable cond to be signaled. The thread execution is suspended and does not consume any CPU time until the condition variable is signaled. The mutex must be locked by the calling thread on entrance to `pthread_cond_wait`. Before returning to the calling thread, `pthread_cond_wait` re-acquires mutex (as per `pthread_lock_mutex`).

pthread_cond_signal - restarts one of the threads that are waiting on the condition variable cond. If no threads are waiting on cond, nothing happens. If several threads are waiting on cond, exactly one is restarted, but it is not specified which.

pthread_cond_broadcast - restarts all the threads that are waiting on the condition variable cond. Nothing happens if no threads are waiting on cond.

As we can see, threads will utilize a *mutex* along with *pthread_cond_wait*. Once waiting, they will essentially stop consuming CPU cycles. Another thread, most likely the main thread, will then either awaken a single thread with *pthread_cond_signal* or all waiting threads with *pthread_cond_broadcast*.

Thread Pool

The last example and the following example contain parts that make up a very simple bare bones version of a **thread pool** where there are threads laying in wait for work to be performed. Lets take a look at how this pans out in our next example.

```
1  #define _XOPEN_SOURCE 500
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <pthread.h>
5  #include <time.h>
6  #include <stdbool.h>
7  #include <unistd.h>
8
9  #include "09-queue.h"
10
11 #define UNUSED(x) (void)(x)
12 #define NUMTHREADS 8
13 #define NUMLISTS 10
14 #define NUMITEMS 20
15
16 static bool processJobs = true;
17 static queue_t *q;
18 static pthread_mutex_t mtxJobs = { 0 };
19 static pthread_mutex_t mtxQ = { 0 };
20 static pthread_cond_t condJobs = { 0 };
```

```
21
22 void *
23 doWork(void *);
24
25 void
26 destroyJob(void *);
27
28 int
29 main() {
30     int retVal = 0;
31     int result;
32     void(*dj)(void *) = destroyJob;
33     time_t clock;
34     time(&clock);
35     srand(clock);
36
37     /* create jobs queue */
38     q = createQueue();
39
40     /* set up mutexes */
41     pthread_mutex_init(&mtxJobs, NULL);
42     pthread_mutex_init(&mtxQ, NULL);
43
44     /* set up a condition variable */
45     pthread_cond_init(&condJobs, NULL);
46
47     /* create threads */
48     pthread_t threads[NUMTHREADS] = { 0 };
49     for (int i = 0; i < NUMTHREADS; i++) {
50         result = pthread_create(&(threads[i]), NULL, doWork, NULL);
51         if (result != 0) {
52             fprintf(stderr, "Unable to create thread.\n");
53             retVal = result;
54             goto exitNow;
55         }
56     }
57
58     /* give the threads time to get set */
59     sleep(2);
60
61     long jobs[NUMLISTS][NUMITEMS];
62     for (int l = 0; l < NUMLISTS; l++) {
63         /* create list of random numbers */
64         for (int i = 0; i < NUMITEMS; i++) {
65             jobs[l][i] = random() % 1000;
66         }
67
68         result = pthread_mutex_lock(&mtxQ);
69         if (0 != result) {
70             continue;
71         }
```

```

72
73     result = enqueue(q, jobs[l]);
74     pthread_mutex_unlock(&mtxQ);
75     if (result != true) {
76         fprintf(stderr, "Unable to enqueue list.\n");
77     } else {
78         printf("Enqueuing and signaling.\n");
79         pthread_cond_signal(&condJobs);
80     }
81 }
82
83 sleep(2);
84 processJobs = false;
85 printf("Send broadcast.\n");
86 pthread_cond_broadcast(&condJobs);
87 for (int i = 0; i < NUMTHREADS; i++) {
88     printf("Joining %d.\n", i);
89     pthread_join(threads[i], NULL);
90 }
91
92 exitNow:
93     destroyQueue(&q, dj);
94     pthread_mutex_destroy(&mtxJobs);
95     pthread_mutex_destroy(&mtxQ);
96     pthread_cond_destroy(&condJobs);
97     return retVal;
98 }
99
100 void *
101 doWork(void *nothing) {
102     UNUSED(nothing);
103     pthread_t pthreadID = pthread_self();
104     int result;
105     unsigned int cnt;
106
107     while (true == processJobs) {
108
109         /* lock job mutex */
110         result = pthread_mutex_lock(&mtxJobs);
111         if (0 != result) {
112             fprintf(stderr, "Failure to gain lock.\n");
113             pthread_exit(NULL);
114         }
115         if (false == processJobs) {
116             break;
117         }
118
119         /* unlock job mutex and begin waiting on jobs condition variable */
120         result = pthread_cond_wait(&condJobs, &mtxJobs);
121         if (0 != result) {
122             fprintf(stderr, "Failure to wait for condition variable.\n");

```

```

123     pthread_exit(NULL);
124 }
125
126 /* unlock the job mutex again */
127 result = pthread_mutex_unlock(&mtxJobs);
128
129 /* check the queue until empty */
130 for (;;) {
131
132     /* lock queue mutex */
133     result = pthread_mutex_lock(&mtxQ);
134     if (0 == result) {
135         long *list = dequeue(q);
136
137         /* unlock queue mutex */
138         result = pthread_mutex_unlock(&mtxQ);
139
140         if (NULL == list) {
141             break;
142         }
143
144         cnt = 0;
145         for (int i = 0; i < NUMITEMS; i++) {
146             if (0 == list[i] % 2) {
147                 cnt++;
148             }
149         }
150         printf("%d - %d/%d items were even.\n", pthreadID, cnt, NUMITEMS);
151     } else {
152         /* lock failure */
153         break;
154     }
155 }
156 pthread_exit(NULL);
157 }
158
159 void
160 destroyJob(void *job) {
161     UNUSED(job);
162 }
163

```

Listing 7.3: src/07-concurrency3.c

On lines 18 and 19, we can see there are two *mutexes* (*pthread_mutex_t*) being declared globally so that each thread has access to them. Additionally, we have a *condition variable* (*pthread_cond_t*) being created as well. One *mutex* will be used in conjunction with this *condition variable*. All three are initialized on lines 41 - 45 and destroyed on lines 94 - 96.

On line 50 we can see each of our different threads being created and added to an array. The threads

are told to perform the function `doWork` which takes no input and returns nothing either.

Inside of the `doWork` function we can see that the thread looks up its ID utilizing `pthread_self` on line 103. We can also see on line 107 where the thread looks to see if it should keep looping.

Lines 110 and 120 are the bread and butter of what will keep the threads from wasting CPU cycles when there is no work to be performed. In this case, they lock the `mtxJobs mutex` and then begin waiting on the *condition variable* by using `pthread_cond_wait`. This unlocks the `mtxJobs mutex` and puts the thread in a wait state.

Once the main thread determines there is work to be performed, on line 79 it utilizes `pthread_cond_signal` to signal one of the waiting threads to wake up. When this is done, the thread has to re-unlock the `mtxJobs mutex`. It then proceeds to perform work until there is nothing for it to do. At this point, it checks to see if it should continue looping and if so, it goes into a wait state once again.

Once the main thread no longer has any work to assign, on line 84 it sets the `processJobs` flag to false and on line 86 sends a broadcast to all the threads with `pthread_cond_signal`. At this point, each thread wakes up and checks to see if there is any work remaining, sees that `processJobs` flag is no longer set, and returns by running `pthread_exit` on line 157.

Now that the work has been created, assigned, and hopefully performed, the main thread performs a `pthread_join` on line 89 for each of the threads.

Lets see what the results look like.

```
$ ./07-concurrency3
Enqueing and signaling.
Enqueing and signaling.
Enqueing and signaling.
Enqueing and signaling.
139982923020032 - 9/20 items were even.
139982923020032 - 7/20 items were even.
Enqueing and signaling.
139982923020032 - 7/20 items were even.
139982931412736 - 9/20 items were even.
139982914627328 - 13/20 items were even.
Enqueing and signaling.
Enqueing and signaling.
139982897841920 - 11/20 items were even.
139982881056512 - 12/20 items were even.
139982881056512 - 9/20 items were even.
Enqueing and signaling.
Enqueing and signaling.
```

```
139982872663808 - 8/20 items were even.  
139982914627328 - 10/20 items were even.  
Enqueing and signaling.  
Send broadcast.  
Joining 0.  
Joining 1.  
Joining 2.  
Joining 3.  
Joining 4.  
Joining 5.  
Joining 6.  
Joining 7.
```

A quick glance at the above output and we can see that ten jobs were created and the work was performed by 6 different threads. Your output will undoubtedly be different than mine.

8. Networking

In this chapter I'll walk through a couple of examples using *sockets* to send information back and forth between a client and a server application. However, I am not going to go into any great depth. What I actually suggest is to visit Brian "Beej Jorgensen" Hall's website at <http://beej.us/guide/bgnet/>. His Guide to Network Programming [3] is a much more thorough resource. I personally look at it nearly every time I use *sockets* and I've had other programmers mention they do the same. Go browse it now. Going forward, I'm going to assume you have.

First things first, you need to make a few decisions about the application you're building. You'll need to decide whether you will be using TCP or UDP. You'll also need to decide whether you'll be working with IPv4, IPv6, or both.

Additionally, there is the concept of endianness. Most end systems are little-endian these days and most networks utilize big-endian. You will therefore often find yourself translating between the two when sending data between systems.

Lets jump into a very simple example of a threaded server application talking to a single remote client:

```
1 #include <arpa/inet.h>
2 #include <stdio.h>
3 #include <sys/socket.h>
4 #include <sys/types.h>
5 #include <string.h>
6 #include <unistd.h>
7 #define BUFFERSIZE 32
```

```

8
9 int
10 main() {
11     int sockFD, remoteFD;
12     int result;
13     char address[INET_ADDRSTRLEN + 1] = { 0 };
14     char recvBuf[BUFFERSIZE] = { 0 };
15     const char *temp;
16
17     ssize_t bytes;
18     struct sockaddr_in addr = { 0 };
19     struct sockaddr_in remote = { 0 };
20     socklen_t remoteSize = sizeof(remote);
21
22     /* create a socket */
23     sockFD = socket(AF_INET, SOCK_STREAM, 0);
24     if (-1 == sockFD) {
25         fprintf(stderr, "Unable to create socket.\n");
26         return -1;
27     }
28
29     addr.sin_family = AF_INET;
30     addr.sin_port = htons(8888);
31     addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
32
33     /* allow for address reuse, great for testing */
34     int reuse = 1;
35     result = setsockopt(sockFD, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(reuse));
36     if (0 != result) {
37         fprintf(stderr, "Unable to set socket option for address reuse.");
38     }
39
40     /* bind the socket */
41     result = bind(sockFD, (struct sockaddr *)&addr, sizeof(addr));
42     if (0 != result) {
43         fprintf(stderr, "Unable to bind.\n");
44         goto exitNow;
45     }
46
47     /* listen for incoming connections */
48     result = listen(sockFD, 1);
49     if (0 != result){
50         fprintf(stderr, "Unable to listen.\n");
51         goto exitNow;
52     }
53
54     /* accept connection */
55     remoteFD = accept(sockFD, (struct sockaddr *)&remote, &remoteSize);
56     if (-1 == remoteFD) {
57         fprintf(stderr, "Accept error.\n");
58         goto exitNow;

```



```

59     }
60
61     temp = inet_ntop(AF_INET, (void *)&(remote.sin_addr), address,
62                     INET_ADDRSTRLEN);
63     if (NULL == temp) {
64         fprintf(stderr, "Unable to convert address.\n");
65     } else {
66         int port = ntohs(remote.sin_port);
67         printf("Accepted a connection from: %s:%d\n", address, port);
68     }
69
70     for (;;) {
71         bytes = send(remoteFD, "Marco\n", 6, 0);
72         if (-1 == bytes) {
73             fprintf(stderr, "Unable to send.\n");
74             break;
75         }
76
77         bytes = recv(remoteFD, recvBuf, BUFFERSIZE - 1, 0);
78         if (-1 == bytes) {
79             fprintf(stderr, "Error receiving.\n");
80             break;
81         } else if (0 == bytes) {
82             printf("Remote closed connection.\n");
83             break;
84         }
85
86         if ((result = strncmp("Polo", recvBuf, 4)) != 0) {
87             break;
88         }
89     }
90     close(remoteFD);
91
92 exitNow:
93     close(sockFD);
94     return 0;
95 }

```

Listing 8.1: src/08-network1_server.c

AS we can see on lines 18-19 we create two *sockaddr_in structs* to hold our addresses both our local *socket* address and remote connections.

On line 23 we use the *socket* function to create a file descriptor. In this instance, we are passing in AF_INET (IPv4) and SOCK_STREAM (TCP).

In lines 29-31 we are manually packing our address. We once again specify we will be using IPv4. We also use *htons* (Host to Network Short) to convert the port number to big-endian. Lastly we use *htonl* (Host to Network Long) to convert the loopback address to big-endian. We could have used

inet_pton (Presentation to Network) to convert the ip address (string) to a network address as well.

On line 35 we use *setsockopt* to allow the address to be used again. In this scenario, it allows us to relaunch our server application quickly after it was shutdown. This is great for testing since you don't have to wait for the previous listener to time out before launching a new one. *SOL_SOCKET* indicates that the option is at socket level and instead of another protocol and the specific option we are adjusting is *SO_REUSEADDR*.

The man page for *bind* indicates:

When a socket is created with *socket(2)*, it exists in a name space (address family) but has no address assigned to it. *bind()* assigns the address specified by *addr* to the socket referred to by the file descriptor *sockfd*

This is exactly what we do on line 41 where we *bind* the address we created to the socket.

On line 48 we begin listening on the socket. This allows for us to *accept* new connections on line 55. The address of the incoming connection ends up in the *remote struct*. Once filled, we use *inet_ntop* (Network to Presentation) on line 61 to convert the address into something you and I would recognize.

Now that we've accepted a new TCP connection, we are able to *send* and *recv* over the new file descriptor as we do on lines 70 and 76.

Once our communications are complete we *close* the file descriptors on lines 89 and 92.

Lets see what this looks like when we run it.

```
$ ./10-network1-server
Accepted a connection from: 127.0.0.1:39412
```

In one tab we fire up our server and once our client (in another tab) connects, we see its IP address and source port. We can verify this by running `netstat -plantu | grep 8888`.

```
tcp    0    0 127.0.0.1:8888    127.0.0.1:39412    ESTABLISHED 9325/./10-network1-
```

Now lets see what this looks like from the client:

```
$ nc localhost 8888
Marco
Polo
```

Marco
NotPolo

In another tab, we connect to the server via the program `netcat`. We immediately receive "Marco" from the server and respond with "Polo". Once again we receive "Marco" from the server and this time respond with "NotPolo". When the server sees this, line 85 causes it to break out of the loop and close the connection. After hitting <ENTER> in `netcat` we are returned to the prompt.

8.1 getaddrinfo

Another way of specifying the address that we want our server to run on is to use **getaddrinfo**. The man page has a nice explanation of why you might use this function rather than the manual method we used in the last example:

Given node and service, which identify an Internet host and a service, `getaddrinfo()` returns one or more `addrinfo` structures, each of which contains an Internet address that can be specified in a call to `bind(2)` or `connect(2)`. The `getaddrinfo()` function combines the functionality provided by the `gethostbyname(3)` and `getservbyname(3)` functions into a single interface, but unlike the latter functions, `getaddrinfo()` is reentrant and allows programs to eliminate IPv4-versus-IPv6 dependencies.

Let take a look at it in action with our next example:

```
1 #define _POSIX_C_SOURCE 200112L
2 #include <errno.h>
3 #include <netdb.h>
4 #include <signal.h>
5 #include <stdbool.h>
6 #include <stdio.h>
7 #include <string.h>
8 #include <sys/socket.h>
9 #include <sys/types.h>
10 #include <unistd.h>
11 #define BUFFERSIZE 8
12
13 static void
14 handleConnection(int);
15
16 static void
17 sigHandler(int sig);
18
19 bool keepRepeating = true;
```

```

20
21 int
22 main() {
23     int sockFD;
24     int result;
25     int returncode = 0;
26     struct addrinfo hints = { 0 };
27     struct addrinfo *res = NULL;
28     struct addrinfo *adapter = NULL;
29
30     /* setup sigaction handling */
31     struct sigaction sa;
32     memset(&sa, 0, sizeof(sa));
33     sa.sa_handler = &sigHandler;
34     sa.sa_flags = 0;
35     if (sigaction(SIGINT, &sa, 0) != 0) {
36         fprintf(stderr, "Sigaction failure.");
37         return -1;
38     }
39
40     hints.ai_family = AF_UNSPEC;
41     hints.ai_socktype = SOCK_STREAM;
42     hints.ai_flags = AI_PASSIVE;
43
44     /* generate addrinfo structs based on hits */
45     result = getaddrinfo(NULL, "8888", &hints, &res);
46     if (0 != result) {
47         fprintf(stderr, "Unable to get address info.\n");
48         returncode = -1;
49         goto exitNow;
50     }
51
52     /* loop through the addresses attempting to bind */
53     for (adapter = res; NULL != adapter; adapter = adapter->ai_next) {
54         /* create the socket */
55         sockFD = socket(adapter->ai_family, adapter->ai_socktype,
56             adapter->ai_protocol);
57         if (-1 == sockFD) {
58             continue;
59         }
60
61         /* allow for address reuse, great for testing */
62         int reuse = 1;
63         result = setsockopt(sockFD, SOL_SOCKET, SO_REUSEADDR, &reuse,
64             sizeof(reuse));
65         if (0 != result) {
66             fprintf(stderr, "Unable to set socket option for address reuse.");
67         }
68
69         /* bind the socket */
70         result = bind(sockFD, adapter->ai_addr, adapter->ai_addrlen);

```

```

71     if (0 != result) {
72         close(sockFD);
73         continue;
74     }
75     break;
76 }
77
78 /* we were unable to successfully create a socket and bind to it */
79 if (-1 == sockFD || 0 != result) {
80     returncode = -1;
81     goto exitNow;
82 }
83
84 /* listen for incoming connections */
85 result = listen(sockFD, 64);
86 if (0 != result){
87     fprintf(stderr, "Unable to listen.\n");
88     returncode = -1;
89     goto exitNow;
90 }
91
92 int remoteFD;
93 pid_t pid;
94 struct sockaddr_in remote = { 0 };
95 socklen_t remoteSize = sizeof(remote);
96
97 printf("Hit <CTRL+C> to stop server.\n");
98 for (;;) {
99     /* accept connection */
100    remoteFD = accept(sockFD, (struct sockaddr *)&remote, &remoteSize);
101    if (-1 == remoteFD) {
102        /* EINTR is most likely from <CTRL+C> */
103        if (EINTR == errno) {
104            printf("Shutting down...\n");
105        } else {
106            fprintf(stderr, "Accept error.\n");
107        }
108        break;
109    }
110
111    /* fork connection */
112    pid = fork();
113    if (0 == pid) {
114        /* child closes socket it doesn't require */
115        close(sockFD);
116        handleConnection(remoteFD);
117        break;
118    } else {
119        /* parent closes socket it doesn't require */
120        close(remoteFD);
121    }

```

```

122     }
123
124     close(sockFD);
125 exitNow:
126     /* free addrinfo struct */
127     freeaddrinfo(res);
128     return returncode;
129 }
130
131 static void
132 handleConnection(int remoteFD) {
133     char recvBuf[BUFFERSIZE] = { 0 };
134     int result;
135     ssize_t bytes;
136
137     /* send and receive */
138     for (;keepRepeating;) {
139         bytes = send(remoteFD, "ping\n", 6, 0);
140         if (-1 == bytes) {
141             fprintf(stderr, "Unable to send.\n");
142             break;
143         }
144
145         bytes = recv(remoteFD, recvBuf, BUFFERSIZE - 1, 0);
146         if (-1 == bytes) {
147             fprintf(stderr, "Error receiving.\n");
148             break;
149         } else if (0 == bytes) {
150             printf("Remote closed connection.\n");
151             break;
152         }
153
154         if ((result = strncmp("pong", recvBuf, 4)) != 0) {
155             break;
156         }
157     }
158
159     close(remoteFD);
160 }
161
162 static void
163 sigHandler(int sig) {
164     if (SIGINT == sig) {
165         keepRepeating = false;
166     }
167 }

```

Listing 8.2: src/08-network2_server.c

I've tried to be generous with the comments as there is a bit to unpack. On lines 40-42 we fill

out an *addrinfo* struct called *hints*. This serves as the guide for the resulting *structs* that we'll use. Here we specify *AF_UNSPEC* meaning we are good with both IPv4 and IPv6. We also specify *SOCK_STREAM* (TCP). Lastly we set it to *AI_PASSIVE*. The man page says this about *AI_PASSIVE*:

If the *AI_PASSIVE* flag is specified in *hints.ai_flags*, and *node* is *NULL*, then the returned socket addresses will be suitable for *bind(2)*ing a socket that will *accept(2)* connections. The returned socket address will contain the "wildcard address" (*INADDR_ANY* for IPv4 addresses, *IN6ADDR_ANY_INIT* for IPv6 address). The wildcard address is used by applications (typically servers) that intend to accept connections on any of the host's network addresses. If *node* is not *NULL*, then the *AI_PASSIVE* flag is ignored.

Now when we call *getaddrinfo* on line 45, the *res struct* is filled with the IPv4 and IPv6 wildcard addresses. The next thing we do in lines 53-76 is attempt to loop through each one until we can create a socket, update it's options, and *bind* to it. In my tests, this always resulted in my program binding to "0.0.0.0:8888". Of note, *getaddrinfo* allocates space for the *addrinfo structs* dynamically on the *heap*. Therefore, you need to call *freeaddrinfo* as I do on line 127 in order to ensure it is free'd.

Now that we have a socket and have bound our address to it, we can now call *listen* on line 85 and *accept* on line 100 much like we did in the previous example. However, instead of interacting with the remote socket, we instead call *fork* on line 112. This allows the child to handle the remote connection and the parent to go back to wait for the next connection. In our discussions of *fork* (7.1) I mentioned that both the parent and the child have copies of the memory in use when the *fork* took place. However, the child doesn't require the main socket field descriptor *sockFD* so it closes it on line 115. The socket doesn't actually get closed though since the parent process is still using it. Additionally, the parent process doesn't need to interact with the new remote socket *remoteFD* so it closes it on line 120. Again, the socket doesn't actually get closed since the child is still using it. If we didn't do this, the sockets wouldn't get closed when they need to because the parent would continue holding *remoteFD* and the child would continue holding *sockFD*.

8.2 sigaction

The last thing worth pointing out in this example is the use of **sigaction**. In order to shut the server down, the user is expected to hit <CTRL+C>. This sends a *SIGINT* (Interrupt from keyboard). Without doing so, the server would be blocking on *accept*.

Using *sigaction* is fairly straight forward. On line 31, I define a *sigaction struct*. On line 33 I pass it a pointer to a function that will be called when a signal is received. On line 35 I then call *sigaction* and indicate that the signal I am concerned with is *SIGINT*. Now when a *SIGINT* is received, the

sigHandler function will be called. Only certain actions should take place here especially in a multithreaded environment (which this is not). In the context of this program, the *sigHandler* function sets a flag causing the children to exit. In the parent process, the SIGINT causes the *accept* to return a -1 and sets *errno* to be set to *EINTR*. Sensing this, I break out of the loop and return from the program after closing *sockFD* and cleaning up memory.



Part Three

9	Data Structures	99
9.1	Linked-List	
9.2	Queue	
9.3	Stack	
9.4	Binary Search Tree	
9.5	HashTable	
	Bibliography	119
	Websites	
	Books	
	Index	121

9. Data Structures

As you've seen with C up until this point, you tend to have to build everything yourself. This is why understanding data structures is an important skill to have. In the following sections, we'll discuss *linked-lists*, *queues*, *stacks*, *binary search tree*, and *hashtable*. There are multiple ways that they could be constructed. The following examples have been made to store data of type *void ** so that they can be used with any data type. An understanding of pointers and dynamic memory will be essential when building and working with custom data structures.

9.1 Linked-List

A *linked-list* is a fairly simple data structure where one node points to the next node. You may see a *singly linked-list*, *doubly linked-list*, and even a *circlly linked-list*. We'll keep it simple with a *singly linked-list*.

```
1 #ifndef LL_H
2 #define LL_H
3 #include <stdbool.h>
4
5 typedef struct node {
6     struct node *next;
7     void *data;
8 } node_t;
9
10 typedef struct ll {
11     unsigned int numNodes;
```

```

12     node_t *head;
13 } ll_t;
14
15 ll_t *
16 createList();
17
18 bool
19 destroyList(ll_t **, void (*)(void *));
20
21 bool
22 addNode(ll_t *, void *data);
23
24 void *
25 removeNode(ll_t *, void *data, bool (*)(void *, void *));
26
27 void
28 printList(ll_t *, void (*)(void *));
29
30 #endif

```

Listing 9.1: src/09-ll.h

In the header file we have two *structs*, *ll_t* and *node_t*. The *ll_t struct* will contain the first *node_t* which is called the "head". The "next" pointer in each *node_t struct* will point to the next node in the list unless there are no additional nodes. In that case, the "next" pointer will be *NULL*. Again, there are a handful of different ways you could construct a linked-list. However, the basis premise is one node points to the next node.

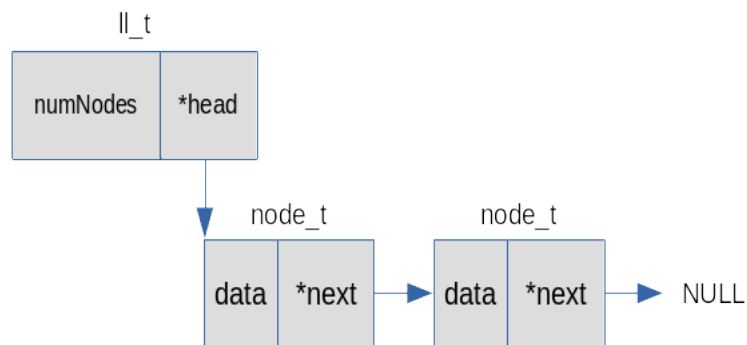


Figure 9.1: Linked-List

In this implementation I am even keeping track of the number of nodes by having a *numNodes* entry in the *ll_t struct*. This isn't necessary nor is it probably common. However, you are free to customize your implementation to your application. Maybe periodically looking up how many nodes are in my list is something my program needs to be aware of. If this wasn't something I was already keeping track of, I would have to traverse the entire list to count the nodes each time I needed to know.

We can also see in the header file that I have implemented a number of functions that will serve as the API to interact with the linked-list. In this case, we have functions to initially create the list, destroy the list, add a node to the list, remove a node from the list, and even print our list. Because we are storing the data as *void **, the implementation requires the passing in of function pointers that know how to handle the particular data type that we've stored.

Let's look to see how all of this is implemented:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include "09-11.h"
5
6 ll_t *
7 createList() {
8     ll_t *list = calloc(1, sizeof(*list));
9     return list;
10 }
11
12 bool
13 destroyList(ll_t **list, void(*destroyData)(void *)) {
14     if (NULL == list || NULL == *list) {
15         return false;
16     }
17
18     node_t *temp = (*list)->head;
19     node_t *next;
20     while (NULL != temp) {
21         next = temp->next;
22         destroyData(temp->data);
23         free(temp);
24         temp = next;
25     }
26
27     free(*list);
28     *list = NULL;
29
30     return true;
31 }
32
33 bool
34 addNode(ll_t *list, void *data) {
35     if (NULL == list) {
36         return false;
37     }
38
39     node_t *node = malloc(sizeof(*node));
40     if (NULL == node) {
41         fprintf(stderr, "Unable to allocate for new node.\n");
```

```
42     return false;
43 }
44
45 node->next = NULL;
46 node->data = data;
47
48 /* no current head */
49 if (NULL == list->head) {
50     list->head = node;
51     return true;
52 }
53
54 /* replace the head */
55 node->next = list->head;
56 list->head = node;
57 return true;
58 }
59
60 void *
61 removeNode(ll_t *list, void *searchData, bool(*compare)(void *, void *)) {
62     if (NULL == list || NULL == list->head) {
63         return NULL;
64     }
65
66     node_t *temp = list->head;
67     node_t *prev = temp;
68     void *rData;
69     bool result;
70
71     /* matched node is the head */
72     if (true == (result = compare(temp->data, searchData))) {
73         list->head = temp->next;
74         rData = temp->data;
75         free(temp);
76         return rData;
77     }
78
79     /* search through the remainder of the list */
80     temp = temp->next;
81     while (NULL != temp) {
82         if (true == (result = compare(temp->data, searchData))) {
83             prev->next = temp->next;
84             rData = temp->data;
85             free(temp);
86             return rData;
87         }
88         prev = temp;
89         temp = temp->next;
90     }
91
92     /* no match */
```

```
93     return NULL;
94 }
95
96 void
97 printList(ll_t *list, void(*printData)(void *)) {
98     if (NULL == list || NULL == list->head) {
99         return;
100     }
101
102     node_t *temp = list->head;
103     while (NULL != temp) {
104         printData(temp->data);
105         temp = temp->next;
106     }
107 }
```

Listing 9.2: src/09-ll.c

9.1.1 createList

On line 8 of *createList* you can see that I use *calloc* to dynamically allocate space for the *ll_t struct*. This means that *numNodes* will be set to zero and *head* will be set to *NULL*.

9.1.2 destroyList

On line 13 we begin the *destroyList* function. Here we see the list is passed in as a pointer to a pointer. This is so that we can *NULL* out the final list as we do on line 28. As you'll see in most of the following functions, we first check to see if we've received a *NULL* for the list. Without doing this, we can expect to have *SEGMENTATION* faults. I don't check for every possible scenario throughout but checking for some of the most common issues goes a long way.

Also notice within *DestroyData* that I am passing in a pointer to a function called *destroyData*. Because the data that is stored in the list may have been dynamically allocated, we need to ensure it can be cleaned up properly. However, since we are storing it as *void **, we don't actually know how to handle the data. What we are expecting is that the function passed in does know how to handle the data so on line 22 I am passing the data to the *destroyData* function.

Overall, the idea is that on lines 18 - 25, we are looping through each node, destroying the data and freeing the node. Once we reach the final node where *next* is set to *NULL* we know we've reached the end of the list. At this time, we free the list itself and set it to *NULL*.

9.1.3 addNode

The *addNode* function allocates space for the new node on line 39. It then checks to see if there is already a node in the list on line 49 by checking the *head*. If one is not present, it is added on line 49. Otherwise, the newly created node sets its *next* to the current *head* on line 55, and then replaces the *head* with its own address on line 56.

9.1.4 removeNode

removeNode may look complicated at first, but in actuality is quite simple. What the function takes in is the list itself, *searchData* to identify the node, and a function pointer called *compare* to perform the match. This allows the calling program that understands the data type stored in the list to properly identify the node to remove.

Lines 72 - 76 are if the data is found at the current *head*. In this case, it saves off the node in a temporary variable on line 66, slides the head down on line 73, saves off the data on line 74, and then frees the temporary node.

If the data wasn't found at the *head*, lines 80-89 traverse the list looking for it. Once identified, the previous node gets patched around the temp node on line 83, the data is saved off off line 84, and then the temp node is free'd.

If after traversing the entire list the data is not identified, *NULL* is returned.

9.1.5 printList

printList is a function that was written just so that I could demonstrate a more complex data type being handled. In this case, it takes in the list as well as a function pointer that interprets the data and prints it. Lines 102-105 merely traverse the list passing the data in each node to the function pointer called *printData*.

9.1.6 Testing the Linked-List

Lets take a look at how we might utilize our linked list:

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 #include <string.h>
4 #include "09-11.h"
5
6 #define UNUSED(x) (void)(x)
7
8 struct student {
```



```
9     int grade;
10     int studentID;
11     char name[20];
12 };
13
14 static void
15 destroyData(void *);
16
17 static bool
18 compareData(void *, void *);
19
20 static void
21 print(void *);
22
23 int
24 main() {
25     /* function pointers */
26     void(*dd)(void *) = &destroyData;
27     bool(*cmp)(void *, void *) = &compareData;
28     void(*pl)(void *) = &print;
29
30     /* create sample data to store in the list */
31     struct student s1 = {85, 1, "Jimmy"};
32     struct student s2 = {76, 2, "Joseph"};
33     struct student s3 = {101, 3, "Julie"};
34     struct student s4 = {101, 4, "Timmy"};
35
36     /* create the list and add nodes to it */
37     ll_t *list = createList();
38     printf("Adding nodes.\n");
39     addNode(list, (void *)&s1);
40     addNode(list, (void *)&s2);
41     addNode(list, (void *)&s3);
42     addNode(list, (void *)&s4);
43     printList(list, pl);
44
45     /* remove nodes */
46     printf("\nRemoving Nodes:\n");
47     int studentID = 2;
48     print(removeNode(list, (void *)&studentID, cmp));
49     studentID = 4;
50     print(removeNode(list, (void *)&studentID, cmp));
51     studentID = 1;
52     print(removeNode(list, (void *)&studentID, cmp));
53     studentID = 3;
54     print(removeNode(list, (void *)&studentID, cmp));
55     studentID = 5;
56     print(removeNode(list, (void *)&studentID, cmp));
57
58     printf("\nAdd Node.\n");
59     addNode(list, (void *)&s2);
```

```

60     printList(list , pl);
61
62     /* destroy the list */
63     printf("\nDestroy list.\n");
64     destroyList(&list , dd);
65     destroyList(&list , dd);
66     return 0;
67 }
68
69 static void
70 destroyData(void *data) {
71     UNUSED(data);
72 }
73
74 static bool
75 compareData(void *d1, void *d2) {
76     struct student s = *(struct student *)d1;
77     return s.studentID == *(int *)d2;
78 }
79
80 static void
81 print(void *data) {
82     if (NULL == data) {
83         printf("NULL\n");
84         return;
85     }
86     struct student s = *(struct student *)data;
87     printf("%s: %d\n", s.name, s.grade);
88 }

```

Listing 9.3: src/09-ll_test.c

As you can see on line 26 - 28 I define function pointers that we'll use later when interacting with the linked-list API. On lines 31 - 34 I define some sample data to be used in our list. On line 37 we finally create our list and begin adding our sample data to it. Notice when I add my data to the list, I am type casting it as *void **. Also notice on line 43 that when I call the *printList* function I also pass in the function pointer to my function called *print*.

Lines 47 - 56 is where I remove nodes. I have to define a value that will be used as the *searchData* in the *removeData* function. Additionally, I have to pass in a pointer to the *compareData* function that begins on line 75 that will identify when we've found the correct node. Out of convenience, I'm passing the result of *removeNode* to my *print* function for an easy way to display it.

Lastly, on lines 64 and 65 I destroy my list. The first time to ensure it works and the second time to see what happens when passing in a *NULL* for the list.

```
$ valgrind --leak-check=full --show-leak-kinds=all ./09-ll
==47082== Memcheck, a memory error detector
```

```

==47082== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==47082== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==47082== Command: ./09-11
==47082==
Adding nodes.
Timmy: 101
Julie: 101
Joseph: 76
Jimmy: 85

Removing Nodes:
Joseph: 76
Timmy: 101
Jimmy: 85
Julie: 101
NULL

Add Node.
Joseph: 76

Destroy list.
==47082==
==47082== HEAP SUMMARY:
==47082==      in use at exit: 0 bytes in 0 blocks
==47082==    total heap usage: 7 allocs, 7 frees, 1,120 bytes allocated
==47082==
==47082== All heap blocks were freed -- no leaks are possible
==47082==
==47082== For lists of detected and suppressed errors, rerun with: -s
==47082== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

9.2 Queue

One variation that we can make with our linked-list is to turn it into a queue. A queue is a First-In-First-Out (FIFO) container. Again, we can implement it in many different ways but I wanted to show you how we can use a plain old linked-list as a queue.

```

1 #ifndef QUEUE_H
2 #define QUEUE_H
3
4 #include <stdbool.h>

```

```

5
6  /*
7   * Dequeue                                Enqueue
8   * |                                     |
9   * | head          tail                 |
10  *  -> node->next node->next NULL <-
11  */
12
13  typedef struct node {
14      void *data;
15      struct node *next;
16  } node_t;
17
18  typedef struct queue {
19      struct node * head;
20      struct node * tail;
21  } queue_t;
22
23  queue_t *
24  createQueue(void);
25
26  bool
27  destroyQueue(queue_t **, void (*)(void *));
28
29  bool
30  enqueue(queue_t *, void *);
31
32  void *
33  dequeue(queue_t *);
34
35  #endif

```

Listing 9.4: src/09-queue.h

The first thing you may notice is that the *node_t* looks the same as it did before. However, *queue_t* look similar but we now have both a *head* and a *tail*. There are two primary function that we'll use with the queue, *enqueue* adds an item to the queue and *dequeue* removes an item from the queue.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include "09-queue.h"
5
6  queue_t *
7  createQueue(void) {
8      queue_t *newQueue = (queue_t *)calloc(1, sizeof(*newQueue));
9      return newQueue;
10 }
11

```

```

12 bool
13 destroyQueue(queue_t **q, void(*destroyData)(void *)) {
14     if (NULL == q || NULL == *q) {
15         return false;
16     }
17
18     /* queue is empty */
19     if (NULL == (*q)->head) {
20         free(*q);
21         *q = NULL;
22         return true;
23     }
24
25     node_t *head = (*q)->head;
26     node_t *next = head->next;
27
28     while(NULL != head) {
29         destroyData(head->data);
30         free(head);
31         head = next;
32         if (NULL != next) {
33             next = next->next;
34         }
35     }
36
37     free(*q);
38     *q = NULL;
39
40     return true;
41 }
42
43 bool
44 enqueue(queue_t *q, void *data) {
45     if (NULL == q) {
46         return false;
47     }
48
49     node_t *newNode = (node_t *)calloc(1, sizeof(*newNode));
50     if (NULL == newNode) {
51         perror("Unable to allocate for new node.");
52         return false;
53     }
54
55     newNode->data = data;
56     if (NULL == q->tail) {
57         q->tail = newNode;
58         q->head = newNode;
59     } else {
60         q->tail->next = newNode;
61         q->tail = newNode;
62     }

```

```
63
64     return true;
65 }
66
67 void *
68 dequeue(queue_t *q) {
69     if (NULL == q || q->head == NULL) {
70         return NULL;
71     }
72
73     void *data = q->head->data;
74     node_t *temp = q->head->next;
75     free(q->head);
76     q->head = temp;
77
78     if (NULL == q->head) {
79         q->tail = NULL;
80     }
81     return data;
82 }
```

Listing 9.5: src/09-queue.c

9.2.1 createQueue

Here we allocate space for a *queue_t*. We use *calloc* so that both the *head* and *tail* are initially set to *NULL*.

9.2.2 destroyQueue

In *destroyQueue* we once again take in a pointer to queue pointer to ensure we can set it to *NULL* when complete. We also take in a function pointer called *destroyData* since we're storing items as *void **. The first thing we check is to make sure the queue isn't already *NULL*. We then loop through the queue if there are leftover nodes and pass the data to the *destroyData* function pointer.

Now that all data and nodes have been free'd, we free the queue itself, set it to *NULL* and return.

9.2.3 enqueue

When adding something to the queue, we *enqueue* that data to the *tail*. This is why on line 56 we first check to see if the *tail* is *NULL*. This would indicate there isn't anything currently queue'd so both the *head* and the *tail* get set to the new node. If however it already points to a node, we point that node's *next* pointer to the new node on line 60 and then point *tail* to the new node on line 61.

9.2.4 dequeue

When we dequeue, we remove a node from the *head*. This is why we must first check on line 69 to see if the current *head* is already pointing to *NULL*. Assuming there are nodes in the queue, we save off the data on line 73, save a temp pointer to *head->next*, free the current *head*, and then assign the temp pointer back to the *head* on line 76. Finally, if the *head* is now set to *NULL*, this means there are no nodes left in the queue so we need to set the current *tail* to *NULL* as well. We do this on lines 78 - 80 and then return the data.

9.2.5 Testing the Queue

Lets take a look at what it looks like to use our queue:

```
1 #define _POSIX_C_SOURCE 200809L
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include "09-queue.h"
6
7 struct bottle {
8     char *msg;
9     char author[24];
10 };
11
12 static struct bottle *
13 newMessage(void);
14
15 static void
16 printMessage(void *);
17
18 static void
19 destroyMessage(void *);
20
21 int
22 main() {
23     void *msg;
24     void(*dm)(void *) = &destroyMessage;
25
26     /* create queue */
27     queue_t *q = createQueue();
28
29     /* add items */
30     printf("Enqueue:\n");
31     enqueue(q, (void *)newMessage());
32     enqueue(q, (void *)newMessage());
33     enqueue(q, (void *)newMessage());
34 }
```

```

35  /* remove items */
36  printf("Dequeue:\n");
37  for (int i = 0; i < 4; i++) {
38      msg = dequeue(q);
39      printMessage(msg);
40      destroyMessage(msg);
41  }
42
43  /* item another item */
44  printf("Enqueue:\n");
45  enqueue(q, (void *)newMessage());
46
47  /* destroy the queue */
48  printf("Destroy:\n");
49  destroyQueue(&q, dm);
50  destroyQueue(&q, dm);
51  }
52
53  static struct bottle *
54  newMessage(void) {
55      struct bottle *b = calloc(1, sizeof(*b));
56      size_t readSize = 0;
57      if (NULL == b) {
58          fprintf(stderr, "Unable to allocate for bottle.\n");
59          return NULL;
60      }
61
62      printf("Message to send: ");
63      ssize_t bytes = getline(&(b->msg), &readSize, stdin);
64      if (-1 == bytes) {
65          fprintf(stderr, "Unable to get input.\n");
66          free(b->msg);
67          free(b);
68          return NULL;
69      }
70      if ('\n' == b->msg[bytes - 1]) {
71          b->msg[bytes - 1] = 0;
72      }
73      strncpy(b->author, "Rick Astley", 23);
74      return b;
75  }
76
77  static void
78  destroyMessage(void *b) {
79      if (NULL == b) {
80          return;
81      }
82      free(((struct bottle *)b)->msg);
83      free(b);
84  }
85

```



```

86 static void
87 printMessage(void *b) {
88     if (NULL == b) {
89         return;
90     }
91     printf("%s: %s\n", ((struct bottle *)b)->author, ((struct bottle *)b)->msg);
92 }

```

Listing 9.6: src/09-queue_test.c

Here we create our queue on line 27. We then *enqueue* three messages. We attempt to *dequeue* four messages to make sure it doesn't break after *dequeuing* more items than were *enqueue'd*. Lastly, we *enqueue* a final message and *destroyQueue*. Lets see if everything works correctly with *valgrind*.

```

$ valgrind --leak-check=full --show-leak-kinds=all ./09-queue
==34017== Memcheck, a memory error detector
==34017== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==34017== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==34017== Command: ./09-queue
==34017==
Enqueue:
Message to send: Never gonna give you up
Message to send: Never gonna let you down
Message to send: Never gonna run around and desert you
Dequeue:
Rick Astley: Never gonna give you up
Rick Astley: Never gonna let you down
Rick Astley: Never gonna run around and desert you
Enqueue:
Message to send: Never gonna make you cry
Destroy:
==34017==
==34017== HEAP SUMMARY:
==34017==    in use at exit: 0 bytes in 0 blocks
==34017== total heap usage: 15 allocs, 15 frees, 2,736 bytes allocated
==34017==
==34017== All heap blocks were freed -- no leaks are possible
==34017==
==34017== For lists of detected and suppressed errors, rerun with: -s
==34017== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

9.3 Stack

```

1  #ifndef STACK_ARRAY_H
2  #define STACK_ARRAY_H
3
4  #include <stdbool.h>
5  #include <stdlib.h>
6
7  typedef struct stack {
8      size_t maxSize;
9      size_t curSize;
10     void **items;
11 } stack_t;
12
13 stack_t *
14 createStack(size_t);
15
16 bool
17 destroyStack(stack_t **, void (freeData)(void *));
18
19 bool
20 push(stack_t *, void *);
21
22 void *
23 pop(stack_t *);
24
25 bool
26 stackFull(stack_t *);
27
28 #endif

```

Listing 9.7: src/09-stack.h

```

1  #include <stdbool.h>
2  #include <stdlib.h>
3  #include "09-stack.h"
4
5  stack_t *
6  createStack(size_t maxSize) {
7      stack_t *newStack = malloc(sizeof(*newStack));
8      newStack->maxSize = maxSize;
9      newStack->curSize = 0;
10     newStack->items = malloc(maxSize * sizeof(*(newStack->items)));
11     return newStack;
12 }
13
14 bool
15 destroyStack(stack_t **stack, void (freeData)(void *)) {
16     if (NULL == stack || NULL == *stack) {
17         return false;
18     }

```

```

19
20     for (size_t i = 0; i < (*stack)->curSize; i++) {
21         freeData((*stack)->items[i]);
22     }
23
24     free((*stack)->items);
25     free(*stack);
26     *stack = NULL;
27     return true;
28 }
29
30 bool
31 push(stack_t *stack, void *data) {
32     if (NULL == stack || stack->curSize == stack->maxSize) {
33         return false;
34     }
35     stack->items[stack->curSize] = data;
36     stack->curSize++;
37     return true;
38 }
39
40 void *
41 pop(stack_t *stack) {
42     if (NULL == stack || 0 == stack->curSize) {
43         return NULL;
44     }
45
46     stack->curSize--;
47     return stack->items[stack->curSize];
48 }
49
50 bool
51 stackFull(stack_t *stack) {
52     if (NULL == stack || stack->curSize == stack->maxSize) {
53         return true;
54     }
55     return false;
56 }

```

Listing 9.8: src/09-stack.c

```

1  #define _POSIX_C_SOURCE 200809L
2  #include <stdio.h>
3  #include <stdbool.h>
4  #include <stdlib.h>
5  #include "09-stack.h"
6  #define MAXSIZE 3
7  #define ITEMSIZE 20
8  #define UNUSED(x) (void)(x)
9
10 static char *

```

```
11 getItem(void);
12
13 static void
14 freeData(void *);
15
16 int
17 main() {
18     void (*fd)(void *) = &freeData;
19
20     printf("Creating stack with maxsize: %d\n", MAXSIZE);
21     stack_t *stack = createStack(MAXSIZE);
22     if (NULL == stack) {
23         fprintf(stderr, "Stack not allocated.\n");
24         return 1;
25     }
26
27     printf("\nPushing onto stack:\n");
28     bool result;
29     char *msg;
30     for (int i = 0; i < MAXSIZE + 1; i++) {
31         msg = getItem();
32         result = push(stack, (void *)msg);
33         if (false == result) {
34             freeData(msg);
35         }
36         printf("Stack full: %s\n", stackFull(stack) ? "true" : "false");
37     }
38
39
40     printf("\nPopping from stack.\n");
41     void *item;
42     for (int i = 0; i < MAXSIZE + 1; i++) {
43         item = pop(stack);
44         printf("%s\n", NULL != item ? (char *)item : "NULL");
45         freeData(item);
46     }
47
48     printf("\nPushing more items onto stack.\n");
49     msg = getItem();
50     result = push(stack, (void *)msg);
51     if (false == result) {
52         freeData(msg);
53     }
54     printf("Stack full: %s\n", stackFull(stack) ? "true" : "false");
55
56     printf("\nDestroying stack.\n");
57     destroyStack(&stack, fd);
58     destroyStack(&stack, fd);
59
60     return 0;
61 }
```

```

62
63 static char *
64 getItem() {
65     char *msg = NULL;
66     size_t readSize = 0;
67
68     printf("Message to send: ");
69     ssize_t bytes = getline(&msg, &readSize, stdin);
70     if (-1 == bytes) {
71         free(msg);
72         return NULL;
73     }
74
75     if ('\n' == msg[bytes - 1]) {
76         msg[bytes - 1] = 0;
77     }
78     return msg;
79 }
80
81
82 static void
83 freeData(void *data) {
84     free(data);
85 }

```

Listing 9.9: src/09-stack_test.c

```

$ valgrind --leak-check=full --show-leak-kinds=all ./09-stack
==45165== Memcheck, a memory error detector
==45165== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==45165== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==45165== Command: ./09-stack
==45165==
Creating stack with maxsize: 3

```

```

Pushing onto stack:
Message to send: ONE
Stack full: false
Message to send: TWO
Stack full: false
Message to send: THREE
Stack full: true
Message to send: FOUR
Stack full: true

```

```

Popping from stack.

```

THREE

TWO

ONE

NULL

Pushing more items onto stack.

Message to send: FIVE

Stack full: false

Destroying stack.

==45165==

==45165== HEAP SUMMARY:

==45165== in use at exit: 0 bytes in 0 blocks

==45165== total heap usage: 9 allocs, 9 frees, 2,696 bytes allocated

==45165==

==45165== All heap blocks were freed -- no leaks are possible

==45165==

==45165== For lists of detected and suppressed errors, rerun with: -s

==45165== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

9.4 Binary Search Tree

9.5 HashTable



Bibliography

Websites

- [2] Free Software Foundation. *A GNU Manual*. URL: <https://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/Type-Attributes.html>. (accessed: 07.25.2020) (cited on page 65).

Books

- [1] Michael Barr. *Embedded C Coding Standard*. BARR-C:2018. Germantown, MD: Barr Group, 2018 (cited on pages 6, 24, 26, 29, 32, 41, 43, 44).
- [3] Brian Hall. *Beej's Guide to Network Programming: Using Internet Sockets*. Brian “Beej Jorgensen” Hall, 16 (cited on page 87).
- [4] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd edition. Pearson, 2015 (cited on page 4).
- [5] Peter Prince and Tony Crawford. *C in a Nutshell*. 2nd edition. Sebastopol, CA: O'Reilly, 2016 (cited on pages 7, 22, 23, 26, 65).



Index

Symbols

#define 29

A

ABC 38

ABC principle 30

array 28

ASCII 20

assembler 8

assembly 34

B

bool 19

C

call stack 34

calloc 36

char 20

character constant 22

cli 45

compile 7

compiler 8

concurrency 73

condition variables 80, 84

const 5

control flow 41

D

deadlock 76

definition 54

double 23

DRY principle 5

E

enum 6, 24

F

fclose 63

fflush 68

fgets 63

float 23

fopen 61

fork 73, 95

fputs 63

free 36

function

 declaration 4

definition 5
 static 4
 fwrite 67

G

gcc 7
 getaddrinfo 91
 guard *see* header, guard

H

header 3, 5
 guard 6
 heap 36

I

int 16

L

linker 8
 long double 23
 long int 16
 long long int 16

M

macro 6
 make 10
 Makefile 11
 malloc 36
 multithreading 76
 mutex 78, 79

N

nanosleep 75
 Networking 87
 NULL 27, 30
 null terminator 30

O

operator precedence 26

P

packed 65
 pointer 27, 30
 preprocessor 7

Q

queue 107

R

rbp 34
 realloc 36
 rip 35
 rsp 34

S

short 16
 sigaction 95
 signed 17
 size_t 19
 sizeof 29
 stack 114
 stack frame 34
 struct 53

T

thread pool 81, 84
 threads 76, 78, 84
 typedef 24

U

unsigned 17

V

void 4, 24

X

XOR 27