

CS 107 READER

STANFORD COMPUTER SCIENCE DEPARTMENT

Copyright © 2018

PUBLISHED BY STANFORD COMPUTER SCIENCE DEPARTMENT

TUFTE-LATEX.GOOGLECODE.COM

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

First printing, January 2018

Contents

<i>Unix, the Command Line, gcc, and Makefiles</i>	13
<i>Number Formats Used in CS 107</i>	19
<i>C Primer</i>	26
<i>gdb</i>	46
<i>Bits and Bytes</i>	54
<i>C-Strings and the C String Library</i>	73
<i>Pointers, Generic functions with void *, and Pointers to Functions</i>	80
<i>IEEE Floating Point</i>	93
<i>x86-64 Assembly Language</i>	107
<i>Managing the Heap</i>	128
<i>Bibliography</i>	129
<i>Index</i>	130

List of Figures

- 1 `test_void_star.c` Simple test programs can often answer questions about syntax, or language usage, and we encourage students to write short test programs for just such a purpose. 10
- 2 A typical Linux terminal window. 14
- 3 *The C Programming Language* by Brian Kernighan and Dennis Ritchie (also known as “K&R”) is the definitive text on C and should be in every C programmer’s library. 26
- 4 Possible memory layout for `swap2` 36
- 5 Possible memory layout for `swap2` with double dereferencing 37
- 6 Memory address for an array of 4-byte ints 56
- 7 The number `0x01234567` stored in Big Endian Format, at address `0x100` 57
- 8 The number `0x01234567` stored in Little Endian Format, at address `0x100` 58
- 9 The AND truth table. 58
- 10 The OR truth table. 58
- 11 The XOR truth table. 58
- 12 The NOT truth table. 58
- 13 The two’s complement circle for a signed 4-bit number 62
- 14 The two’s complement circle for an unsigned 4-bit number 62
- 15 An explicit cast between signed and unsigned ints. 64
- 16 An implicit cast between signed and unsigned ints. 64
- 17 A cast in C does not change the underlying bit pattern for integers of the same bit width. 65
- 18 `printf`’s format string performs a cast on its values. 66
- 19 Example program demonstrating the `sizeof` operator. Note the difference between the size reported for an array and a pointer to the array. 68
- 20 Example program demonstrating the loss of information when converting from a larger integer type to a smaller integer type. 70
- 21 There is no loss of precision when converting from a smaller integer type to a larger integer type. 71
- 22 The following is one way to determine if addition of two unsigned ints will overflow 71
- 23 The following is one way to determine if addition of two signed ints will overflow 71

24	A C string pointer and the associated memory	75
25	An array of longs in memory	81
26	Possible memory layout for nums array and numsptr	82
27	The command line arguments array	91
28	The command line arguments array after being sorted by the sum of the character values of each argument string (except the program name): pear, peach, apple, banana, orange, nectarine.	92
29	Single precision (32-bit) IEEE floating point format	96
30	32-bit FLT_MAX and FLT_MIN values	105
31	The sixteen x86-64 integer registers	112
32	The Linux address space (not to scale)	115
33	A push followed by a pop. Notice that after the push %rax, the stack pointer has been decremented by 8, and after the pop %rdx, the stack pointer has been incremented by 8. Also note that the value on the stack after the pop has not been cleared.	116

List of Tables

1	Hexadecimal to binary conversion for 0 - 7	22
2	Hexadecimal to binary conversion for 8 - 15	23
3	Common printf specifiers	29
4	C Data Sizes on the Myth Computers	57
5	Ranges for integer data types on the Myth machines	61
6	C expressions that have both unsigned and signed integers perform the calculation assuming all values are unsigned.	66
7	Four and eight bit signed two's complement representation.	69
8	Intel Data Types	110
9	The common conditional suffixes for set / j / cmov	121

List of URLs

http://stanford.edu/~cgregg/107-Reader/107-Reader-code.zip	p. 11
http://vxer.org/lib/pdf/Reflections%20on%20Trusting%20Trust.pdf	p. 13
http://web.stanford.edu/class/cs107/	p. 9
http://www.apache.org/licenses/LICENSE-2.0	p. 2
http://www.cplusplus.com/reference/cstdio/printf/	p. 28
http://www.imada.sdu.dk/Courses/DM18/Litteratur/IntelATT.htm	p. 107
https://books.google.com/books?id=KfM2rgEACAAJ	pp. 9, 129
https://books.google.com/books?id=Yi5FI5QcdmYC	pp. 9, 129
https://cdecl.org	p. 44
https://docs.oracle.com/cd/E19957-01/816-2464/ncg_math.html	p. 99
https://en.wikipedia.org	p. 11
https://en.wikipedia.org/wiki/C_(programming_language)	p. 13
https://en.wikipedia.org/wiki/C_standard_library	p. 29
https://en.wikipedia.org/wiki/Dennis_Ritchie	p. 13
https://en.wikipedia.org/wiki/Donald_Knuth	p. 13
https://en.wikipedia.org/wiki/Free_Software_Foundation	p. 14
https://en.wikipedia.org/wiki/Git	p. 16
https://en.wikipedia.org/wiki/Gratis_versus_libre	p. 14
https://en.wikipedia.org/wiki/IEEE_754	p. 93
https://en.wikipedia.org/wiki/IEEE_754#Roundings_to_nearest	p. 98
https://en.wikipedia.org/wiki/Ken_Thompson	p. 13
https://en.wikipedia.org/wiki/Linus_Torvalds	p. 13
https://en.wikipedia.org/wiki/Make_(software)	p. 15
https://en.wikipedia.org/wiki/Setun	p. 19
https://en.wikipedia.org/wiki/TeX	p. 13
https://en.wikipedia.org/wiki/The_Art_of_Computer_Programming	p. 13
https://en.wikipedia.org/wiki/Two%27s_complement	p. 62
https://en.wikipedia.org/wiki/Unicode	p. 73
https://en.wikipedia.org/wiki/Unix	p. 13
https://en.wikipedia.org/wiki/Unix_philosophy	p. 14
https://en.wikipedia.org/wiki/William_Kahan	p. 95
https://en.wikipedia.org/wiki/X86	p. 107
https://github.com	p. 16
https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/	p. 103
https://software.intel.com/en-us/articles/intel-sdm	p. 107
https://stackoverflow.com/a/1641963/561677	p. 32
https://stallman.org	p. 14

<https://stanford.edu/~cgregg/107-Reader/float/convert.html>
 pp. 95, 97, 98
https://wikimediafoundation.org/wiki/Ways_to_Give ... p. 11
<https://www.bell-labs.com/usr/dmr/www/chist.pdf> pp. 82, 129
<https://www.h-schmidt.net/FloatConverter/IEEE754.html> p. 98
https://www.strchr.com/x86_machine_code_statistics . p. 111
[https://www.tutorialspoint.com/cprogramming/c_preprocessors.](https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm)
 htm p. 27
<https://www.wired.com/2011/10/thedennisritchieeffect/> p. 13
<mailto:cgregg@stanford.edu> p. 11

Introduction

THIS COURSE READER is meant to be a guide for Stanford students as they progress through CS 107, *Computer Organization & Systems*, which serves as the first systems course in the Stanford Computer Science Curriculum. The guide is not intended to replace the outstanding course textbooks, *Computer Systems, A Programmers Perspective*, 3rd Edition by Randy Bryant and David O'Hallaron¹ and (the ultimate classic) *The C Programming Language* by Brian Kernighan and David Ritchie², nor is it intended to replace attending lectures, participating in labs, and completing the course assignments. The only way to really learn the material in CS 107 is by doing the labs and assignments, and the only way to do the labs and assignments is to use the course materials (this guide included) as scaffolding.

The material in this guide follows the outline of the course, though it should not be considered a definitive guide to the class. Some material covered in class may not be in the guide, and vice-versa. During a particular quarter, the course website, at

<http://web.stanford.edu/class/cs107/>

holds up-to-date information about the course, and this guide should be used as an additional reference.

CS 107 is not a trivial course, as many current and former Stanford Computer Science students can attest. The introductory courses, CS 106A, CS 106B, and CS 106X, provide a computer science kiddie pool where students learn how to swim, under the guidance of a flotilla of Section Leader lifeguards who are available in numbers that allow frequent one-on-one guidance. In CS 107, students jump into the ocean, with a few Course Assistant Coast Guardspeople who can provide limited assistance when students are in dire need. In other words, many of the safety nets that students get used to in the introductory courses are not available in CS 107, and we expect students to be better self-learners when they become stuck on a programming problem. This guide can be a good resource, as can the textbook, Piazza, and office hours. Additionally, students should consider doing Internet searches as well, although answers found through Google or Stack Overflow (for example) will not always yield answers that are appropriate for

¹ R.E. Bryant and D.R. O'Hallaron. *Computer Systems : A Programmer's Perspective*. Pearson, 2015. ISBN 9781292101767. URL <https://books.google.com/books?id=KfM2rgEACAAJ>

² B.W. Kernighan and D. Ritchie. *The C Programming Language*. Pearson Education, 1988. ISBN 9780133086218. URL <https://books.google.com/books?id=Yi5FI5QcdmYC>

the course material. That said, searching online for the meaning of error messages, or for syntax questions is encouraged.

A frequent source of questions about the course material come in a form that could easily be tested by the student with a bit of C code. For example, a student might ask, *“I know that pointer arithmetic cannot be done on void* pointers. So... if we have two void* pointers, void* x and void* y, can we find the difference between their addresses by doing y - x or would that count as pointer arithmetic?”* This is a great question! However, it is also a question that can be tested with a little bit of C code, as shown in Figure 1. The student who can produce C code to answer questions such as this one have truly learned to swim in the programming ocean!

```
// file: piazza_question.c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int m = 0;
    int n = 1;

    void *x = &m;
    void *y = &n;

    printf("x - y: %lu\n", x - y);

    return 0;
}

$ gcc -g -O0 -std=gnu99 -Wall -Wfloat-equal -Wtype-limits \
    -Wpointer-arith -Wlogical-op -Wshadow \
    -fno-diagnostics-show-option void_star_test.c \
    -o void_star_test
void_star_test.c: In function `main':
void_star_test.c:12:29: warning: pointer of type `void *'
used in subtraction
    printf("x - y: %lu\n", x - y);
                           ^
```

Figure 1: test_void_star.c Simple test programs can often answer questions about syntax, or language usage, and we encourage students to write short test programs for just such a purpose.

In this course reader I have attempted to produce a detailed overview of what we expect students to be able to do by the end of the course. I have included practice problems that should serve as models of the type of questions that you might see on a CS 107 exam, in a lab, or on a homework assignment. If you can do all of the problems at the end of each chapter, you will be well on your way to a good grade on exams, and you will have a head start on the assignments.

Running the Code in the Reader

Throughout this reader, there are many examples of full (though small) programs that you can run, test, and play around with. We strongly encourage you to try all the programs, and to modify them to test your own understanding of how the programs are run.

The code for the assignments is located here:

<http://stanford.edu/~cgregg/107-Reader/107-Reader-code.zip>

If you are on a Myth machine and want to use the code, type the following (without the \$, which is the prompt):

```
$ cd
$ wget http://stanford.edu/~cgregg/107-Reader/107-Reader-code.zip
$ unzip 107-Reader-code.zip
$ cd 107-Reader-code
$ make
```

All of the programs will be compiled (there will be a few warnings). All code examples in the text have the file name as the first line of the code, so you should be able to quickly determine which file to run.

The code should work on any Linux system with gcc and make installed³, and on a Mac with XCode and the “command line tools” installed⁴. The easiest way to install and run the tools on a Windows 10 or higher machine is to enable “Bash” and then install gcc and make⁵.

Once you unzip the file, all the programs can be compiled with the make command.

Links in the Text and Footnotes

This reader has multiple Internet links to extra information, in both the main text, and primarily in the footnotes. The links provide a combination of online reference material, further information, and general information. Many of the links are from [Wikipedia](#), which is a triumph of 21st Century knowledge and information, and completely free.⁶

Contact Information

Please forward any typos or other comments on the text to cgregg@stanford.edu, and I hope you have a terrific experience learning about the lower depths of your computer!

³ To install the tools on Ubuntu, try the following: `sudo apt-get install gcc make build-essential linux-headers-$(uname -r)`.

⁴ To install XCode, open the App store and search for XCode. You most likely won't have to download anything else, although the first time you try to make a file, you might be prompted to install the command line tools.

⁵ A web search for “install gcc windows 10 bash” should direct you to some helpful guides.

⁶ which means that you should [donate to Wikipedia](#) if you use it! My general rule for donating to Wikipedia: one cent per visit. If you find yourself visiting Wikipedia ten times a day, pony up the \$36.5 once a year when you see that “donate now!” link!

Unix, the Command Line, gcc, and Makefiles

IN YOUR INTRODUCTORY CS courses, you may have learned to program using an *Integrated Development Environment* (IDE) such as *Eclipse* or *Qt Creator*. As their name implies, IDEs provide a full toolset to write, build, compile, run, and debug your programs. This is a terrific environment for beginning programmers to start learning the “art of computer programming” (as coined by Donald Knuth⁷). However, IDEs also hide the mechanics of turning text code (e.g., written in C or Java) into the *binary* representation that we will study in CS 107. Instead of using an IDE to create your programs, you will use a number of tools provided through the *Linux terminal*, and together they will take you from program creation in C to compilation (via *make* and *gcc*) to debugging via *gdb* and running your program on the *command line*. If all of those terms are foreign to you, you are not alone – many students who take CS 107 have never used the terminal, nor have they directly used any of the tools used to turn their programs into runnable applications.

In order to get up to speed using the Linux terminal, you should see the course website, which has a number of tutorial videos that describe how to set up your computer to log into the *Myth* computers at Stanford. If you are going through this reader and not at Stanford, unfortunately you won’t have access to the *Myth* computers, but much of the online material can be used on a generic Linux command line. If you don’t have a computer that natively runs Linux as its operating system, I suggest installing a virtual machine for Linux.⁸ Once you install a virtual machine, you should install the latest version of *gcc* and *gdb*, which will enable you to follow most of the examples in the reader.

Unix and Linux

Unix is an operating system that dates from the early 1970s and was originally created primarily by *Ken Thompson*⁹ and *Dennis Ritchie*¹⁰ at Bell Labs. It was the first operating system that was written to be portable, and it was completely written in the *C programming language*¹¹, also created by Dennis Ritchie. In the early 1990s, *Linus Torvalds* began an open-source version of Unix that he called *Linux*, and Linux has become a standard, free, open source operating system used on millions of computers around the world.

⁷ *Don Knuth*, Professor Emeritus of *The Art of Computer Programming* at Stanford, is a legendary computer scientist one of the world’s foremost algorithmists. He is the author of a magnum opus, *The Art of Computer Programming*. He is also the creator of *T_EX*, the typesetting engine that this reader was typeset in.

⁸ An online search for “how to install a Linux VM” will lead to many results.

⁹ Thompson’s *Reflections on Trusting Trust* is a terrifying look at why you can never completely trust your compiler

¹⁰ See this *Wired Magazine* tribute about Dennis Ritchie to understand why he deserves more name recognition

¹¹ Except for the machine specific parts, which were written in Assembly Language for each particular computer. But, the bulk of the operating system was written in C

Figure 2 shows an example of the Linux terminal, which is the interface that you will use in CS 107 to edit, compile, run, and debug your programs. Once you have worked out how to log into the Myth machines (again, see the course website), the next critical event is to choose an editor. There are multiple choices, but `vim` and `emacs` are two popular choices. Whichever editor you end up choosing, you should learn as many keyboard shortcuts as you can: the faster you are able to navigate around your programs, the easier it will be to concentrate on the coding itself.

The Unix Philosophy

The Unix creators had a simple philosophy¹²: command-line programs should be simple and should do one thing well. For example, the `cat` program has one job, and that is to read a file and print the file to the terminal. It is a simple program, and it does the job well. We will spend a lot of time in CS 107 writing programs that do one thing well – sometimes you may think we abuse the notion of *simple* (because the programs we write can be complex to write), but all of the programs you write for class will do a single thing.¹³

Compiling programs using `gcc`

CS 107 is taught primarily in the C programming language, and the compiler we will use to turn our C programs into binary code is the *Gnu Compiler Collection*, otherwise known as `gcc`¹⁴. `Gcc` has many options for compiling our code, and it can compile code to *binary* (machine) code that can be run directly, or into *assembly* code, which can then be compiled into binary code. For the first half of the course, we will be compiling with the `-O0` optimization flag, which will produce code that is able to be debugged easily in the `gdb` debugger (more on the debugger later). Once we begin discussing assembly code, we will compile with the `-Og` optimization flag, which will make less verbose assembly code to look at.

Specifically, we will be compiling our code in the following way:¹⁵

```
$ gcc -g -O0 -std=gnu99 -Wall -Wfloat-equal -Wtype-limits \
    -Wpointer-arith -Wlogical-op -Wshadow \
    -fno-diagnostics-show-option program_name.c \
    -o program_name
```

The following briefly describes the options we use (and a backslash means that the line is continued onto the next line, so the entire text above is technically one long line):

- `gcc`] The name of the gcc program
- `-g` Include debugging information (necessary when using `gdb` for debugging)

```
cgregg@myth4:~/tmp$ cat hello.c
#include<stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}

cgregg@myth4:~/tmp$ gcc -g -O0 -std=gnu99 -Wall -Wfloat
-equal -Wtype-limits -Wpointer-arith -Wlogical-op -Wsha
-dow -fno-diagnostics-show-option hello.c -o hello
cgregg@myth4:~/tmp$ ./hello
Hello, World!
cgregg@myth4:~/tmp$
```

Figure 2: A typical Linux terminal window.

¹² as described by Ken Thompson.

¹³ Possibly in different ways. For example, you will write a sorting program that will sort based on options, such as reverse-ordering, or based on the length of a line.

¹⁴ `gcc` was initially created by Richard Stallman, a unique character in computer science, and the founder of the Free Software Foundation. All of the software we will use in CS 107 is both “free as in speech and free as in beer”

¹⁵ Only type the code in **bold**. The dollar sign at the beginning of the line is the command-line prompt

-O0 The optimization level (no optimization, good for debugging)

-std=gnu99 Use the gnu99 C standard, which allows (among other things) the ability to define for loop variables inside the loop declaration, e.g., `for (int i=0; i < 10; i++)`

-Wall, -W... Warning options, to provide useful warnings when compiling code.

-fno-diagnostics-show-option Suppress showing which warning option triggered a particular warning

`program_name.c` The C program name that we are compiling.

-o `program_name` The output program name that can be run by typing `./program_name`

Compiling programs using make

As the previous section describes, there are a lot of options to compiling a program using gcc! There are additional options to compile multiple files into one executable, too, and this will depend on the programs you are writing.

Instead of typing the entire gcc line every time you want to compile your program, we will provide you a text file called “Makefile” that has all of the necessary commands to properly compile all of your code for the projects. To compile your projects using a Makefile, simply run the following command¹⁶:

¹⁶ `make` is another GNU program that automates building programs.

```
$ make
```

That’s it! Make also performs another helpful function: if any of your source programs have changed, running make again re-builds the projects, automatically, and only re-builds the necessary programs.

A Makefile can contain instructions to build multiple programs, and many of the CS 107 assignments have multiple programs in them. If you only want to build a particular program, you can do so by telling make to build just one. For example, assignment 1 has four programs, called `code`, `sat`, `automata`, and `utf8`. To just build `sat` (for instance), you would type:

```
$ make sat
```

In order to remove the programs and re-compile all of them, you type:

```
$ make clean
$ make
```

Using git

You will work on all of your projects for CS 107 in your local directory on the Stanford computer network. However, your code will be copied from a different directory, also located on the network. Specifically, for each assignment, you will run a command similar to the following (which is the command for assignment 0):

```
$ git clone /afs/ir/class/cs107/repos/assign0/{$USER} assign0
```

This command runs a program called `git`¹⁷, which is a *version control system* that is designed to keep versions of your code so that you can always go back to previous versions. One thing that `git` does that might be important as you go through the course is that it can, in essence, keep backups for you. Any time you change the text in a file, you can *commit* that file into the repository, and then you can go back to that version whenever you want. If you want even more backup security, you can *push* your changes to the master repository where the initial cloned version resides. In fact, whenever you run `sanitycheck` (see the next section for details), this is what that program does.

The following example demonstrates how you might use `git` for assignment 1. Assume you change the `utf8.c` program, and want to save the current state as a backup. You would run the following commands:

```
$ git commit -m "Added code to test on 1-byte examples" utf8.c
$ git push
```

The first command commits the `utf8.c` file to your local repository, with a message (`-m`) that says what the change accomplished¹⁸

The second command pushes the change to the master repository, where it originally came from. All previous commits are stored, as well.

Let's say that you made another change and committed it, but it turns out that you deleted an important function, accidentally. If you need to go back to a previous commit temporarily, you can do so as follows:

```
$ git log
commit d0fcf945876592e2bb74e26f31fcad7ffb850451
Author: Chris Gregg <tofergregg@gmail.com>
Date:   Fri Dec 15 14:38:27 2017 -0800
```

Took out unneeded function

```
commit fb8f84f57672b0b9ded5cd4dd737141650cf7aea
Author: Chris Gregg <tofergregg@gmail.com>
Date:   Fri Dec 15 14:33:34 2017 -0800
```

Added code to test on 1-byte examples

¹⁷ `git` is distinct from `GitHub`; the former is the program that provides the versioning tools, and the latter is a website that provides repository storage and sharing.

You might get warning message that "warning: push.default is unset;" To stop the warning, type the following: `git config --global push.default simple`

¹⁸ Good commit messages are important for finding old versions!

...

```
$ git checkout fb8f84f57672b0b9ded5cd4dd737141650cf7aea
M      code.c
Note: checking out 'fb8f84f57672b0b9ded5cd4dd737141650cf7aea'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the checkout command again. Example:

```
git checkout -b new_branch_name
```

HEAD is now at fb8f84f... Added code to test on 1-byte examples

The big long number from the commit log (fb8f84f57672b0b9ded5cd4dd737141650cf7aea) is used to uniquely identify that version. Now that you have checked out that version, you can go look at it, and copy anything you need from that file to a temporary file. Then, you can go back to the latest version and fix your error:

```
$ git checkout master
Previous HEAD position was fb8f84f... Added code to test on 1-byte examples
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 2 commits.
(use "git push" to publish your local commits)
```

Testing and using sanitycheck

A critical task you must do while writing your programs is compiling, running, and testing your code regularly, and often. The definition of *often* does not mean “when you are done with the program” – it means every time you make a change to your code. Some programmers compile after *every block of code* they write, and this is a very good idea. A good strategy is to write a block of code, compile it, and then fix all warnings and errors. Then you compile again. You can catch errors quickly, and you know exactly where in the code the error is happening. In CS 107, we frown upon¹⁹ code that has warnings, and your code should always compile cleanly with no errors or warnings. You are not allowed to even submit code with compilation errors, in fact.

¹⁹ i.e., take off points

Learning how to test your code properly takes time. In CS 107, for each assignment we provide you with a program called `sanitycheck` that runs some rudimentary tests on your code. When you run `sanitycheck`, you get a list of tests and whether your code passed or failed each test. When we grade your code, we run an additional set of tests in a similar manner, and those tests are comprehensive and are designed to find any corner cases in your code.

To run sanitycheck on your code, type the following from a project directory:

```
cgregg@myth32:~/cs107/assign1$ tools/sanitycheck
```

```
Will run default sanity check for assign1 in current directory
.ir.stanford.edu/users/c/g/cgregg/cs107/assign1.
```

```
+++ Test A-Make on .ir.stanford.edu/users/c/g/cgregg/cs107/assign1
Descr:  verify project builds cleanly using 'make'
Command: make
OK:  Clean build
```

```
+++ Test B-SatRange on .ir.stanford.edu/users/c/g/cgregg/cs107/assign1
Descr:  min and max value of range for 12-bit signed
Command: ./sat 12
MISMATCH: Submission output does not match sample
Sample output:
12-bit signed integer range
min: -2048  0xfffffffffff800
max:  2047  0x000000000000007ff
Your output:
12-bit signed integer range
min: -2048  0xfffffffffff800
max:  2048  0x00000000000000800
```

```
+++ Test C-SatAdd on .ir.stanford.edu/users/c/g/cgregg/cs107/assign1
Descr:  test saturating addition overflow case
Command: ./sat 8 127 45
MISMATCH: Submission output does not match sample
Sample output: 127 + 45 = 127
Your output:   127 + 45 = 128
... (more follows)
```

For this example, there was a mismatch with Test B-SatRange, and you can see the expected output and your output.

You can also create your own (somewhat rudimentary) tests using sanitycheck, too. Each assignment comes with a custom_tests file that you can modify to add your own tests. For example, the default assignment 1 custom_tests file is:

```
sat 16 -32768 -32768
automata 90
utf8 0x20AC
```

You run your custom tests as follows:

```
$ tools/sanitycheck custom_tests
```

It is hard to overstate the importance of testing your code for CS 107. Students who learn to test their code early and often have fewer problems writing good code, and they ultimately do better in the course.

Number Formats Used in CS 107

TODAY'S COMPUTERS ARE, at the lowest level, *binary* machines²⁰, and all numbers in a computer are represented in base 2 notation, which consists entirely of the digits 0 and 1. Humans are generally used to base 10, which has eight additional digits: 2, 3, 4, 5, 6, 7, 8, and 9. Additionally, base 16 is also used when representing numbers in a computer, because 16 is a multiple of 2, and therefore it is easy to convert between base 2 and base 16. Base 16 has the following digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f.

²⁰ There have been **computers** based in a *ternary* format, as well

Therefore, in CS 107, you must be extremely familiar with all three bases that are used to represent numbers, and you must be able to convert between them quickly (or be familiar with online conversion tools). We will describe what you need to know in this chapter, and you would be well-advised to learn the details as soon as you can.

Base 10

Let's start by reminding ourselves about the number system we use every day. The other two bases we will discuss are analogous, but it might help to think a bit deeper about base 10 before we move onto the other bases.

Base 10 numbers have the digits 0-9, and counting proceeds by adding one digit at a time, until the digits are exhausted, at which point you add a power of 10, and then repeat from 0 again. That is a fancy way of saying that when you get to 9, you continue counting at 10:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ..., 97, 98, 99, 100, 101, ..., 998, 999, 1000, ...

A base 10 number with three digits, $d_2d_1d_0$ is actually represented as follows:

$$d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0$$

So, for example, the number 425 is:

$$4 \times 10^2 + 2 \times 10^1 + 5 \times 10^0 = 400 + 20 + 5 = 425$$

When adding two base 10 numbers together, we *carry* numbers greater than 9 to the next higher power-of-10 position:

$$\begin{array}{r} 11 \\ 178 \\ + 456 \\ \hline 634 \end{array}$$

When subtracting two base 10 numbers, we *borrow* when we are trying to subtract a digit from a smaller digit, treating the subtraction as the $10 + d_{\text{smaller}} - d_{\text{larger}}$, and reducing the digit that was carried from by one:

$$\begin{array}{r} 214 \\ 6\cancel{3}4 \\ - 416 \\ \hline 218 \end{array}$$

Two definitions you should become familiar with are *least significant digit* and *most significant digit*:

Least significant digit: The digit farthest to the right of a number.

Most significant digit: The digit farthest to the left of a number.

The least significant digit is the digit that matters the least in a number. In the number 456, the 6 represents just 6, and the 4 represents 400, so the 6 is least significant and the 4 is most significant.

Base 2 (binary)

Base 2, or *binary* notation, has only the digits 0 and 1. Just like counting in base 10, when the digits are exhausted, you must start with a new 1:

0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, ...

As you can see, binary numbers can get long quickly, as there are only two digits to write with!

Base 2 notation is comprised of powers of 2 (instead of powers of 10), so a base 2 number with four digits, $b_3b_2b_1b_0$ is represented as follows:

$$b_3 \times 2^3 + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

For example, 1101₂ is:

$$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13_{10}$$

In other words, 1101₂ is 13₁₀, and this is the method we use to determine a base 10 number if we have a base 2 number.

To convert a base 10 number into base 2, we must perform a series of divisions, taking the remainder each time, and growing the binary number from least significant digit to most significant digit, until the division produces zero. For example, to convert 1234₁₀ to binary:

To differentiate between binary and decimal representations, we use b and d following each number

$1234d/2 = 617$, remainder 0	the least significant digit is 0
$617d/2 = 308$, remainder 1	the next digit to the left is 1
$308d/2 = 154$, remainder 0	the next digit to the left is 0
$154d/2 = 77$, remainder 0	the next digit to the left is 0
$77d/2 = 38$, remainder 1	the next digit to the left is 1
$38d/2 = 19$, remainder 0	the next digit to the left is 0
$19d/2 = 9$, remainder 1	the next digit to the left is 1
$9d/2 = 4$, remainder 1	the next digit to the left is 1
$4d/2 = 2$, remainder 0	the next digit to the left is 0
$2d/2 = 1$, remainder 0	the next digit to the left is 0
$1d/2 = 0$, remainder 1	the next digit to the left is 1

Therefore, the final result is

$$1234d = 10011010010b$$

We won't expect you to convert from decimal to binary regularly, but you should be able to do it. To convert $10011010010b$ to decimal (it is only necessary to calculate bits that are 1)

$$10011010010b = 2^{10} + 2^7 + 2^6 + 2^4 + 2^1 = 1024 + 128 + 64 + 16 + 2 = 1234d$$

Because we will use them often, you should simply memorize the first ten powers of 2:

$$2, 4, 8, 16, 32, 64, 128, 256, 512, 1024$$

When adding two binary numbers together, there are many more carries than in decimal addition, because every time 1 is added to 1, there is a carry. Also, remember that $1 + 1 = 10$, and $1 + 1 + 1 = 11$:

$$\begin{array}{r} 11 \\ 1011 \\ + 1011 \\ \hline 10110 \end{array}$$

When subtracting two binary numbers, the borrow happens any time a 1 is subtracted from a 0:

$$\begin{array}{r} 10 \\ 10110 \\ - 1010 \\ \hline 1100 \end{array}$$

Base 16 (hexadecimal)

Base 16 may at first seem like an odd choice for a base for computing. It has more digits than decimal, and computers are fundamentally binary. However, one of the downsides of binary is that binary

numbers are long and difficult for humans to cognitively process.²¹ Hexadecimal (hex) numbers, on the other hand, are four times shorter than their equivalent binary representations, and although reading a number with letters for some of the digits does take some time to get used to, reading hex numbers becomes second nature with practice. Additionally, and the underlying reason why we often use hexadecimal notation is because the conversion from binary to hexadecimal, and back again, is extremely easy (as we shall see in a few paragraphs).

²¹ i.e., they are hard to read!

A hexadecimal number (represented with 0x preceding the number, e.g., 0x4a) has sixteen digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f. The letters a–f represent the numbers 10–15. When counting, a 1 is added after the last digit is f:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, 10, 11, 12, 13, ..., fc, fd, fe, ff, 100, 101, ...

Because both the base for both binary and hexadecimal formats are divisible by 2, the conversion between them is straightforward. Every four binary digits (or *bits*) is equivalent to one hexadecimal digit. Tables 1 and 2 show the conversion from hex digit to binary (and decimal). You should memorize the binary representations for each hex digit. One trick is to memorize A (1010), C (1100), and F (1111), and the others are easy to figure out.

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111

Table 1: Hexadecimal to binary conversion for 0 – 7

Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

Table 2: Hexadecimal to binary conversion for 8 - 15

Translating between binary and hexadecimal is trivial if you know the digits. For example:

$$0x135af2 = 0001\ 0011\ 0101\ 1010\ 1111\ 0010b$$

Each hex digit converts directly to a 4-digit binary number:

$$0x1 = 0001$$

$$0x3 = 0011$$

$$0x5 = 0101$$

$$0xa = 1010$$

$$0xf = 1111$$

$$0x2 = 0010$$

Translating between hexadecimal and binary is just as easy. For example:

$$0101\ 1111\ 0111\ 1100\ 1111\ 0010b = 0x5f7cf2$$

Translating from hex to decimal is similar to converting from binary to decimal. A 4-digit hexadecimal number in the form $x_3x_2x_1x_0$ is equal to:

$$x_3 \times 16^3 + x_2 \times 16^2 + x_1 \times 16^1 + x_0 \times 16^0$$

For example:

$$0x1ea5 = 1 \times 16^3 + 14 \times 16^2 + 10 \times 16^1 + 5 \times 16^0 = 7845d$$

To translate from decimal to hexadecimal, you perform a series of divisions by 16 with remainders, which is similar to the decimal to binary translation. You will not have to perform those translations by hand in CS 107, and it is reasonable to use a computer to do so.²²

Using `man ascii`

The `man` command in Linux is one of the most helpful commands for quickly finding something out about another command, or a library function. If you need to quickly determine the ASCII character values in decimal or hex, or if you need to quickly determine the decimal or hex conversion for numbers less than 255, you can do so with the `man ascii` command:

²² You can perform the conversion on the command line as follows (this assumes you have Python installed):

```
$ python -c "print bin(1234)"
0b10011010010
$ python -c "print
hex(0b1010101)"
0x55
$ python -c "print 0xabcd1234"
2882343476
$ python -c "print bin(0xa1b2)"
0b1010000110110010
```

ASCII(7) Linux Programmer's Manual ASCII(7)

NAME

ascii - ASCII character set encoded in octal, decimal, and hexadecimal

DESCRIPTION

ASCII is the American Standard Code for Information Interchange. It is a 7-bit code. Many 8-bit codes (such as ISO 8859-1, the Linux default character set) contain ASCII as their lower half. The international counterpart of ASCII is known as ISO 646.

The following table contains the 128 ASCII characters.

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
000	0	00	NUL '\0'	100	64	40	@
001	1	01	SOH (start of heading)	101	65	41	A
002	2	02	STX (start of text)	102	66	42	B
003	3	03	ETX (end of text)	103	67	43	C
004	4	04	EOT (end of transmission)	104	68	44	D
005	5	05	ENQ (enquiry)	105	69	45	E
006	6	06	ACK (acknowledge)	106	70	46	F
007	7	07	BEL '\a' (bell)	107	71	47	G
010	8	08	BS '\b' (backspace)	110	72	48	H
011	9	09	HT '\t' (horizontal tab)	111	73	49	I
012	10	0A	LF '\n' (new line)	112	74	4A	J
013	11	0B	VT '\v' (vertical tab)	113	75	4B	K
014	12	0C	FF '\f' (form feed)	114	76	4C	L
015	13	0D	CR '\r' (carriage ret)	115	77	4D	M
016	14	0E	SO (shift out)	116	78	4E	N
017	15	0F	SI (shift in)	117	79	4F	O
020	16	10	DLE (data link escape)	120	80	50	P
021	17	11	DC1 (device control 1)	121	81	51	Q
022	18	12	DC2 (device control 2)	122	82	52	R
023	19	13	DC3 (device control 3)	123	83	53	S
024	20	14	DC4 (device control 4)	124	84	54	T
025	21	15	NAK (negative ack.)	125	85	55	U
026	22	16	SYN (synchronous idle)	126	86	56	V
<hr/>							
Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
027	23	17	ETB (end of trans. blk)	127	87	57	W
030	24	18	CAN (cancel)	130	88	58	X
031	25	19	EM (end of medium)	131	89	59	Y
032	26	1A	SUB (substitute)	132	90	5A	Z
033	27	1B	ESC (escape)	133	91	5B	[
034	28	1C	FS (file separator)	134	92	5C	\ '\\'
035	29	1D	GS (group separator)	135	93	5D]
036	30	1E	RS (record separator)	136	94	5E	^

037	31	1F	US (unit separator)	137	95	5F	_
040	32	20	SPACE	140	96	60	`
041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	\$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	`	147	103	67	g
050	40	28	(150	104	68	h
051	41	29)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

C Primer

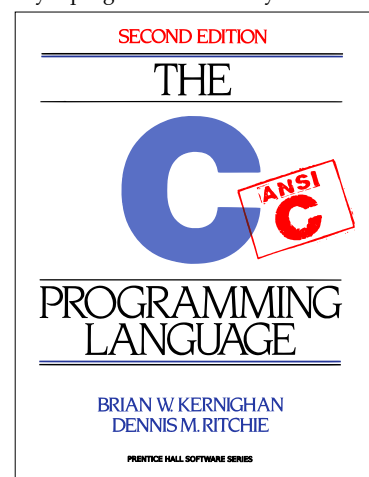
AS YOU BEGIN CS 107, we expect that you have had a good introduction to programming course, and a good follow-on course that covers typical data structures content for a second programming course. At Stanford, CS 106A and CS 106B are excellent preparation for CS 107 and they are taught in Java and C++, which are both based on C syntax, and both have similar function structure, loop constructs, variable naming and scoping. The transition to C after coding in Java and/or C++ is relatively straightforward, and you should quickly feel at home. If your background is in a language like Python, Ruby, R, PHP, or a functional language like Lisp or Haskell, you may need more time to assimilate to C.

Whatever your background, C does come with its own intricacies, some of which are historical in nature, and some of which might be, to some, a bit dated. You will write a great deal of of pointer-based code that provides very little safety, and this is just how C rolls. C does not have any notion of classes, and memory management is neither garbage collected like in Java, nor object-based like in C++. But, C is actually a language that is simple enough that you can become an expert in the language – once you learn a few crucial details, C becomes a language that allows you to do exactly what you want, and fast, though sometimes at the expense of “ease.”

As discussed in *Unix, the Command Line, gcc, and Makefiles*, C was invented in the 1970s by Dennis Ritchie as a *systems* language that could be used to write low-level, portable code that could be run on any computer with a C compiler. Ritchie and Ken Thompson wrote Unix in C, and the popularity of Unix cemented C into a key role in systems programming ever since.

This chapter is designed to provide an introduction to programming in C *-vs-* other C-syntax based languages – we primarily want to show you some of the nuances in C that you will need to get used to as you start CS 107. C++ actually shares many of the features described in the chapter (and Java shares some, as well), but in CS 106B we generally focus on other C++ features than the ones discussed here. The chapter isn’t in any particular order, and can be used as a reference to some of the syntax as you begin CS 107.

Figure 3: *The C Programming Language* by Brian Kernighan and Dennis Ritchie (also known as “K&R”) is the definitive text on C and should be in every C programmer’s library.



C Basics

If you are comfortable with either Java or C++, C will be familiar, although if you haven't used pointers before, you will need to learn about them. The following annotated C program describes many of the typical parts of a C program:

```
/* file: personhours.c
 * Calculates persondays worked for an array
 * of jobs and an array of people per job
 */

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

#define WORKDAYSPERYEAR 250
#define WORKHOURSPERDAY 8

void scalearray(int *input, const int *scale,
                size_t nelems);
void converttohours(int *input, size_t nelems);
int workyearstohours(int days);

int main(int argc, char *argv[])
{
    // jobdays initially holds
    // the number of days worked per job
    int jobdays[] = {1, 5, 7, 2, 4, 8, 3};

    // workersperday holds the number of workers
    // who worked each day on a job
    int workersperday[] = {2, 5, 7, 1, 9, 4, 5};

    size_t arrsz = sizeof(jobdays) /
        sizeof(jobdays[0]);

    converttohours(jobdays, arrsz);
    scalearray(jobdays, workersperday, arrsz);

    for (int i=0; i < arrsz; i++) {
        printf("%d", jobdays[i]);
        i == arrsz-1 ? printf("\n") : printf(", ");
    }
    return 0;
}
```

Comments come in two forms. Multiline comments start with `/*` and end with `*/`, and cannot be nested. Single-line comments start with `/*` and only go to the end of a line.

These are the two typical library include statements, for input/output and the C standard library.

The `#define` syntax defines a constant that the C Preprocessor replaces in your code before compilation.

Function prototypes declare functions before they are defined. A function must be declared before it can be used in C, and we normally put function declarations at the top of a file so we can use them from any function below.

All C programs begin at the `main` function, which must return an `int`. See below for more information about the `argc` and `argv` command line arguments.

C arrays can be declared with the `[]` notation and an initial comma separated list of values. The compiler will determine the size of the array accordingly.

The `size_t` type is an unsigned integer type that holds the size of an object (like an array).

See *Bits and Bytes* for information about `sizeof`. This is the standard way to determine the number of elements in a fixed-size array. This *does not* work for arrays that are created via `malloc`.

See below for details about `printf`, which is the standard way to print to the terminal in C.

The "ternary operator" is a shorthand if/else statement. The syntax is:

```
a ? x : y
```

and can be read as "if *a* then *x* else *y*".

```

/* multiply each element in input by the
 * corresponding scaling element in scale
 */
void scalearray(int *input, const int *scale,
                size_t nelems)
{
    for (size_t i=0; i < nelems; i++) {
        input[i] *= scale[i];
    }
}

/* convert workdays array to hours */
void converttohours(int *input, size_t nelems)
{
    for (size_t i=0; i < nelems; i++) {
        *(input+i) = workyearstohours(*(input+i));
    }
}

/* convert a workday into a number of hours */
int workyearstohours(int days)
{
    return days * WORKDAYSPERYEAR *
           WORKHOURSPERDAY;
}

```

Functions must be declared with a return type, although that type can be void if nothing will be returned. See below for information about const.

for loops may declare the loop variable inside the declaration, with gcc flag -std=gnu99.

Array elements can be accessed using the [index] notation. Note that any pointer can act as an array in this way. Also note that arrays and pointers do not carry information about the number of elements, and you need to pass along this information as a parameter (in this case, nelems).

Array elements can also be accessed using *pointer arithmetic* with dereferencing.

Function names in C are generally either squished together (as in workyearstohours or with underscores as separators, e.g., workyears_to_hours. You should get used to both. camelCase is not generally preferred.

The printf statement

The printf statement is the preferred way in C to output text to the screen or to *standard out* (stdout). The printf function has the following function declaration:

```
int printf(const char *format, ...);
```

The first parameter is the *format string*, which contains the string that will be printed, along with optional *embedded format specifiers*, which start with the “%” character. For every format specifier, there is an additional parameter that provides the value to be formatted.²³ Most often, the return value for printf is ignored, but the return value is the number of characters that were printed. For example,

```

int x = 5;
char c = 'A';
char *str = "some words";
printf("x: %d, a character: %c, a string: %s\n", x, c, str);

```

Special characters in the format string are prepended with a backslash, “\”. The most common special character is the newline

²³ for a comprehensive list, see [this reference](#).

character, `\n`, which prints a newline (similar to `"cout << endl;"` in C++).

The output from the above `printf` statement is:

```
x: 5, a character: A, a string: some words
```

Table 3 shows the common `printf` specifiers we will use in CS 107.

The print specifier can also have a length sub-specifier, which modifies the length of the data type. For example, if you want to print out an unsigned long, you need to use the `%lu` format specifier, which will expect an unsigned long variable in the parameter list. E.g.,

```
unsigned long x = 5000000000;
printf("x: %lu\n", x);
```

Output:

```
5000000000
```

Gcc will give a warning if you attempt to specify an incorrect length for the parameter you pass in. E.g.,

```
int x = 50;
printf("x: %lu\n", x);
```

```
$ gcc -g -O0 -std=gnu99 printf_ex.c -o printf_ex
printf_ex.c: In function 'main':
printf_ex.c:7:5: warning: format '%lu' expects argument of type 'long unsigned int',
but argument 2 has type 'int' [-Wformat=]
    printf("x: %lu\n", x);
    ^
```

The Standard Library

C comes with a robust *standard library* that has hundreds of functions available to C programmers. The two most common libraries are `stdlib` and `stdio`, and they will be included in all of our programs. Other common libraries include `string`, `math`, `assert`, `errno`, `limits`, and `stdbool`.²⁴ We will focus on the `string` library (`string.h`) in a later chapter. For CS 107, we will have assignments where you will write some standard library functions, and you should be familiar with a number of the libraries.

All of the standard library function specifications are available in the Linux reference manual, which you can access via the `man` command. For example, to look up the `atoi` function, which converts a string to an integer, you type:

```
$ man atoi
```

which brings up the manual entry for the function from the `stdlib` library. All the standard library functions are in section 3

Table 3: Common `printf` specifiers

Spec.	Output	Example
d or i	Signed integer	-5
u	Unsigned integer	12
x	Unsigned hex int	9f02
f	Floating point	123.45
g	Alt. floating point	1.23E+02
c	Character	a
s	String	text
p	Pointer addr	b8000000
%%	Prints a single %	%

²⁴ See a complete list [on Wikipedia](#).

of the Linux reference manual, although sometimes there are also Linux command line functions listed, as well. For example, typing `man printf` will bring up the manual page for the Linux command line `printf`, and if you want the C library `printf`, you must type `man 3 printf` to bring up the version from the standard library.

For CS 107, you should generally utilize the standard library as often as possible. Unless we specifically ask you to program a particular library function²⁵, you should always use a library function for the task you are trying to accomplish, if one exists. Many students have tried to re-write standard library functions unsuccessfully, and it isn't worth the time²⁶ to try and write them on your own. The standard library functions are fast, efficient, and well-tuned, as they are used by millions of programs daily.

²⁵ For example, you will write a version of `cat`, a version of `ls`, and a version of `which`.

²⁶ or the potential for missed points because you wrote them incorrectly!

The command line arguments, argv and argc

One way to provide your C program data is through *command line arguments*. For example, your program might accept three words as input, and you could type them into the command line when you start the program:

```
$ ./alphabetize dog hamster cat
cat dog hamster
```

The command line arguments are available to C programs through two variables, `argc` and `argv`, and are parameters to `main`:²⁷

```
int main(int argc, char *argv[]);
```

or, alternatively (and exactly equivalent):

```
int main(int argc, char **argv);
```

`int argc`: This parameter holds the number of parameters, *including the program name*. For our example above, `argc` would have the value 4, because there are four arguments: `./alphabetize`, `dog`, `hamster`, and `cat`.

`char *argv[]`: This parameter holds an array of C string pointers. It is equivalent to `char **argv`. Each string can be accessed using bracket notation, e.g., `argv[1]` or pointer arithmetic and dereference notation, e.g., `*(argv + 1)`.

The following is a full example using the command line arguments:

```
// file: argument_list.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    for (int i=0; i < argc; i++) {
```

²⁷ There is a third parameter we will use in assignment 2, called `envp`, which provides access to the Linux environment variables.

```

        printf("Argument %d: %s\n", i, argv[i]);
    }
    return 0;
}

```

```

$ ./argument_list dog hamster cat
Argument 0: ./argument_list
Argument 1: dog
Argument 2: hamster
Argument 3: cat

```

C Arrays and C Strings

Arrays in C are simple, although they have some caveats to be aware of. Arrays are a contiguous block of memory with a fixed length. A programmer can access elements in an array using either bracket notation (e.g., `arr[2]`) or by dereferencing an offset from the beginning of the array (e.g., `*(arr + 2)`). Although arrays sometimes behave like pointers (see below), they are distinct and should not be confused with pointers. The following program is an example of an array declaration and accessing the array elements:

```

// file: array.c
#include<stdio.h>
#include<stdlib.h>

void sizeof_test(int arr[]) {
    printf("sizeof(arr) in function: %d\n", (int)sizeof(arr));
}

int main()
{
    int arr[] = {1,3,4,2,7,9};

    printf("%d\n",arr[2]); // prints 4
    printf("%d\n",*(arr+5)); // prints 9

    // set the value of the third element
    arr[2] = 42;
    printf("%d\n",arr[2]); // prints 42

    printf("sizeof(arr) in main: %d\n",(int) sizeof(arr));
    sizeof_test(arr);

    return 0;
}

$ gcc -g -O0 -std=gnu99 -Wall array.c -o array
$ ./array

```

```

4
9
42
sizeof(arr) in main: 24
sizeof(arr) in function: 8

```

In the program above, `arr` is an array, but critically, *not* a pointer. When accessing the array elements by either the bracket notation or by dereferencing an offset, the array name is converted to a pointer to the first element of the array to do the calculation²⁸, however, so it does behave like a pointer in many circumstances.

²⁸ See [this StackOverflow post](#) for more information.

When applied to arrays, the `sizeof` operator returns the number of bytes of the entire array. For the array in the example above inside `main`, `sizeof(arr)` returned 24 because each `int` is 4-bytes long²⁹. In the `sizeof_test` function, the array parameter is converted into a pointer to the first element in the array, and therefore `sizeof(arr)` returns the size of a pointer in bytes, which is 8 bytes on the Myth machines.

²⁹ see Chapter *Bits and Bytes* for information about the size of C data types.

C does not provide any array bounds checking. You as the programmer must ensure that you do not try to read or write past either end of an array, lest you cause a memory error. As seen above, when passing arrays into a function, you must also explicitly pass the number of elements if the function needs that information.

Often, we will declare an array of a particular size, without initializing the values:

```
char buffer[256]; // a 256 byte buffer
```

The scope for the array is within the function (or block) that it is declared in, and it has space for exactly 256 bytes of memory.

Arrays in C are not hard to understand, but you do need to be careful to work with them properly.

C strings are simply arrays of chars, with a truncating 0 byte.³⁰ Unlike Java and C++, C strings are not objects, and do not have any built-in functionality. They are simply char arrays. C strings can be declared directly in two different ways, but there are key differences. The following program demonstrates both methods:

³⁰ This is called “null-terminating” a string. Don’t confuse this with the `NULL` pointer. Although `NULL` has the value 0, its type is `void *`, and not `char`.

```

// file: c_string_decl.c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    char cstr1[] = "A string";
    char *cstr2 = "Another string";

    printf("cstr1: %s\n", cstr1);
    printf("cstr2: %s\n", cstr2);
    return 0;
}

```



```
$ gcc -g -O0 -std=gnu99 c_string_decl.c -o c_string_decl
$ ./c_string_decl
cstr1: A string
cstr2: Another string
```

Both declarations of strings work, but `cstr1` is an array, and `cstr2` is a pointer to an array. The difference is subtle, but there is another issue that is more insidious: `cstr1` is placed in memory onto the *stack* (which we will discuss in Chapter *Pointers, the Stack, and the Heap, and Pointers to Functions*), and we are able to read and write to the string characters. On the other hand, the string that `cstr2` points to gets placed into *read only memory* and we are only allowed to read from the contents pointed to by `cstr2`. The proper declaration for `cstr2` should have been:

```
const char *cstr2 = "Another string";
```

If, in the original program, we had attempted this:

```
// change the first characters to lowercase
cstr1[0] = 'a';
cstr2[0] = 'a';
```

The program would have crashed with a *segmentation fault* when attempting to change the value of `cstr2` (but changing the value of `cstr1` is fine). To be clear: the compiler would not have caught this error, and it would not have produced a warning, either.

Because Cstrings are terminated with a 0 byte, any time you use a string in a function that expects a string, you must ensure that the last byte in the string is a 0. In C, there are (at least) two ways to directly set the last character to NULL:

```
char buffer[5];
buffer[4] = '\0';
buffer[4] = 0;
```

The first case, `'\0'` is generally preferred, as it is a proper char type. The second, `0`, will work, as well (without warning), but `0` on its own has a type of `int`.

Note that we have set `buffer`'s index 4 to 0, because as a 5-integer buffer the indices go from 0-4. This means that a string buffer of length 5 *can only hold 4 meaningful string characters*. The string "hello" needs a buffer of length 6 to be a proper Cstring. Forgetting this causes many off-by-one errors with strings!

Pointers and void *

If you took CS 106B, you have worked with pointers before. If you only have Java experience, you may not have seen pointers before. A pointer is, simply, a variable that holds a memory address. On the Myth machines, all pointers are 8 bytes (64 bits) long. Although under the hood all pointers are identical, C enforces a type on pointers so the compiler can calculate the element offsets for arrays.³¹

³¹ This has an added benefit of providing some type safety when compiling, and the compiler can warn you when you use a pointer of the incorrect type.

Take, for instance, the following code:

```
// file: array_offsets.c
#include<stdlib.h>
#include<stdio.h>

void print_array(int *arr, size_t nelems)
{
    for (size_t i=0; i < nelems; i++) {
        printf("%d",arr[i]);
        i == nelems - 1 ? printf("\n") :
            printf(", ");
    }
}

int main()
{
    int values[] = {1,3,5,2,4,6};
    print_array(values, sizeof(values) /
        sizeof(values[0]));
    return 0;
}

$ gcc -g -O0 -std=gnu99 -Wall array_offsets.c -o array_offsets
$ ./array_offsets
1, 3, 5, 2, 4, 6
```

In the function, `print_array`, the compiler knows that each element in the array is an `int`, and it can calculate where in memory the next element is when executing `arr[i]`.³²

We can access the address of all variables in C with the *address-of* operator, `&`. We say that this address of a value is a *reference* to that value.³³ For example:

```
int x = 12;
int *xptr = &x; // xptr now points to x
```

As we have seen above, we can get to the value stored by a pointer by *dereferencing* the pointer with the `*` operator:

```
int y = *xptr; // y now holds the value 12
```

When we have a pointer to a value, we can modify the original value in the location at that pointer. This makes functions such as `swap` possible:

```
// file: swap.c
#include<stdio.h>
#include<stdlib.h>

void swap(int *x, int *y)
{
    int tmp = *x;
```

³² We will discuss the details in future chapters, but an `int` is a 4-byte value, and memory addresses refer to one-byte locations. Therefore, each `int` in `arr` takes up four memory locations, and the compiler needs to know that fact. If we had instead, declared values to be of type `long *`, each element would take up 8 bytes, because a `long` is an 8-byte data type. See Table 4 for more information.

³³ Do not confuse C references with C++ references! C references are simply pointers.

```

        *x = *y;
        *y = tmp;
    }

int main()
{
    int a = 5;
    int b = 12;
    printf("a: %d, b: %d\n",a,b);
    swap(&a,&b);
    printf("a: %d, b: %d\n",a,b);
    return 0;
}

$ gcc -g -O0 -std=gnu99 -Wall swap.c -o swap
$ ./swap
a: 5, b: 12
a: 12, b: 5

```

We will spend a lot of time during CS 107 looking at one further level of indirection, the *pointer to a pointer*. A pointer to a pointer is also a memory address, and it holds a pointer type. Take the following swap2 program:

```

// file: swap2.c
#include<stdio.h>
#include<stdlib.h>

void swap2(int **x, int **y)
{
    int *tmp = *x;
    *x = *y;
    *y = tmp;
}

int main()
{
    int arr[] = {5, 12};

    int *aptr = &arr[0];
    int *bptr = &arr[1];

    printf("a: %d, b: %d\n",*aptr,*bptr);
    swap2(&aptr,&bptr);
    printf("a: %d, b: %d\n",*aptr,*bptr);
    return 0;
}

$ gcc -g -O0 -std=gnu99 -Wall swap.c -o swap
$ ./swap

```

a: 5, b: 12

a: 12, b: 5

It is helpful to draw pictures for the situation covered in swap2. This is a critical understanding point in CS 107 – if you are at all in doubt about pointers or pointers-to-pointers, *draw a picture!* Figure 4 shows a possible memory layout for swap2; the addresses are made up for the purpose of this example. The arrows graphically show the pointer references, but often it is instructive to see the numbers, just to make it more concrete. Recall from above that pointers are 8 bytes, and ints are 4 bytes. Also note that once in swap2, the function no longer has access to the variables from main.

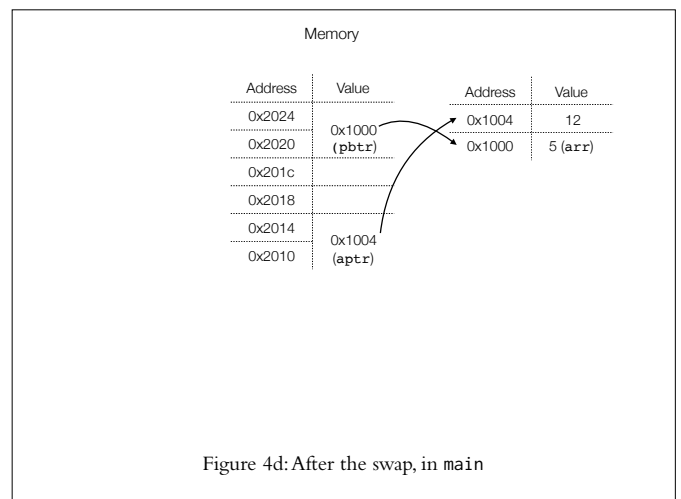
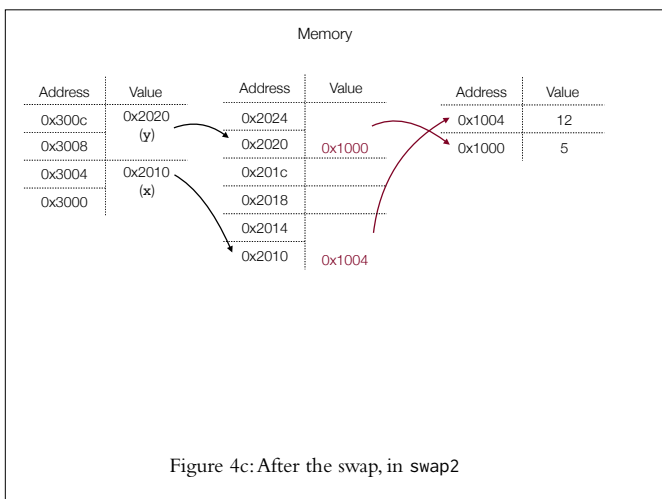
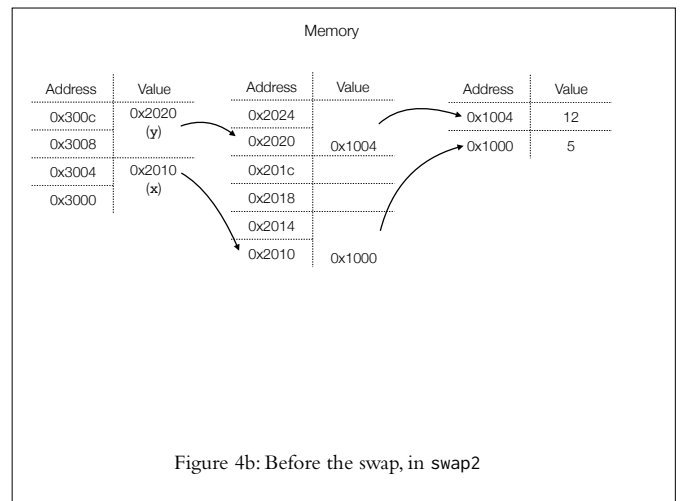
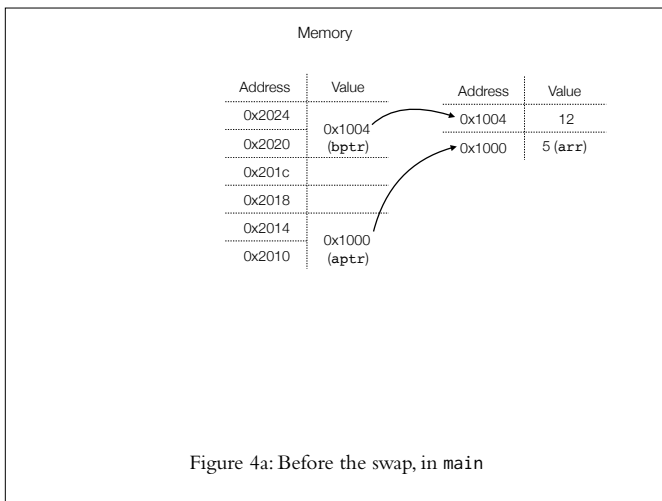


Figure 4: Possible memory layout for swap2

If our swap function actually did want to swap the two values in the original array and still had a pointer to a pointer passed in as a

parameter, it would be possible by *double dereferencing* the pointer:

```
void swap2(int **x, int **y)
{
    int tmp = **x;
    **x = **y;
    **y = tmp;
}
```

Note that a double-dereferenced `int **` is an `int`.

In this case, the diagram would look like Figure 5, where the original array elements were swapped.

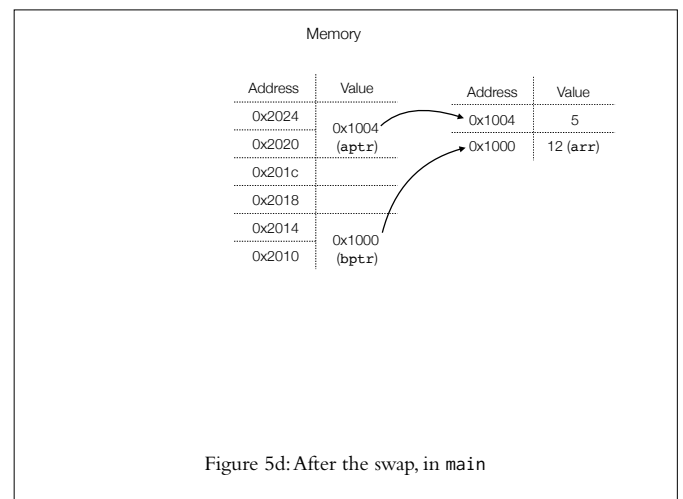
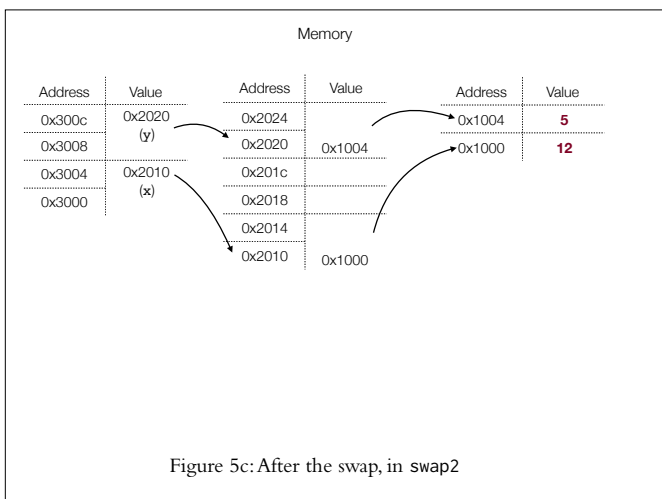
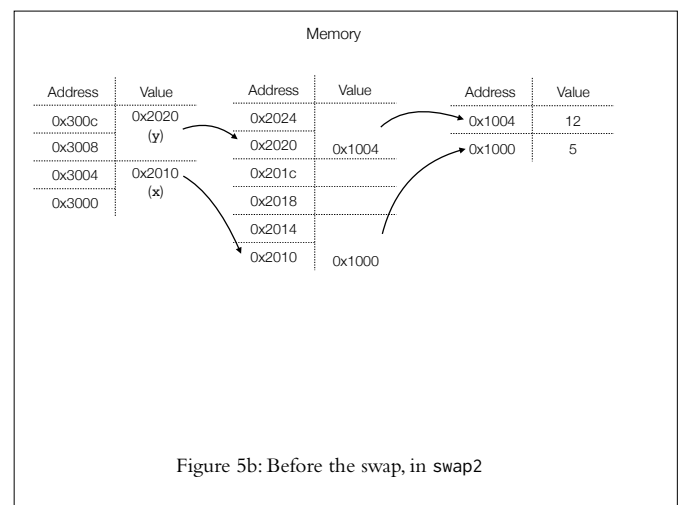
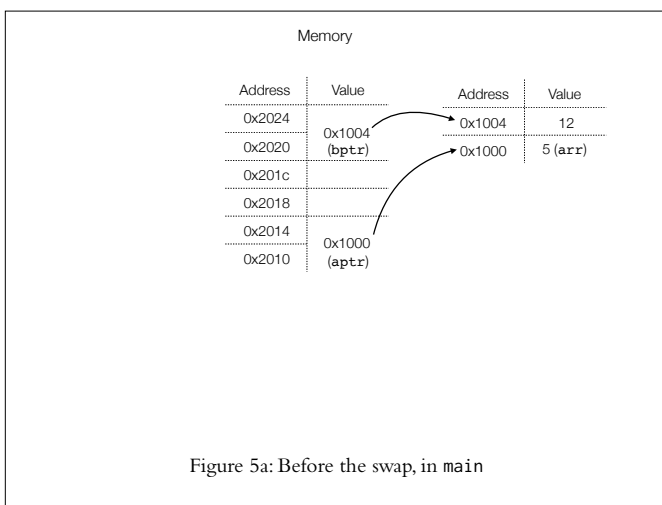


Figure 5: Possible memory layout for swap2 with double dereferencing

The one type of pointer that is significantly different than other pointers is the “void *” pointer. A void * pointer does not have

any type information associated with the data it points to. This necessarily means that when faced with a `void *` pointer, the compiler must be told to interpret the pointer as a particular type.³⁴ This means that you must cast `void *` pointers to some type. Often, we will not have enough information about the type to tell exactly what it is, so we have to rely on simply walking through the bytes one-by-one. In order to walk an array one byte at a time, the compiler must be told to interpret the pointer as pointing to a 1-byte data type. The only 1-byte data type we have in C is the `char` type, so you will frequently see casts to `char *` pointers, even though the underlying data has nothing at all to do with “characters,” and are simply bytes of another data type. For example, the following (annotated) program prints out the raw bytes (in hex) of any array, which is passed in as a `void *` array. Notice that we must have the length of the array in bytes:

```
// file: rawbytes.c
#include<stdio.h>
#include<stdlib.h>

void printbytes(const void *arr, size_t size)
{
    for (size_t i = 0; i < size; i++) {
        printf("%x",*((unsigned char *)arr + i));
        i == size-1 ? printf("\n") : printf(", ");
    }
}

int main(int argc, char **argv)
{
    // create buffer to hold longs
    long buffer[argc-1];

    // convert command line args to longs
    for (int i=1; i < argc; i++) {
        buffer[i-1] = atol(argv[i]);
    }

    // print the raw bytes
    printbytes(buffer, (argc-1) * sizeof(long));

    return 0;
}
```

```
$ gcc -g -O0 -std=gnu99 rawbytes.c -o rawbytes
$ ./rawbytes 1234 9876 1111 42
d2, 4, 0, 0, 0, 0, 0, 0, 94, 26, 0, 0, 0, 0, 0, 0, 57, 4,
0, 0, 0, 0, 0, 0, 2a, 0, 0, 0, 0, 0, 0, 0, 0
```

There is a good deal more to pointers that we will cover in *Pointers, the Stack, and the Heap, and Pointers to Functions!*

³⁴ But not necessarily the exact underlying type!

We are casting `arr` to an `unsigned char *` because an `unsigned char` is a 1-byte length data type, and we want the byte it returns to be an unsigned value between 0 and 255. We have to dereference the byte from the array after we do the pointer arithmetic.

We ignore the first argument, which is the program name, so we only need space for `argc-1` values, which are longs in this case.

Convert each argument from a string to a long.

Pass in the total number of bytes, which is the size of the array times the size of a long.

The memcpy and memmove Functions

As we go through CS 107, we will quickly reach a point where we need to copy raw bytes around in memory. There are two functions that accomplish this, `memcpy` and `memmove`. Both copy the bytes (rather than “moving”) them, but the difference is that `memcpy` is not guaranteed to work on *overlapping* memory. In other words, if you are copying part of an array to another part of the same array, and the bytes overlap, you must use `memmove` instead of `memcpy`. You should spend some time understanding how both functions work by looking at their respective man pages³⁵, but we will go over the details here. The function prototypes are as follows:

³⁵ man `memcpy` and man `memmove`

```
void *memcpy(void *dest, const void *src, size_t n);

void *memmove(void *dest, const void *src, size_t n);
```

Both functions expect a `void *` pointer to the destination address (`dest`) to copy the data, a `const void *` address of the location copying from (the source, `src`), and the number of bytes to copy.

Much like the example code above, neither function knows the underlying type of the source or destination pointers, but it moves the bytes one at a time.

C structs and the typedef statement

Although C does not have objects and is not an object oriented programming language, it does have the ability to package data together in the form of a struct. C structs allow you to define a contiguous region of memory that will hold other variables based on their types. To declare a struct, we use the following form:

```
struct tag {
    type a;
    type b;
    ...
};
```

You may have learned about structs in C++, but the C struct has simpler functionality³⁶. To declare a variable of a struct type in C, we use the following form:

³⁶ structs in C++ are true objects, with the ability to include functions as well as data.

```
struct tag varname;
```

This is different from C++, where the struct modifier is not needed. You may optionally typedef a shortcut for the struct, and this can be the same name as the struct tag:

```
typedef struct tag {
    int a;
    char b;
    long c;
} tag;
```

```
...
tag var; // declare var to be of type struct tag
```

To access a struct variable's fields, you use dot notation (e.g., `var.a`), and to access a pointer to a struct variable's fields, you use arrow notation (e.g., `var->a`). Here is a complete example to demonstrate how to use structs:

```
// file: fraction.c
#include<stdio.h>
#include<stdlib.h>

struct fraction {
    int numerator;
    int denominator;
};

struct fraction multiply(struct fraction *a, struct fraction *b);
void printfrac(struct fraction *f);
int gcd(int a, int b);

int main(int argc, char **argv)
{
    struct fraction f1 = {1,2}; // 1/2
    struct fraction f2 = {2,3}; // 2/3

    printfrac(&f1);
    printfrac(&f2);

    struct fraction f3 = multiply(&f1,&f2);
    printfrac(&f3);

    return 0;
}

struct fraction multiply(struct fraction *a, struct fraction *b)
{
    struct fraction f;
    f.numerator = a->numerator * b->numerator;
    f.denominator = a->denominator * b->denominator;

    // reduce
    int fgcd = gcd(f.numerator,f.denominator);
    f.numerator /= fgcd;
    f.denominator /= fgcd;
    return f;
}

void printfrac(struct fraction *f)
{
```



```

    printf("%d/%d\n", f->numerator, f->denominator);
}

int gcd(int a, int b)
{
    if (a==0) return b;
    return gcd(b%a, a);
}

$ gcc -g -O0 -std=gnu99 fraction.c -o fraction
$ ./fraction
1/2
2/3
1/3

```

C Memory Management: malloc, free, calloc, and realloc

As we have seen earlier, one method for creating an array is by declaring local storage:

```
int values[] = {1, 3, 5, 2, 4, 6};
```

There are some downsides to this approach:

1. The memory allocated only remains in scope during the function or block where it is declared. This precludes creating an array in a function and returning a pointer to it.
2. The memory is declared on the *stack*, and there is limited stack memory, meaning that the arrays cannot be too large.

A more robust method for creating an array is by dynamically allocating memory, using the following functions:

malloc : request an array of bytes from the operating system.

free : return bytes previously requested back to the operating system.

calloc : request an array of memory and set the values to zero.

realloc : change the size of a previously requested array

Dynamic memory is allocated from the heap³⁷, and a program has access to that memory for the duration of the program, if desired. Heap allocated memory does not suffer from the same size restrictions as memory from the stack, so it also allows you to request large blocks of memory.

It is good practice to free memory that is no longer needed, to avoid *memory leaks*. The operating system will reclaim any unfreed memory when a program ends, but you should ensure that your program frees all of its memory before the program ends³⁸

The details for all of the dynamic memory allocation functions are located on malloc's man page³⁹, but here are the function signatures:

³⁷ more on that in *Pointers, the Stack, and the Heap*, and *Pointers to Functions*

³⁸ And you will lose points in CS 107 for not freeing memory. We will discuss the use of valgrind to find memory errors and leaks as the course progresses.

³⁹ man malloc

```

void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);

```

Notes:

1. malloc's only parameter is the number of bytes requested, and it returns a void * pointer to the memory allocated (and NULL if it cannot allocate the requested number of bytes).
2. calloc's two parameters, when multiplied together, give the total number of bytes requested.
3. free must be given a previously allocated pointer, or NULL. Once freed, a program is not allowed to access the memory region again, even though the pointer value has not been changed.
4. realloc is given a previously allocated pointer (or NULL), and a new size, and returns a pointer to the (potentially) new block (it returns the same pointer if it can do an in-place size change). If it cannot resize the block of memory, it will return NULL, and *the original pointer will be unchanged*, allowing the program to clean up if it can. realloc also frees the old pointer during a size change when a new block is needed.

The following program is an example that uses the dynamic memory functions, and demonstrates how to use the valgrind program to check for memory leaks and/or errors:

```

// file: dynamic_memory.c
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

int main(int argc, char **argv)
{
    int num_ints = atoi(argv[1]);

    int *values = malloc(num_ints * sizeof(int));
    assert(values);

    // fill with even numbers
    for (int i=0; i < num_ints; i++) {
        values[i] = i * 2;
    }

    for (int i=0; i < num_ints; i++) {
        printf("%d", values[i]);
        i == num_ints - 1 ? printf("\n") : printf(", ");
    }

    // double the size

```

```

int *new_values = realloc(values, num_ints * 2 * sizeof(int));
assert(new_values);

values = new_values;
num_ints *= 2;

// fill with more even numbers
for (int i=num_ints / 2; i < num_ints; i++) {
    values[i] = i * 2;
}

for (int i=0; i < num_ints; i++) {
    printf("%d", values[i]);
    i == num_ints - 1 ? printf("\n") : printf(", ");
}

// clean up
free(values);
return 0;
}

```

```

$ gcc -g -O0 -std=gnu99 dynamic_memory.c -o dynamic_memory
$ ./dynamic_memory
0, 2, 4, 6, 8, 10, 12, 14, 16, 18
0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38
$ valgrind ./dynamic_memory 10
==1797== Memcheck, a memory error detector
==1797== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==1797== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==1797== Command: ./dynamic_memory 10
==1797==
0, 2, 4, 6, 8, 10, 12, 14, 16, 18
0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38
==1797==
==1797== HEAP SUMMARY:
==1797==      in use at exit: 0 bytes in 0 blocks
==1797==    total heap usage: 2 allocs, 2 frees, 120 bytes allocated
==1797==
==1797== All heap blocks were freed -- no leaks are possible
==1797==
==1797== For counts of detected and suppressed errors, rerun with: -v
==1797== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Using the assert Function

In the program above, we used the `assert` function to ensure that the memory allocation functions were able to allocate the requested number of bytes. The `assert` function has a single expression parameter, and if that expression is false, the program aborts

(crashes) with a diagnostic message. Because the program crashes when an assert fails, it is best used as a debugging tool instead of an error checking tool. In CS 107, you should use asserts instead of proper error checking, mainly to save time for concentrating on other details of your program.

Using cdecl

Sometimes, it is difficult to tell what a particular type declaration actually means. For example, how does `const` apply in the following declarations?

```
const char **x;
char *const *y;
char **const z;
```

There is a handy tool on the Myth computers called `cdecl` that will decode type declarations for you:⁴⁰

⁴⁰ And there is a [cdecl website](#) as well.

```
$ cdecl explain const char **x
declare x as pointer to pointer to const char
```

```
$ cdecl explain char *const *y
declare y as pointer to const pointer to char
```

```
$ cdecl explain char **const z
declare z as const pointer to pointer to char
```

In the first case, the chars themselves may not be changed, in the second case, the pointer to char may not be changed, and in the final case, the pointer to pointer to the char may not be changed. The `cdecl` program can come in handy when you start to get confused by type declarations!

Boolean Values

Most languages have a built-in *boolean* type, which provide the common true and false boolean values. C actually does not have built-in boolean values, and in C, an integer 0 is considered false and *any integer not equal to zero* is considered true. In the version of C that we are using⁴¹, there is a boolean library you can use to provide the `bool` type, `stdbool.h`:

⁴¹ `gnu C99`

```
#include <stdbool.h>
bool a = true;
```

Final Thoughts

Believe it or not, C is a relatively lightweight language, and although there are nuances, the language can be learned thoroughly. Once you have a good grasp of the language, and are comfortable

using the man pages to look up library functions, you will be ready to excel in CS 107.

gdb

PROBABLY THE MOST IMPORTANT tool you will use during CS 107 is the command line debugger, *gdb*. *Gdb* allows you to step through your code line-by-line, investigate variables while your program is running, set breakpoints to stop your code at particular lines, look at the assembly code that *gcc* generates, and many more functions that will help you write and debug your code. It is imperative that you become proficient at *gdb* early during CS 107!

For the next part of this chapter, we will use the following buggy example program to demonstrate the use of *gdb*:

```
// file: digits_buggy.c

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

// returns the number of digits in a base-10
// number
int numdigits(int n)
{
    int count = 0;
    while ((n /= 10) != 0){
        count++;
    }
    return count;
}

// calculates 10^exp
int powerof10(int exp)
{
    int result = 1;
    for (int i=0; i < exp; i++) {
        result *= 10;
    }
    return result;
}

// returns the digit from d at position tensplace
// e.g., digit_place(5678,2) returns 6 because
// 6 is at the 10^2, or 100s place in 5678
```

```

int digit_place(int d, int tensplace)
{
    int p10 = powerof10(tensplace);

    return d / (p10 * 10) % p10;
}

int main(int argc, char **argv)
{
    char *digit_str[] =
        {"zero", "one", "two", "three",
         "four", "five", "six", "seven",
         "eight", "nine"};

    int number = atoi(argv[1]); // command line
    arg
    int ndigits = numdigits(number);
    for (int i=ndigits-1; i >= 0; i--) {
        int digit = digit_place(number, i);
        printf("%s ", digit_str[digit]);
    }
    printf("\n");

    return 0;
}

```

We can compile our program as follows:

```
$ gcc -g -O0 -std=gnu99 -Wall digits_buggy.c -o digits_buggy
```

The program is supposed to take a number as an argument to the program, and print out the english version of all the digits. For example, this is what we would like:

```
$ ./digits_buggy 8675309
eight six seven five three zero nine
```

But, what we actually get is:

```
$ ./digits_buggy 8675309
Segmentation fault (core dumped)
```

Obviously, the most expedient method to debug the code would probably be to look at it, and find the bugs! But, this isn't always easy, and we have tools such as gdb to help with the process.

Below is a sample gdb run that demonstrates many of the commands you should become familiar with. When a command is used, it is defined in the right margin.

```
$ gdb digits_buggy
```

```
The target architecture is assumed to be i386:x86-64
```

```
Reading symbols from digits_buggy...done.
```

```
(gdb) run 8675309
```

```
Starting program:
```

```
/afs/.ir.stanford.edu/users/c/g/cgregg/cs107
```

```
/reader/digits_buggy 8675309
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x00007ffff7a5c943 in _IO_vfprintf_internal
```

```
(s=<optimized out>, format=<optimized out>,
```

```
    ap=ap@entry=0x7ffff7ffe888) at vfprintf.c:1661
```

```
1661      vfprintf.c: No such file or directory.
```

```
(gdb) where
```

```
#0 0x00007ffff7a5c943 in _IO_vfprintf_internal
```

```
(s=<optimized out>, format=<optimized out>,
```

```
    ap=ap@entry=0x7ffff7ffe888) at vfprintf.c:1661
```

```
#1 0x00007ffff7a653d9 in __printf (format=<optimized out>)
```

```
at printf.c:33
```

```
#2 0x000000000400720 in main (argc=2, argv=0x7ffff7feab8)
```

```
at digits_buggy.c:45
```

```
(gdb) up
```

```
#1 0x00007ffff7a653d9 in __printf (format=<optimized out>)
```

```
at printf.c:33
```

```
33      printf.c: No such file or directory.
```

```
(gdb) up
```

```
#2 0x000000000400720 in main (argc=2, argv=0x7ffff7feab8)
```

```
at digits_buggy.c:45
```

```
45      printf("%s ",digit_str[digit]);
```

```
(gdb) p digit
```

```
$1 = 86
```

```
(gdb) p digit_str[digit]
```

```
$2 = 0xb <error: Cannot access memory at address 0xb>
```

```
(gdb) list
```

```
40
```

```
41      int number = atoi(argv[1]); // from command line
```

```
42      int ndigits = numdigits(number);
```

```
43      for (int i=ndigits-1; i >= 0; i--) {
```

```
44          int digit = digit_place(number,i);
```

```
45          printf("%s ",digit_str[digit]);
```

The **run** command starts the program. It can take command line arguments just as if you were running the program on the command line.

The **where** command provides a stack trace. Assuming that the bug is in your code (and not a library function – a good assumption!), you can look for the first instance down the stack trace where your function appears. In this case, the error seems to be coming from line 45 of the program, even though it technically crashed during the **printf** call.

The **up** command follows the stack trace up one level (to the previous function) and allows you to investigate the variables and program state at that point. In this case, we need to type **up** twice to get to the main function.

The **p** (abbreviated from **print**) command prints variables, constants, etc. To print a hexadecimal digit, type **p/x variable**.

The **list** command lists your C code around the line that is in process. If you continue to type **l**, further lines of code will print.


```

46         }
47         printf("\n");
48
49         return 0;
(gdb) p number
$3 = 8675309
(gdb) p i
$4 = 4
(gdb) call digit_place(number,i)
$10 = 86
(gdb) break 44 if i==4
Breakpoint 1 at 0x4006f2: file digits_buggy.c, line 44.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107
/reader/gdb/digits_buggy 8675309

Breakpoint 1, main (argc=2, argv=0x7fffffffefaa8)
at digits_buggy.c:44
44         int digit = digit_place(number,i);
(gdb) p i
$3 = 4
(gdb) step
digit_place (d=8675309, tensplace=4) at digits_buggy.c:30
30         int p10 = powerof10(tensplace);
(gdb) next
32         return d / (p10 * 10) % p10;
(gdb) p d
$4 = 8675309
(gdb) p p10
$5 = 10000
(gdb) p d / (p10 * 10) % p10
$6 = 86
(gdb) p d % (p10 * 10) / p10
$7 = 7
(gdb) quit
A debugging session is active.

        Inferior 1 [process 14432] will be killed.

Quit anyway? (y or n) y

```

We are investigating the **number** and **i** variables.

The **call** command will run a function. Here, we are confirming that the **digit_place** function is giving us an incorrect result (it should be **5** instead of **86**).

The **break** command tells gdb to stop running at a particular line; in this case, line 44. We have also told gdb to stop when **i** is equal to 4, so we can continue at that point (the condition is optional). When we **run** the program again, it uses the last command line arguments, and it starts the program over. Then, it breaks on the line or function where we have set a breakpoint.

The program stopped at line 44 when **i** was equal to 4, before that line runs, and we use the **step** command to run the next line in the program, which steps into the **digit_place** function. We use the **next** command to run the next line but to run the entire function (in this case, the **powerof10** function).

Now we can start to analyze the bug. We print out the **d** and **p10** variables, and see that they are what we expect.

When we print out the details of the line that is about to be returned, we realize that this is incorrect. We then (hopefully!) recognize that we have the modulus (%) and division swapped, so we print out the expression with the operators in the correct position, and see that we get the correct answer, which is the **7** extracted from **8675309**. We have found a bug, and we can **quit** gdb to go fix it.

After we fix the bug in the **digit_place** function, we re-run the program:

```
$ ./digits_buggy 8675309
six seven five three zero nine
```

This is better! But, we are missing the **eight** that we are expecting. Back to gdb!

```
$ gdb digits_buggy
The target architecture is assumed to be i386:x86-64
Reading symbols from digits_buggy...done.
(gdb) run 8675309
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107
/reader/gdb/digits_buggy 8675309
six seven five three zero nine
[Inferior 1 (process 22983) exited normally]
(gdb) l
```

If we run the program from gdb, we find see that it is missing the first digit (**eight**).

```
29      {
30          int p10 = powerof10(tensplace);
31
32          //return d / (p10 * 10) % p10; // fix: swap mod and div
33          return d % (p10 * 10) / p10;
34      }
35
36      int main(int argc, char **argv)
37      {
38          char *digit_str[] = {"zero","one","two","three",
(gdb)
39              "four","five","six","seven",
40              "eight","nine"};
41
42          int number = atoi(argv[1]); // from command line
43          int ndigits = numdigits(number);
44          for (int i=ndigits-1; i >= 0; i--) {
45              int digit = digit_place(number,i);
46              printf("%s ",digit_str[digit]);
47          }
48          printf("\n");
```

We list the program (l) to determine where to break.

```
(gdb) break 45
Breakpoint 1 at 0x4006f2: file digits_buggy.c, line 45.
(gdb) run
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107
/reader/gdb/digits_buggy 8675309
```

We decide to break on line 45 to investigate some more.

```

Breakpoint 1, main (argc=2, argv=0x7fffffffefa98)
at digits_buggy.c:45
45             int digit = digit_place(number,i);
(gdb) p i
$1 = 5
(gdb) p ndigits
$3 = 6
(gdb) break numdigits
Breakpoint 2 at 0x4005c4: file digits_buggy.c, line 9.
(gdb) info break
Num      Type           Disp Enb Address            What
1        breakpoint     keep y   0x00000000004006f2 in main
at digits_buggy.c:45
           breakpoint already hit 1 time
2        breakpoint     keep y   0x00000000004005c4 in numdigits
at digits_buggy.c:9
(gdb) delete 1
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

```

Hmm...**ndigits** should be 7, not 6.

We will put a breakpoint on the **numdigits** function, to investigate that function. The **info break** command tells us what breakpoints we already have set.

We **delete** the first breakpoint we set because we don't need it any more, and we re-run the program.

```

Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107
/reader/gdb/digits_buggy 8675309

```

```

Breakpoint 2, numdigits (n=8675309) at digits_buggy.c:9
9             int count = 0;
(gdb) display count
3: count = 32767
(gdb) n
10            while ((n /= 10) != 0){
3: count = 0
(gdb) l
5
6            // returns the number of digits in a base-10 number
7            int numdigits(int n)
8            {
9                int count = 0;
10               while ((n /= 10) != 0){
11                   count++;
12               }
13               return count;
14           }
(gdb) break 13

```

The **display count** command will show the **count** variable value after each gdb command. In this case, **count** is a non-initialized value, because we have stopped on line 9, before it has been initialized to 0. The **n** command (**next**) runs line 9, and stops on line 10, where we **l** (**list**) the code. Note that at this point, **count** has been initialized to 0.

We **break** on line 13 to investigate the count at that line.

Breakpoint 7 at 0x4005f2: file digits_onefix.c, line 13.

(gdb) **continue**

Continuing.

Breakpoint 7, numdigits (n=0) at digits_onefix.c:13

13 return count;

3: count = 6

(gdb)

The **continue** command continues program execution at the current line (and only stops at breakpoints).

The program breaks at line 13 (as we directed it to do), and we find that we have an off-by-one problem with count, after the while loop. We realize that there must be an error in the while loop, and we find that we should have initialized **count** to **1**.

Other Important gdb Commands

The commands in the above gdb traces are probably the most useful commands, and you will use almost all of them frequently. The list below has other commands that you will want to know when debugging C code, as well.

break 12 if x == 42 This is a conditional breakpoint to break on line 12 when the variable **x** is equal to 42. This command is useful if you need to stop the program in the middle of a loop, when a variable reaches a certain value, so you can investigate the program in that state.

help This is an obvious command, but it is useful to give you information about other commands in gdb. For example, the next command we will look at is the **x** command, and we can find help on it:

(gdb) **help x**

Examine memory: x/FMT ADDRESS.

ADDRESS is an expression for the memory address to examine.

FMT is a repeat count followed by a format letter and a size letter.

Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char), s(string) and z(hex, zero padded on the left).

Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes).

The specified number of objects of the specified size are printed according to the format.

Defaults for format and size letters are those previously used.

Default count is 1. Default address is following last thing printed with this command or "print".

x/16bx 0x4007eb The **x** command prints out a range of memory. The form here will print out a range of bytes at a particular memory location in hexadecimal format, e.g., the following prints out sixteen hexadecimal bytes at memory address **0x4007eb**:

(gdb) **x/16bx 0x4007eb**

0x4007eb: 0x65 0x69 0x67 0x68 0x74 0x00 0x6e 0x69

```
0x4007f3:  0x6e  0x65  0x00  0x25  0x73  0x20  0x00  0x00
```

The **x** command is also useful if you want to print out the memory contents of a variable. For example, let's say we had the following function, which simply increments the value of the **int** that **xptr** points to:

```
void add1(int *xptr)
{
    (*xptr)++;
}
```

In **gdb**, you could investigate **xptr** in a number of different ways:

Breakpoint 1, add1 (xptr=0x7fffffff9b0) at arrays.c:6

```
(gdb) p *xptr
```

```
$1 = 1234
```

```
(gdb) x/1d xptr
```

```
0x7fffffff9b0:      1234
```

```
(gdb) x/4b xptr
```

```
0x7fffffff9b0:      -46      4      0      0
```

```
(gdb) p/x 1234
```

```
$2 = 0x4d2
```

```
(gdb) x/4bx xptr
```

```
0x7fffffff9b0:      0xd2      0x04      0x00      0x00
```

```
(gdb)
```

Dereference the pointer and print it.

Print one 4-byte integer at the address of the pointer.

Print four bytes at the address. This may look strange! In the *Bits and Bytes* chapter, we will see why. Let's print the hexadecimal value for **1234**. If we print the four bytes in hexadecimal, and then print the memory at **xptr** as hex bytes, we can see some similarity, although the bytes seem to be in reverse order. Read ahead in the *Bits and Bytes* chapter for more information!

Bits and Bytes

AT THE LOWEST LEVEL, computers are good at determining the difference between “on” and “off”. They are built using billions of transistors, each of which can either conduct electrical current, or not, based on other transistors connected in a circuit. Together, these fundamental building blocks contribute to a computer’s ability to perform calculations, and logical and numerical calculation is the basis for computation.

We call a single on or off representation a *bit*, and numerically a bit is either a 0 (off) or a 1 (on), representing one state (either off or on). However, having one bit (and a single state) is not particularly helpful! In order to have more states, we simply add more bits. As discussed in *Numerical Formats Used in CS 107*, the binary number format is represented by 0s and 1s, and therefore it is the perfect base to represent a series of bits.

If we have two bits, we have the following four representations:

```
00
01
10
11
```

Three bits gives us eight representations: 000, 001, 010, 011, 100, 101, 110, 111. It follows that adding an extra bit produces two times more states, and therefore for n bits, we have 2^n states available to us.

As you can see, we are actually just counting, assigning each state a different number. In this way, we can encode anything we want with bits. We can encode text characters in the ASCII format, for instance. The character value for “A” is 65d, or 01000001b⁴² We encode images, music, and video numerically, and therefore, using bits.

In this chapter, we will discuss how computers represent and perform calculations on integers⁴³, and the important role bits play in those calculations. We will also delve into the way C handles integers, including how the language performs calculations, converts between different integer data types, and prints out integers.

⁴² (See Tables 1 and 2 for the decimal and hexadecimal ASCII character values.

⁴³ We will tackle non-integers, or *floating point numbers* in a later chapter

Unsigned and Signed Integers

There are two forms of integers that we will use in CS 107:

Unsigned Integers: these are the positive integers, and zero.

Signed Integers: these are the negative and positive integers, and zero.

Because computers use a fixed number of bits to represent integers, we have a fixed number of values for a particular number representation. For example, the C `int` data type uses 32 bits to represent a number⁴⁴, and therefore we can represent 2^{32} (or 4,294,967,296) different numbers. If we want to represent both negatives and positives, half the numbers in our range will be negative, and the other half will be positive (plus another value for zero). This describes the signed integers, and indeed `int` is a signed representation, able to represent numbers in the range $-2,147,483,648$ through $2,147,483,647$.⁴⁵

But, if we know that we only want to represent positive numbers (e.g., length, or weight), we can double our range by using an unsigned number format. The `unsigned int` data type also uses 32 bits to represent integers, but because they are only positives (and zero), the range is from 0 to 4,294,967,295.

The fixed range for an integer data type leads to interesting issues. Take, for example, the following program:

```
// file: multtest.c

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

int main() {
    int a = 200;
    int b = 300;
    int c = 400;
    int d = 500;

    int answer = a * b * c * d;
    printf("%d\n", answer);
    return 0;
}
```

When we compile and run the program, we get the following output:

```
$ ./multtest
-884901888
```

Why did we get a negative result? The answer is that we just did not have enough bits to represent the number we were trying to calculate:

⁴⁴ on the Myth machines. The `int` data type can have a different number of bits depending on the machine being used

⁴⁵ We will soon see why this is the exact range!

$$200 \times 300 \times 400 \times 500 = 12,000,000,000$$

The maximum `int` value we can represent is 2,147,483,647, so the number we were trying to calculate *overflowed* our representation, and cannot be represented by an `int`. We will cover overflow directly later in the chapter.

Information Storage

Most of CS 107 concentrates on the C programming language, and in C, all integers can be thought of as blocks of 8 bits, which we call a *byte*. We will discuss how we manipulate integers on a bit-by-bit level, but the C language does not allow us to consider an individual bit on its own.⁴⁶

A computer's memory system is simply a large array of bytes. Each *address* in memory represents a byte. Figure 6 demonstrates this concept for the range of addresses from 200d to 236d decimal (0xcc to 0xec hex).

values (ints):	7	2	8	3	14	99	-6	3	45	11
address (decimal):	200d	204d	208d	212d	216d	220d	224d	228d	232d	236d
address (hex):	0xc8	0xcc	0xd0	0xd4	0xd8	0xdc	0xe0	0xe4	0xe8	0xec

⁴⁶ The reason is more because the underlying hardware only represents numbers at the byte level, so C does not need to represent numbers more granularly than the byte level.

Figure 6: Memory address for an array of 4-byte ints

You cannot address a particular bit, and you must address at the byte level.

How much memory can our computers address? The limit happens to be the underlying *word size* of the computer, and the Myth computers have a word size of 64 bits.⁴⁷ In other words, an address is allocated a 64-bit value, and because each byte is addressable, the Myth machines could reference up to 2^{64} bytes of memory, which is *a lot* of memory. The Linux operating system imposes a memory limit of 2^{48} bytes, which is still a tremendous amount of memory.⁴⁸

Because a byte is made up of 8 bits, the range of a byte is 00000000 to 11111111. As discussed in *Numerical Formats Used in CS 107*, it is more convenient to represent bytes in hexadecimal, and the hexadecimal range for a byte is 0x00 to 0xff.

As discussed above, the Myth computers represent ints with 32 bits. Table 4 shows the other representations used by gcc on the Myth computers.

⁴⁷ They are 64-bit machines.

⁴⁸ A high-end computer these days might have 64 Gigabytes (GB) of memory, or 68,719,476,736 bytes of memory ("Giga" in this case referring to 2^{30} bytes). Linux is capable of addressing over four thousand times more memory than 64GB. It will be a long time before Linux needs to modify the 2^{48} byte limit.

C Declaration		Bytes
Signed	Unsigned	
char	unsigned char	1
short	unsigned short	2
int	unsigned int	4
long	unsigned long	8
int32_t	uint32_t	4
int64_t	uint64_t	8
char *		8
float		4
double		8

Table 4: C Data Sizes on the Myth Computers

Byte Storage in Memory (*Little Endian and Big Endian*)

We saw in Figure 6 that a four-byte `int` has a single address that refers to the entire integer. Gcc keeps track of these addresses, and we often don't need to know more than this fact to refer to ints in our C variables. However, there are times when you will need to analyze or manipulate data types at the byte level, so it is important to understand how the data is layed out in memory.

A four byte `int` is stored in consecutive bytes in memory, but we actually have a choice on ordering those bytes. For example, we can represent a 4-byte `int` as an 8-digit hexadecimal number (two hex digits represent one byte):

```
0x01234567
```

We can separate out the bytes:

```
0x  01  23  45  67
```

There are two natural orderings for storing those bytes into memory:

Big Endian: the order in memory is from the “big end” of the number (i.e., the *most significant byte*) at the lowest memory location, to the “little end” of the number (i.e., the *least significant byte*) at the highest memory location. Writing out our number from above, big endian format would look like Figure 7:

byte:	01	23	45	67
address:	0x100	0x101	0x102	0x103

Figure 7: The number `0x01234567` stored in Big Endian Format, at address `0x100`

Little Endian: the order in memory is from the little end of the number at the lowest memory location, to the big end of the number at the highest memory location. Writing out our number from above, little endian format would look like Figure 8, and this is the form in which integers on the Myth computers are stored in:

byte:	67	45	23	01
address:	0x100	0x101	0x102	0x103

Figure 8: The number 0x01234567 stored in Little Endian Format, at address 0x100

To be clear: the *endianness* of a number is simply the order of the individual bytes in memory. Big endian numbers have the most significant byte first in memory, and little endian numbers have the least significant byte in memory first. What endianness is not, is a complete reversal of the binary bits of a number, but rather the ordering of the *bytes* in memory. The Intel/AMD x86 computers we use in class are little-endian, although we will rarely need to worry about it.⁴⁹

Boolean Algebra

Because computers store values in binary, we often perform operations on numbers that involve *boolean algebra*, or arithmetic that performs operations directly on the individual bits in a number.

Boolean algebra is defined over a 2-element set, 0 and 1, where 0 represents false, and 1 represents true. The following C operations are defined, and can be used to apply the boolean functions on two numbers (one in the case of the NOT (~) operator):⁵⁰

Bitwise AND, &, example: $z = x \& y$ The & operator produces a 1 bit when both operand bits are 1, and a 0 otherwise. Figure 9 shows the AND truth table.

Bitwise OR, |, example: $z = x | y$ The | operator produces a 1 bit when either of the operand bits are 1, and a 0 if both operand bits are 0. Figure 10 shows the truth table for OR.

Bitwise XOR (Exclusive OR), ^, example: $z = x ^ y$ The ^ operator produces a 1 bit when only one of the operand bits is 1, and a 0 otherwise. Figure 11 shows the truth table for XOR.

Bitwise NOT, ~, example: $z = \sim x$ The ~ operator inverts a bit, so a 0 becomes a 1, and a 1 becomes a 0. Figure 12 shows the truth table for NOT.

The bitwise operators apply to all the bits on a number at once. For example:

⁴⁹ At the end of the quarter, you may have to think about the endianness of numbers when you are closely analyzing memory for your final project.

⁵⁰ Be careful when writing your C programs to differentiate the bitwise boolean operators from the *logical* boolean operators. Gcc does not always produce a warning when logical AND, &&, is substituted for bitwise AND, &, and this can lead to difficult-to-track-down bugs.

Figure 9: The AND truth table.

INPUT		OUTPUT
A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Figure 10: The OR truth table.

INPUT		OUTPUT
A	B	A B
0	0	0
0	1	1
1	0	1
1	1	1

Figure 11: The XOR truth table.

INPUT		OUTPUT
A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 12: The NOT truth table.

INPUT	OUTPUT
A	~A
0	1
1	0

$$\begin{array}{r}
 10011001 \\
 \&10100111 \\
 \hline
 10000001
 \end{array}$$

The AND operator is applied to the corresponding bits in both numbers to produce the resulting number.

Bit Masking

An important result of being able to perform bitwise boolean operations on numbers is the ability to *mask* an integer variable, to access only particular bits of the number. For example:

```
int x = 0x89ABCDEF;
int y = x & 0xFF; // y now holds the value 0xEF,
                  // which is the low-order byte
                  // of x
```

Let's break down what is happening in the code. Here are the values of `x` and `0xFF` in binary:

```
x      == 0b10001001101010111100110111101111
0xFF   == 0b00000000000000000000000011111111
```

When the AND operator is applied, the only bits that remain are the lowest eight bits, and we have “masked” `x` to get the lower eight bits alone.

Example: write an expression that sets the most significant byte of a 4-byte integer, `n` to all ones, and all other bytes of the number left unchanged. E.g.

```
0x87654321 → 0xFF654321
```

One possible solution would be:

```
x | 0xFF000000
```

Example: write an expression that complements all but the most significant byte of a 4-byte integer, `n`, with the least significant byte unchanged. E.g.

```
0x87654321 → 0x879abcde
```

One possible solution would be:

```
x ^ ~0xFF000000
```

Shift Operations

C provides operations to *shift* bit patterns to the left or right. The `<<` operator moves the bits to the left, replacing the lower order bits with zeros and dropping any values that would be bigger than the type can hold:

`x << k` will shift `x` to the left by `k` number of bits.

Examples for an 8-bit binary number:

```
00100111 << 2 returns 10011100
01101011 << 4 returns 10110000
10011101 << 4 returns 11010000
```

There are actually two flavors of *right shift*, which work differently depending on the value and type of the number you are shifting.

A *logical* right shift moves the values to the right, replacing the upper bits with zeros.

An *arithmetic* right shift moves the values to the right, replacing the upper bits with a copy of the most significant bit.⁵¹

Example logical shift for an 8-bit binary number:

```
00100111 >> 2 returns 00001010
10100111 >> 2 returns 00101001
01101011 >> 4 returns 00000110
10011101 >> 4 returns 00001001
```

Example arithmetic shift for an 8-bit binary number:

```
00100111 >> 2 returns 00001010
10100111 >> 2 returns 11101001
01101011 >> 4 returns 00000110
10011101 >> 4 returns 11111001
```

There are two important things you need to consider when using the shift operators:

1. The C standard does not precisely define whether a right shift for signed integers is logical or arithmetic. Almost all compilers and computer hardware uses arithmetic shifts for signed integers, and you can most likely assume this. All unsigned integers will always use a logical right shift (more on this later!)
2. Operator precedence can be tricky! Example:

$1 << 2 + 3 << 4$ means this: $1 << (2 + 3) << 4$, because addition and subtraction have a higher precedence than shifts!

Always parenthesize to be sure:

$(1 << 2) + (3 << 4)$

Integer Representations in C

The C language has different data types to represent both signed and unsigned integers.⁵² C also has rules that properly convert between one integer data type and another when moving data between variable types and when performing arithmetic on different integer data types. Table 5 shows the integer data type ranges for the Myth (64-bit) machines.

Unsigned integers in C are simply represented by their true binary representation. For example, an eight-bit unsigned integer (in C, this would be declared as an unsigned char) fourteen would be

⁵¹ This may seem weird! But, we will see why this is useful soon!

⁵² In fact, C and C++ are somewhat odd in this regard – most languages stick to signed integers only.

C data type	MIN	MAX	Bytes
char	-128	127	1
unsigned char	0	255	1
short	-32,768	32,767	2
unsigned short	0	65,535	2
int, int32_t	-2,147,483,648	2,147,483,647	4
unsigned int, uint32_t	0	4,294,967,295	4
long, int64_t	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	8
unsigned long, uint64_t	0	18,446,744,073,709,551,615	8

Table 5: Ranges for integer data types on the Myth machines

00001110, which is the true binary representation for 14d. The range for an unsigned integer is $0 \rightarrow 2^w - 1$, where w is the number of bits in our integer. For example, a 32-bit int can represent numbers from 0 to $2^{32} - 1$, or 0 to 4,294,967,295.

Signed integers are treated differently than unsigned integers. In particular, we don't specifically have a "negate" sign, as we do in regular mathematics, and we assign a bit to represent whether the number is positive (0) or negative (1). While this *sign bit* is sufficient to determine whether a number is signed or not, this leads to some irregularities. For example, let's say we have an 4-bit number, and we reserve the most significant bit as the sign bit. We can represent sixteen (or 2^4) total numbers, and we might simply assume that the lower three bits are the magnitude of the number:

0 001 = 1	1 001 = -1
0 010 = 2	1 010 = -2
0 011 = 3	1 011 = -3
0 100 = 4	1 100 = -4
0 101 = 5	1 101 = -5
0 110 = 6	1 110 = -6
0 111 = 7	1 111 = -7

We have successfully represented fourteen of our sixteen possible representations, but what about the remaining two numbers?

0 000 = ?	1 000 = ?
-----------	-----------

It seems that we are left with two representations for zero: positive zero and negative zero. For integers, this does not make sense, and we would rather have a single zero representation.⁵³ Additionally, and perhaps more importantly, performing calculations on positive and negative numbers in the representation above is not particularly easy, because there is an inherent need to subtract with numbers that are negative.

Instead of the method outlined above, signed integers in a computer are represented in a format called *Two's complement*. The two's complement of an n -bit number is defined to be the complement of the number with respect to 2^n . For example, the four bit number 0110 has a two's complement of 1010 because

⁵³ In the Floating Point chapter, we will see that we do have two zero representations for non-integers.

$0110 + 1010 = 10000$.⁵⁴ As can be seen, to calculate the two's complement of a number, you must know how many bits the number holds.

The standard method for calculating the two's complement of a number is to invert all the bits⁵⁵, and then add 1. For the example above:

$$\begin{aligned}\sim 0110 &= 1001 \\ 1001 + 1 &= \mathbf{1010}\end{aligned}$$

The process is reversible. To find the two's complement of 1010:

$$\begin{aligned}\sim 1010 &= 0101 \\ 0101 + 1 &= \mathbf{0110}\end{aligned}$$

If a number *overflows* to the left past the bit length, we just ignore the bits that overflowed. For example, to find the two's complement of the four-bit number, 0000:

$$\begin{aligned}\sim 0000 &= 1111 \\ 1111 + 1 &= \mathbf{10000}\end{aligned}$$

Interestingly, there is a second number whose two's complement is itself:

$$\begin{aligned}\sim 1000 &= 0111 \\ 0111 + 1 &= \mathbf{1000}\end{aligned}$$

We will soon see why this is the case!

The Two's Complement Circle

Figure 13 shows the “two's complement circle” for a 4-bit signed integer. The figure shows the correspondence between the decimal and binary numbers in the 4-bit range. There are a few key ideas to take away from the diagram:

1. Positive numbers are simply the binary equivalent of their value.
2. Negative one is comprised of all 1s, which is equal to the binary value $2^n - 1$, where n is the number of bits in the integer (in this case, $n = 4$). Negative two is the binary equivalent of $2^n - 2$, and the pattern continues as the negative numbers get smaller.
3. The largest positive number is $2^{n-1} - 1$. The number has a 0 as its most significant bit (denoting that it is a positive number), and the rest of the bits are 1s.
4. The smallest negative number is -2^{n-1} . This accounts for the fact that because of 0, there is one more negative than positive.
5. *Overflow* happens at the bottom of the circle. If we try to add two positive numbers together that are bigger than 2^{n-1} , the number will become negative, and be incorrect mathematically.

⁵⁴ The [Wikipedia article on two's complement](#) is excellent.

⁵⁵ called the *one's complement*, or simply the bitwise NOT operation.

Figure 13: The two's complement circle for a signed 4-bit number

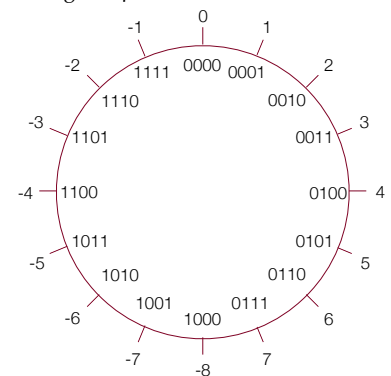
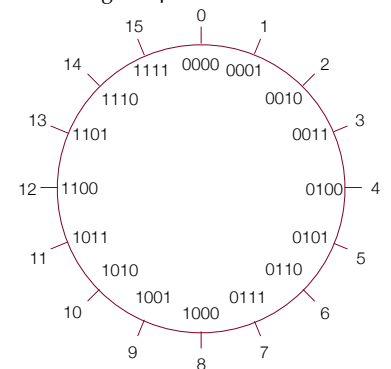


Figure 14: The two's complement circle for an unsigned 4-bit number



Likewise, if we add two negative numbers together and the result will be less than -2^n , the result will become positive and overflow.

6. It is not possible to add a positive and a negative number together and overflow in signed arithmetic.

Figure 14 shows the two's complement circle for a 4-bit unsigned integer. Note the differences between this signed and unsigned circle:

1. All numbers are the binary equivalent of their value.
2. The largest number is $2^n - 1$, where n is the number of bits in the number.
3. Overflow happens at the *top* of the circle. If we try to add two positive numbers together that are bigger than $2^n - 1$, the number will wrap around in a clockwise direction, and be incorrect mathematically. If we subtract two numbers and the result will be less than zero, the result will wrap around in a counterclockwise direction, and the result will be incorrect mathematically.⁵⁶

⁵⁶ But correct in a modular sense.

It is important to understand the number circle for both signed and unsigned integers, particularly when considering the overflow points, and the maximum and minimum values. Although Figures 13 and 14 are drawn for 4-bit numbers, they can be extrapolated to any bit length, and the properties remain the same.

Two's Complement Number Properties

So far, you may be asking yourself what the big deal is with two's complement numbers. It turns out that the use of two's complement to store signed integers leads to a number of interesting and helpful properties:

1. There is only one representation for 0. As we saw earlier, a simple sign-bit representation leads to having two zeros, and two's complement avoids this.
2. The most-significant bit is interpreted as a sign bit. It is nice that this property remains when using two's complement.
3. Addition with all numbers is simply addition. This is an amazing development: to add two two's complement numbers together, you simply add them, and throw away any overflow bits. The answer will be correct, as long as the correct mathematical result would not overflow. For example, performing the addition of $-3 + 7$ for our 4-bit numbers, we should get an answer of 4:

$$\begin{array}{r} 1101 \quad -3 \\ +0111 \quad 7 \\ \hline \cancel{1}0100 \quad 4 \end{array}$$

and indeed, we do!

4. To subtract two numbers, we first perform a two's complement conversion on the number we are subtracting away, and then we add them. For example, $3 - 5$: Convert 5 from 0101 to its two's complement:

$$\begin{aligned}\sim 0101 &= 1010 \\ 1010 + 1 &= 1011 \text{ } (-5)\end{aligned}$$

Add the original number and the converted number:

$$\begin{array}{r} 0011 \quad 3 \\ +1011 \quad -5 \\ \hline 1110 \quad -2 \end{array}$$

5. Multiplication is also a matter of ignoring the fact that one or both numbers is negative, and simply throwing away overflow bits.⁵⁷ For example: -2×3 :

$$\begin{array}{r} 1110 \quad -2 \\ \times 0011 \quad 3 \\ \hline 1110 \\ 1110 \\ 0000 \\ 0000 \\ \hline 0101010 \quad -6 \end{array}$$

As another example, -1×-2 :

$$\begin{array}{r} 1111 \quad -1 \\ \times 1110 \quad -2 \\ \hline 0000 \\ 1111 \\ 1111 \\ 1111 \\ \hline 10010010 \quad 2 \end{array}$$

One thing to note about two's complement notation is that the bits we are looking at are still powers of two, with the leading bit being a (possibly) negated power of two. For example, let's look at the bit representations for -5 for a 4-bit number:

$$\begin{aligned}-5 &= 1 \quad 0 \quad 0 \quad 1 \\ &= 1 \times -2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0\end{aligned}$$

Casting Between Signed and Unsigned Integers

In C, we can cast between different number types either explicitly, or implicitly. Figure 15 demonstrates an explicit cast, using parenthesized type notation for the cast. Figure 16 demonstrates an implicit cast, where the compiler performs the cast for us.

When casting, **the underlying bits do not change**. In other words, casting between an unsigned integer and a signed integer

From a hardware perspective, the benefit of two's complement arithmetic is that there is no need for separate subtraction hardware, and performing a two's complement conversion is a simple hardware circuit.

⁵⁷ One caveat: multiplication is much more likely to produce actual overflow mathematically. But as long as the product fits into the number of bits in the format, the multiplication is straightforward.

Figure 15: An explicit cast between signed and unsigned ints.

```
int tx, ty;
unsigned ux, uy;
...
tx = (int) ux;
uy = (unsigned) ty;
```

Figure 16: An implicit cast between signed and unsigned ints.

```
int tx, ty;
unsigned ux, uy;
...
tx = ux; // cast to signed
uy = ty; // cast to unsigned
```


of the same bit-width does not undergo any conversion. Casting does not perform an absolute value function on a negative signed number when converting to a signed number, either. For example, figure 17 shows the output when implicitly casting an 8-bit unsigned char to a signed char in C. `-5` as a signed char has the same bit values as `251` as an unsigned char, which is `11111011`.

```
// file: cast.c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    char tx;
    unsigned char ux;

    tx = -5;
    ux = tx;

    printf("tx: %d\n", tx);
    printf("ux: %u\n", ux);
    return 0;
}

$ gcc -g -O0 -std=gnu99 cast.c -o cast
$ ./cast
tx: -5
ux: 251
```

Figure 17: A cast in C does not change the underlying bit pattern for integers of the same bit width.

C's `printf` command has three different ways to print an integer⁵⁸:

`%d` : signed int

`%u` : unsigned int

`%x` : hexadecimal formatted int

Unless there is an extra width-specifier in the format string, `printf` will treat the number according to the format string. Figure 18 demonstrates this with 32-bit ints.

Comparison Between Signed and Unsigned Integers

Often, we want to compare integers to each other. When a C expression has combinations of signed and unsigned integers, you have to be careful – if an operation is performed that has both a signed and an unsigned value, **C implicitly casts the signed argument to unsigned** to perform the operation, assuming both integers are non-negative. Table 6 shows the results of such operations.

⁵⁸ And there are multiple format-string modifications that can be made to augment these three.

```
// file: cast_printf.c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int d = -1;
    unsigned int u = 1<<31;

    printf("d = %d = %u = %x\n", d, d, d);
    printf("u = %d = %u = %x\n", u, u, u);

    return 0;
}

$ gcc -g -O0 -std=gnu99 cast_printf.c -o cast_printf
$ ./cast_printf
d = -1 = 4294967295 = ffffffff
u = -2147483648 = 2147483648 = 80000000
```

Figure 18: printf’s format string performs a cast on its values.

Expression	Type	Evaluation
0 == 0U	Unsigned	1
TRUE	Signed	1
-1 < 0U	Unsigned	0
2147483647 > -2147483647 - 1	Signed	1
2147483647U > -2147483647 - 1	Unsigned	0
2147483647 > (int)2147483648U	Signed	1
TRUE	Signed	1
(unsigned)-1 > -2	Unsigned	1

Table 6: C expressions that have both unsigned and signed integers perform the calculation assuming all values are unsigned.

The sizeof Operator

As we have seen, integer data types are limited by the number of bits that they hold. In C, we use the `sizeof` operator to determine the size of a variable. The `sizeof` operator is a *compile time* expression, which means that although it looks like a function, the result is *always* determined before the program runs, and the compiler replaces the result as a constant into your code.⁵⁹ Figure 19 shows an example program that prints out the number of bytes of each of the variables in the program, and the output is from the Myth machines.

Pay close attention to the output of the `sizeof` operator for arrays and pointers. For statically-defined arrays, `sizeof` prints out the number of bytes used by the entire array. In figure 19, the 5-element `int` array is 20 bytes long, because each `int` is 4-bytes. Although arrays can act as pointers⁶⁰, the compiler can report the actual length. For a pointer, the compiler reports the number of bytes that the pointer itself occupies, which is 8-bytes on the Myth machines.

⁵⁹ This happens behind the scenes, but you should understand how it works so you don't get caught off-guard when it happens.

⁶⁰ But they are distinct! We will see this in the chapter on low level C.

Expanding the Bit Representation of an Integer

Sometimes we want to convert between two integers having different sizes. E.g., we want to convert a `short` to an `int`, or an `int` to a `long`. We might not be able to convert from a bigger data type to a smaller data type without losing some information by virtue of the fact that there are fewer bits in a smaller data type, but we do want to always be able to convert from a smaller data type to a bigger data type without losing any information.

Conversion is straightforward for unsigned values: simply add leading zeros to the representation. This *zero extension* is analogous to the zero-extension when performing a logical right-shift:

```
unsigned short s = 4;
// short is a 16-bit format, so          s = 0000 0000 0000 0100b

unsigned int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0100b
```

Note that there is an implicit cast, from a 16-bit unsigned `short` to a 32-bit unsigned `int`. As with the casting we saw earlier, the low-order bits that are shared between the two values do not change based on the cast. However, the upper bits of the larger number do change to all zeros.

For signed values, we want the number to remain the same, just with more bits. In this case, we perform a *sign extension* by repeating the sign of the value for the new digits, in an analogous fashion to the way arithmetic right shift works. Recall that negative signed numbers start with all 1s for -1 . Table 7 shows a few 4-bit and 8-bit numbers and their respective two's complement representations.

```

// file: sizeof.c
#include<stdio.h>
#include<stdlib.h>

int main() {
    int iarray[] = {5, 4, 3, 2, 1};
    char *iarr = (char *)iarray;

    printf("sizeof(char): %d\n", (int) sizeof(char));
    printf("sizeof(short): %d\n", (int) sizeof(short));
    printf("sizeof(int): %d\n", (int) sizeof(int));
    printf("sizeof(unsigned int): %d\n", (int) sizeof(unsigned int));
    printf("sizeof(long): %d\n", (int) sizeof(long));
    printf("sizeof(long long): %d\n", (int) sizeof(long long));
    printf("sizeof(size_t): %d\n", (int) sizeof(size_t));
    printf("sizeof(void *): %d\n", (int) sizeof(void *));
    printf("sizeof(iarray): %d\n", (int) sizeof(iarray));
    printf("sizeof(iarr): %d\n", (int) sizeof(iarr));

    return 0;
}

$ gcc -g -O0 -std=gnu99 sizeof.c -o sizeof
$ ./sizeof
sizeof(char): 1
sizeof(short): 2
sizeof(int): 4
sizeof(unsigned int): 4
sizeof(long): 8
sizeof(long long): 8
sizeof(size_t): 8
sizeof(void *): 8
sizeof(iarray): 20
sizeof(iarr): 8

```

Figure 19: Example program demonstrating the `sizeof` operator. Note the difference between the size reported for an array and a pointer to the array.

Notice that the only difference between the negative numbers in both 8-bit and 16-bit representations is the leading 1s in the upper bits of the 16-bit representation.

Unsigned Integer Two's Complement Representation		
Decimal value	8-bit value	16-bit value
5	00000101	0000000000000101
82	01010010	0000000001010010
127	01111111	0000000011111111
-5	11111011	1111111111111011
-82	10101110	1111111110101110
-127	10000001	1111111110000001

Table 7: Four and eight bit signed two's complement representation.

E.g., for positive numbers:

```
short s = 4;
// short is a 16-bit format, so          s = 0000 0000 0000 0100b
```

```
int i = s;
// conversion to 32-bit int, so i = 0000 0000 0000 0000 0000 0000 0000 0100b
```

and for negative numbers:

```
short s = -4;
// short is a 16-bit format, so          s = 1111 1111 1111 1100b
```

```
int i = s;
// conversion to 32-bit int, so i = 1111 1111 1111 1111 1111 1111 1111 1100b
```

Now that we have discussed signed and unsigned integer types in C, we can see why there are two types of right shifts in C. Unsigned numbers are always logically right-shifted when we use the `>>` operator, and signed numbers are always arithmetically right-shifted when we use the `>>` operator.

Truncating Numbers

In the previous section, we converted numbers from a smaller type to a larger type, and we were able to do this without a loss of information. Negative numbers regain their value by sign-extension, and positive values retain their value by zero-extension. But what if we want to convert from a larger type into a smaller type, e.g., from a 32-bit `int` to a 16-bit `short`? If the magnitude of the number in the larger type contains more bits than the smaller type can hold, we will necessarily lose information. For example, let's look at converting a 32-bit signed `int` with the value 53191 into a signed `short`:

```
53191 = 0000 0000 0000 0000 1100 1111 1100 0111
```

When we convert to a 16-bit `short`, we *truncate* the number by removing the upper 16-bits, and we are left with:

1100 1111 1100 0111

In the short form, the number is negative, because the most significant bit is a 1, and the number is now -12345 . We have lost the information relating to the positive sign, and we have also lost information about the actual value of the number. Figure 20 shows the result in C.

```
// file: cast_lost_precision.c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int x = 53191;          // 53191
    short sx = (short) x; // -12345
    int y = sx;

    printf("x: %d\n", x);
    printf("sx: %d\n", sx);
    printf("y: %d\n", y);
    return 0;
}

$ gcc -g -O0 -std=gnu99 cast_lost_precision.c -o cast_lost_precision
$ ./cast_lost_precision
x: 53191
sx: -12345
y: -12345
```

Figure 20: Example program demonstrating the loss of information when converting from a larger integer type to a smaller integer type.

This is actually another form of overflow. We have altered the value of the number, and you must be careful to avoid overflows, or to write code that checks for this type of overflow when necessary.⁶¹

The truncation works fine if there isn't going to be any loss of information. Figure 21 demonstrates this. The 32-bit and 16-bit values after truncation both represent -3 :

```
x: 1111 1111 1111 1111 1111 1111 1111 1101
sx:                1111 1111 1111 1101
```

We can also lose information when we convert unsigned numbers due to truncation. For example, take the following snippet of code:

```
unsigned int x = 128000;
unsigned short sx = (short) x;
```

the 32-bit unsigned int representation in x of 128,000 is:

```
0000 0000 0000 0001 1111 0100 0000 0000
```

⁶¹ To check for this kind of overflow, you might compare the original number to the maximum of the smaller type, or you might reason about the sign of the number – for instance, if the sign changes, this would indicate overflow.

```
// file: cast_no_lost_precision.c
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int x = -3;
    short sx = (short) x;
    int y = sx;

    printf("x:  %d\n", x);
    printf("sx: %d\n", sx);
    printf("y:  %d\n", y);
    return 0;
}

$ gcc -g -O0 -std=gnu99 cast_no_lost_precision.c -o cast_no_lost_precision
$ ./cast_no_lost_precision
x:  -3
sx: -3
y:  -3
```

Figure 21: There is no loss of precision when converting from a smaller integer type to a larger integer type.

and the truncated unsigned short representation of sx is:

1111 0100 0000 0000

The value of sx is 62,464, which has lost the information about the most significant bit in the 32-bit version.

When integer operations overflow in C, the runtime does not produce an error. According to the C specification, unsigned integers will overflow in a defined behavior (as we saw on the number circle), although you should be wary of using calculations that could potentially overflow, and you should be careful with your code if that is a possibility. Figure 22 demonstrates one way to determine if addition on two unsigned ints will overflow.

According to the C specification, signed integers *produce undefined behavior* when they overflow. On most machines, the overflow will behave as we saw in the number circle, however, the compiler is free to ignore this and can assume that numbers will not overflow. So, you should write your programs to be careful not to overflow with signed numbers, or to check if that will be the case. Figure 23 demonstrates one way to check for overflow addition.

Final Thoughts

Having an excellent grasp of how integers are stored and manipulated in a computer is fundamental to understanding how C handles integers in programs. The two's complement format does take

Figure 22: The following is one way to determine if addition of two unsigned ints will overflow

```
#include <limits.h>
unsigned int a = <something>;
unsigned int x = <something>;
if (a > UINT_MAX - x)
    /* 'a + x' would
       overflow */;
```

Figure 23: The following is one way to determine if addition of two signed ints will overflow

```
#include <limits.h>
int a = <something>;
int x = <something>;
if ((x > 0) && (a > INT_MAX - x))
    /* 'a + x' would
       overflow */;
if ((x < 0) && (a < INT_MIN - x))
    /* 'a + x' would
       underflow */;
```

some time to become comfortable with, but it is worth the effort. Understanding the number wheels will enable you to reason out what will happen at overflow and underflow points in your code, as well.

Bits and Bytes Practice Problems

C-Strings and the C String Library

C STRINGS ARE SIMPLY A sequence of characters (chars), followed by a terminating 0. The char data type represents a fundamental type in C, as a char is 1-byte in length⁶², and the byte is the minimal addressable unit on our machines. A char may be signed or unsigned by default⁶³

The ASCII character set only defines characters for 0-127 (0x00-0x7f hex). The eighth (most significant) bit was historically used for parity checking, or error detection when transmitting characters, but ASCII is only defined for the low seven bits. There is no standard character representation for char values greater than 127.

There is a standard called Unicode⁶⁴ that is much more robust than 7-bit ASCII, and it includes over one hundred thousand characters covering hundreds of languages, scripts, and symbol sets. However, C has no built-in support for Unicode, and we will limit ourselves in CS 107 to ASCII chars.

The ctype Library

One of standard libraries for C is ctype, which includes many functions that take an int as a parameter (representing a character⁶⁵), and either test something about the character (e.g, if it is a digit, or punctuation) or to perform an operation on the character (e.g., convert the character to uppercase). The following is a list of ctype functions we will concern ourselves with:⁶⁶

isalpha: returns non-zero (true) if the character is alphabetic.

isdigit: returns non-zero if the character is a digit (0-9).

isalnum: returns non-zero if the character is alphabetic or a digit.

islower: returns non-zero if the character is lowercase.

isupper: returns non-zero if the character is uppercase.

isspace: returns non-zero if the character is whitespace.

isxdigit: returns non-zero if the character is a hex digit (0-f).

tolower: converts the character to lowercase, if possible, and returns the lowercase char as an int. If it is not possible, the behaviour is undefined.

⁶² And defined by C to be 1-byte. There is no need to ever apply sizeof on a char.

⁶³ C does not specify, and this is up to the compiler. Interestingly, most implementations (including gcc) define char to be signed.

⁶⁴ See <https://en.wikipedia.org/wiki/Unicode>

⁶⁵ The functions' parameter is not an 8-bit char, but a 32-bit int. This is because we want to be able to represent all 256 chars, but also have the ability to determine if the number is "EOF" (end of file), which is not a character in that range.

⁶⁶ The full list can be seen with a combination of man isalpha and man tolower

`toupper` : converts the character to uppercase, if possible, and returns the uppercase char as an `int`. If it is not possible, the behaviour is undefined.

The following is an example of a program that uses some of the functions above:

```
// file: ctypedemo.c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

int main(int argc, char **argv)
{
    char *string = argv[1];

    // count alpha characters, digits,
    // whitespace, and punctuation
    int alphacount = 0;
    int digitcount = 0;
    int spacecount = 0;
    int punctcount = 0;
    int total = 0;
    int i = 0;
    while (string[i] != 0) {
        if (isalpha(string[i])) alphacount++;
        if (isdigit(string[i])) digitcount++;
        if (isspace(string[i])) spacecount++;
        if (ispunct(string[i])) punctcount++;
        total++;
        i++;
    }
    printf("Alphabetic characters: %d\n", alphacount);
    printf("Digits: %d\n", digitcount);
    printf("Spaces: %d\n", spacecount);
    printf("Punctuation: %d\n", punctcount);
    printf("Total characters: %d\n", total);

    // convert to uppercase
    while (*string) {
        if (isalpha(*string)) {
            printf("%c", toupper(*string));
        } else {
            printf("%c", *string);
        }
        string++;
    }

    return 0;
}
```

```
$ gcc -g -O0 -std=gnu99 -Wall ctypedemo.c -o ctypedemo
$ ./ctypedemo "The Earth has a radius of 6,353km."
Alphabetic characters: 22
Digits: 4
Spaces: 6
Punctuation: 2
Total characters: 34
THE EARTH HAS A RADIUS OF 6,353KM.
```

C Strings in Memory

A C string is referenced by a pointer to its first character⁶⁷, as seen in Figure 24. We have drawn the string vertically, with the lower memory on the bottom, as this is the way we will depict memory for most of the class. Because a string is an array, regular pointer arithmetic can be used to traverse the string, as seen in the example above. A properly null-terminated string has 0 as its last byte, so a loop traversing the string can stop based on the null value.

Because C strings are referenced by a pointer, it is generally meaningless to compare two strings via their pointers. E.g., in the following program, the comparison is between the *addresses* of the first byte of each string, not the value of the char at the respective locations:

```
// file: pointer_compare.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    char *s1 = argv[1];
    char *s2 = argv[2];

    // the following two lines do not compare
    // the two strings!
    if (s1 < s2) printf("%s is less than %s\n", s1, s2);
    if (s1 == s2) printf("%s is equal to %s\n", s1, s2);
    if (s1 > s2) printf("%s is greater than %s\n", s1, s2);

    printf("%s address: %p\n", s1, s1);
    printf("%s address: %p\n", s2, s2);

    return 0;
}
```

```
$ gcc -g -O0 -std=gnu99 -Wall pointer_compare.c -o pointer_compare
$ ./pointer_compare cat dog
cat is less than dog
cat address: 0x7ffee0e9962
dog address: 0x7ffee0e9966
```

⁶⁷ Or by an array variable, which is converted to a pointer when we need to access the elements

Figure 24: A C string pointer and the associated memory

`char *str = "apple";`

Address	Value
0x105	\0
0x104	e
0x103	l
0x102	p
0x101	p
0x100	a

`str` 0x100 → 0x100

```
$ ./pointer_compare dog cat
dog is less than cat
dog address: 0x7ffeeb6b7962
cat address: 0x7ffeeb6b7966
```

Assigning one string pointer to another string pointer does not make a copy of the original string. Instead, both pointers point to the same string. In the following example, because the two pointers point to the same string, changing a character via either pointer changes the same string:

```
// file: string_pointers.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    char *s1 = argv[1];
    char *s2 = s1; // not a copy!

    s1[0] = 'x';
    s2[1] = 'y';

    printf("address: %p, string:%s\n", s1, s1);
    printf("address: %p, string:%s\n", s2, s2);

    return 0;
}

$ gcc -g -O0 -std=gnu99 -Wall string_pointers.c -o string_pointers
$ ./string_pointers cs107
$ ./string_pointers cs107
address: 0x7ffee837f962, string:xy107
address: 0x7ffee837f962, string:xy107
```

The string Library

One of the more important libraries we will discuss is the string library. You must be familiar with all of the following library functions, and you should rely on them for your assignments and on the exams. Unless explicitly directed to re-write part or all of a string library function, you should always utilize the functions from the library in your code. A general overview of the functions and their declarations can be found with `man string`, and as always, `man function_name` will provide a more detailed reference. All strings are expected to be null-terminated. Note: all string functions have a worst-case complexity of $\mathcal{O}(n)$, where n is the length of the string (or both strings in such cases), including `strlen`. The functions must traverse the entire string or strings.

`strlen` : Calculates and returns the length of the string. Prototype:

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

`strcmp` : Compares two strings, character-by-character, and returns 0 for identical strings, < 0 if *s* is before *t* in the alphabet, and > 0 if *s* is after *t* (digits are less than alphabetic characters).

Prototype:

```
int strcmp(const char *s, const char *t);
```

`strncmp` : Performs the same comparison as `strcmp` except that it stops after *n* characters (and does not traverse past null characters). Prototype:

```
int strncmp(const char *s, const char *t, size_t n);
```

`strchr` : Returns a pointer to the first occurrence of *ch* in *s*, or NULL if the character is not in the string. Prototype:

```
char *strchr(const char *s, int ch);
```

`strstr` : Locate a substring. Returns a pointer to the first occurrence of *needle* in *haystack*, or NULL if the substring does not exist. Prototype:

```
char *strstr(const char *haystack, const char *needle);
```

`strcpy` : Copies *src* to *dst*, including the null byte. The caller is responsible for ensuring that there is enough space in *dst* to hold the entire copy. The strings may not overlap.⁶⁸ Prototype:

```
char *strcpy(char *dst, const char *src);
```

`strncpy` : Similar to `strcpy`, except that at most *n* bytes will be copied. If there is no null byte in the first *n* bytes of *src*, then *dst* will *not* be null-terminated!⁶⁹ Prototype:

```
char *strncpy(char *dst, const char *src, size_t n);
```

`strcat` : Appends *src* onto the end of *dst*.⁷⁰ The null-terminator in *dst* is overwritten by the first character in *src*. The two strings may not overlap, and the caller is responsible for ensuring that there is enough space in *dst* to hold the resulting string.⁷¹ Prototype:

```
char *strcat(char *dst, const char *src);
```

`strncat` : Similar to `strcat` but only copies at most *n* bytes. *dst* is always null-terminated, and the caller must be careful to ensure that this is taken into account. Prototype:

```
char *strncat(char *dst, const char *src, size_t n);
```

`strspn` : Calculates and returns the length in bytes of the initial part of *s* which contains only characters in *accept*. For example, `strspn("hello", "efgh")` returns 2 because only the first two characters in "hello" are in "efgh." Prototype:

```
size_t strspn(const char *s, const char *accept);
```

⁶⁸ Be careful! `strcpy` is responsible for many *buffer overruns*, which are a primary attack vector for malicious programmers.

⁶⁹ `strcpy` and `strncpy` are not replacements for `memcpy`, although they perform similar functions.

⁷⁰ The cat stands for "concatenate," or to link together.

⁷¹ This is another relatively unsafe function!

`strcspn`: Similar to `strspn` except that `strcspn` returns the length in bytes of the initial part of `s` which *does not* contain any characters in `reject`. For example, `strspn("hello", "mnop")` returns 4 because the first four characters in "hello" are not in "mnop."

Prototype:

```
size_t strcspn(const char *s, const char *reject);
```

`strdup`: Returns a pointer to a heap-allocated string⁷² which is a copy of `s`. It is the responsibility of the caller to free the pointer when it is no longer needed. Prototype:

⁷² Obtained with `malloc`.

```
char *strdup(const char *s);
```

`strndup`: Like `strdup` but only copies up to `n` bytes. The resulting string will be null-terminated. Prototype:

```
char *strndup(const char *s, size_t n);
```

The following program demonstrates the use of some of the string library functions.

```
// file: extract_numbers.c
// purpose: print all numbers in the first
// command line argument
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, char **argv)
{
    char *s = argv[1];
    const char *digits = "0123456789";

    while (*s) {
        // count how many digits are at the front of s
        size_t prefixDigits = strspn(s,digits);

        // copy just the initial digits
        char *numstr = strndup(s,prefixDigits);

        printf("%s\n",numstr);

        // clean up
        free(numstr);

        // update s
        s += prefixDigits + strcspn(s+prefixDigits,digits);
    }

    return 0;
}
```

```
$ gcc -g -O0 -std=gnu99 -Wall extract_numbers.c -o extract_numbers
$ ./extract_numbers 123abc456x7b9
123
456
7
9
```

Final Thoughts

Strings in C are simply a string of chars, followed by a 0 byte. The character and string libraries in C provide a robust set of string manipulation tools, but it takes practice to understand them. There are many caveats to using C strings, and you should be careful to understand the nuances present when using the libraries on C strings. But, when you do understand the details, using the string libraries can save you time when writing your programs.

C Strings Practice Problems

Pointers, Generic functions with void *, and Pointers to Functions

IT IS TIME TO DIG DEEPER into C than we did in the *C Primer*. We will investigate C pointers in more detail, and we will look at the differences between stack-allocated memory and heap-allocated memory. Finally, we will look at how we can pass pointers to functions, which will allow us to make functions that are more generic.

Pointers

As covered in the *C Primer*, pointers are integers that hold a memory address. On our 64-bit Myth machines, pointers are 8-byte unsigned long values, which means that they are, ultimately, just numbers. Because of this, we can perform *pointer arithmetic* on pointers, and access addresses nearby – this is the basis for accessing an array’s values.

Every pointer has a type, declared as follows:

```
type *varname;
```

where type is any type (e.g., int, char, etc.). When declared as above, the initial value of the pointer is undefined. Pointers that have the value 0 are called NULL pointers.

The value of a pointer can be set to the address of another variable by using the “&” character:

```
int x = 7;
int *xptr = &x; // xptr points to x
```

To reiterate, the value of xptr in this case is the *address* of x in memory, whatever that may be.

If a pointer has a non-NULL value that points to a memory location a program can access, the value at that location can be *dereferenced* using the * operator:⁷³

```
printf("%d\n", *xptr); // prints 7
```

When compiling the above code, gcc knows that xptr points to an int, and therefore it knows that when it dereferences the value, it needs to get four bytes from the address pointed to by xptr. This is a key benefit of having typed pointers: if gcc only knew that

⁷³ Yes, this is an *overloaded* meaning for the asterisk character. It can be used to define a pointer, or to dereference a pointer (and it can also be used as the multiplication operator).

xptr held an address, but it didn't know what that address pointed to, it could not successfully determine how to encode the printf function parameters.⁷⁴

The type information that is embedded into the pointer is also necessary for performing pointer arithmetic on arrays. Take the following example:

```
void printarr(long *array, size_t nelems)
{
    for (size_t i=0; i < nelems; i++) {
        printf("%lu", array[i]);
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
}
```

Figure 25 shows the layout of memory – note that each long takes up 8 bytes of memory. Also note that in hexadecimal, $0x8 + 0x8$ is $0x10$.

Another way to express the printf statement from above would be:

```
printf("%lu", *(array + i));
}
```

In order to dereference the proper element in the array, gcc needs to know that each element in the array has a width of 8 bytes, and assuming the value of array is $0x100$ (as in Figure 25), the calculation of the address of the i^{th} element of the array is:

$$\text{array} + i * \text{sizeof}(\text{long})$$

which would be $0x100 + i * 8$, leading to the addresses seen in Figure 25.

Pointers -vs- Arrays

Although we covered this in the *C Primer*, it is worth repeating the differences between pointer and array variables. Pointers are variables that have enough space to hold an unsigned long, which is assumed to be a valid address, or NULL. As variables, they can be changed to have a different address. E.g., `xptr++` increments the pointer to point to the next element in an array.

Array variables are defined locally as follows:

```
int nums[10]; //space for 10 ints
```

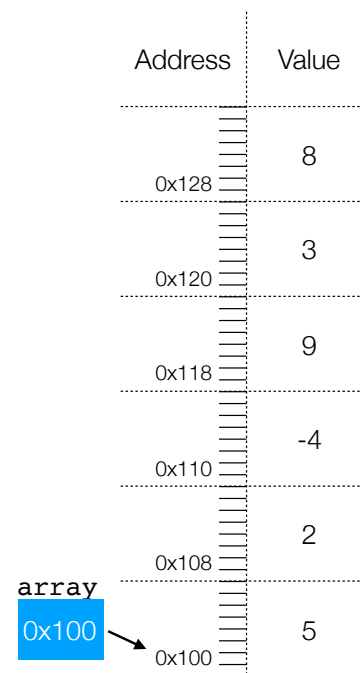
The type of `nums` above is “10-element array of int.” The type is *not* “pointer to int.” Array variables cannot be re-assigned to a different array, and they always point to the same array in memory for their lifetime. The compiler keeps track of array locations automatically (much like it keeps track of variable locations like `int x;` automatically), and there is not an actual pointer associated with an array declared in this way.

It can be instructive to look at the memory layout for a program that uses both an array and a pointer the array:

⁷⁴ Well, it probably could use the printf string as a guide, knowing that %d expects an int. But, take the following example:

```
long z = *xptr;
gcc could assume that xptr pointed to a long, but that would be incorrect.
```

Figure 25: An array of longs in memory



```
// file: array_vs_pointer.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    int nums[5];
    int nelems_nums = sizeof(nums) / sizeof(int);
    for (int i=0; i < nelems_nums; i++) {
        // fill with index
        nums[i] = i;
    }
    int *numsptr = nums;
    // this is equivalent to:
    // int *numsptr = &nums[0]

    for (int i=0; i < nelems_nums; i++) {
        printf("%d, ", nums[i]); // one way of printing
        printf("%d, ", *(nums + i)); // print again
        printf("%d, ", numsptr[i]); // print again
        printf("%d", *(numsptr + i)); // print again
        i == nelems_nums - 1 ? printf("\n")
                               : printf(", ");
    }
    return 0;
}

$ gcc -g -O0 -std=gnu99 array_vs_pointer.c -o array_vs_pointer
$ ./array_vs_pointer
0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4
```

Figure 26 shows a potential memory layout for the array and pointer. Notice that the pointer `numsptr` has a location in memory dedicated to hold the address, and the array variable `nums` just refers to the location of the array, without dedicating a pointer to hold the value.

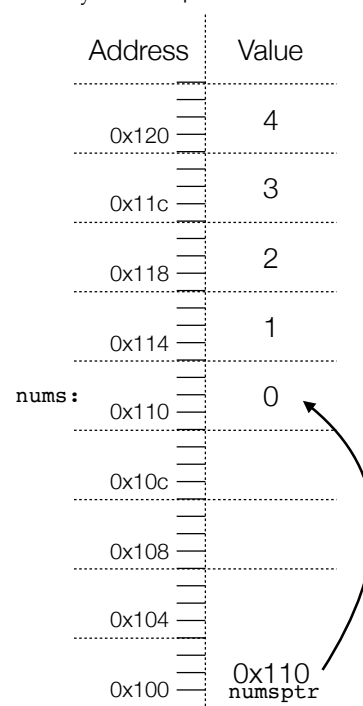
Because arrays can be dereferenced and because you can use bracket notation to refer to the elements of an array, they behave like a pointer when used that way. In the code above, we have printed each array element in four different ways.

For more detailed information on Dennis Ritchie's reasoning behind developing arrays the way he did, see the article he wrote on his process.⁷⁵

Generic functions with `void *`

The beauty of C's type system as it applies to pointers is that the compiler has the ability to automatically handle the underlying data type that a pointer points to. However, this necessitates associating a pointer to a particular type, and this results in functions

Figure 26: Possible memory layout for `nums` array and `numsptr`



⁷⁵ Dennis M Ritchie. The development of the c language. 1993. URL <https://www.bell-labs.com/usr/dmr/www/chist.pdf>

that only work on a specific type of data. Take the following program, which swaps the beginning and end elements of an array, as an example:

```
// file: nongeneric.c
#include<stdio.h>
#include<stdlib.h>

void swap_ends_short(short *arr, size_t nelems)
{
    short tmp = *arr;
    *arr = *(arr + nelems - 1);
    *(arr + nelems - 1) = tmp;
}

void swap_ends_int(int *arr, size_t nelems)
{
    int tmp = *arr;
    *arr = *(arr + nelems - 1);
    *(arr + nelems - 1) = tmp;
}

void swap_ends_long(long *arr, size_t nelems)
{
    long tmp = *arr;
    *arr = *(arr + nelems - 1);
    *(arr + nelems - 1) = tmp;
}

int main(int argc, char **argv)
{
    short s_array[] = {1,4,8,2,-3,5};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    int i_array[] = {10,40,80,20,-30,50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {100,400,800,200,-300,500};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    swap_ends_short(s_array, s_nelems);
    swap_ends_int(i_array, i_nelems);
    swap_ends_long(l_array, l_nelems);

    printf("s_array[0]:%d\t", s_array[0]);
    printf("s_array[%lu]:%d\n", s_nelems - 1, s_array[s_nelems-1]);

    printf("i_array[0]:%d\t", i_array[0]);
    printf("i_array[%lu]:%d\n", i_nelems - 1, i_array[i_nelems-1]);
}
```

```

    printf("l_array[0]:%lu\t", l_array[0]);
    printf("l_array[%lu]:%lu\n", l_nelems - 1, l_array[l_nelems - 1]);

    return 0;
}

$ gcc -g -O0 -std=gnu99 nongeneric.c -o nongeneric
$ ./nongeneric
s_array[0]:5,   s_array[5]:1
i_array[0]:50,  i_array[5]:10
l_array[0]:500, l_array[5]:100

```

We had to write three virtually identical functions to perform the same work, because we have three different types of arrays that we want to perform the swap on. This is a problem with the type system, and we would like to only have to write a function once to accomplish the task. Our goal fits into the category called *generic programming*, and while it may be easier to accomplish in other languages, we can write generic functions in C.

To do so, we are going to utilize the `void *` pointer, which we saw briefly in the *C Primer*. Recall that a `void *` pointer is a pointer without an associated type, and it can point to any underlying type. Because we necessarily lose that information, we will have to pass along the type information in a generic way in our functions.

With a `void *` pointer, the missing piece turns out to be, simply, the width of the type that we are passing in. In other words, as Figure 25 showed, each element in an array has a width, and we can perform a calculation to get to the next element if we know the width. Because we are losing the width information by using a `void *` pointer, we will have to perform the calculation manually, without the help of the compiler.

When we perform the array-offset calculation, we will have to fool the compiler into providing us with the correct offset in our pointer. To do this, we cast the `void *` pointer to a 1-byte representation, which, in C, is the `char *` pointer. Now, when we perform pointer arithmetic on the `char *` pointer, we can get the address of the type by manually multiplying by the width.

This is best described with an example. The following program in a generic version of the program above. The key ideas are:

1. The pointer type information in the parameter is now `void *`.
2. We have added a width parameter, to be able to determine how far apart each element is in the array. We must calculate and pass the width information to the function.
3. We cannot simply use an assignment to swap the values, as the compiler does not know the width of the type on its own, and assignments (e.g., `tmp = arr[0]`) is not possible unless the information is known at compile time. So, instead, we create a `char` array that allocates the space we need, in this case, the

width of one element. We then use the `memcpy` function to move width number of bytes to perform the swap.

4. We must manually do the pointer calculation, remembering to cast the pointer to `char *` (fooling the compiler).

```
// file: generic.c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

void swap_ends(void *arr, size_t nelems, int width)
{
    // allocate space for the copy
    char tmp[width];

    // copy the first element to tmp
    memcpy(tmp, arr, width);

    // copy the last element to the first
    memcpy(arr, (char *)arr + (nelems - 1) * width, width);

    // copy tmp to the last element
    memcpy((char *)arr + (nelems - 1) * width, tmp, width);
}

int main(int argc, char **argv)
{
    short s_array[] = {1,4,8,2,-3,5};
    size_t s_nelems = sizeof(s_array) / sizeof(s_array[0]);

    int i_array[] = {10,40,80,20,-30,50};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {100,400,800,200,-300,500};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    swap_ends(s_array, s_nelems, sizeof(s_array[0]));
    swap_ends(i_array, i_nelems, sizeof(i_array[0]));
    swap_ends(l_array, l_nelems, sizeof(l_array[0]));

    printf("s_array[0]:%d,\t", s_array[0]);
    printf("s_array[%lu]:%d\n", s_nelems - 1, s_array[s_nelems-1]);

    printf("i_array[0]:%d,\t", i_array[0]);
    printf("i_array[%lu]:%d\n", i_nelems - 1, i_array[i_nelems-1]);

    printf("l_array[0]:%lu,\t", l_array[0]);
    printf("l_array[%lu]:%lu\n", l_nelems - 1, l_array[l_nelems-1]);
}
```

```

    return 0;
}

$ gcc -g -O0 -std=gnu99 generic.c -o generic
$ ./generic
s_array[0]:5,   s_array[5]:1
i_array[0]:50,  i_array[5]:10
l_array[0]:500, l_array[5]:100

```

Generic programming takes some time to understand! Look over the examples above carefully, and ensure that you are completely clear on why we needed each change from the non-generic functions to the single generic function.

Pointers to Functions

Looking at the generic example program in the previous section, you might have some lingering concerns about the necessity of having to create a char array to store the data for the swap. What if we had to perform some application where we really did need to know the type of the data – could we still use a generic function?

Let's look at the following example:

```

// file: printf_generic_problem.c
#include<stdio.h>
#include<stdlib.h>

void print_array(void *arr, size_t nelems, int width)
{
    for (int i=0; i < nelems; i++) {
        void *element = (char *)arr + i * width;
        printf("%?", element); // what goes in place of the ?
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
}

int main(int argc, char **argv)
{
    int i_array[] = {0,1,2,3,4,5};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    int l_array[] = {0,10,20,30,40,50};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    print_array(i_array, i_nelems, sizeof(i_array[0]));
    print_array(l_array, l_nelems, sizeof(l_array[0]));

    return 0;
}

```

We have a problem. The `printf` call in the `print_array` function needs a format specifier, which is dependent on the type of data we have in the array. For standard data types, we might be able to get away with passing in the correct format specifier, but what if our array had pointers to a struct, or something else that we wanted to print in a custom way?

What we really want is to have the `print_array` function perform a custom function, based on the type of data, and that is exactly what a *pointer to a function* provides. A pointer to a function allows the generic function to pass its data to a custom function, provided by the calling function, that will handle the individual array element in the way that makes sense to the calling function. In the case of our example, the calling function can provide a print function that will do the work, and the `print_array` function only needs to know how to provide the function the data, which is going to be embedded into the type of the function pointer.

The function pointer syntax can be confusing, and it takes practice to get used to reading the function signature. Here is an example, which we will use for our `print_array` function:

```
void(*pr_func)(void *)
```

This is interpreted from the inside out, starting with the pointer to the function name, which is `pr_func`. The function takes one parameter, which is a `void *`, and the function's return value is `void`. Functions that would meet this definition (which we will use in our program) is:

```
void print_int(void *arr);
void print_long(void *arr);
```

In other words, we will pass in one of the functions above into our `print_array` function. We have to write a custom `pr_func` for any data type that we want to use in the `print_array` function. For example, here is the full `print_int` function:

```
void print_int(void *arr)
{
    int i = *(int *)arr;
    printf("%d", i);
}
```

A key idea about this function is that because it is custom-built to take an `int *`, we *know* that the `void *` can be cast to an `int *` inside the function. The function signature must use a `void *`, but inside the function we know that it will be an `int *`, and we can cast it.

The following is the completed program that includes the function pointer in the `print_array` function:

```
// file: printf_generic.c
#include<stdio.h>
#include<stdlib.h>
```

```

void print_array(void *arr, size_t nelems, int width, void(*pr_func)(void *))
{
    for (int i=0; i < nelems; i++) {
        void *element = (char *)arr + i * width;
        pr_func(element);
        i == nelems - 1 ? printf("\n") : printf(", ");
    }
}

void print_int(void *arr)
{
    int i = *(int *)arr;
    printf("%d", i);
}

void print_long(void *arr)
{
    long l = *(long *)arr;
    printf("%lu", l);
}

int main(int argc, char **argv)
{
    int i_array[] = {0,1,2,3,4,5};
    size_t i_nelems = sizeof(i_array) / sizeof(i_array[0]);

    long l_array[] = {0,10,20,30,40,50};
    size_t l_nelems = sizeof(l_array) / sizeof(l_array[0]);

    print_array(i_array, i_nelems, sizeof(i_array[0]), print_int);
    print_array(l_array, l_nelems, sizeof(l_array[0]), print_long);

    return 0;
}

```

```

$ gcc -g -O0 -std=gnu99 printf_generic.c -o printf_generic
$ ./printf_generic
0, 1, 2, 3, 4, 5
0, 10, 20, 30, 40, 50

```

The following is an example of how we could use the same function⁷⁶ to use a more complicated printing function:

```

// file: printf_coords.c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

struct coordinate {
    double lat;

```

⁷⁶ with a minor adjustment to the list format to take out the newline and comma.


```

    double lon;
};

void print_array(void *arr, size_t nelems, int width, void(*pr_func)(void *))
{
    for (int i=0; i < nelems; i++) {
        void *element = (char *)arr + i * width;
        pr_func(element);
    }
}

void print_coord(void *arr)
{
    struct coordinate *coord = (struct coordinate *)arr;
    int lat_deg = (int)coord->lat;
    double lat_minsec = fabs((coord->lat - lat_deg) * 60.0);
    int lat_min = (int)lat_minsec;
    double lat_sec = (lat_minsec - lat_min)* 60.0;

    int lon_deg = (int)coord->lon;
    double lon_minsec = fabs((coord->lon - lon_deg) * 60.0);
    int lon_min = (int)lon_minsec;
    double lon_sec = (lon_minsec - lon_min)* 60.0;

    printf("%f Latitude, %f Longitude =\n", coord->lat, coord->lon);
    printf("Latitude: %d deg %d min %f sec\n", lat_deg, lat_min, lat_sec);
    printf("Longitude: %d deg %d min %f sec\n\n", lon_deg, lon_min, lon_sec);
}

int main(int argc, char **argv)
{
    struct coordinate coords[] = {{37.4301566, -122.1756849},
                                   {38.8975062, -77.0388237},
                                   {-82.8627513, 134.9978113},
                                   {51.5287718, -0.2416818}
    };
    size_t coords_nelems = sizeof(coords) / sizeof(coords[0]);

    print_array(coords, coords_nelems, sizeof(coords[0]), print_coord);

    return 0;
}

```

```

$ gcc -g -O0 -std=gnu99 printf_coords.c -o printf_coords
$ ./printf_coord
37.430157 Latitude, -122.175685 Longitude =
Latitude: 37 deg 25 min 48.563760 sec
Longitude: -122 deg 10 min 32.465640 sec

```

```
38.897506 Latitude, -77.038824 Longitude =
Latitude: 38 deg 53 min 51.022320 sec
Longitude: -77 deg 2 min 19.765320 sec
```

```
-82.862751 Latitude, 134.997811 Longitude =
Latitude: -82 deg 51 min 45.904680 sec
Longitude: 134 deg 59 min 52.120680 sec
```

```
51.528772 Latitude, -0.241682 Longitude =
Latitude: 51 deg 31 min 43.578480 sec
Longitude: 0 deg 14 min 30.054480 sec
```

You must be careful with the arguments and return values for functions to pointers, and we will show an additional example that demonstrates this.

A frequent use of functions to pointers is the *comparison function*. The standard C comparison function compares two elements, and returns a negative value if the first element is less than the second element, zero if the elements are the same, and a positive value if the first element is larger than the second element. The definition of “less than,” “equal to,” and “greater than” depends on the things that are being compared⁷⁷ One example of a standard library function that accepts a generic comparison function is the `qsort` function, which (as its name suggests) performs a sort on an array of elements. The prototype for `qsort` is:

⁷⁷ although it is normally based on a numeric property of the elements.

```
void qsort(void *base, size_t nmem, size_t size,
           int (*compar)(const void *, const void *));
```

The `compar` argument in `qsort` is a function pointer that takes two `const void *` pointers to compare, and returns an `int`. The following program sorts the command line arguments (after the program name) by *the sum of the character values in the names*:

```
// file: compare_words.c
#include<stdio.h>
#include<stdlib.h>

int compare_words(const void *p, const void *q)
{
    int psum = 0;
    int qsum = 0;

    char *pword = *(char **)p;
    while (*pword) psum += *(pword++);

    char *qword = *(char **)q;
    while (*qword) qsum += *(qword++);

    return psum - qsum;
}
```

`qsort` passes in a pointer to the pointer of the two words to compare!

```

int main(int argc, char **argv)
{
    qsort(argv+1, argc-1, sizeof(argv[0]), compare_words);
    for (int i=1; i < argc; i++) {
        printf("%s", argv[i]);
        i == argc - 1 ? printf("\n") : printf(", ");
    }
    return 0;
}

```

```

$ gcc -g -O0 -std=gnu99 compare_words.c -o compare_words
$ ./compare_words apple nectarine banana orange peach pear
pear, peach, apple, banana, orange, nectarine

```

Students tend to gloss over some of the critical details of this example, and it is instructive to take a good look at the various pointers and the memory footprint of what is happening here. Figure 27 shows the layout in memory of the command line arguments.

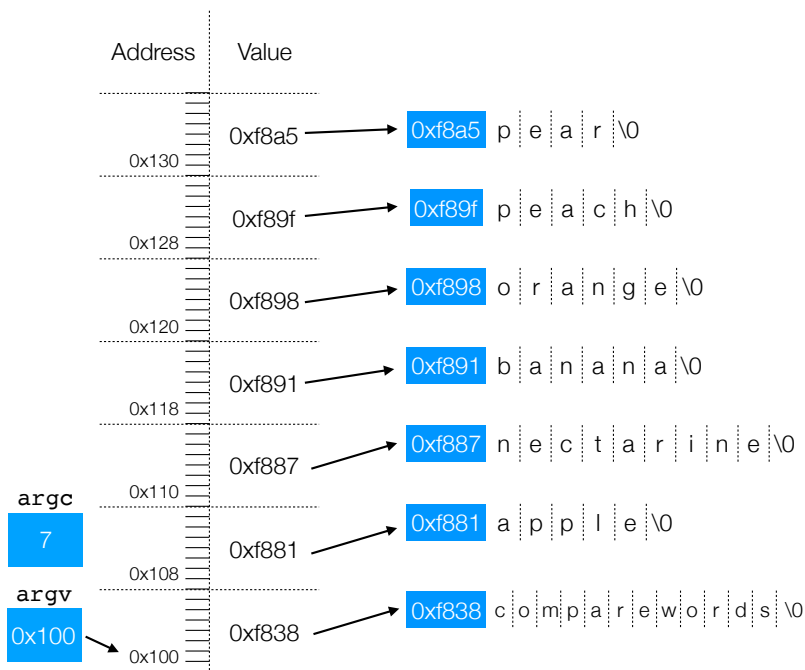


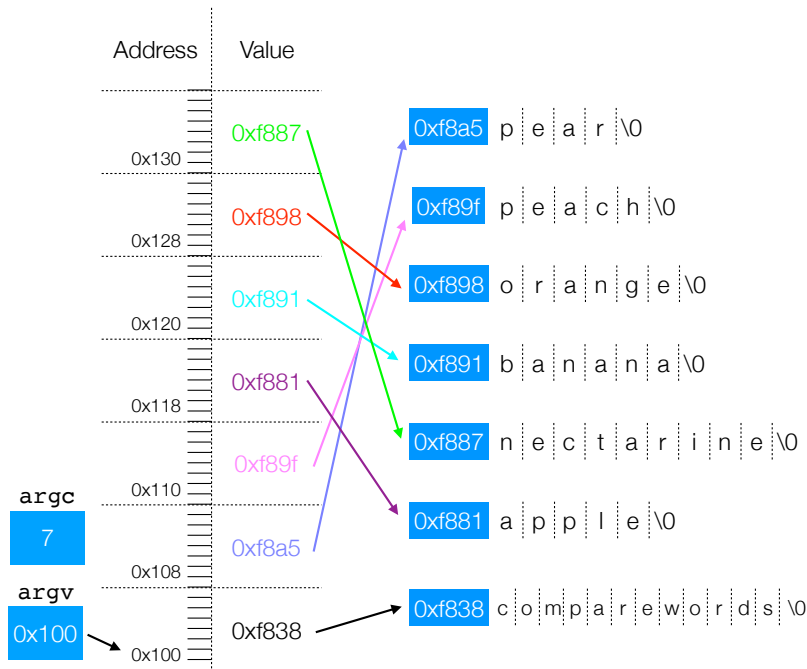
Figure 27: The command line arguments array

The argv variable is a pointer to an array of char * pointers. Because it is an array, the pointers must be contiguous in memory, and they are, from address 0x100 through 0x107. Because they are pointers, they each take up 8 bytes in memory (e.g., argv[0]'s bytes are from 0x100 - 0x107).

The pointers for each char * *do not have to be in any particular place in memory*. Notice that they are not a fixed distance apart. Because they are command line arguments, each string happens to be next to each other, but *this is not a critical point*.

The `qsort` function is passed in the value for `(argv + 1)`⁷⁸, which (in our diagram) is the value `0x108`. As far as `qsort` is concerned, it simply has an array of some type of pointer, and it will rearrange those pointers in the array based on the comparison function. For example, `qsort` may, at some point, pass the values `0x108` and `0x110` to the comparison function. Those values are still pointers to `char *` pointers, although `qsort` does not know this information (it simply knows that they are pointers). This is why the comparison function, `compare_words` knows that the arguments are both `char **` pointers, because that is what it must receive from `qsort`.

The `qsort` function sorts the array, and by this we mean that it *rearranges the elements in the array itself*. This is critical to understand: `qsort` never touches the characters in the strings (though the comparison function does) – it simply moves the pointers in the array to correctly sort the elements. Figure 28 shows the result after the sort.



⁷⁸ Because we are ignoring the program name, `argv[0]`.

Figure 28: The command line arguments array after being sorted by the sum of the character values of each argument string (except the program name): pear, peach, apple, banana, orange, nectarine.

Final Thoughts

Pointers take time to understand, especially when it comes to pointers to pointers, void `*`, generic functions, and pointers to functions. It would be well-worth your time to go back over this chapter and really understand the nuances of the code.

Pointer Practice Problems

IEEE Floating Point

AS WE HAVE LEARNED, ALL DATA used by a computer is represented by bits and bytes. Integer types each have a fixed number of bytes that represent their exact numeric value, characters are represented by 1-byte ASCII encodings, and strings are simply arrays of null-terminated characters. For a given integer type, we know the range⁷⁹ and we have the exact representations of each integer in that range.

The real world, however, does not fit perfectly into this completely-representable model – numbers such as $\frac{1}{3}$ and π cannot be simply broken down into a data type that can exactly represent their values.⁸⁰ Additionally, between any two numbers there is an infinite number of rational and real numbers, so representing ranges of numbers exactly is impossible. Therefore, computer designers are forced to compromise when creating a type to hold real numbers, and one of the most successful solutions to the problem is called *The IEEE Standard for Floating Point Arithmetic* (IEEE 754, or IEEE Floating Point)⁸¹, which has become a standard that is implemented in most computer hardware today. The IEEE Floating Point standard was designed to encode the representation of real numbers into a fixed-width format, and to provide a large range with a definable precision, precise rounding rules, and special values such as negative and positive infinity, and special *not a number* (NaN) values that denote a calculation that has become not-real⁸². The IEEE Floating Point format is a clever solution to the problem, and in this chapter we will take a look at the format, and we will investigate some of the decisions that were made and the trade-offs that this implies. We will learn how to convert between a decimal representation and the binary IEEE Floating Point representation, and you should become familiar enough with the format to convert some categories of numbers⁸³ between the two formats.

Fixed Point Format

When deciding how to represent real numbers in binary form, we have some choices:

1. We want to represent real numbers in a fixed number of bits.
This means that we aren't going to be able to represent all real

⁷⁹ INT_MIN to INT_MAX for the int type, for example.

⁸⁰ Rational numbers can be exactly represented with a numerator / denominator model.

⁸¹ The [Wikipedia Page](#) is outstanding.

⁸² e.g., division by zero, or the square root of -1 .

⁸³ For example, powers of two, and other small numbers such as 1.25.

numbers exactly, nor even all rational numbers exactly. Furthermore, we can't even represent all rational numbers in a range exactly (there are infinitely many rational numbers in any fixed range).

2. We want to represent a large range of numbers.
3. We want to be able to perform calculations on the numbers.

One idea we might consider is called *fixed point* format. Integers represent a fixed point format, with the decimal point (or binary point) located implicitly to the right of the least significant digit. We also limited ourselves to numbers with factors that were positive multiples of the base. E.g., the decimal integer 1234 is really $1234.000\dots$, and the binary equivalent, 10011010010 is really $10011010010.000\dots$. If we wanted to, we could have a system for representing real numbers just move the point to the left some number of digits, so that we can represent numbers between 0 and 1. Maybe we want to have two decimal digits of precision:

$$d_2d_1d_0.d_{-1}d_{-2} = d_2 \times 10^2 + d_1 \times 10^1 + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2}$$

E.g., we could now represent numbers like 123.45. The range of those numbers would be between 0 and 999.99, and we have five decimal digits of precision, to the 100^{th} decimal place. With our format, however, we cannot represent some numbers exactly, e.g., 123.456 or 0.33...

Arithmetic with fixed point numbers is relatively easy, because we simply line up the decimal place:

$$\begin{array}{r} 123.45 \\ + 678.90 \\ \hline 802.35 \end{array}$$

and

$$\begin{array}{r} 100.22 \\ * 1.08 \\ \hline 80176 \\ 000000 \\ 1002200 \\ \hline 1082376 \end{array} = 108.2376 = 108.24 \text{ (rounded)}$$

Fixed point format has one glaring problem: the range of our numbers is severely limited. For our 5-digit example above, if we set the decimal point to the left of the most significant digit, we could only represent numbers in the range of 0 to 0.99999. Although fixed point format has its place,⁸⁴ it is too limiting for a general format for real numbers.

⁸⁴ It can provide better accuracy in some cases.

Floating Point Format

The way we obtain a worthwhile range for our number format is to take advantage of the fact that we can scale numbers efficiently by

multiplying them by a power of the base we are working in. We can thus decide to represent the numbers as follows:

$$N = x \times 10^y \text{ (decimal)}$$

or

$$N = x \times 2^y \text{ (binary)}$$

We will concentrate on the binary representation from now on, as it is the way we will eventually encode the data. When representing binary numbers, digits after the *binary point* are represented by negative powers of two:

$$b_2b_1b_0.b_{-1}b_{-2} = b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0 + b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2}$$

For example, 101.11b is:

$$\begin{aligned} 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} \\ = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4} \end{aligned}$$

We have written an online binary to decimal (and vice-versa) calculator that you can play with: <https://stanford.edu/~cgregg/107-Reader/float/convert.html>. If you type in 101.11 into the Binary to Decimal converter, you will find that it is 5.75, as we calculated above.

Just like in decimal format, binary cannot represent numbers such as $\frac{1}{3}$ and $\frac{1}{6}$ exactly, nor can binary represent some other numbers that can be represented in base 10 exactly, like $\frac{1}{10}$. If you type 0.1 into the Decimal to Binary converter mentioned above, you will get:

$$0.000110011\overline{0011} \dots$$

which, as you can see, is a non-terminating rational number when expressed in binary. Binary notation can exactly represent numbers in the form

$$x \times 2^y$$

where x and y are integers.

Once we decide to use this format for representing real numbers, we have entered the realm of *floating point* formats. In other words, we will keep track of the values of x and y , and the binary point will “float” in a particular number. For a large number, the binary point might be far to the right of the significant digits, and for a small number the binary point might be far to the left of the significant digits. In other words, the power 2^y scales the location of the binary point.

The IEEE Floating Point standard was first designed by William Kahan⁸⁵, and he had many choices to make while architecting the format. As we go through the format, we will see some of those choices, and why they ended up being good ones.⁸⁶

⁸⁵ Kahan is known as *The father of Floating Point*.

⁸⁶ And sometimes confusing ones, too.

The IEEE Floating Point format defines a bit pattern to represent real numbers in the following form:

$$V = (-1)^s \times M \times 2^E$$

where:

- s is a sign bit (0 for positive, 1 for negative, with the sign for numerical zero as a special case)
- M is the *significand*⁸⁷, which represents the fractional binary number in the range of $1 \leq M < 2$, or in the range of $0 \leq M < 1$ for special cases (which we will discuss).
- E is the exponent, which scales the value by a (possibly negative) power of 2.

For example, if we have $s = 0$, $M = 1.5$, and $E = 9$:

$$V = (-1)^0 \times 1.5 \times 2^9 = 768$$

To encode the bits themselves, IEEE Floating Point defines many bit-widths, but we will concentrate on the 32-bit format (and briefly discuss some details about the similar 64-bit format). The bit representation of a floating point number is divided into three fields to encode s , M , and E :

- A single sign bit, s directly encodes the sign.
- A k -bit exponent field (8-bits for the 32-bit format), $e_{k-1} \cdots e_1 e_0$ encodes the exponent, E (though not directly, as we shall see).
- An n -bit fraction field (23-bits for the 32-bit format), $f_{n-1} \cdots f_1 f_0$ encodes the significand, M , and this value depends on whether or not the exponent field equals zero.

Figure 29 shows the bit representation for a *single precision* (32-bit) floating point number in IEEE Floating Point format.

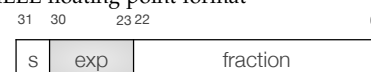
Normalized Floats

The IEEE Floating Point format has three different possible interpretations, depending on the value of the exponent field: *normalized*, *denormalized*, and *exceptional*. Most of our time will be spent learning about the normalized form, which handles the vast majority of the numbers represented by the format. The denormalized numbers are a special case for numbers with the very smallest magnitude, and exceptional numbers are numbers such as infinity and NaN.

A float is considered to have a normalized value if the exponent is not all 0s, and not all 1s. In a normalized number, the exponent is encoded as a *signed integer*⁸⁸ that is in *biased* form. The exponent has a value of $exp - bias$, where $bias$ is $2^{k-1} - 1$, and where k is the number of bits in the exponent. For a 32-bit number, there are

⁸⁷ sometimes called the *mantissa* (thus the M designation).

Figure 29: Single precision (32-bit) IEEE floating point format



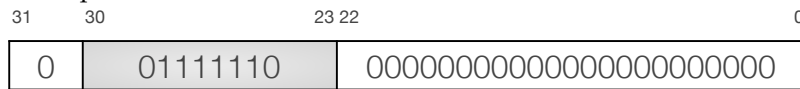
⁸⁸ Not a two's complement number! It is an 8-bit binary number that is interpreted as a signed number, based on the bias.

eight bits in the exponent field, so $bias = 2^{8-1} - 1 = 2^7 - 1 = 127$. As an example, if the *exp* field has a value of 10000001 (129, decimal), the value of the numerical exponent would be interpreted as $129 - 127 = 2$. The range of exponents for a normalized number is -126 to 127 , because the range of the *exp* field is between 1 and 254 (because normalized numbers are only defined when the *exp* field is not all 0s (00000000) or all 1s (11111111)).⁸⁹

For normalized numbers, the *fraction* is interpreted as having a fractional value f , where $0 \leq f < 1$, and having a value of $0.f_{n-1} \dots f_1 f_0$, with the binary point to the left of the most significant bit. However, this is not yet the significand!

The significand is defined to be $M = 1 + f$. This is an *implied leading 1 representation*, and it is a trick for getting an actual digit of precision for free. Here is why: for any number that has ones in it at all, we can always adjust the exponent so that the significand is in the range of $1 \leq M < 2$. When multiplying by a power of two (the exponent), this simply shifts the number to the left or to the right, so we can always adjust the exponent to shift the number into a form of $1.f$. Therefore, because this is *always* possible, it is unnecessary to actually encode the 1 into the representation of the number.⁹⁰

Let's take a look at an example, which should clear up potential misconceptions:



The sign bit, 0 means that the number is positive.

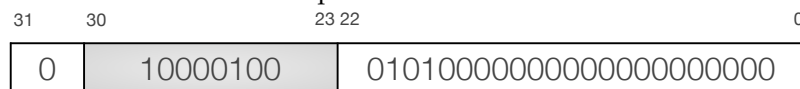
The *exp* field has the value of 01111110, or 126 decimal, and it has been biased by 127. Therefore, the exponent value we will interpret *exp* as will be $126 - 127 = -1$.

The *fraction* field has a value of all 0s, but we interpret this as the binary value of $1.000\dots$.

Therefore, the number represents:

$$+1.0 \times 2^{-1} = 0.5$$

Let's look at another example:



The sign bit, 0 means that the number is positive.

The *exp* field has the value of 10000100, or 132 decimal, and it has been biased by 127. Therefore, the exponent value we will interpret *exp* as will be $132 - 127 = 5$.

The *fraction* field has a value of 010100000000000000000000, but we interpret this as the binary value of $1.01010\dots$:

$$1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 1.3125$$

Therefore, the number represents:

$$+1.3125 \times 2^5 = 42.0$$

⁸⁹ For example, if the exponent of the real number we wanted to encode was -126 , we would add 127 to bias the number giving us 00000001 for the *exp* field. We could not encode an exponent of -127 because $-127 + 127 = 0$, or 00000000, which would not represent a normalized number.

⁹⁰ This is an example of where the designers of the IEEE Floating Point format did something extremely clever, at the possible expense of being harder to understand.

We can check this at the website from above: <https://stanford.edu/~cgregg/107-Reader/float/convert.html>

To put a decimal number into normalized 32-bit floating point format, we must determine the bits for each of the three parts of the number. Let's take the example of converting the decimal number 0.4. If we plug that number into the Decimal to Binary converter at <https://stanford.edu/~cgregg/107-Reader/float/convert.html>, we get the following number:

0.011001100110011001100110011

This is a non-terminating rational number, so we *won't be able to represent it exactly*. Next, we find the most significant 1 in the number, and then take the following 23 bits after that, in order to form the significand (in bold, below):

0.01**1001100110011001100110011**

In this case, we have one more tiny step to do: we must round the number. Notice that the number after the 23rd digit is a 1. Because 0.1 in binary is $\frac{1}{2}$, the rest of the number is *greater than one half*. Therefore, just like when we round up in decimal when digits when the digit after the rounding place is 5, we round up when the number after the 23rd digit is 1 and there are any more 1s after it (and we do not round up if the number is 0, or if there is a single 1⁹¹. In order to round up a number, add 1 to it. So, we have:

0.01**10011001100110011001101**

The number in bold above will be the 23 bits of the significand.

Next, we have to scale the number so that it falls between 1.0 and 2, which would look like this:

1.**10011001100110011001101**

Notice that we had to shift the number two binary places to the *right*. Therefore, the exponent of two that we will multiply by is going to be -2 . We need to bias this number by 127, so the exponent will be $-2 + 127 = 125$, or 01111101, which is the exponent field for the number.

Finally, we know that the sign bit will be 0 because it is a positive number. The following depicts the bits in the floating point representation for 0.4:



There are websites to check if you are correct⁹², or you can write a quick program and run it in gdb. The gdb command "`x/tw &f`" means, *print the memory located at the address of f in binary ("t") and in word size ("w", or 4-bytes)*.

```
// file: floatingpt.c
#include<stdio.h>
#include<stdlib.h>
```

⁹¹ This is called *rounding towards nearest, ties to even*. There are actually five rounding rules, but they are beyond the scope of CS 107. See [the Wikipedia page](#) for more information.

⁹² This is an outstanding site: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>

```
int main(int argc, char **argv)
{
    float f = 0.4;
    printf("%g\n", f);
    return 0;
}
```

```
$ gcc -g -O0 -std=gnu99 floatingpt.c -o floatingpt
```

```
$ gdb floatingpt
```

The target architecture is assumed to be i386:x86-64

Reading symbols from floatingpt...done.

```
(gdb) break main
```

Breakpoint 1 at 0x40053c: file floatingpt.c, line 7.

```
(gdb) run
```

Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107/
reader/107-Reader-code/floatingpt

Breakpoint 1, main (argc=1, argv=0x7fffffffefaa8) at floatingpt.c:7

```
7         float f = 0.4;
```

```
(gdb) n
```

```
8         printf("%g\n", f);
```

```
(gdb) x/tw &f
```

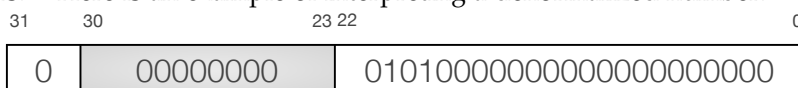
```
0x7fffffffef9bc:      00111110110011001100110011001101
```

```
(gdb)
```

Denormalized Floats

When the exponent is all zeros, this is called *denormalized* form. We interpret the exponent differently: the exponent is now $1 - \text{bias}$ (or, you can think of the bias now being 1 less, or 126 instead of 127 in the case of 32-bit floats). The significand value is simply the fraction, without a leading 1. Why do we do this? For one, we now have a way to represent zero (all 0s for the entire number). All 0s is $+0.0$, and a 1 followed by all 0s is -0.0 . A nice effect is that just like integers, when all the bits in a floating point number are zero, the number means zero.

Second, changing the bias to 126 leads to *gradual underflow*, meaning that it allows us to extend the lower range of representable numbers, and to limit the amount of error with very small numbers.⁹³ Here is an example of interpreting a denormalized number:



The sign bit, 0 means that the number is positive.

The *exp* field has the value of 00000000, meaning that the number is denormalized, and has a bias of 126. Therefore, the exponent value we will interpret *exp* as will be $0 - 126 = -126$.⁹⁴

The *fraction* field has a value of 010100000000000000000000, and

⁹³ See here for more information than you may ever want: https://docs.oracle.com/cd/E19957-01/816-2464/ncg_math.html

⁹⁴ and all denormalized numbers have that exponent.

we interpret this as the binary value of $0.01010\dots$:

$$0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 0.3125$$

Therefore, the number represents:

$$+0.3125 \times 2^{-126} = 3.67341985 \times 10^{-39}$$

The following program and gdb trace demonstrates that we are correct:

```
// file: denormalized.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    float f = 3.6734198463196485e-39;
    printf("%g\n",f);
    return 0;
}

$ gcc -g -O0 -std=gnu99 denormalized.c -o denormalized
$ gdb denormalized
The target architecture is assumed to be i386:x86-64
Reading symbols from denormalized...done.
(gdb) break main
Breakpoint 1 at 0x40053c: file denormalized.c, line 7.
(gdb) run
Starting program: /afs/.ir.stanford.edu/users/c/g/cgregg/cs107/
reader/107-Reader-code/denormalized

Breakpoint 1, main (argc=1, argv=0x7fffffffefaa8) at denormalized.c:7
7          float f = 3.67341985e-39;
(gdb) n
8          printf("%g\n",f);
(gdb) x/tw &f
0x7fffffffef9bc:      00000000001010000000000000000000
(gdb)
```

Exceptional Floats

When the exponent bits in an IEEE float are all 1s, the number is considered *exceptional*. Two interesting exceptional numbers are ∞ , which is 0 11111111 000000000000000000000000 (for a 32-bit float), and $-\infty$, which is 1 11111111 000000000000000000000000. Positive infinity can arise when a floating point number overflows, and negative infinity can arise when a number underflows (even smaller than the denormalized values), and other calculations such as $\frac{1}{0}$ can also produce infinity.

The other exceptional numbers⁹⁵ are either NaNs, or representations that have a special meaning for a particular processor.

⁹⁵ and there are over 8-million of them!

Unrepresentable Numbers

An important understanding of floating point numbers is that while the range is large (roughly 2^{-126} to 2^{127} for 32-bit floats, and 2^{-1022} to 2^{1023} for 64-bit floats), there are infinitely many real numbers that cannot be represented by the format. We have already seen that 0.4 decimal cannot be exactly represented. A 32-bit float can, by definition, only represent 2^{32} values, so most, including many integers, are not representable in the format. 0.4 is an example of a non-terminating rational number that cannot be represented in binary, and irrational numbers such as π and $\sqrt{2}$ also (by definition) cannot be exactly represented. Another category of number that fit within the floating point range but cannot be represented are numbers that “run out of bits” in the significand. Let’s look at an example. Let’s convert 16,777,217 to normalized 32-bit floating point format.

First, we calculate the binary representation, which is

10000000000000000000000000000001

Next, we put a binary point to the right of the most significant 1:

1.0000000000000000000000000000001

Next, we count off the 23-bits after the binary point, which will become our significand (in bold, below):

1.00000000000000000000000**000000000000000000000001**

This forms the significand, and it has lost information, so we cannot represent it exactly.

To finish off the conversion, we calculate the exponent, which is 24 (because we have to shift left by 24 places to get a number between 1 and 2), and after biasing, our exponent bits are $24 + 127 = 151$, or 10010111. The sign bit is 0 because the number is positive.

Below are the binary representations of 16,777,216, 16,777,217, and 16,777,218 – notice that the representations for 16,777,216 and 16,777,217 are identical:

16,777,216:	31	30	23	22	0
	0	10010111	000000000000000000000000		
16,777,217:	31	30	23	22	0
	0	10010111	000000000000000000000000		
16,777,218:	31	30	23	22	0
	0	10010111	000000000000000000000001		

The following program demonstrates that even if we set a float to the integer 16,777,217, it cannot be represented:

```
// file: unrepresentable.c
#include<stdio.h>
```

```
#include<stdlib.h>

int main(int argc, char **argv)
{
    float f1 = 16777216.0;
    float f2 = 16777217.0;
    float f3 = 16777218.0;

    printf("16,777,216: %f\n", f1);
    printf("16,777,217: %f\n", f2);
    printf("16,777,218: %f\n", f3);

    printf("f1 == f2? %s", f1 == f2 ? "true" : "false");
    return 0;
}
```

```
$ gcc -g -O0 -std=gnu99 unrepresentable.c -o unrepresentable
$ ./unrepresentable
16,777,216: 16777216.000000
16,777,217: 16777216.000000
16,777,218: 16777218.000000
f1 == f2? true
```

Comparing Floating Point Values

As the last example in the previous section demonstrated, comparing floating point values can be troublesome, and in fact it is unwise to compare floating point values directly without using the idea of an *epsilon*, which is a value that you determine is within the error bounds of your calculation. Let's look at an example:

```
// file: double_comparisons.c
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char **argv)
{
    double a = 0.1;
    double b = 0.2;
    double c = 0.3;
    double d = a + b;

    printf("a (0.1): %.30g\n", a);
    printf("b (0.2): %.30g\n", b);
    printf("c (0.3): %.30g\n", c);
    printf("d (0.3): %.30g\n", d);
    printf("c == d? %s\n", c == d ? "true" : "false");
    printf("c < d? %s\n", c < d ? "true" : "false");
    return 0;
}
```

The `%.30g` formatter means *print the floating point value to a precision of 30 decimal places*.

```
$ gcc -g -O0 -std=gnu99 double_comparisons.c -o double_comparisons
$ ./double_comparisons
a (0.1): 0.100000000000000005551115123126
b (0.2): 0.2000000000000000011102230246252
c (0.3): 0.299999999999999988897769753748
d (0.3): 0.3000000000000000044408920985006
c == d? false
c < d? true
```

What can we do to see if c is equal to d above, with some error bounds? One option is to choose a value (called the epsilon) that is known to us to be within the bounds. For example, let's presume that for our case above, we want to determine if our result is within 0.1% of the value we think it should be. We could re-write our program as follows:

```
// file: double_comparisons_epsilon.c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
```

```
const float EPSILON = 0.001; // 1 %
```

```
int main(int argc, char **argv)
{
```

```
    double a = 0.1;
    double b = 0.2;
    double c = 0.3;
    double d = a + b;
```

```
    printf("a (0.1): %.30g\n", a);
    printf("b (0.2): %.30g\n", b);
    printf("c (0.3): %.30g\n", c);
    printf("d (0.3): %.30g\n", d);
```

```
    float max_cd_percent = fmax(fabs(c), fabs(d)) * EPSILON;
    float diff_cd = fabs(c-d);
```

```
    printf("c == d? %s\n", (diff_cd <= max_cd_percent) ? "true" : "false");
    printf("c < d? %s\n", (diff_cd <= max_cd_percent) ? "false" : "true");
    return 0;
}
```

```
$ gcc -g -O0 -std=gnu99 double_comparisons_epsilon.c -o double_comparisons_epsilon
$ ./double_comparisons_epsilon
a (0.1): 0.100000000000000005551115123126
b (0.2): 0.2000000000000000011102230246252
c (0.3): 0.299999999999999988897769753748
d (0.3): 0.3000000000000000044408920985006
c == d? true
c < d? false
```

An outstanding and more detailed description of using epsilons to compare floats is here: <https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/>

floating point formats we have been discussing. Figure 30 shows the 32-bit FLT_MAX and FLT_MIN values (defined in float.h).

We can generate those values directly in a roundabout way, as demonstrated with the following program:

```
// file: float_extremities.c
#include<stdio.h>
#include<stdlib.h>
#include<float.h>

int main(int argc, char **argv)
{
    int flt_max_int_form = 0b01111111011111111111111111111111;
    int flt_min_int_form = 0b00000000100000000000000000000000;

    float flt_max = *((float *)&flt_max_int_form);
    float flt_min = *((float *)&flt_min_int_form);

    printf("flt_max: %g = %g\n", flt_max, FLT_MAX);
    printf("flt_min: %g = %g\n", flt_min, FLT_MIN);
    return 0;
}
```

```
$ gcc -g -O0 -std=gnu99 float_extremities.c -o float_extremities
$ gdb float_extremities
flt_max: 3.40282e+38 = 3.40282e+38
flt_min: 1.17549e-38 = 1.17549e-38
```

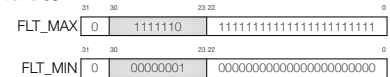
There are some properties of FLT_MAX and FLT_MIN that you should be aware of:

1. FLT_MAX is the largest normalized representable number, which is similar to the integer definition of INT_MAX. Infinity is a larger number, but not finite.
2. FLT_MIN is the smallest *in magnitude* normalized number. It is still a positive number, and in fact it is not the smallest representable number, which would be a denormalized number.

Final Thoughts

The IEEE Floating Point Standard was a carefully thought out way to get the most out of a discrete set of bits. It may not be simple, but it is a great study in good engineering design. Floating point numbers represent a very large range, in a limited number of bits. A 32-bit float can only hold a bit over 4 billion numbers but it has a range of -3.4×10^{38} to $+3.4 \times 10^{38}$. Not only is this literally an infinite number of reals that the format must try and represent, but that is a phenomenal range of numbers. The 64-bit double range is -1.7×10^{308} to $+1.7 \times 10^{308}$, which is enormous. Most numbers are,

Figure 30: 32-bit FLT_MAX and FLT_MIN values



therefore, only represented approximately in float format, including many integers.

When working with floating point numbers, we as programmers must be cognizant of the limitations and the nuances, but it is an excellent and versatile format for representing the real numbers.

Floating Point Practice Problems

x86-64 Assembly Language

EVERY MICROPROCESSOR HAS AN underlying *machine code* that it uses to perform calculations and to write and read data to and from memory. It is this machine code that high level languages like C get compiled into by gcc and other compilers and interpreters. The machine code is in binary, and the binary directly maps to a textual and human-readable representation, called *assembly language*. One assembly language instruction generally maps to one (or possibly two) machine code instructions, based on simple rules. In this chapter, we will investigate the assembly language used in the Intel / AMD *x86-64* series of microprocessors, which are the microprocessors used in the Myth machines.⁹⁶ Other microprocessor families, like the one that is in your phone, use a different assembly language *instruction set*, and unlike higher level languages, assembly language programs are not compatible with one another, and do not enable portable programs. However, once you learn assembly language for one family of processors, learning a different assembly language is a bit like learning a dialect of a spoken language, and the skills are largely transferable.

Because assembly language is low level, everything in the language is explicit, and assembly language programs do not have features you might be used to from C, such as loops and variables. This means that assembly language tends to be much more verbose than an equivalent high level language, and it does make reading it more involved. That said, there is a simplicity to assembly language that enables you to get a feel for the actual way a computer is processing your data that is illuminating.

One goal of this chapter is for you to understand how a section of assembly language code translates into C code, and we will build up to those constructs as we go along. We don't expect you to necessarily translate in the other direction (from C to assembly language), nor do we expect you to write much assembly language code. Programmers used to write in assembly language when they wanted to tune their code for maximum efficiency, but compiler optimization technology has advanced such that most compilers can turn C code into extremely efficient assembly code and it is usually not worth the effort to try and do better by hand.

The Intel-based Myth computers we use are direct descendants of Intel's 16-bit, 1978 processor with the name 8086.⁹⁷ Intel has

⁹⁶ The comprehensive set of x86-64 manuals can be found here: <https://software.intel.com/en-us/articles/intel-sdm>. We are going to be using AT&T syntax; see the following for information on the differences between AT&T and Intel format: <http://www.imada.sdu.dk/Courses/DM18/Litteratur/IntelnATT.htm>

⁹⁷ See the excellent [Wikipedia article on the history of the x86 family of processors](#)

taken a strict backwards-compatibility approach to new processors, and their 32- and 64-bit processors have built upon the original 8086 Assembly code. These days, when we learn x86 assembly code, we have to keep this history in mind. Naming of *registers*, for example, has historical roots, and there is some terminology based on the original processor that has remained to this day.

The Intel Instruction Set Architecture and a Sample Program

The machine code for a processor is based on the *instruction set architecture (ISA)* for a processor. The ISA defines the behavior and layout of a system, and the behavior is defined as if instructions are run one after the other. Memory appears as a very large byte-addressable array.⁹⁸ There are a number of parts of the underlying machine that we have not discussed yet (in some sense, they have been hidden from us by C), but that will be necessary to understand how the machine works. Some of these ideas are listed below:

- Program values are stored in a *register file*, and there are sixteen named locations that store 64-bit values, plus other registers that hold information you cannot directly access. Registers are the fastest memory on your computer. Registers can hold addresses, or integer data. Some registers are used to keep track of your program's state, and others hold temporary data. Registers are used for arithmetic, local variables, and return values for functions, and your entire processor core shares the small number of registers.
- The computer has a register *program counter* called either `%rip` or `%pc`⁹⁹ which indicates the address of the next machine code instruction.
- The condition code registers hold status information about the most recently executed arithmetic or logical instruction. These are used to control program flow – e.g., if the result of an addition is negative, exit a loop.
- There are registers called *vector registers* which are specifically designed for integer and floating point calculations.

Unlike in C, assembly language and the ISA does not differentiate between signed and unsigned integers, between different types of pointers, or even between pointers and integers. As we will see, there is a set of data widths that can be referred to when moving data between registers and memory, and these do map onto specific C data types, depending on the machine¹⁰⁰

A single machine instruction performs only a very elementary operation. For example, there is an instruction to add two numbers in registers, there is an instruction that transfers data between a register and memory, and there is an instruction that conditionally

⁹⁸ As you may guess, C is a high level language that is based on an underlying machine architecture that is byte-addressable, and operates linearly. Unlike some other languages, C code translates in a relatively straightforward manner into assembly code.

⁹⁹ *r* stands for *register*, *ip* stands for *instruction pointer*, and *pc* stands for *program counter*.

¹⁰⁰ We will discuss the mapping for 64-bit machines, as you might expect.

branches to a new instruction address. As you shall see, assembly instructions are simple, and often, one C statement generates multiple assembly code instructions.

Let's start by looking at some assembly code. We will compile with the `-S` flag to tell gcc to create the assembly code output of our program. We will also use the `-Og` optimization flag to produce lightly optimized code that will be easy to debug. The gcc output has more detail than we really care about at this point, and we have annotated the interesting portions. We will show a different view of it below the following code.

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

int main()
{
    int i = 1;
    printf("Hello, World %d!\n", i);
    return 0;
}
```

```
$ gcc -S -Og -std=gnu99 -Wall simpleasm.c
```

```
$ cat simpleasm.s
```

```
.file      "simpleasm.c"
.section   .rodata.str1.1,"aMS",@progbits,1
.LC0:
.string   "Hello, World %d!\n"
.text
.globl    main
.type     main, @function
main:
.LFB39:
.cfi_startproc
subq     $8, %rsp
.cfi_def_cfa_offset 16
movl     $1, %edx
movl     $.LC0, %esi
movl     $1, %edi
movl     $0, %eax
call     __printf_chk
movl     $0, %eax
addq     $8, %rsp
.cfi_def_cfa_offset 8
ret
.cfi_endproc
.LFE39:
.size     main, .-main
.ident    "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.5) 5.4.0 20160609"
.section   .note.GNU-stack,"",@progbits
```

The string is placed into memory, and there is a label called `.LC0`

The main function begins here.

The `movl` instruction moves information between registers, and between registers and memory. This is actually setting up the arguments for the `printf` function.

The call to the `printf` function.

If we compile the code to a binary program as we normally do (using the `-Og` optimization flag), we can look at the assembly in `gdb` as well:

```
$ gcc -g -Og -std=gnu99 -Wall simpleasm.c -o simpleasm
$ gdb simpleasm
The target architecture is assumed to be i386:x86-64
Reading symbols from simpleasm...done.
(gdb) disas main
Dump of assembler code for function main:
    0x000000000400546 <+0>:    sub     $0x8,%rsp
    0x00000000040054a <+4>:    mov     $0x1,%edx
    0x00000000040054f <+9>:    mov     $0x4005f4,%esi
    0x000000000400554 <+14>:   mov     $0x1,%edi
    0x000000000400559 <+19>:   mov     $0x0,%eax
    0x00000000040055e <+24>:   callq   0x400430 <__printf_chk@plt>
    0x000000000400563 <+29>:   mov     $0x0,%eax
    0x000000000400568 <+34>:   add     $0x8,%rsp
    0x00000000040056c <+38>:   retq
End of assembler dump.
```

By the end of this chapter, you will have no trouble understanding the code above, although at this point it may look confusing! For now, notice how many `mov` instructions there are – these are the instructions that move data between registers and memory. Also notice on line `<+9>` that there is a value of `0x4005f4` – that happens to be the location where the `Hello, World %d!\n` is kept, which we can also see in `gdb`:

```
(gdb) p (char *) (0x4005f4)
$2 = 0x4005f4 "Hello, World %d!\n"
```

A couple of other things to point out about the assembly code above:

- A single C statement can lead to multiple assembly instructions.
- Setting up function calls takes some work.
- The `%rsp` register refers to the stack, so something is happening on the stack.
- Two instructions before the `ret` (*return*) instruction, a value is placed into the `%eax` register, which will become the return value for the `main` function. In this case, we are returning 0.

Data Formats

Intel assembly language has four basic data width formats, to denote byte widths of 1 (*b*), 2 (*w*), 4 (*l*), and 8 (*q*). Because of its 16-bit origins, Intel uses *word* to mean 16-bits, or 2 bytes. Thirty-two bit words are referred to *double words*, and 64-bit words are referred to as *quad words*. Table 8 shows how the assembly code suffix aligns with common C data types.

Table 8: Intel Data Types

C type	Suf.	B	Intel data type
<code>char</code>	<code>b</code>	1	Byte
<code>short</code>	<code>w</code>	2	Word
<code>int</code>	<code>l</code>	4	Double word
<code>long</code>	<code>q</code>	8	Quad word
<code>char *</code>	<code>q</code>	8	Quad word
<code>float</code>	<code>s</code>	4	Sing. precision
<code>double</code>	<code>l</code>	8	Dbl. precision

x86-64 Registers

x86 CPUs have 16 general purpose registers, which store 64-bit values. Registers store integer data and pointers. Register names begin with *r*, but the naming is historical: the original 16-bit registers were *%ax*, *%bx*, *%cx*, *%dx*, *%si*, *%di*, *%bp*, and *%sp*. Each register had a purpose, and were named as such.

When 32-bit x86 arrived, the register names expanded to 32-bits each, and changed to *%eax*, *%ebx*, etc. When x86-64 arrived, the registers were again renamed to *%rax*, *%rbx*, etc., and expanded to 64-bits. Additionally, eight more registers were added, *%r8* - *%r15*.

Figure 31 shows the integer registers, which are *nested* such that (for example), *%rax* is the full 64-bit register, and *%eax* is the low 32 bits of the register, *%ax* is the low 16 bits of the register, and *%al* is the low 8 bits of the register.

Operand Forms and the *mov* and *lea* Instructions

Most assembly language instructions take one or two *operands* to perform an operation on, and there are multiple operand forms. These forms can be broken down into a few general examples.

\$IMM : An *immediate* value is a constant, which is preceded by a dollar sign. Examples: *\$1*, *\$0x1A*, *\$-42*.

%r_a : A register. Examples: *%rax*, *%edx*, *%r8d*.

Imm(*r_b*, *r_i*, *s*) The general form of a *scaled indexed* operand. The value is computed with the following linear formula:

$$\text{Imm} + r_b + s * r_i$$

The *s* stands for the *scaling factor*, and it is limited to either 1, 2, 4, or 8. Each part may be left out, except that if *r_i* is left out, then *s* must, as well. Examples:

$$\begin{aligned} 4(\text{\%rax}, \text{\%rdx}, 2) &= 4 + \text{\%rax} + 2 * \text{\%rdx} \\ (\text{\%rdx}, \text{\%rax}, 4) &= \text{\%rdx} + 4 * \text{\%rax} \\ 5(\text{\%rax}) &= 5 + \text{\%rax} \\ -7(, \text{\%rdx}, 8) &= -7 + 8 * \text{\%rdx} \\ (\text{\%rax}, \text{\%rdx}) &= \text{\%rax} + \text{\%rdx} \end{aligned}$$

The most common x86-64 assembly instruction is the *mov* instruction¹⁰¹, which copies *immediate* data (e.g., constant numbers) into a register or memory, copies values between two registers, or copies values between a register and memory (in either direction). In x86-64 assembly, it is not legal to copy directly from one memory location to another memory location without first copying the data into a register. The general form of the *mov* instruction is as follows:

```
movx src, dest
```

where *x* can be replaced by *b*, *w*, *l*, or *q* to denote the width of the value being moved. The register source and/or destination must

¹⁰¹ See an interesting pie chart about it at https://www.strchr.com/x86_machine_code_statistics

63	31	15	7	
%rax	%eax	%ax	%al	return value
%rbx	%ebx	%bx	%bl	caller owned
%rcx	%ecx	%cx	%cl	4th argument
%rdx	%edx	%dx	%dl	3rd argument
%rsi	%esi	%si	%sil	2nd argument
%rdi	%edi	%di	%dil	1st argument
%rbp	%ebp	%bp	%bpl	caller owned
%rsp	%esp	%sp	%spl	stack pointer
%r8	%r8d	%r8w	%r8b	5th argument
%r9	%r9d	%r9w	%r9b	6th argument
%r10	%r10d	%r10w	%r10b	callee owned
%r11	%r11d	%r11w	%r11b	callee owned
%r12	%r12d	%r12w	%r12b	caller owned
%r13	%r13d	%r13w	%r13b	caller owned
%r14	%r14d	%r14w	%r14b	caller owned
%r15	%r15d	%r15w	%r15b	caller owned

Figure 31: The sixteen x86-64 integer registers

also be the proper width for the instruction (see the examples below), and in most cases, only the bits in the specified width are modified in a 64-bit register. For example, if you are moving 1 byte (with the *b* specifier), only the lowest byte of a destination register will be affected, and the rest will remain the same. There is one exception: the *movl* instruction moves 32-bits into the low bits of a register, and *clears the upper 32-bits* with zeros.

If the operand is in the form **Imm(*r_b*, *r_i*, *s*)**, then the location is a *memory address*, and the copy will be from the location calculated with the linear equation above.

Examples:

movl \$0x20,%eax	copies the immediate value 0x20 into the low 32 bits of %rax, and clears the upper 32 bits to zero.
movq 4(%rax,%rdx,2),%rax	copies the 64-bit (8 byte) value at memory location $4 + \%rax + 2 * \%rdx$ into register %rax.
movw %bx, (%rdx,%rax,4)	copies two bytes from %bx into the memory location $\%rdx + 4 * \%rax$
movl %eax,%edx	copies the low 32-bits (four bytes) of %eax into the low 32-bits of %rdx, and clears the upper 32 bits to zero.
movb (,%rdx,8),%al	copies one byte from $8 * \%rdx$ to the lowest 8 bits of %rax.
movq %rcx, (%rax, %rdx)	copies the 64-bit value in %rcx into the memory location at $\%rax + \%rdx$.

There is another mov instruction, *movabsq \$IMM, Reg*, which is used to put a 64-bit immediate value into a register. This instruction is needed because in the regular mov instruction, the maximum immediate value is only 32 bits. Example:

movabsq \$0x0011223344556677, %rax

There are two additional mov instructions, which are used to copy a smaller source into a larger destination (which must be a register): *movz* and *movs*. These instructions perform either a zero-fill of the remaining bytes, or a *sign-extension* of the remaining bytes. There are six ways to move a 1- or 2-byte operand into a 2-, 4- or 8-byte operand. There is not an explicit instruction to zero-extend a 4-byte source into an 8-byte destination, because the *movl* instruction already accomplishes the zero-fill operation. Note that the instructions have the size designations embedded in each instruction:

movzbw and movsbw : move zero- or sign-extended byte to word.

movzbl and movsbl : move zero- or sign-extended byte to double word.

movzbq and movsbq : move zero- or sign-extended byte to quad word.

movzwl and movswl : move zero- or sign-extended word to double word.

`movzqw` *and* `movswq` : move zero- or sign-extended word to quad word.

`movslq` : move sign-extended double word to quad word.

An important instruction that is related to `movq` is the `leaq` instruction. The difference is that instead of actually reading from memory, it simply puts the result of the calculation of the operand into the destination. You can think of it as the *address of* (`&`) operator in C:

```
leaq src, dest
```

where the destination is a register. For example:

```
leaq (%rax,%rdx,4), %rax           copies the value of %rax + 4 * %rdx into %rax
```

Often, you will see the `leaq` instruction used to perform arithmetic on values in registers, and the registers do not have to hold addresses.¹⁰²

¹⁰² Keep this in mind! You may see the `leaq` instruction in what seem like odd sections of code (where there are no memory references), but in those cases it is being used for arithmetic.

The Linux Address Space, and the Stack and the Heap

At this point, we will take a brief detour into the land of the *stack* and the *heap* in a program. We have discussed stack variables (local variables and local arrays), and heap allocation (using `malloc` and `free`), but as we get into assembly language, we need to discuss how the stack and heap are layed out in memory.

Figure 32 shows the layout of the Linux address space.

The following is a brief introduction to each section of memory:

stack The stack grows *downward* in memory, and every program is, by default, given an 8MB stack partition.¹⁰³ The stack holds local variables, arrays, and parameters¹⁰⁴, and it is fast to allocate variable space on the stack, because a program already “owns” the memory when it begins. Addresses on the stack are in the `0x7ffffxxxxxx` range.

heap The heap grows *upward* in memory, and the heap holds values allocated with `malloc` and `calloc`. The heap is typically much larger (measured in GB instead of MB), and is managed by the operating system. It is slower to request heap memory because the operating system is directly involved. The heap grows as is necessary for a program (i.e., there isn’t a fixed limit). Heap memory begins in the range `0x602010`, and we will see many addresses in this range as we continue the chapter.

shared library text and data The C libraries (and other shared libraries) are placed into a location accessible by all programs. Each program gets its own data segment for shared libraries, but the code is completely shared.

global data Any global or static variables get stored in global memory.

text The text segment (in the `0x400000` range is where a program’s code resides. You will see these addresses when you analyze code in more detail.

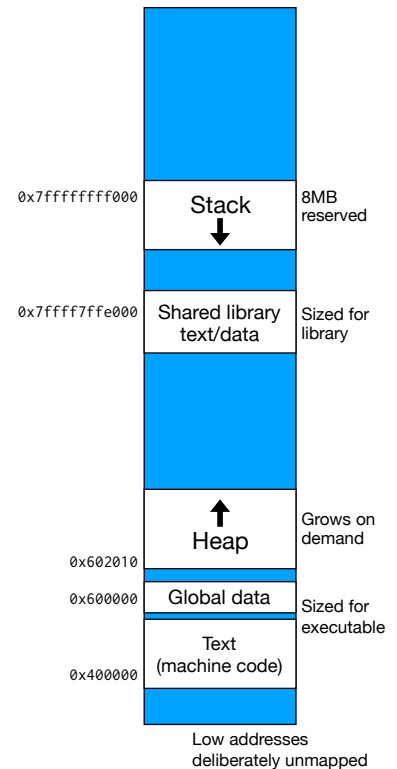
low addresses The low addresses are used by the operating system and hardware.

Pushing and Popping from the Stack

As we saw in the previous section, the stack is an important part of our program. Assembly language has two built-in instructions, called `push` and `pop` to place values onto, and to take values off of the stack. Just like the stack abstract data type, the instructions have a first-in-first-out discipline.

We saw above that the stack grows downward in memory. The `push` instruction places a value from a register onto the stack at location of the stack pointer, `%rsp`. Then it *decrements* `%rsp` by the amount of bytes that it pushed. The `pop` instruction accomplishes

Figure 32: The Linux address space (not to scale)



¹⁰³ But the hardware memory manager only doles out the memory as needed.

¹⁰⁴ It actually infrequently holds the parameters, as we will see later in the chapter

the reverse process: it saves the value at the location of `%rsp` into a register, and then it increments `%rsp` by the amount of bytes that it popped. Figure 33 demonstrates a push followed by a pop.

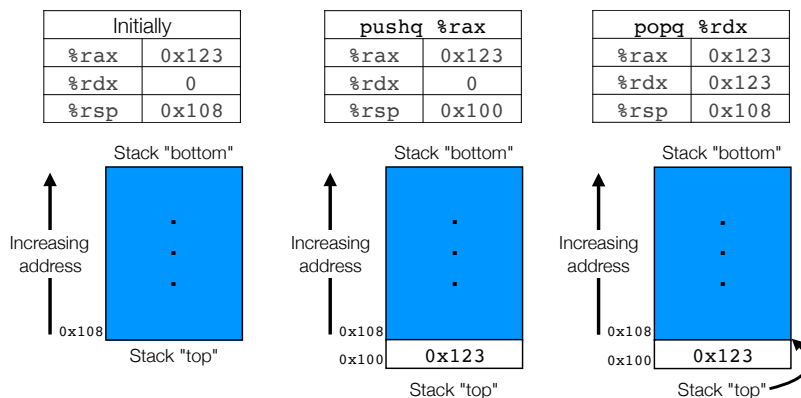


Figure 33: A push followed by a pop. Notice that after the push `%rax`, the stack pointer has been decremented by 8, and after the pop `%rdx`, the stack pointer has been incremented by 8. Also note that the value on the stack after the pop has not been cleared.

Arithmetic in Assembly

In assembly languages, arithmetic forms a substantial set of the instructions. The *unary* instructions act on a single operand (register or memory), and the *binary* operations act on two operands (one of which can be memory). There are also some special instructions (e.g., `mul` and `div`) that act on one operand but affect other operands, as well. The *shift* instructions operate on one operand, but take a second argument to determine how much to shift the operand by.

The unary instructions we care about are defined as follows:

inc *dst* : Increments the register or memory location by 1. Example: `incq %rax`

dec *dst* : Decrements the register or memory location by 1. Example: `decq 4(%rdx)`

neg *dst* : Performs a two's complement conversion on the operand. Example: `negl %eax`

not *dst* : Performs a bitwise complement on the operand. Example: `notb %ax`

The binary instructions are defined as follows¹⁰⁵:

add *src, dst* : Adds *src* to *dst* and puts the result into *dst*. Example: `addq %rax, %rdx`

sub *src, dst* : Subtracts *src* from *dst* and puts the result into *dst*.¹⁰⁶ Example: `subl %eax, %edx`

imul *src, dst* : Performs a signed multiply on *src* and *dst*, and puts the result into *dst*. There is a possibility of losing information, as

¹⁰⁵ Note the lack of a division instruction; we will cover division as a special case below

¹⁰⁶ This syntax can be tricky. `addq %rax, %rdx` is read "subtract `%rax` from `%rdx`."

the result of two n -bit multiplications can produce a $2n$ -bit product.¹⁰⁷ Can be used on unsigned operands as well, because the lower half of the product is the same. Example: `imulb %a1,%d1`

¹⁰⁷ See below for a different form of `imul` that does not lose information.

xor src,dst : Performs the *exclusive-OR* operation on *src* and *dst* and puts the result into *dst*. Example: `addq %rax,%rdx`

or src,dst : Performs the *OR* operation on *src* and *dst* and puts the result into *dst*. Example: `andl (%rax,%rdx,2),%edx`

and src,dst : Performs the *AND* operation on *src* and *dst* and puts the result into *dst*. Example: `andw %ax,%dx`

The shift instructions are defined as follows. Note that the first operand can be either an immediate value or `%cl` (and only `%cl`):

sal k,dst or shl src,dst : Performs a left shift of k bits on *dst* and puts the result into *dst*.¹⁰⁸ Example: `shlq $4,%rdx`

sar k,dst : Performs an arithmetic right shift of k bits on *dst* and puts the result into *dst*. Example: `sarb $2,%d1`

shr k,dst : Performs an logical right shift of k bits on *dst* and puts the result into *dst*. Example: `shrq %cl,8(%rax)`

¹⁰⁸ The reason there are two instructions that do the exact same thing is simply to be analogous to the right shift instructions, which have a logical and arithmetic form.

Because the multiplication of two n -bit numbers can produce a $2n$ -bit product, we have a different form of `imul` and an additional `mul` instruction to perform the multiplication of two 64-bit operands without losing information. Both instructions take a single 64-bit operand, but also implicitly use the `%rax` register as one of the multiplicands. The 128-bit result is *always* put into the combination of `%rdx:%rax`, with the upper 64-bits placed into `%rdx` and the lower 64-bits placed into `%rax`. The following example demonstrates the idea:

```
// file: octmult.c
#include<stdio.h>
#include<stdlib.h>

// multiply64
// x : the first multiplicand
// y : the second multiplicand
// result : a pointer to an array of 2 longs
//          that will be in little-endian format
void multiply64(unsigned long x, unsigned long y, unsigned long *result);

int main(int argc, char **argv)
{
    unsigned long x = strtol(argv[1], NULL, 0);
    unsigned long y = strtol(argv[2], NULL, 0);

    unsigned long result[] = {0,0};
```

```

    multiply64(x,y,result);

    printf("0x%016lx%016lx\n",result[1],result[0]);
    return 0;
}

# File: multiply64.s
# -----
# Demonstrates the 1-operand imul / mul
# instructions with two signed 64-bit multiplicands
# and the result placed into %rdx:%rax

.section .text

    .type    multiply64, @function
    .globl multiply64

# %rdi will hold first multiplicand
# %rsi will hold the second multiplicand
# %rdx will hold a pointer to an array
# for the little endian 128-bit result
multiply64:
    # store %rdx in %r8 because %rdx will
    # hold the upper bits of the product
    movq %rdx,%r8

    # copy the first multiplicand into %rax
    movq %rdi,%rax

    # perform the multiplication
    mulq %rsi

    # result is now in %rdx:%rax
    # now we copy the result into
    # the array
    movq %rax,(%r8)
    movq %rdx,8(%r8)
    ret

$ gcc -g -Og -std=gnu99 -Wall octmult.c -x assembler multiply64.s -o octmult
$ ./octmult 0x3000000000000000 0x2000000000000000
x06000000000000000000000000000000

```

The division instruction seemed to be missing from the list of arithmetic instructions above. However, division is another special case. There are two division instructions (`idivq` and `divq`), and a third helper instruction (`ctqo`) that is used as well (described below). Instead of using a single 64-bit dividend, the `div` instructions take a 128-bit dividend, located in `%rdx:%rax`, and then divide the single operand into the 128-bit dividend. The resulting quotient is

placed into `%rax`, and the *remainder* of the integer division is placed into `%rdx`. If the quotient ends up as too large for the destination register, `%rax`), the runtime throws an exception.¹⁰⁹ The divide instructions are explained below:

¹⁰⁹ One way to check for this potential case is to ensure that `%rdx` is less than the divisor operand.

`idivq src` : Divides the signed 128-bit combination of `%rdx:%rax` by the signed `src`, and puts the quotient into `%rax`, and puts the remainder into `%rdx`. Example: `idivq %rdi`

`divq src` : Divides the unsigned 128-bit combination of `%rax:%rdx` by the unsigned `src`, and puts the quotient into `%rax`, and puts the remainder into `%rdx`. Example: `idivq %rdi`

`cqto` : Sign-extends `%rax` into `%rdx` to produced a correctly-signed 128-bit `%rdx:%rax` combination.

The following shows an example of the `idiv` instruction:

```
// file: div_example.c
#include<stdio.h>
#include<stdlib.h>

long divide_with_remainder(long x, long y, long *rem)
{
    *rem = x % y;
    return x / y;
}

int main(int argc, char **argv)
{
    long x = strtol(argv[1], NULL, 0);
    long y = strtol(argv[2], NULL, 0);
    long remainder;
    long quotient = divide_with_remainder(x, y, &remainder);
    printf("%ld / %ld = %ld remainder %ld\n", x, y, quotient, remainder);
    return 0;
}
```

```
$ gcc -Og -std=gnu99 -Wall div_example.c
```

```
$ ./div_example 10000 3
```

```
10000 / 3 = 3333 remainder 1
```

```
$ gdb div_example
```

```
The target architecture is assumed to be i386:x86-64
```

```
Reading symbols from div_example...done.
```

```
(gdb) disas divide_with_remainder
```

```
Dump of assembler code for function divide_with_remainder:
```

```
0x0000000000000760 <+0>:    mov    %rdi,%rax
0x0000000000000763 <+3>:    mov    %rdx,%rcx
0x0000000000000766 <+6>:    cqto
0x0000000000000768 <+8>:    idiv   %rsi
0x000000000000076b <+11>:   mov    %rdx,(%rcx)
```

```
0x0000000000000076e <+14>:    retq
End of assembler dump.
```

Control

So far, we have only been discussing "straight-line" code, where one instruction happens directly after the previous instruction. However, it is often necessary to perform one instruction or another instruction based on the logic in our programs, and assembly code gives us tools to do this. We can alter the flow of code using a *jump* instruction, which indicates that the next instruction will be somewhere else in the program (this is called a *branch*), or we can modify the instruction path by calling and returning from functions. We will start by discussing *condition codes* that are set when we do arithmetic and some other operations, and then we will talk about jump instructions to change control flow.

Besides the registers we have already discussed, the x86-64 processors have a separate set of single-bit *condition code* registers that describe attributes of the result of recent instructions. We can use the registers (by testing them) to perform branches in our code.

The condition code registers we will use in class are:

- CF** : Carry flag. The most recent instruction generated a carry out of the most significant bit. The carry flag is used to detect overflow for unsigned operations.
- ZF** : Zero flag. The most recent instruction yielded zero.
- SF** : Sign flag. The most instruction yielded a negative value.
- OF** : Overflow flag. The most recent instruction caused a two's-complement overflow, either negative or positive.

For example, if `%rax` contains 5 and `%rdx` contains -5 , the instruction `add %rax,%rdx` will set the ZF flag, because the result of the addition was 0. If, instead, `%rdx` holds -20 , then the same instruction would set the SF flag, because the result of the addition is negative.

The `leaq` and `mov` instructions do not set any condition codes, but the arithmetic instructions do set them, as do the logical instructions¹¹⁰. For shift instructions, the carry flag is set to the last bit shifted out, while the overflow flag is set to zero. For historical reasons, the `inc` and `dec` instructions do not set the carry flag, but do set the overflow and zero flags.

There are two types of instructions, `cmp` and `test` that we can use to set condition codes without altering any other registers:

`cmpx s1,s2`: Compares two numbers by subtracting `s2-s1`.¹¹¹ The `x` is replaced by a bit width, `b`, `w`, `l`, or `q`. Example:

```
movl $5,%eax
movl $1,%edx
```

¹¹⁰ The logical instructions, e.g., `xor` set the carry and overflow flags to 0.

¹¹¹ Be careful to get the order correct! Think, "subtract `s1` from `s2`."


```

cmpl %eax,%edx # sets the sign flag because 1-5 is less than 0
               # also sets the carry flag, because adding
               # -1 to 5 produces a carry-out

```

testx s1,s2: Compares two numbers by performing a bitwise AND on the operands. The *x* is replaced by a bit width, b, w, l, or q. The result from the AND sets the flags; for instance, a result that has its most significant bit set would set the Sign Flag, and a number that is all zeros would set the zero flag. We often use the test instruction with the same register as both operands in order to determine if the register's value is negative, positive, or zero. Example:

```

movl $0,%eax
test %eax,%eax # sets the zero flag because 0 AND 0 == 0

```

There are three common ways to use the condition codes to affect assembly language behavior. The set instruction will set a single byte based on condition codes. The j (jump) instruction sets the instruction pointer to a different instruction than the one that is next in the program.¹¹² The cmov (conditional move) instruction performs a regular mov instruction based on the condition codes.

¹¹² There is also an *unconditional jump* instruction, jmp, which changes the instruction pointer unconditionally.

All three conditional instructions use the same suffixes to determine which flags will affect their behavior. Table 9 shows the suffix, english description, and the condition code flags. The suffixes are preceded by either set (e.g., setz), j (e.g., jz, or cmov (e.g., cmovz):

Suffix	Behavior	Flags used
e / z	set, jump, move if equal/zero	ZF
ne / nz	set, jump, move if not equal/nonzero	~ZF
s	set, jump, move if negative	SF
ns	set, jump, move if nonnegative	~SF
g / nle	set, jump, move if greater (signed)	~(SF ^ OF) & ~ZF
ge / nl	set, jump, move if greater or equal (signed)	~(SF ^ OF)
l / nge	set, jump, move if less (signed)	SF ^ OF
le / ng	set, jump, move if less or equal	(SF ^ OF) ZF
a / nbe	set, jump, move if above (unsigned)	~CF & ~ZF
ae / nb	set, jump, move if above or equal (unsigned)	~CF
b / nae	set, jump, move if below (unsigned)	CF
be / na	set, jump, move if below or equal (unsigned)	CF ZF

Table 9: The common conditional suffixes for set / j / cmov

For completeness, we have put all of the flags used in the table above. However, most of the time it is far simpler to think about the behavior as it relates to the suffix, and to think about two instructions at a time. For example:

```

cmpq $0x5,%rax
jg 0x821 <main+193>

```

To read this, say, “jump if %rax is greater than 5.” Here is another example:

```
    cmpw %dx,%ax
    cmovne %rcx,%r8
```

To read this, say, “move %rcx to %r8 if %ax is not equal to %dx.”

The `jmp` instruction is *unconditional*, meaning that it always performs the requested jump to a different instruction:

`jmp Label` : Directly jump to the *Label*, which is usually replaced in C generated code with an offset (e.g., a certain number of bytes away from the instruction pointer). Example: `jmp .L0`

`jmp *Operand` : Jump indirectly, based on the value in memory pointed to by the operand. This is used when we have a function pointer. Example: `jmp *(%rdx)`

The following program demonstrates how an unconditional and a conditional jump work in a loop in C:

```
// file: loopjumps.c
#include<stdio.h>
#include<stdlib.h>

void loop()
{
    int i = 0;
    while (i < 100) {
        printf("i:%d\n",i);
        i++;
    }
}

int main(int argc, char **argv)
{
    loop();
    return 0;
}
```

```
$ gcc -Og -std=gnu99 -Wall loopjumps.c -o loopjumps
```

```
$ gdb loopjumps
```

The target architecture is assumed to be i386:x86-64

Reading symbols from loopjumps...done.

```
(gdb) disas loop
```

Dump of assembler code for function loop:

```
0x000000000400546 <+0>:    push    %rbx
0x000000000400547 <+1>:    mov     $0x0,%ebx
0x00000000040054c <+6>:    jmp     0x400567 <loop+33>
0x00000000040054e <+8>:    mov     %ebx,%edx
0x000000000400550 <+10>:   mov     $0x400614,%esi
0x000000000400555 <+15>:   mov     $0x1,%edi
0x00000000040055a <+20>:   mov     $0x0,%eax
0x00000000040055f <+25>:   callq   0x400430 <__printf_chk@plt>
0x000000000400564 <+30>:   add     $0x1,%ebx
```

```

0x0000000000400567 <+33>:    cmp     $0x63,%ebx
0x000000000040056a <+36>:    jle     0x40054e <loop+8>
0x000000000040056c <+38>:    pop     %rbx
0x000000000040056d <+39>:    retq

```

End of assembler dump.

Let's take a closer look at the assembly code above. We will refer to the code by the values in angle brackets (<>), which are actually counts of how many bytes the instruction is away from the beginning of the function.¹¹³

The function starts on line <+0> by pushing the register %rbx onto the stack, to save it.¹¹⁴

The value of %rbx, which is the loop counter, i, is set to 0 on line <+1>.

The unconditional jmp on line <+6> sets the instruction pointer to instruction <+33> (address 0x400567), where the initial while loop check for i < 100 is completed.

The jle instruction on line <+36> jumps back to line <+8> if %rbx is less than or equal to 99 (0x63), at which point the function enters into the while loop body.

Lines <+8> through <+25> set up and call the printf function.

On line <+30>, the counter is incremented, and the code falls through to the loop counter check again. When the counter has reached a value of 99, the jle instruction does *not* branch, and the code continues with line <+38>, restoring %rbx (line <+38> and returning from the function (line <+39>.

More on Conditional Moves

The conditional moves we discussed in the previous section have an interesting use, especially in today's computers. Some times, a program that has a conditional statement that normally requires a conditional jump can be re-written to use a conditional move, instead. Although the resulting change may take extra lines of assembly code, the removal of a branch can actually speed up the program. Modern processors perform what is called *branch prediction* when they read a conditional branch; i.e., they *guess* whether the program will branch, or not. The reason this is necessary is because modern processors have the ability to perform many¹¹⁵ of instructions in parallel, and by guessing which way a conditional branch will go can keep the processor working on instructions before it actually gets to a point where it knows whether the branch will happen or not. Branch prediction technology has an extremely high rate of correct behavior (over 90% correct), but when it mispredicts, the processor has to *stall* and throw away the result of instructions that it thought would get executed.

Therefore, if we can remove a branch from our programs, this can often speed up the program, at the cost of a few extra instructions.

¹¹³ x86-64 instructions are *variable length*, and generally take between 1 and 5 bytes. The address of each instruction is, therefore, not exactly the same number of bytes from the previous instruction.

¹¹⁴ We will discuss why this is the case soon!

¹¹⁵ sometimes up to hundreds

Take the following two C functions that accomplish the same thing:

```
// file: conditionalmov_prog.c
#include<stdio.h>
#include<stdlib.h>

int maxdifference_A(int a, int b)
{
    int diffab;
    if (a - b >= b - a) {
        diffab = a - b;
    } else {
        diffab = b - a;
    }
    return diffab;
}

int maxdifference_B(int a, int b)
{
    int diffab = a - b;
    int diffba = b - a;
    int maxab = diffab < diffba;
    if (maxab) diffab = diffba;

    return diffab;
}

int main(int argc, char **argv)
{
    int a = atoi(argv[1]);
    int b = atoi(argv[2]);
    printf("max_diff_A: %d\n", maxdifference_A(a,b));
    printf("max_diff_B: %d\n", maxdifference_B(a,b));

    return 0;
}

$ gcc -O3 -std=gnu99 -Wall conditionalmov.c -o conditionalmov
$ ./conditionalmov 4 5
max_diff_A: 1
max_diff_B: 1
```

In `maxdifference_A`, there is an explicit `if/else` statement, and it would seem that a conditional jump is warranted. In `maxdifference_B`, there is no `else` statement to go along with the `if` statement, because both calculations, `diffab` and `diffba` are actually computed. Now take a look at the output the compiler produces when the optimization level is set to a high level of optimization, `-O3`:

```
$ gdb conditionalmov
```

The target architecture is assumed to be i386:x86-64

Reading symbols from conditionalmov...done.

(gdb) **disas maxdifference_A**

Dump of assembler code for function maxdifference_A:

```
0x000000000400620 <+0>:    mov    %edi,%eax
0x000000000400622 <+2>:    sub    %esi,%eax
0x000000000400624 <+4>:    sub    %edi,%esi
0x000000000400626 <+6>:    cmp    %esi,%eax
0x000000000400628 <+8>:    cmovl  %esi,%eax
0x00000000040062b <+11>:   retq
```

End of assembler dump.

(gdb) **disas maxdifference_B**

Dump of assembler code for function maxdifference_B:

```
0x000000000400630 <+0>:    mov    %edi,%eax
0x000000000400632 <+2>:    sub    %esi,%eax
0x000000000400634 <+4>:    sub    %edi,%esi
0x000000000400636 <+6>:    cmp    %esi,%eax
0x000000000400638 <+8>:    cmovl  %esi,%eax
0x00000000040063b <+11>:   retq
```

End of assembler dump.

Notice that the compiler has discovered that both of the loops can be re-written using the `cmovl` instruction, without needing to use any conditional jumps.

From C to Assembly: while Loops

In this and the following sections, we are going to analyze how the compiler creates code for various C programs. Note that these are unoptimized compilations, and in many cases the compiler can make faster and more efficient code (as we saw in the last section) when it is allowed to optimize. However, for instructional purposes, we will look at the unoptimized code.

The first code transformation we will look at is the `while` loop. There are two forms that we will look at, and depending on what the compiler can deduce from your code, it will perform one of the following transformations.

In C, the general form of the `while` loop is as follows:

```
while (test_expr)
    body\_statement
```

Gcc often translates a `while` loop into assembly that looks like this:

```
    jmp test // unconditional jump
loop:
    body_statement instructions
test:
    cmp ... // comparison instruction
```

```
j.. loop // conditional jump based on comparison
        // can think of as "if test, goto loop"
```

The following program demonstrates a real example (with comments manually added to the gdb output):

```
// file: whileloop1.c
#include<stdio.h>
#include<stdlib.h>

long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n - 1;
    }
    return result;
}

int main(int argc, char **argv)
{
    long x = 5;
    long x_fact = fact_while(x);
    printf("%ld! = %ld\n", x, x_fact);
    return 0;
}
```

```
$ gcc -Og -std=gnu99 -Wall whileloop1.c
```

```
$ ./whileloop1
```

```
5! = 120
```

```
$ gdb whileloop1
```

```
The target architecture is assumed to be i386:x86-64
```

```
Reading symbols from whileloop1...done.
```

```
(gdb) disas fact_while
```

```
Dump of assembler code for function fact_while:
```

```
0x00000000040055d <+0>:    mov     $0x1,%eax           # set result to 1
0x000000000400562 <+5>:    jmp     0x40056c <fact_while+15> # jump to test
0x000000000400564 <+7>:    imul    %rdi,%rax          # loop start, compute result *= n
0x000000000400568 <+11>:   sub     $0x1,%rdi           # decrement n
0x00000000040056c <+15>:   cmp     $0x1,%rdi           # compare n:1
0x000000000400570 <+19>:   jg      0x400564 <fact_while+7> # if >, jmp to loop
0x000000000400572 <+21>:   repz retq                   # return from function
```

```
End of assembler dump.
```

```
(gdb)
```

From C to Assembly: for Loops

From C to Assembly: Functions

From C to Assembly: Data Transfer

From C to Assembly: Local Stack Storage

From C to Assembly: Recursion

From C to Assembly: Array Allocation and Access

From C to Assembly: Structures

From C to Assembly: Function Pointers in Assembly

Managing the Heap

Bibliography

R.E. Bryant and D.R. O'Hallaron. *Computer Systems : A Programmer's Perspective*. Pearson, 2015. ISBN 9781292101767. URL <https://books.google.com/books?id=KfM2rgEACAAJ>.

B.W. Kernighan and D. Ritchie. *The C Programming Language*. Pearson Education, 1988. ISBN 9780133086218. URL <https://books.google.com/books?id=Yi5FI5QcdmYC>.

Dennis M Ritchie. The development of the c language. 1993. URL <https://www.bell-labs.com/usr/dmr/www/chist.pdf>.

Index

ASCII, [23](#), [54](#)

bit, [54](#)

bitmask, [59](#)

byte, [56](#)

license, [2](#)

overflow, [62](#)

printf, [28](#)

sizeof, [67](#)

standard library, [29](#)

swap, [36](#), [37](#)