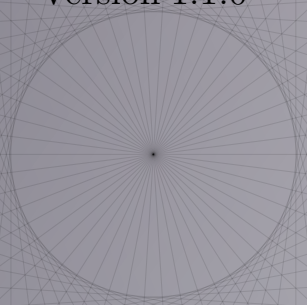
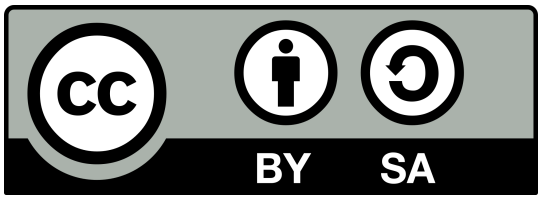
Computer Science I

Dr. Chris Bourke
cbourke@cse.unl.edu
Department of Computer Science & Engineering
University of Nebraska–Lincoln
Lincoln, NE 68588, USA

2015/09/15 21:49:32
Version 1.1.0



Copyleft (Copyright)



Draft Notice

This book is a draft that has been released for evaluation and comment. The draft contains mostly complete chapters up through functions (Chapter 5) for three languages (C, Java, and PHP). Subsequent chapters are included as placeholders and indicators for the intended scope of the final draft, but are intentionally left blank. The author encourages people to send feedback including suggestions, corrections, and reviews to inform and influence the final draft. Thank you in advance to anyone helping out or sending constructive criticisms.

Preface

> "If you really want to understand something, the best way is to try and explain it to someone else. That forces you to sort it out in your own mind... that's really the essence of programming. By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've certainly learned something about it yourself." —Douglas Adams, *Dirk Gently's Holistic Detective Agency* [8]

> "The world of A.D. 2014 will have few routine jobs that cannot be done better by some machine than by any human being. Mankind will therefore have become largely a race of machine tenders. Schools will have to be oriented in this direction. All the high-school students will be taught the fundamentals of computer technology, will become proficient in binary arithmetic and will be trained to perfection in the use of the computer languages that will have developed out of those like the contemporary Fortran' —Isaac Asimov 1964

I've been teaching Computer Science since 2008 and was a Teaching Assistant long before that. Before that I was a student. During that entire time I've been continually disappointed in the value (note, not quality) of textbooks, particularly Computer Science textbooks and especially introductory textbooks. Of primary concern are the costs which have far outstripped inflation over the last 20 years while not providing any real additional value. New editions with trivial changes are released on a regular basis in an attempt to nullify the used book market. Publishers engage in questionable business practices and unfortunately many institutions are complicit in this process.

In established fields such as mathematics and physics, new textbooks are especially questionable as the material and topics don't undergo many changes. However, in Computer Science, new languages and technologies are created and change at breakneck speeds. Faculty and students are regularly trying to give away stacks of textbooks ("Learn Java 4!", "Introduction to Cold Fusion", etc.) that are only a few years old and yet are completely obsolete and worthless. The problem is that such books have built-in obsolescence by focusing too much on technological specifics and not enough on concepts. There are dozens of introductory textbooks for Computer Science; add in the fact that there are multiple languages and many gimmicks ("Learn Multimedia Java", "Gaming with JavaScript", "Build a Robot with C!"), its publisher paradise: hundreds of variations, a growing market, and customers with few alternatives.

That's why I like organizations like Openstax (http://openstaxcollege.org/) that attempt to provide free and "open" learning materials. Though they have textbooks for

a variety of disciplines, Computer Science is not one of them (currently that is). This might be due to the fact that there are already a huge amount of resources available online such as tutorials, videos, online open courses, and even interactive code learning tools. With such a huge amount of resources, why write this textbook then? Firstly, layoff. Secondly, I don't really expect this book to have much impact beyond my own courses or department. I wanted a resource that presented Computer Science how I teach it in my courses and it wasn't available. However, if it does find its way into another instructor's classes or into the hands of an aspiring student that wants to learn, then great!

Several years ago our department revamped our introductory courses in a "Renaissance in Computing" initiative in which we redeveloped several different "flavors" of Computer Science I (one intended for Computer Science majors, one for Computer Engineering majors, one for non-CE engineering majors, one for humanities majors, etc.). The courses are intended to be equivalent in content but have a broader appeal to those in different disciplines. The intent was to provide multiple entry points into Computer Science. Once a student had a solid foundation, they could continue into Computer Science II and pick up a second programming language with little difficulty.

This basic idea informed how I structured this book. There is a separation of concepts and programming language syntax. The first part of this book uses pseudocode for example with a minimum of language-specific elements. Subsequent parts of the book recapitulate these concepts but in the context of a specific programming language. This allows for a "plug-in" style approach to Computer Science: the same book could theoretically be used for multiple courses or the book could be extended by adding another part for a new language with minimal effort.

Another inspiration for the structure of this book is the Computer Science I Honors course that I developed. Usually Computer Science majors take CS1 using Java as the primary language while CE students take CS1 using C. Since the honors course consists of both majors (as well as some of the top students), I developed the Honors version to cover *both* languages at the same time in parallel. This has led to many interesting teaching moments: by covering two languages, it provides opportunities to highlight fundamental differences and concepts in programming languages. It also keeps concepts as the focus of the course emphasizing that syntax and idiosyncrasies of individual languages are only of secondary concern. Finally, actively using multiple languages in the first class provides a better opportunity to extend knowledge to other programming languages–once a student has a solid foundation in one language learning a new one should be relatively easy.

The exercises in this book are a variety of exercises I've used in my courses over the years. They have been made as generic as possible so that they could be assigned using any language. While some have emphasized the use of "real-world" exercises (whatever that means), my exercises have focused more on solving problems of a mathematical nature (most of my students have been Engineering students). Some of them are more easily understood if students have had Calculus but it is not absolutely necessary.

It may be cliché, but the two quotes above exemplify what I believe a Computer Science I course is about. The second is from Isaac Asimov who was asked at the 1964 World's Fair what he though the world of 2014 would look like. His depiction isn't entirely true, but I do believe we are on the verge of a fundamental social change that will be caused by more and more automation. Like the industrial revolution, but on a much smaller time scale and to a far greater extent, automation will fundamentally change how we live and not work (I say "not work" because automation will very easily destroy the vast majority of today's jobs–this a huge economic and political issue that will need to be addressed). The time is quickly approaching where being able to program and develop software will be considered a fundamental skill as essential as arithmetic. I hope this book plays some small role in helping students adjust to that coming world.

The second quote describes programming, or more fundamentally Computer Science and "problem solving." Computers do not solve problems, humans do. Computers only make automating solutions possible quickly and on a large scale. At the end of the day, the human race is still responsible for tending the machines and will be for some time despite what Star Trek and the most optimistic of AI advocates think.

I hope that people find this book useful. If value is a ratio of quality vs cost then this book has already succeeded in having infinite value.[1] If you have suggestions on how to improve it, please feel free to contact me. If you end up using it and finding it useful, please let me know that too!

[1] or it might be undefined, or NaN, or this book is `Exception`al depending on which language sections you read

Acknowledgements

I'd like to thank the Department of Computer Science & Engineering at the University of Nebraska–Lincoln for their support during my writing and maintaining this book.

This book is dedicated to my family.

Contents

List of Algorithms

List of Code Samples

List of Figures

1. Introduction

Computers are awesome. The human race has seen more advancements in the last 50 years than in the entire 10,000 years of human history. Technology has transformed the way we live our daily lives, how we interact with each other, and has changed the course of our history. Today, everyone carries smart phones which have more computational power than supercomputers from 20 years ago. Computing has become ubiquitous, the "internet of things" will soon become a reality in which every device will become interconnected and data will be collected and available even about the smallest of minutiae.

However, computers are also dumb. Despite the most fantastical of depictions in science fiction and and hopes of Artificial Intelligence, computers can only do what they are told to do. The fundamental art of Computer Science is problem solving. Computers are not good at problem solving; *you* are the problem solver. It is still up to you, the user, to approach a complex problem, study it, understand it, and develop a solution to it. Computers are only good at automating solutions once you have solved the problem.

Computational sciences have become a fundamental tool of almost every discipline. Scholars have used textual analysis and data mining techniques to analyze classical literature and historic texts, providing new insights and opening new areas of study. Astrophysicists have used computational analysis to detect dozens of new exoplanets. Complex visualizations and models can predict astronomical collisions on a galactic scale. Physicists have used big data analytics to push the boundaries of our understanding of matter in the search for the Higgs boson and study of elementary particles. Chemists simulate the interaction of millions of combinations of compounds without the need for expensive and time consuming physical experiments. Biologists use massively distributed computing models to simulate protein folding and other complex processes. Meteorologists can predict weather and climactic changes with ever greater accuracy.

Technology and data analytics have changed how political campaigns are run, how products are marketed and even delivered. Social networks can be data mined to track and predict the spread of flu epidemics. Computing and automation will only continue to grow. The time is soon coming where basic computational thinking and the ability to develop software will be considered a basic skill necessary to every discipline, a requirement for many jobs and an essential skill akin to arithmetic.

Computer Science is not programming. Programming is a necessary skill, but it is only the beginning. This book is intended to get you started on your journey.

1.1. Problem Solving

At its heart, Computer Science is about problem solving. Not problem solving as a discipline, it would be hubris to think that Computer Science holds a monopoly on "problem solving." Indeed, it would be hard to find any discipline in which solving problems was not a substantial aspect or motivation if not integral. Instead, Computer Science is the study of computers and computation. It involves studying and understanding computational processes and the development of algorithms and techniques and how they apply to problems.

Problem solving skills are not something that can be distilled down into a single step-by-step process. Each area and each problem comes with its own unique challenges and considerations. General problem solving techniques can identified, studied and taught, but problem solving skills are something that come with experience, hard work, and most importantly, failure. Problem solving is part and parcel of the human experience.

That doesn't mean we can't identify techniques and strategies for approaching problems, in particular problems that lend themselves to computational solutions. A prerequisite to solving a problem is *understanding* it. What is the problem? Who or what entities are involved in the problem? How do those entities interact with each other? What are the problems or deficiencies that need to be addressed? Answering these questions, we get an idea of *where we are*.

Ultimately, what is desired in a solution? What are the objectives that need to be achieved? What would an ideal solution look like or what would it do? Who would use the solution and how would they use it? By answering these questions, we get an idea of *where we want to be*.

Once we know where we are and where we want to be, the problem solving process can begin: how do we get from point A to point B?

One of the first things a good engineer asks is: does a solution already exist? If a solution already exists, then the problem is already solved! Ideally the solution is an "off-the-shelf" solution: something that already exists and which may have been designed for a different purpose but that can be *repurposed* for our problem. However, there may be exceptions to this. The existing solution may be infeasible: it may be too resource intensive or expensive. It may be too difficult or too expensive to adapt to our problem. It may solve most of our problem, but may not work in some corner cases. It may need to be heavily modified in order to work. Still, this basic question may save a lot of time and effort in many cases.

In a very broad sense, the problem solving process is one that involves

1. Design
2. Implementation
3. Testing

4. Refinement

After one has a good understanding of a problem, they can start designing a solution. A design is simply a plan on the construction of a solution. A design "on paper" allows you to see what the potential solution would look like before investing the resources in building it. It also allows you to identify possible impediments or problems that were not readily apparent. A design allows you to an opportunity to think through possible alternative solutions and weigh the advantages and disadvantages of each. Designing a solution also allows you to understand the problem better.

Design can involve gathering requirements and developing use cases. How would an individual *use* the proposed solution? What features would they need or want?

Implementations can involve building prototype solutions to test the feasibility of the design. It can involve building individual components and integrating them together.

Testing involves finding, designing, and developing test cases: actual instances of the problem that can be used to test your solution. Ideally, the a test case instance involves not only the "input" of the problem, but also the "output" of the problem: a feasible or optimal solution that is known to be correct via other means. Test cases allow us to test our solution to see if it gives correct and perhaps optimal solutions.

Refinement is a process by which we can redesign, reimplement and retest our solution. We may want to make the solution more efficient, cheaper, simpler or more elegant. We may find there are components that are redundant or unnecessary and try to eliminate them. We may find errors or bugs in our solution that fail to solve the problem for some or many instances. We may have misinterpreted requirements or there may have been miscommunication, misunderstanding or differing expectations in the solution between the designers and stakeholders. Situations may change or requirements may have been modified or new requirements created and the solution needs to be adapted. Each of these steps may need to be repeated many times until an ideal solution, or at least acceptable, solution is achieved.

Yet another phase of problem solving is maintenance. The solution we create may need to be maintained in order to remain functional and stay relevant. Design flaws or bugs may become apparent that were missed in the process. The solution may need to be updated to adapt to new technology or requirements.

In software design there are two general techniques for problem solving; top-down and bottom-up design. A top-down design strategy approaches a problem by breaking it down into smaller and smaller problems until either a solution is obvious or trivial or a preexisting solution (the aforementioned "off-the-shelf" solution) exists. The solutions to the subproblems are combined and interact to solve the overall problem.

A bottom-up strategy attempts to first completely define the smallest components or entities that make up a system first. Once these have been defined and implemented, they are combined and interactions between them are defined to produce a more complex system.

1.2. Computing Basics

Everyone has some level of familiarity with computers and computing devices just as everyone has familiarity with automotive basics. However, just because you drive a car everyday doesn't mean you can tell the difference between a crankshaft and a piston. To start, let's familiarize ourselves with some basic concepts.

A computer is a device, usually electronic, that stores, receives, processes, and outputs information. Modern computing devices include everything from simple sensors to mobile devices, tablets, desktops, mainframes/servers, supercomputers, to huge grid clusters consisting of multiple computers networked together.

Computer hardware usually refers to the physical components in a computing system which includes input devices such as a mouse/touchpad, keyboard, or touchscreen, output devices such as monitors, storage devices such as hard disks and solid state drives, as well as the electronic components such as graphics cards, main memory, motherboards and chips that make up the Central Processing Unit (CPU).

Computer processors are complex electronic circuits (referred to as Very Large Scale Integration (VLSI)) which contain thousands of microscopic electronic transistors–electronic "gates" that can perform logical operations and complex instructions. In addition to the CPU a processor may contain an Arithmetic and Logic Unit (ALU) that performs arithmetic operations such as addition, multiplication, division, etc.

Computer Software usually refers to the actual machine instructions that are run on a processor. Software is usually written in a high-level programming language such as C or Java and then converted to machine code that the processor can execute.

Computers "speak" in binary code. Binary is nothing more than a structured collection of 0s and 1s. A single 0 or 1 is referred to as a bit. Bits can be collected to form larger chunks of information: 8 bits form a byte, 1024 bytes is referred to as a kilobyte, etc. Table 1.1 contains a several more binary units. Each unit is in terms of a power of 2 instead of a power of 10. As humans, we are more familiar with decimal–base-10 numbers and so units are usually expressed as powers of 10, kilo- refers to 10^3, mega- is 10^6, etc. However, since binary is base-2 (0 or 1), units are associated with the closest power of 2.

Computers are binary machines because it is the most practical to implement in electronic devices. 0s and 1s can be easily represented by low/high voltage; low/high frequency; on-off; etc. It is much easier to design and implement systems that switch between only two states.

Computer *memory* can refer to *secondary memory* which are typically longterm storage devices such as hard disks, flash drives, SD cards, optical disks (CDs, DVDs), etc. These generally have large capacity but are slower (the time it takes to access a chunk of data is longer). Or, it can refer to *main memory* (or primary memory): data stored on chips that is much faster but also more expensive and thus generally smaller.

Unit	2^n	Number of bytes
Kilobyte (KB)	2^{10}	1,024
Megabyte (MB)	2^{20}	1,048,576
Gigabyte (GB)	2^{30}	1,073,741,824
Terabyte (TB)	2^{40}	1,099,511,627,776
Petabyte (PB)	2^{50}	1,125,899,906,842,624
Exabyte (EB)	2^{60}	1,152,921,504,606,846,976
Zettabyte (ZB)	2^{70}	1,180,591,620,717,411,303,424
Yottabyte (YB)	2^{80}	1,208,925,819,614,629,174,706,176

Table 1.1.: Various units of digital information with respect to bytes.

The first hard disk (IBM 350) was developed in 1956 by IBM and had a capacity of 3.75MB and cost $3,200 ($27,500 in 2015 dollars) per month to lease. For perspective, the first commercially available TB hard drive was released in 2007. As of 2015, terabyte hard disks can be commonly purchased for $50-100.

Main memory, sometimes referred to as Random Access Memory (RAM) consists of a collection of *addresses* along with *contents*. An address usually refers to a single byte of memory. The content, that is the byte of data that is stored at an address, can be anything. It can represent a number, a letter, etc. In the end, to the computer its all just a bunch of 0s and 1s. For convenience, data, in particular memory addresses are represented using hexadecimal, which is a base-16 counting system using the symbols $0, 1, \ldots, 9, a, b, c, d, e, f$. Numbers are prefixed with a `0x` to indicate they represent hexadecimal numbers.

Separate computing devices can be connected to each other through a *network*. Networks can be wired which provide large bandwidth (the amount of data that can be sent at any one time), but expensive to build and maintain. They can also be wireless, but provide shorter range and lower bandwidth.

1.3. Basic Program Structure

Programs start out as *source code*, a collection of instructions usually written in a high-level programming language. A source file containing source code is nothing more than a plain text file that can be edited by any text editor. However, many developers and programmers utilize modern Integrated Development Environment (IDE) that provide a text editor with *code highlighting*: various elements are displayed in various colors to make the code more readable and various elements can be easily identified. Mistakes such as unclosed comments or curly brackets can be readily apparent with such editors. IDEs can also provide automated compile/build features and other tools that make the development process easier and faster.

Address	Contents
⋮	⋮
0x7fff58310b8f	
0x7fff58310b8b	0x32
0x7fff58310b8a	0x3e
0x7fff58310b89	0xcf
0x7fff58310b88	0x23
0x7fff58310b87	0x01
0x7fff58310b86	0x32
0x7fff58310b85	0x7c
0x7fff58310b84	0xff
0x7fff58310b83	
0x7fff58310b82	
0x7fff58310b81	
0x7fff58310b80	
0x7fff58310b7f	
0x7fff58310b7e	
0x7fff58310b7d	
0x7fff58310b7c	3.14159265359
0x7fff58310b7b	
0x7fff58310b7a	
0x7fff58310b79	
0x7fff58310b78	32,321,231
0x7fff58310b77	
0x7fff58310b76	
0x7fff58310b75	
0x7fff58310b74	1,458,321
0x7fff58310b73	\0
0x7fff58310b72	o
0x7fff58310b71	l
0x7fff58310b70	l
0x7fff58310b6f	e
0x7fff58310b6e	H
0x7fff58310b88	0xfa
0x7fff58310b87	0xa8
0x7fff58310b86	0xba
⋮	⋮

Table 1.2.: Depiction of Computer Memory. Each address refers to a byte, but different types of data (integers, floating-point numbers, characters) take different amounts of memory. Memory addresses and some data is represented in *hexadecimal*.

Some languages are *compiled* languages meaning that a source file must be translated into the machine code that a processor can understand and execute. This is actually a multistep process. A compiler may first preprocess the source file(s) and perform some pre-compiler operations. It may then transform the source code into another language such as an assembly language, a lower level, more machine-like language. Ultimately, the compiler transforms the source code into object code, a binary format that the machine can understand.

To produce an executable file that can actually be run, a *linker* may then take the object code and link in any other necessary objects or precompiled library code necessary to produce a final program. Finally, an executable file (still just a bunch of binary code) is produced.

We can now execute the program. When a program is executed, a request is sent to the operating system to load and run the program. The operating system loads the executable file into memory and may setup additional memory for its variables as well as its *call stack* (memory to enable the program to make function calls). Once loaded and setup, the operating system begins executing the instructions at the program's entry point.

In many languages, a program's entry point is defined by a *main* function or method. A program may contain many functions and pieces of code, but this special function is defined as the one that gets called when a program starts. Without a main function, the code may still be useful: libraries contain many useful functions and procedures so that you don't have to write a program from scratch. However, these functions are not intended to be run by themselves. Instead, they are written so that other programs can use them. A program becomes executable when a main entry point is provided.

This compile-link-execute process is roughly depicted in Figure 1.1. An example of a simple C program can be found in Figure 1.1 along with the resulting assembly code produced by a compiler (gcc) in Figure 1.2 and the final machine code represented in hexadecimal in Figure 1.3.

In contrast, some languages are *interpreted*, not compiled. Instead, the source code is contained in a file usually referred to as a *script*. Rather than being run directly by an operating system, the operating system loads and execute another program called an *interpreter*. The interpreter then loads the script, parses, and execute its instructions. Interpreted languages may still have a predefined main function, but in general, a script starts executing starting with the first instruction in the script file. Adhering to the syntax rules is still important, but since interpreted languages are not compiled, syntax errors become runtime errors. A program may run fine until its first syntax error at which point it fails.

There are other ways of compiling and running programs. Java for example represents a compromise between compiled and interpreted languages. Java source code is compiled into Java bytecode which is not actually machine code that the operating system and hardware can run directly. Instead, it is compiled code for a Java Virtual Machine (JVM).

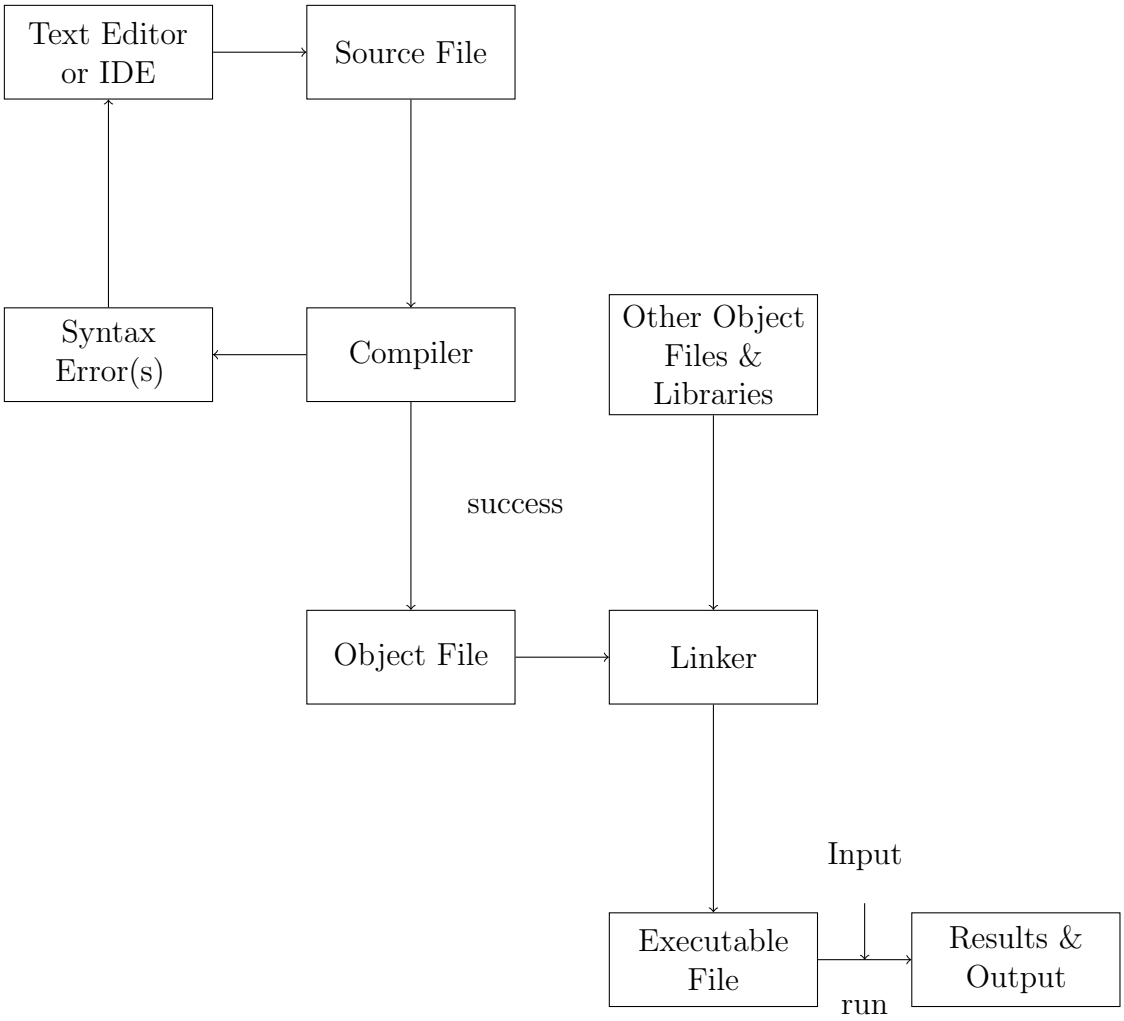


Figure 1.1.: A Compiling Process

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

int main(int argc, char **argv) {

  if(argc != 2) {
    fprintf(stderr, "Usage: %s x\n", argv[0]);
    exit(1);
  }

  double x = atof(argv[1]);
  double result = sqrt(x);

  if(x < 0) {
    fprintf(stderr, "Cannot handle complex roots\n");
    exit(2);
  }

  printf("square root of %f = %f\n", x, result);

  return 0;
}
```

Code Sample 1.1: A simple program in C

This allows a developer to write highly portable code, compile it and it is runnable on any JVM on any system (write-once, compile-once, run-anywhere).

In general, interpreted languages are slower than compiled languages because they are being run through another program (the interpreter) instead of being executed directly by the processor. Modern tools have been introduced to solve this problem. Just In Time (JIT) compilers have been developed that take scripts that are not usually compiled, and compile them to a native machine code format which has the potential to run much faster than when interpreted. Modern web browsers typically do this for JavaScript code (Chrome's V8 JavaScript engine for example).

Transpilers are source-to-source compilers. They don't produce assembly or machine code, instead they take one language and translate it to another language. This is sometimes done to ensure that scripting languages like JavaScript are backwards compatible with previous versions of the language. Transpilers can also be used to translate one language into the same language but with different aspects (such as parallel or synchronized code) automatically added. They can also be used to translate older languages such as Pascal to more modern languages as a first step in updating a legacy system.

```
        .section        __TEXT,__text,regular,pure_instructions
        .globl        _main
        .align        4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        pushq        %rbp
Ltmp2:
        .cfi_def_cfa_offset 16
Ltmp3:
        .cfi_offset %rbp, -16
        movq        %rsp, %rbp
Ltmp4:
        .cfi_def_cfa_register %rbp
        subq        $48, %rsp
        movl        $0, -4(%rbp)
        movl        %edi, -8(%rbp)
        movq        %rsi, -16(%rbp)
        cmpl        $2, -8(%rbp)
        je        LBB0_2
## BB#1:
        leaq        L_.str(%rip), %rsi
        movq        ___stderrp@GOTPCREL(%rip), %rax
        movq        (%rax), %rdi
        movq        -16(%rbp), %rax
        movq        (%rax), %rdx
        movb        $0, %al
        callq        _fprintf
        movl        $1, %edi
        movl        %eax, -36(%rbp)         ## 4-byte Spill
        callq        _exit
LBB0_2:
        movq        -16(%rbp), %rax
        movq        8(%rax), %rdi
        callq        _atof
        xorps        %xmm1, %xmm1
        movsd        %xmm0, -24(%rbp)
        movsd        -24(%rbp), %xmm0
        sqrtsd        %xmm0, %xmm0
        movsd        %xmm0, -32(%rbp)
        ucomisd        -24(%rbp), %xmm1
        jbe        LBB0_4
## BB#3:
        leaq        L_.str1(%rip), %rsi
        movq        ___stderrp@GOTPCREL(%rip), %rax
        movq        (%rax), %rdi
        movb        $0, %al
        callq        _fprintf
        movl        $2, %edi
        movl        %eax, -40(%rbp)         ## 4-byte Spill
        callq        _exit
LBB0_4:
        leaq        L_.str2(%rip), %rdi
        movsd        -24(%rbp), %xmm0
        movsd        -32(%rbp), %xmm1
        movb        $2, %al
        callq        _printf
        movl        $0, %ecx
        movl        %eax, -44(%rbp)         ## 4-byte Spill
        movl        %ecx, %eax
        addq        $48, %rsp
        popq        %rbp
        retq
        .cfi_endproc

        .section        __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
        .asciz        "Usage: %s x\n"

L_.str1:                                ## @.str1
        .asciz        "Cannot handle complex roots\n"

L_.str2:                                ## @.str2
        .asciz        "square root of %f = %f\n"

.subsections_via_symbols
```

Code Sample 1.2: A simple program in C, compiled to assembly

```
00000e40  55 48 89 e5 48 83 ec 30  c7 45 fc 00 00 00 00 89  |UH..H..0.E......|
00000e50  7d f8 48 89 75 f0 81 7d  f8 02 00 00 00 0f 84 2c  |}.H.u..}.......,|
00000e60  00 00 00 48 8d 35 f2 00  00 00 48 8b 05 9f 01 00  |...H.5....H.....|
00000e70  00 48 8b 38 48 8b 45 f0  48 8b 10 b0 00 e8 94 00  |.H.8H.E.H.......|
00000e80  00 00 bf 01 00 00 00 89  45 dc e8 81 00 00 00 48  |........E......H|
00000e90  8b 45 f0 48 8b 78 08 e8  6e 00 00 00 0f 57 c9 f2  |.E.H.x..n....W..|
00000ea0  0f 11 45 e8 f2 0f 10 45  e8 f2 0f 51 c0 f2 0f 11  |..E....E...Q....|
00000eb0  45 e0 66 0f 2e 4d e8 0f  86 25 00 00 00 48 8d 35  |E.f..M...%...H.5|
00000ec0  a5 00 00 00 48 8b 05 45  01 00 00 48 8b 38 b0 00  |....H..E...H.8..|
00000ed0  e8 41 00 00 00 bf 02 00  00 00 89 45 d8 e8 2e 00  |.A.........E....|
00000ee0  00 00 48 8d 3d 9d 00 00  00 f2 0f 10 45 e8 f2 0f  |..H.=.......E...|
00000ef0  10 4d e0 b0 02 e8 22 00  00 00 b9 00 00 00 00 89  |.M....".........|
00000f00  45 d4 89 c8 48 83 c4 30  5d c3 ff 25 08 01 00 00  |E...H..0]..%....|
00000f10  ff 25 0a 01 00 00 ff 25  0c 01 00 00 ff 25 0e 01  |.%.....%.....%..|
00000f20  00 00 00 00 4c 8d 1d dd  00 00 00 41 53 ff 25 cd  |....L......AS.%.|
00000f30  00 00 00 90 68 00 00 00  00 e9 e6 ff ff ff 68 0c  |....h.........h.|
00000f40  00 00 00 e9 dc ff ff ff  68 18 00 00 00 e9 d2 ff  |........h.......|
00000f50  ff ff 68 27 00 00 00 e9  c8 ff ff ff 55 73 61 67  |..h'........Usag|
00000f60  65 3a 20 25 73 20 78 0a  00 43 61 6e 6e 6f 74 20  |e: %s x..Cannot |
00000f70  68 61 6e 64 6c 65 20 63  6f 6d 70 6c 65 78 20 72  |handle complex r|
00000f80  6f 6f 74 73 0a 00 73 71  75 61 72 65 20 72 6f 6f  |oots..square roo|
00000f90  74 20 6f 66 20 25 66 20  3d 20 25 66 0a 00 00 00  |t of %f = %f....|
00000fa0  01 00 00 00 1c 00 00 00  00 00 00 00 1c 00 00 00  |................|
00000fb0  00 00 00 00 1c 00 00 00  02 00 00 00 40 0e 00 00  |............@...|
00000fc0  34 00 00 00 34 00 00 00  0b 0f 00 00 00 00 00 00  |4...4...........|
00000fd0  34 00 00 00 03 00 00 00  0c 00 01 00 10 00 01 00  |4...............|
00000fe0  00 00 00 00 00 00 00 01  14 00 00 00 00 00 00 00  |................|
00000ff0  01 7a 52 00 01 78 10 01  10 0c 07 08 90 01 00 00  |.zR..x..........|
00001000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001010  00 00 00 00 00 00 00 00  34 0f 00 00 01 00 00 00  |........4.......|
00001020  3e 0f 00 00 01 00 00 00  48 0f 00 00 01 00 00 00  |>.......H.......|
00001030  52 0f 00 00 01 00 00 00  00 00 00 00 00 00 00 00  |R...............|
00001040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00002000  11 22 18 54 00 00 00 00  11 40 5f 5f 5f 73 74 64  |."".T.....@___std|
00002010  65 72 72 70 00 51 72 10  90 40 64 79 6c 64 5f 73  |errp.Qr..@dyld_s|
00002020  74 75 62 5f 62 69 6e 64  65 72 00 80 e8 ff ff ff  |tub_binder......|
00002030  ff ff ff ff ff 01 90 00  72 18 11 40 5f 61 74 6f  |........r..@_ato|
00002040  66 00 90 00 72 20 11 40  5f 65 78 69 74 00 90 00  |f...r .@_exit...|
00002050  72 28 11 40 5f 66 70 72  69 6e 74 66 00 90 00 72  |r(.@_fprintf...r|
00002060  30 11 40 5f 70 72 69 6e  74 66 00 90 00 00 00 00  |0.@_printf......|
00002070  00 01 5f 00 05 00 02 5f  6d 68 5f 65 78 65 63 75  |.._...._mh_execu|
00002080  74 65 5f 68 65 61 64 65  72 00 21 6d 61 69 6e 00  |te_header.!main.|
00002090  25 02 00 00 00 03 00 c0  1c 00 00 00 00 00 00 00  |%...............|
000020a0  c0 1c 00 00 00 00 00 00  fa de 0c 05 00 00 00 14  |................|
000020b0  00 00 00 01 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000020c0  02 00 00 00 0f 01 10 00  00 00 00 00 01 00 00 00  |................|
000020d0  16 00 00 00 0f 01 00 00  40 0e 00 00 01 00 00 00  |........@.......|
000020e0  1c 00 00 00 01 00 00 01  00 00 00 00 00 00 00 00  |................|
000020f0  27 00 00 00 01 00 00 01  00 00 00 00 00 00 00 00  |'...............|
00002100  2d 00 00 00 01 00 00 01  00 00 00 00 00 00 00 00  |-...............|
00002110  33 00 00 00 01 00 00 01  00 00 00 00 00 00 00 00  |3...............|
00002120  3c 00 00 00 01 00 00 01  00 00 00 00 00 00 00 00  |<...............|
00002130  44 00 00 00 01 00 00 01  00 00 00 00 00 00 00 00  |D...............|
00002140  03 00 00 00 04 00 00 00  05 00 00 00 06 00 00 00  |................|
00002150  07 00 00 00 00 00 00 40  02 00 00 00 03 00 00 00  |.......@........|
00002160  04 00 00 00 05 00 00 00  06 00 00 00 20 00 5f 5f  |............ .__|
00002170  6d 68 5f 65 78 65 63 75  74 65 5f 68 65 61 64 65  |mh_execute_heade|
00002180  72 00 5f 6d 61 69 6e 00  5f 5f 5f 73 74 64 65 72  |r._main.___stder|
00002190  72 70 00 5f 61 74 6f 66  00 5f 65 78 69 74 00 5f  |rp._atof._exit._|
000021a0  66 70 72 69 6e 74 66 00  5f 70 72 69 6e 74 66 00  |fprintf._printf.|
000021b0  64 79 6c 64 5f 73 74 75  62 5f 62 69 6e 64 65 72  |dyld_stub_binder|
000021c0  00 00 00 00                                       |....|
000021c4
```

Code Sample 1.3: A simple program in C, resulting machine code formatted in hexadecimal (partial)

1.4. Syntax Rules & Pseudocode

Programming languages are a lot like spoken languages in that they have *syntax* rules. These rules dictate the appropriate arrangements of words, punctuation, and other symbols that form valid statements in the language. For example, many programming languages, commands or statements are terminated by semicolons (just as most sentences are terminated with a period). This is an example of "punctuation" a programming language.

In general, individual executable commands are written one per line. When a program executes, each command executes one after the other. This is known as sequential control flow.

A *block* of code is a section of code that has been logically grouped together. Many languages allow you to define a block by enclosing the grouped code around opening and closing curly brackets. Blocks can be *nested* within each other to form sub-blocks.

Most languages also have reserved words and symbols that have special meaning. For example, many languages assign special meaning to keywords such as `for`, `if`, `while`, etc. that are used to define various *control structures* such as conditionals and loops. Special symbols include operators such as + and ∗ for performing basic arithmetic.

Failure to adhere to the syntax rules of a particular language will lead to bugs and programs that fail to compile and/or run. Natural languages are very forgiving: we can generally discern what someone is trying to say even if they speak in broken English (to a point). However, a compiler or interpreter isn't as smart as a human. Even a small syntax error will cause a compiler to completely fail to understand the code you have written. Learning a programming language is a lot like learning a new spoken language (but, fortunately a lot easier).

In subsequent parts of this book we focus on particular languages. However, in order to focus on concepts, we'll avoid specific syntax rules by using pseudocode, informal, high-level descriptions of algorithms and processes. Good pseudocode makes use of plain English and mathematical notation, making it more readable and abstract. A small example can be found in Algorithm 1.1.

1.5. Documentation, Comments, and Coding Style

Good code is not just functional, its also beautiful. Good code is organized, easy to read, and well documented. Organization can be achieved by separating code into useful functions and collecting functions into *modules* or libraries. Good organization means that at any one time, we only need to focus on a small part of a program.

It would be difficult to read an essay that contained random line breaks, paragraphs were not indented, it contained different spacing or different fonts, etc. Likewise, code

```
   INPUT  : A collection of numbers, A = {a_1, a_2, ..., a_n}
   OUTPUT : The minimal element in A
 1 Let min be equal to a_1
 2 FOREACH element a_i in A DO
 3 |   IF a_i < min THEN
 4 |   |   a_i is less than the smallest element we've found so far
 5 |   |   Update min to be equal to a_i
 6 |   END
 7 END
 8 output min
```

Algorithm 1.1: An example of pseudocode: finding a minimum value

should be legible. Well written code is consistent and makes good use of whitespace and indentation. Code in the same code block should be indented at the same level. Nested blocks should be further indented just like the outline of an essay or table of contents.

Code should be well-documented. The code itself should be clear enough that it tells the user *what* the code does and *how* it does it. This is called "self-documenting" code. In addition, well-written code should contain sufficient and clear *comments*. A comment in a program is intended for a human user to read. A comment is ultimately ignored and has no effect on the actual program. Good comments tell the user *why* the code was written or why it was written the way it was. Comments provide a high-level description of what a block of code, function, or program does. If the particular method or algorithm is of interest, it should also be documented.

There are typically two ways to write comments. Single line comments usually begin with two forward slashes, `//`. Everything after the slashes until the next line is ignored by the program. Multiline comments begin with a `/*` and end with a `*/`; everything between them is ignored even if it spans multiple lines. This syntax is shared among many languages including C, Java, PHP and others. Some examples:

```
double x = sqrt(y); //this is a single line comment

/*
  This is a multiline comment
  each line is ignored, but allows
  for better formatting
*/

/**
 * This is a doc-style comment, usually placed in
 * front of major portions of code such as a function
 * to provide documentation
 * It begins with a forward-slash-star-star
 */
```

The last example above is a doc-style comment. It originated with Java, but has since been adopted by many other programming languages. Syntactically it is a normal multiline comment, but begins with a `/**`. Asterisks are aligned together on each line. Certain commenting systems allow you to place other marked up data inside these comments such as labeling parameters (`@param x`) or use HTML code to provide links. These doc-style comments are used to provide documentation for major parts of the code especially functions and data structures. Though not part of the language, other documentation tools can be used to gather the information in doc-style comments to produce documentation documents (such as web pages).

Comments should not be trivial: they should not explain something that should be readily apparent to an experienced user or programmer. For example, if a piece of code adds two numbers together and stores the result, there should not be a comment that explains the process. It is a simple and common enough operation that is self-evident. However, if a function uses a particular process or algorithm such as a Fourier Transform to perform an operation, it would be appropriate to document it in a series of comments.

Comments can also detail how a function or piece of code should be used. This is typically done when developing an Application Programmer Interface (API) for use by other programmers. The API's available functions should be well-documented so that users will know how and when to use a particular function. It can document the function's expectations and behavior such as how it handles bad input or error situations.

2. Basics

2.1. Control Flow

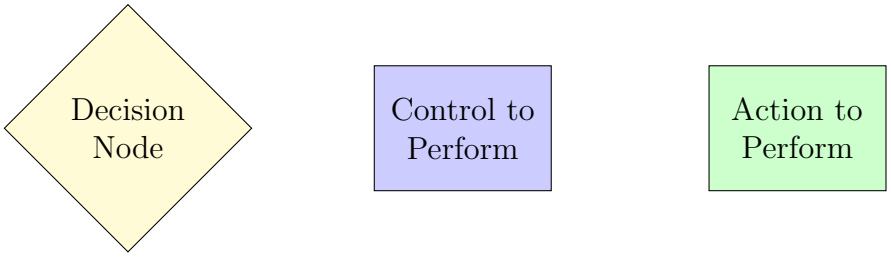
The flow of control (or simply control flow) is how a program processes its instructions. Typically, programs operate in a linear or *sequential* flow of control. Executable statements or instructions in a program are performed one after another. In source code, the order that instructions are written defines their order. Just like English, a program is "read" top to bottom. Each statement may modify the *state* of a program. The state of a program is the value of all its variables and other information/data stored in memory at a given moment during its execution. Further, an executable statement may instead invoke (or call or execute) another *procedure* (also called subroutine, function, method, etc.) which is another unit of code that has been encapsulated into one unit so that it can be reused.

This type of control flow is usually associated with a procedural programming paradigm (which is closely related to imperative or structured programming paradigms). Though this text will mostly focus on languages that are procedural (or that have strong procedural aspects), it is important to understand that there are other programming language paradigms. Functional programming languages such as Scheme and Haskell achieve computation through the evaluation of mathematical functions with as little or no ("pure" functional) state at all. Declarative languages such as those used in database languages like SQL or in spreadsheets like Excel specify computation by expressing the logic of computation rather than explicitly specifying control flow. For a more formal introduction to programming language paradigms, a good resource is *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages* by Tate [21].

2.1.1. Flow Charts

Sometimes processes are described using diagrams called flowchartss. A flow chart is a visual representation of an algorithm or process consisting of boxes or "nodes" connected by directed edges. Boxes can represent an individual step or a decision to be made. The edges establish an *order* of operations in the diagram.

Some boxes represent *decisions* to be made which may have one or more alternate routes (more than one directed edge going out of the box) depending on the the result of the decision. Decision boxes are usually depicted with a diamond shaped box.



(a) Decision Node (b) Control Node (c) Action Node

Figure 2.1.: Types of Flowchart Nodes. Control and action nodes are distinguished by color. Control nodes are automated steps while action nodes are steps performed as part of the algorithm being depicted.

Other boxes represent a process, operation, or action to be performed. Boxes representing a process are usually rectangles. We will further distinguish two types of processes using two different colorings: we'll use green to represent boxes that are steps directly related to the algorithm being depicted. We'll use blue for actions that are necessary to the control flow of the algorithm such as assigning a value to a variable or incrementing a value as part of a loop. Figure 2.1 depicts the three types of boxes we'll use. Figure 2.2 depicts a simple ATM (Automated Teller Machine) process as an example.

2.2. Variables

In mathematics, variables are used as placeholders for values that aren't necessarily known. For example, in the equation,

$$x = 3y + 5$$

the variables x and y represent numbers that can take on a number of different values.

Similarly, in a computer program, we also use variables to store values. A variable is essentially a memory location in which a *value* can be stored. Typically, a variable is referred to by a *name* or identifier (like x, y, z in mathematics). In mathematics variables are usually used to hold numerical values. However, in programming, variables can usually hold different *types* of values such as numbers, strings (a collection of characters), Booleans (*true* or *false* values), or more complex types such as objects.

2.2.1. Naming Rules & Conventions

Most programming languages have very specific rules as to what you can use as variable identifiers (names). For example, most programming languages do not allow you to use

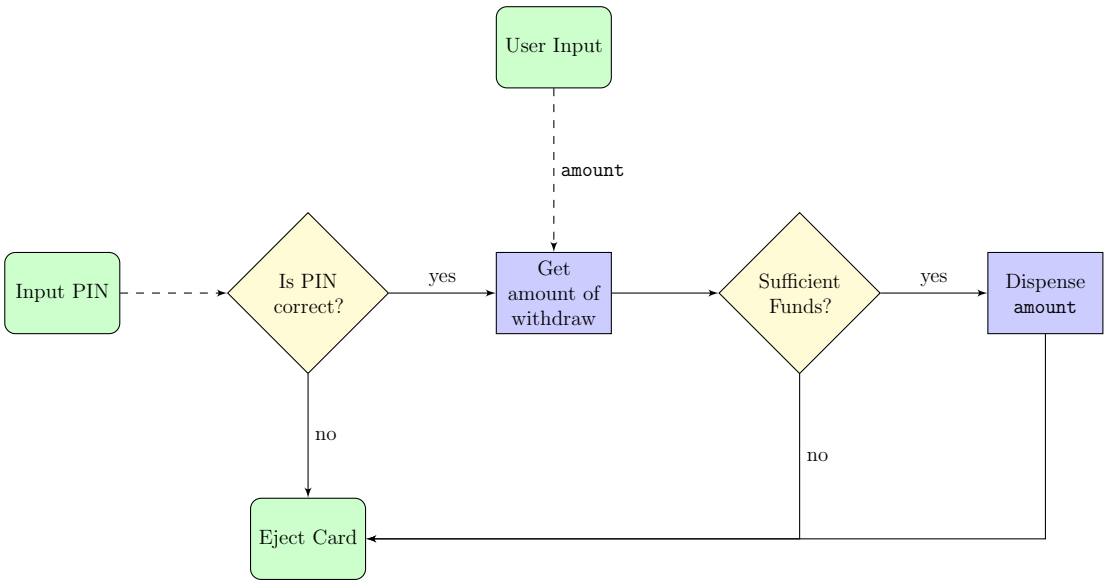


Figure 2.2.: Example of a flow chart for a simple ATM process

whitespace characters (space, tab, etc.) in a variable's identifier. Allowing spaces would make variable names ambiguous: where does the variable's name end and the rest of the program continue? How could you tell the difference between "average score" and two separate variables named "average" and "score"? Many programming languages also have reserved word–words or terms that are used by the programming language itself and have special meaning. Variable names cannot be the same as any reserved word as the language wouldn't be able to distinguish between them.

For similar reasons, many programming languages do not allow you to start a variable name with a number as it would make it more difficult to decipher a program. Yet other languages require that variables begin with a specific character (PHP for example *requires* that all variables begin with a dollar sign, `$`).

In general, most programming languages allow you to use upper `[A-Z]` and lowercase `[a-z]` letters as well as numbers, `[0-9]` and certain special characters such as underscores `_` or dollar signs, `$`. Moreover, most programming languages (like English) are case sensitive meaning that a variable name using lower case letters is not the same variable as one that uses upper case letters. For example, the variables `x` and `X` are *different*; the variables `average`, `Average` and `AVERAGE` are all different as well. A few languages are *case-insensitive* meaning that they do not recognize differences in lower and upper case letters when used in variable identifiers. Even in these languages, however, using a mixture of lower and upper case letters to refer to the same variable is discouraged: it is difficult to read, inconsistent, and just plain ugly.

Beyond the naming rules that languages may enforce, most languages have established naming conventions; a set of guidelines and best-practices for choosing identifier names for variables (as well as functions, methods, and class names). Conventions may be widely

adopted on a per-language basis or may be established within a certain library, framework or by an organization. Naming conventions are intended to give source code consistency which ultimately improves readability and makes it easier to understand. Following a consistent convention can also greatly reduce the chance for errors and mistakes. Good naming conventions also has an aesthetic appeal; code should be beautiful.

There are several general conventions when it comes to variables. An early convention, but still in common use is *underscore casing* in which variable names consisting of more than one word have words separated by underscore characters with all other characters being lower case. For example:

`average_score`, `number_of_students`, `miles_per_hour`

A variation on this convention is to use all uppercase letters such as `MILES_PER_HOUR`. A more modern convention is to use *lower camel casing* (or just *camel casing*) in which variable names with multiple words are written as one long word with the first letter in each new word capitalized but with the first word's first letter lower case. For example:

`averageScore`, `numberOfStudents`, `milesPerHour`

The convention refers to the capitalized letters resembling the humps of a camel. One advantage that camel casing has over underscore casing is that you're not always straining to type the underscore character. Yet another similar convention is upper camel casing, also known as *PascalCase*[1] which is like camel casing, but the first letter in the first word is also capitalized:

`AverageScore`, `NumberOfStudents`, `MilesPerHour`

Each of these conventions is used in various languages in different contexts which we'll explore more fully in subsequent sections (usually underscore lowercasing and camel casing are used to denote variables and functions, PascalCase is used to denote user defined types such as classes or structures, and underscore uppercasing is used to denote static and constant variables). However, for our purposes, we'll use camel casing for variables in our pseudocode.

There are exceptions and special cases to each of these conventions such as when a variable name involves an acronym or a hyphenated word, etc. In such cases sensible extensions or compromises are employed. For example, `xmlString` or `priorityXMLParser` (involving the acronym Extensible Markup Language (XML)) may be used which keep all letters in the acronym consistent (all lower or all uppercase).

In addition to these conventions, there are several best-practice principles when deciding on identifiers.

[1] Rarely this is referred to as DromedaryCase; a Dromedary is an Arabian camel.

- Be descriptive, but not verbose – Use variable names that describe what the variable represents. The examples above, `averageScore`, `numberOfStudents`, `milesPerHour` clearly indicate what the variable is intended to represent. Using good, descriptive names makes your code self-documenting (a reader can make sense of it without having to read extensive supplemental documentation).

  Avoid meaningless variable names such as `value`, `aVariable`, or some cryptic combination of `v10` (its the 10th variable I've used!). Ambiguous variables such as `name` should also be avoided unless the context makes its clear what you are referring to (as when used inside of a Person object).

  Single character variables are commonly used, but used in a context in which their meaning is clearly understood. For example, variable names such as `x`, `y` are okay if they are used to refer to points in the Euclidean plane. Single character variables such as `i`, `j` are often used as index variables when iterating over arrays. In this case, terseness is valued over descriptiveness as the context is very well-understood.

  As a general rule, the more a variable is used, the shorter it should be. For example, the variable `numStudents` may be preferred over the full variable `numberOfStudents`.

- Avoid abbreviations (or at least use them sparingly) – You're not being charged by the character in your code; you can afford to write out full words. Abbreviations can help to write shorter variable names, but not all abbreviations are the same. The word "abbreviation" itself could be abbreviated as "abbr.", "abbrv." or "abbrev." for example. Abbreviations are not always universally understood by all users, may be ambiguous, or non-standard. Moreover, modern IDEs provide automatic code completion, relieving you of the need to type longer variable names. If the abbreviation is well-known or understood from context, then it may make sense to use it.

- Avoid acronyms (or at least use them sparingly) – Using acronyms in variable names come with many of the same problems as abbreviations. However, if it makes sense in the context of your code and has little chance of being misunderstood or mistaken, then go for it. For example, in the context of a financial application, APR (Annual Percentage Rate) would be a well-understood acronym in which case the variable `apr` may be preferred over the longer `annualPercentageRate`.

- Avoid pluralizations, use singular forms – English is not a very consistent language when it comes to rules like pluralizations. For most cases you simply add "s"; for others you add "es" or change the "y" to "i" and add "es". Some words are the same form for singular and plural such as "glasses".[2] Other words completely different forms forms ("focus" becomes "foci"). Still yet there are instances in which *multiple* words are acceptable: the plural of "person" can be "persons" *or* "people".

[2]These are called *plurale tantum* (nouns with no singular form) and *singular tantum* (nouns with no plural form) for you grammarians. Words like "sheep" are *unchanging irregular plurals*; words whose singular and plural forms are the same.

> Avoiding plural forms keeps things simple and consistent: you don't need to be a grammarian in order easily read code. One potential exception to this is when using a collection such as an array to hold more than one element or the variable represents a quantity that is pluralized (as with `numberOfStudents` above).

Though the guidelines above provide a good framework from which to write good variable names, reasonable people can and do disagree on best practice because at some point as you go from generalities to specifics, conventions become more of a matter of personal preference and subjective aesthetics. Sometimes an organization may establish its own coding standards that must be followed which of course trumps any of the guidelines above.

In the end, a good balance must be struck between readability and consistency. Rules and conventions should be followed, until they get in the way of good code that is.

2.2.2. Types

A variable's type (or *data type*) is the characterization of the data that it represents. As mentioned before, a computer only "speaks" in 0s and 1s (binary). A variable is merely a memory location in which a series of 0s and 1s is stored. That binary string could represent a number (either an integer or a floating point number), a single alphanumeric character or series of characters (string), a boolean type or some other, more complex user-defined type.

The type of a variable is important because it affects how the raw binary data stored at a memory location is interpreted. Moreover, some types take a different amount of memory. For example, an integer type could take 32 bits while a floating point type could take 64 bits.

Programming languages may support different types and may do so in different ways. In the next few sections we'll describe some common types that are supported by many languages.

Numeric Types

At their most basic, computers are number crunching machines. Thus, the most basic type of variable that can be used in a computer program is a *numeric type*. There are several numeric types that are supported by various programming languages. The most simple is an *integer* type which can represent whole numbers 0, 1, 2, etc. and their negations, $-1, -2, \ldots$. *Floating point* numeric types represent decimal numbers such as $0.5, 3.14, 4.0$, etc. However, floating point numbers cannot represent *every* real number possible since they use a finite number of bits to represent the number. We will examine this in detail below. For now, let's understand how a computer represents both integers and floating point numbers in memory.

As humans, we "think" in base-10 (decimal) because we have 10 fingers and 10 toes.[3] When we write a number with multiple digits in base-10 we do so using places (ones place, tens place, hundreds place, etc.). Mathematically, a number in base-10 can be broken down into powers of ten; for example:

$$3,201 = 3 \times 10^3 + 2 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$$

In general, any number in base-10 can be written as the summation of powers of 10 multiplied by numbers 0–9,

$$c_k \times 10^k + c_{k-1} \times 10^{k-1} + \cdots c_1 \cdot 10^1 + c_0$$

In binary, numbers are represented in the same way, but in base-2 in which we only have 0 and 1 as symbols. To illustrate, let's consider counting from 0: in base-10, we would count $0, 1, 2, \ldots, 9$ at which point we "carry-over" a 1 to the tens spot and start over at 0 in the ones spot, giving us $10, 11, 12, \ldots, 19$ and repeat the carry-over to 20.

With only two symbols, the carry-over occurs much more frequently, we count $0, 1$ and then carry over and have 10. It is important to understand, this is not "ten": we are counting in base-2, so 10 is actually equivalent to 2 in base-10. Continuing, we have 11 and again carry over, but we carry it over twice giving us 100 (just like we'd carry over twice when going from 99 to 100 in base-10). A full count from 0 to 16 in binary can be found in Table 2.1. In many programming languages, a prefix of `0b` is used to denote a number represented in binary. We use this convention in the table.

Base-10	Binary
0	`0b0`
1	`0b1`
2	`0b10`
3	`0b11`
4	`0b100`
5	`0b101`
6	`0b110`
7	`0b111`
8	`0b1000`
9	`0b1001`
10	`0b1010`
11	`0b1011`
12	`0b1100`
13	`0b1101`
14	`0b1110`
15	`0b1111`
16	`0b10000`

Table 2.1.: Counting in Binary

As a fuller example, consider again the number 3,201. This can be represented in binary as follows.

$$\begin{aligned}
\texttt{0b110010000001} &= 1 \times 2^{11} + 1 \times 2^{10} + 0 \times 2^9 + 0 \times 2^8 + \\
&\quad 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 0 \times 2^4 + \\
&\quad 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 2^{11} + 2^{10} + 2^7 + 2^0 \\
&= 2,048 + 1,024 + 128 + 1 \\
&= 3,201
\end{aligned}$$

[3] At least, the most fortunate among us.

Representing negative numbers is a bit more complicated and is usually done using a scheme called two's complement. We omit the details for now, but essentially the first bit in the representation serves as a *sign bit*: zero indicates positive, while 1 indicates negative. Negative values are represented as a complement with respect to 2^n (a complement is where 0s and 1s are "flipped" to 1s and 0s).

When represented using Two's Complement, binary numbers with n bits can represent numbers x in the range

$$-2^{n-1} \leq x \leq 2^{n-1} - 1$$

Note that the upper bound follows from the fact that

$$\underbrace{\texttt{0b}\,\texttt{11}\ldots\texttt{11}}_{n \text{ bits}} = \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Which captures the idea that we start at zero. The lower bound represents the idea that we have 2^{n-1} possible values ($n-1$ since we need one bit for the sign bit) and we don't need to start at zero, we can start at -1. Table 2.2 contains ranges for common integer types using various number of bits.

n (number of bits)	minimum	maximum
8	-128	127
16	-32,768	32,767
32	-2,147,483,648	2,147,483,647
64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
128	$\approx -3.4028 \times 10^{38}$	$\approx 3.4028 \times 10^{38}$

Table 2.2.: Ranges for various signed integer types

Some programming languages allow you to define variables that are *unsigned* in which the sign bit is not used to indicate positive/negative. With the extra bit we can represent numbers twice as big; using n bits we can represent numbers x in the range

$$0 \leq x \leq 2^n - 1$$

Floating point numbers in binary are represented in a manner similar to scientific notation. Recall that in scientific notation, a number is *normalized* by multiplying it by some power of 10 so that it its most significant digit is between 1 and 9. The resulting normalized number is called the *significand* while the power of ten that the number was scaled by is called the *exponent* (and since we are base-10, 10 is the *base*). In general, a number in scientific notation is represented as:

$$\text{significand} \times \text{base}^{\text{exponent}}$$

For example,

$$14326.123 = \underbrace{1.4326123}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{4}^{\text{exponent}}}$$

Equivalently, sometimes the notations $1.4326123e+4$, $1.4326123e4$ or $1.4326123E4$ are used. As before, we can see that a fractional number in base-10 can be seen as a summation of powers of 10:

$$\begin{aligned} 1.4326123 = &1 \times 10^{1} \quad + 4 \times 10^{-1} + 3 \times 10^{-2} + 2 \times 10^{-3} + \\ &6 \times 10^{-4} + 1 \times 10^{-5} + 2 \times 10^{-6} + 3 \times 10^{-7} \end{aligned}$$

In binary, floating point numbers are represented in a similar way, but the base is 2, consequently a fractional number in binary is a summation of powers of 2. For example,

$$\begin{aligned} 110.011 =& 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ =& 1 \times 4 + 1 \times 2 + 0 \times 1 + 0 \times \frac{1}{2} + 1 \times \frac{1}{4} + 1 \times \frac{1}{8} \\ =& 4 + 2 + 0 + 0 + \frac{1}{4} + \frac{1}{8} \\ =& 6.375 \end{aligned}$$

In binary, the significand is often referred to as a mantissa. We also normalize a binary floating point number so that the mantissa is between $\frac{1}{2}$ and 1. This is where the term *floating point* comes from: the decimal point (more generally called a *radix point*) "floats" left and right to ensure that the number is always normalized. The example above would normalized to

$$0.110011 \times 2^3$$

Here, 0.110011 is the mantissa and 3 is the exponent (which in binary would be `0b11`).

Most modern programming languages implement floating point numbers according to the Institute of Electrical and Electronics Engineers (IEEE) 754 Standard [13] (also called the International Electrotechnical Commission (IEC) 60559 [12]). When represented in binary, the total number of bits must be used to represent the sign, mantissa and exponent. The standard defines several precisions that each use a fixed number of bits with a resulting number of significant digits (base-10) of precision. Table 2.3 contains a summary of a few of the most commonly implemented precisions.

Just as with integers, the finite precision of floating-point numbers results in several limitations. First, irrational numbers such as $\pi = 3.14159\ldots$ can only be approximated out to a certain number of digits. For example, with single precision $\pi \approx 3.1415927$ which is accurate only to the 6th decimal place and with double precision, $\pi \approx 3.1415926535897931$ approximate to only 15 decimal places.[4] In fact, *regardless* of how many bits we allow in

[4] The first 80 digits of π are
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899
though only only 39 digits of π are required to accurately calculate the volume of the known universe to within one atom.

Name	Bits	Exponent Bits	Mantissa Bits	Significant Digits of Precision	Approximate Range
Half	16	5	10	≈ 3.3	$10^3 \sim 10^{4.5}$
Single	32	8	23	≈ 7.2	$10^{-38} \sim 10^{38}$
Double	64	11	52	≈ 15.9	$10^{-308} \sim 10^{308}$
Quadruple	128	15	112	≈ 34.0	$10^{-4931} \sim 10^{4931}$

Table 2.3.: Summary of Floating-point Precisions in the IEEE 754 Standard. Half and quadruple are not widely adopted.

our representation, an irrational number like π (that never repeats and never terminates) will only ever be an approximation. Real numbers like π require an infinite precision, but computers are only finite machines.

Even numbers that have a finite representation (rational numbers) such as $\frac{1}{3} = 0.\overline{333}$ are not represented exactly when using floating-point numbers. In double precision binary,

$$\frac{1}{3} = \texttt{0b1.0101010101010101010101010101010101010101010101010101} \times 2^{-2}$$

which when represented in scientific notation in decimal is

$$3.3333333333333330 \times 10^{-1}$$

That is, there are only 16 digits of precision, after which the remaining (infinite) sequence of 3s get cut off.

Programming languages usually only support the common single and double precisions defined by the IEEE 754 standard as those are commonly supported by hardware. However, there are languages that support arbitrary precision (also called multiprecision) numbers and yet other languages that have many libraries to support "big number" arithmetic. Arbitrary precision is still not infinite: instead, as more digits are needed, more memory is allocated. If you want to compute 10 more digits of π, you can but at a cost. To support the additional digits, more memory is allocated. Also, operations are performed in software using many operations which can be much slower than performing fixed-precision arithmetic directly in hardware. Still, there are many applications where such accuracy or large numbers are absolutely essential.

Characters & Strings

Another type of data is textual data which can either be single characters or a sequence of characters which are called strings. Strings are sometimes used for human readable data such as messages or output, but may model general data. For example, DNA is usually encoded using strings consisting of C, G, A, T (corresponding to the nucleases cytosine, guanine, adenine, and thymine). Numerical characters and punctuation can

also be used in strings in which case they do not represent numbers, but instead may represent textual versions of numerical data.

Different programming languages implement characters and strings in different ways (or may even treat them the same). Some languages implement strings by defining *arrays* of characters. Other languages may treat strings as dynamic data types. However, all languages use some form of *character encoding* to represent strings. Recall that computers only speak in binary: 0s and 1s. To represent a character like the capital letter "A", the binary sequence `0b1000001` is used. In fact, the most common alphanumeric characters are encoded according to the American Standard Code for Information Interchange (ASCII) text standard. The basic ASCII text standard assigns characters to the decimal values 0–127 using 7 bits to *encode* each character as a number. Table 2.4 contains a complete listing of standard ASCII character set.

The ASCII table was designed to enforce a lexicographic ordering: letters are in alphabetic order, uppercase precede lowercase versions, and number precede both. This design allows for an easy and natural comparison among strings, "alpha" would come before "beta" because they differ in the first letter. The characters have numerical values 97 and 98 respectively; since $97 < 98$, the order follows. Likewise, "Alpha" would come before "alpha" (since $65 < 97$), and "alpha" would come before "alphanumeric": the sixth character is empty in the first string (usually treated as the null character with value 0) while it is "n" in the second (value of 110). This is the ordering that we would expect in a dictionary.

There are several other nice design features built into the ASCII table. For example, to convert from upper to lower case versions, you only need to "flip" the second bit (0 for uppercase, 1 for lowercase). There are also several characters that need to be *escaped* to be defined. For example, though your keyboard has a tab and an enter key, if you wanted to code those characters, you would need to specify them in some way other than those keys (since typing those keys will affect what you are typing rather than specifying a character). The standard way to escape characters is to use a backslash along with another, single character. The three most common are the (horizontal) tab, `\t`, the endline character, `\n`, and the null terminating character, `\0`. The tab and endline character are used to specify their whitespace characters respectively. The null character is used in some languages to denote the *end* of a string and is not printable.

ASCII is quite old, originally developed in the early sixties. President Johnson first mandated that all computers purchased by the federal government support ASCII in 1968. However, it is quite limited with only 128 possible characters. Since then, additional extensions have been developed. The Extended ASCII character set adds support for 128 additional characters (numbered 128 through 255) by adding 1 more bit (8 total). Included in the extension are support for common international characters with diacritics such as ü, ñ and £ (which are characters 129, 164, and 156 respectively).

Even 256 possible characters are not enough to represent the wide array of international characters when you consider languages like Chinese, Japanese, and Korean (CJK for

Binary	Dec	Character	Binary	Dec	Character	Binary	Dec	Character
0b000 0000	0	\0 Null character	0b010 1011	43	+	0b101 0110	86	V
0b000 0001	1	Start of Header	0b010 1100	44	,	0b101 0111	87	W
0b000 0010	2	Start of Text	0b010 1101	45	-	0b101 1000	88	X
0b000 0011	3	End of Text	0b010 1110	46	.	0b101 1001	89	Y
0b000 0100	4	End of Transmission	0b010 1111	47	/	0b101 1010	90	Z
0b000 0101	5	Enquiry	0b011 0000	48	0	0b101 1011	91	[
0b000 0110	6	Acknowledgment	0b011 0001	49	1	0b101 1100	92	\
0b000 0111	7	\a Bell	0b011 0010	50	2	0b101 1101	93	]
0b000 1000	8	\b Backspace	0b011 0011	51	3	0b101 1110	94	^
0b000 1001	9	\t Horizontal Tab	0b011 0100	52	4	0b101 1111	95	_
0b000 1010	10	\n Line feed	0b011 0101	53	5	0b110 0000	96	‘
0b000 1011	11	\v Vertical Tab	0b011 0110	54	6	0b110 0001	97	a
0b000 1100	12	\f Form feed	0b011 0111	55	7	0b110 0010	98	b
0b000 1101	13	\r Carriage return	0b011 1000	56	8	0b110 0011	99	c
0b000 1110	14	Shift Out	0b011 1001	57	9	0b110 0100	100	d
0b000 1111	15	Shift In	0b011 1010	58	:	0b110 0101	101	e
0b001 0000	16	Data Link Escape	0b011 1011	59	;	0b110 0110	102	f
0b001 0001	17	Device Control 1	0b011 1100	60	<	0b110 0111	103	g
0b001 0010	18	Device Control 2	0b011 1101	61	=	0b110 1000	104	h
0b001 0011	19	Device Control 3	0b011 1110	62	>	0b110 1001	105	i
0b001 0100	20	Device Control 4	0b011 1111	63	?	0b110 1010	106	j
0b001 0101	21	Negative Ack	0b100 0000	64	@	0b110 1011	107	k
0b001 0110	22	Synchronous idle	0b100 0001	65	A	0b110 1100	108	l
0b001 0111	23	End of Trans. Block	0b100 0010	66	B	0b110 1101	109	m
0b001 1000	24	Cancel	0b100 0011	67	C	0b110 1110	110	n
0b001 1001	25	End of Medium	0b100 0100	68	D	0b110 1111	111	o
0b001 1010	26	Substitute	0b100 0101	69	E	0b111 0000	112	p
0b001 1011	27	Escape	0b100 0110	70	F	0b111 0001	113	q
0b001 1100	28	File Separator	0b100 0111	71	G	0b111 0010	114	r
0b001 1101	29	Group Separator	0b100 1000	72	H	0b111 0011	115	s
0b001 1110	30	Record Separator	0b100 1001	73	I	0b111 0100	116	t
0b001 1111	31	Unit Separator	0b100 1010	74	J	0b111 0101	117	u
0b010 0000	32	(space)	0b100 1011	75	K	0b111 0110	118	v
0b010 0001	33	!	0b100 1100	76	L	0b111 0111	119	w
0b010 0010	34	"	0b100 1101	77	M	0b111 1000	120	x
0b010 0011	35	#	0b100 1110	78	N	0b111 1001	121	y
0b010 0100	36	$	0b100 1111	79	O	0b111 1010	122	z
0b010 0101	37	%	0b101 0000	80	P	0b111 1011	123	{
0b010 0110	38	&	0b101 0001	81	Q	0b111 1100	124	|
0b010 0111	39	’	0b101 0010	82	R	0b111 1101	125	}
0b010 1000	40	(	0b101 0011	83	S	0b111 1110	126	~
0b010 1001	41	)	0b101 0100	84	T	0b111 1111	127	Delete
0b010 1010	42	*	0b101 0101	85	U			

Table 2.4.: ASCII Character Table. The first and second column indicate the binary and decimal representation respectively. The third column visualizes the resulting character when possible. Characters 0–31 and 127 are control characters that are not printable or print whitespace. The encoding is designed to impose a lexicographic ordering: A–Z are in order, uppercase letters precede lowercase letters, numbers precede letters and are also in order.

short). Unicode was developed to solve this problem by establishing a standard encoding that supports 1,112,064 possible characters, though only a fraction of these are actually currently assigned.[5] Unicode is backward compatible, so it works with plain ASCII characters. In fact, the most common encoding for Unicode, UTF-8 uses a *variable* number of bytes to encode characters. 1-byte encodings correspond to plain ASCII, there are also 2, 3, and 4-byte encodings.

In most programming languages, strings literals are defined by using either single or double quotes to *delimit* where the string begins and ends. For example, one may be able to define the string `"Hello World"`: the double quotes are *not* part of the string, but instead specify where the string begins and ends. Some languages allow you to use *either* single or double quotes. PHP for example would allow you to also define the same string as `'Hello World'`. Yet other languages, such as C distinguish the usage of single and double quotes: single quotes are for single characters such as `'A'` or `'\n'` while double quotes are used for full strings such as `"Hello World"`.

In any case, if you want a single or double quote to appear in your string you need to escape it similar to how the tab and endline characters are escaped. For example, in C `'\''` would refer to the single quote character and `"Dwayne \"The Rock\" Johnson"` would allow you to use double quotes within a string. In our pseudocode we'll use the stylized double quotes, "Hello World" in any strings that we define. We will examine string types more fully in Chapter 8.

Boolean Types

A Boolean is another type of variable that is used to hold a truth value, either *true* or *false*, of a logical statement. Some programming languages explicitly support a built-in Boolean type while others implicitly support them. For languages that have explicit types, typical the keywords `true` and `false` are used, but logical expressions can also be evaluated and assigned to Boolean variables.

Some languages do not have an explicit Boolean type and instead support Booleans implicitly, sometimes by using numeric types. For example, in C, *false* is associated with zero while *any* non-zero value is associated with *true*. In either case, Boolean values are used to make decisions and control the flow of operations in a program (see Chapter 3).

Object & Reference Types

Not everything is a number or string. Often, we wish to model real-world entities such as people, locations, accounts, or even interactions such as exchanges or transactions.

[5] As of 2012, 110,182 are assigned to characters, 137,468 are reserved for private use (they are valid characters, but not defined so that organizations can use them for their own purposes), with 2,048 surrogates and 66 non-character control codes. 864,348 are left unassigned meaning that we are well-prepared for encoding alien languages when they finally get here.

Most programming languages allow you to create *user-defined types* by using objects or structures. Objects and structures allow you to group multiple pieces of data together into one logical entity; this is known as encapsulation. For example, a Student object may consist of a first-name, last-name, GPA, year, major, etc. Grouping these separate pieces of data together allows us to define a more complex type. We explore these concepts in more depth in Chapter 10.

In contrast to the built-in numeric, character/string, and Boolean types (also called primitive data types) user-defined types do not necessarily take a fixed amount of memory to represent. Since they are user-defined, it is up to the programmer to specify how they get created and how they are represented in memory. A variable that refers to an object or structure is usually references or pointers: a reference to where the object is stored in memory on a computer.

Many programming languages use the keyword `null` (or sometimes `NULL` or a some variation) to indicate an invalid reference. The `null` keyword is often used to refer to uninitialized or "missing" data.

Another common user-defined type is an *enumerated* type which allows a user to define a *list* of keywords associated with integers. For example, the cardinal directions, "north", "south", "east", and "west" could be associated with the integers 0, 1, 2, 3 respectively. Defining an enumerated type then allows you to use these keywords in your program directly without having to rely on mysterious numerical values, making a program more readable and less prone to error.

2.2.3. Declaring Variables: Dynamic vs. Static Typing

In some languages, variables must be declared before they can be referred to or used. When you declare a variable, you not only give it an identifier, but also define its type. For example, you can declare a variable named $numberOfStudents$ and define it to be an integer. For the life of that variable, it will *always* be an integer type. You can only give that variable integer values. Attempts to assign, say, a string type to an integer variable may either result in a syntax error or a runtime error when the program is executed or lead to unexpected or undefined behavior. A language that requires you to declare a variable and its type is a statically typed language.

The declaration of a variable is typically achieved by writing a statement that includes the variable's type (using a built-in keyword of the language) along with the variable name. For example, in C-style languages, a line like

```
int x;
```

would create an integer variable associated with the identifier `x`.

In other languages, typically *interpreted* languages, you do *not* have to declare a variable before using it. Such languages are generally referred to as dynamically typed languages.

Instead of declaring a variable to have a particular type, the type of a variable is determined by the type of value that is assigned to it. If you assign an integer to a variable it becomes an integer. If you assign a string to it, it becomes a string type. Moreover, a variable's type can *change* during the execution of a program. If you reassign a value to a variable, it dynamically changes its type to match the type of the value assigned.

In PHP for example, a line like

```
$x = 10;
```

would create an integer variable associated with the identifier `$x`. In this example, we did not declare that `$x` was an integer. Instead, it was inferred by the value that we assigned to it (10).

At first glance it may seem that dynamically typed languages are better. Certainly they are more flexible (and allow you to write less so-called "boilerplate" code), but that flexibility comes at a cost. Dynamically typed variables are generally less efficient. Moreover, dynamic typing opens the door to a lot of potential type mismatching errors. For example, you may have a variable that is assumed to *always* be an integer. In a dynamically typed language, no such assumption is valid as any reassignment can change the variable's type. It is impossible to enforce this rule by the language itself and may require a lot of extra code to check a variable's type and deal with "type safety" issues. The advantages and disadvantages of each continue to be debated.

2.2.4. Scoping

The scope of a variable is the section of code in which a variable is valid or "known." In a statically typed language, a variable must be declared before it can be used. The code block in which the variable is declared is therefore its scope. Outside of this code block, the variable is invalid. Attempts to reference or use a variable that is out-of-scope typically result in a syntax error. An example using the C programming language is depicted in Code Sample 2.1.

```
1 {
2   int a;
3   {
4     //this is a new code block inside the outer block
5     int b;
6     //at this point in the code, both a and b are in-scope
7   }
8   //at this point, only a is in-scope, b is out-of-scope
9 }
```

Code Sample 2.1: Example of variable scoping in C

Scoping in a dynamically typed language is similar, but since you don't declare a variable, the scope is usually defined by the block of code where you first use or reference the variable. Moreover, in some languages using a variable may cause that variable to become *globally* scoped.

A globally scoped variable is valid throughout the entirety of a program. A global variable can be accessed and referenced on every line of code. Sometimes this is a good thing: for example, we could define a variable to represent π and then use it anywhere in our program. We would then be assured that every computation involving π would be using the same definition of π (rather than one line of coding using the estimate 3.14 while another uses 3.14159).

On the same token, however, global variables make the state and execution of a program less predictable: if any piece of code can access a global variable, then potentially any piece of code could *change* that variable. Imagine some questionable code changing the value of our global π variable to 3. For this reason, using global variables are generally considered bad practice.[6] Even if no code performs such an egregious operation, the fact that anything *can* change the value means that when testing, you must test for the potential that anything *will* change the value, greatly increasing the complexity of software testing.

To capture the advantages of a global variable while avoiding the disadvantages, it is common to only allow global *constants*; variables whose values cannot be changed once set.

Another argument against globally scoped variables is that once the identifier has been used, it cannot be reused or redefined for other purposes (a floating-point variable with the identifier `pi` means we cannot use the identifier `pi` for any other purpose) as it would lead to conflicts. Defining many globally scoped variables (or functions, or other elements) starts to *pollute the namespace* by reserving more and more identifiers. Problems arise when one attempts to use multiple libraries that have both used the same identifiers for different variables or functions. Resolving the conflict can be difficult or impossible if you have no control over the offending libraries.

2.2.5. Binary Representations

2.3. Operators

Now that we have variables, we need a way to work with variables. That is, given two variables we may wish to add them together. Or we may wish to take two strings and combine them to form a new string. In programming languages this is accomplished through operators which operate on one or more operands. An operator takes the values

[6] Coders often say "globals are evil" and indeed have often demonstrated that they have low moral standards. Global variables that is. Coders are above reproach.

of its operands and combines them or changes them in some way to produce a new value. If an operator is applied to variable(s), then the values used in the operation are the values stored in the variable at the time that the operator is evaluated.

Many common operators are *binary* in that they operate on two operands such as common arithmetic operations like addition and multiplication. Some operators are *unary* in that they only operate on one variable. The first operator that we look at is a unary operator and allows us to assign values to variables.

2.3.1. Assignment Operators

The assignment operator is a unary operator that allows you to take a value and *assign* it to a variable. The assignment operator usually takes the following form: the value is placed on the right-hand-side of the operator while the variable to which we are assigning the value is placed on the left-hand-side of the operator. For our pseudocode, we'll use a generic "left-arrow" notation:

$$a \leftarrow 10$$

which should be read as "place the value 10 into the variable a." Many C-style programming languages commonly use the single equals sign for the assignment operator. The example above might be written as

```
a = 10;
```

It is important to realize that when this notation is used, it is not a declaration like it would be in algebra: $a = b$ for example is an algebraic assertion that the variables a and b are equal. An assignment operator is different: it means place the value on the right-hand-side into the variable on the left-hand-side. For that reason, writing something like

```
10 = a;
```

is invalid syntax. The left-hand-side *must* be a variable.

The right-hand-side, however, may be a literal, another variable, or even a more complex expression. In the example before,

$$a \leftarrow 10$$

the value 10 was acting as a numerical literal: a way of expressing a (human-readable) value that the computer can then interpret as a binary value. In code, we can conveniently write numbers in base-10; when compiled or interpreted, the numerical literals are converted into binary data that the computer understands and placed in a memory location corresponding to the variable. This entire process is automatic and transparent to the user. Literals can also be strings or other values. For example:

$$message \leftarrow \text{“hello world”}$$

We can also "copy" values from one variable to another. Assuming that we've assigned the value 10 to the variable a, we can then copy it to another variable b:

$$b \leftarrow a$$

This does not mean that a and b are the same variable. The value that is stored in the variable a at the time that this statement is executed is *copied* into the variable b. There are now *two* different variables with the same value. If we reassign the value in a, the value in b is unaffected. This is illustrated in Algorithm 2.1

```
1 a ← 10
2 b ← a
  //a and b both store the value 10 at this point
3 a ← 20
  //now a has the value 20, but b still has the value 10
4 b ← 25
  //a still stores a value of 20, b now has a value of 25
```

Algorithm 2.1: Assignment Operator Demonstration

The right-hand-side can also be a more complex expression, for example the result of summing two numbers together.

2.3.2. Numerical Operators

Numerical operators allow you to create complex expressions involving either numerical literals and/or numerical variables. For most numerical operators, it doesn't matter if the operands are integers or floating-point numbers. Integers can be added to floating-point numbers without much additional code for example.

The most basic numerical operator is the unary negation operator. It allows you to negate a numerical literal or variable. For example,

$$a \leftarrow -10$$

or

$$a \leftarrow -b$$

The usage of a negation is so common that it is often not perceived to be an operator but it is.

Addition & Subtraction

You can also add (sum) two numbers using the + (plus) operator and subtract using the − (minus) operator in a straightforward way. Note that most languages can distinguish

the minus operator and the negation operator by how you use it just like a mathematical expression. If applied to one operand, it is interpreted as a negation operator. If applied to two operands, it represents subtraction. Some examples can be found in Algorithm 2.2.

```
1 a ← 10
2 b ← 20
3 c ← a + b
4 d ← a − b
  //c has the value 30 while d has the value −10
5 c ← a + 10
6 d ← −d
  //c now has the value 20 and d now has the value 10
```

Algorithm 2.2: Addition and Subtraction Demonstration

Multiplication & Division

You can also multiply and divide literals and variables. In mathematical expressions multiplication is represented as $a \times b$ or $a \cdot b$ or simply just ab and division is represented as $a \div b$ or a/b or $\frac{a}{b}$. In our pseudocode, we'll generally use $a \cdot b$ and $\frac{a}{b}$, but in programming languages it is difficult to write these some of these symbols. Usually programming languages use $*$ for multiplication and / for division. Similar examples are provided in Algorithm 2.3.

```
1 a ← 10
2 b ← 20
3 c ← a · b
4 d ← a/b
  //c has the value 200 while d has the value 0.5
```

Algorithm 2.3: Multiplication and Division Demonstration

Careful! Some languages specify that the result of an arithmetic operation on variables of a certain type *must* match. That is, an integer plus an integer results in an integer. A floating-point number divided by a floating-point number results a floating-point number. When we mix types, say an integer and a floating-point number, the result is generally a floating-point number. For the most part this is straightforward. The one tricky case is when we have an integer divided by another integer, 3/2 for example.

Since both operands are integers, the result must be an integer. Normally, $3/2 = 1.5$, but since the result must be an integer, the fractional part gets truncated (cut-off) and

only the integral part is kept for the final result. This can lead to weird results such as $1/3 = 0$ and $99/100 = 0$. The result is *not* rounded down or up; instead the fractional part is completely thrown out. Care must be taken when dividing integer variables in a statically typed language. Type casting can be used to force variables to change their type for the purposes of certain operations so that the full answer is preserved. For example, in C we can write

```
int a = 10;
int b = 20;
double c;
int d;
c = (double) a / (double) b;
d = a / b;
//the value in c is correctly 0.5 but the value in d is 0
```

Integer Division

Recall that in arithmetic, when you divide integers a/b, b might not go into a evenly in which case you get a remainder. For example, $13/5 = 2$ with a remainder $r = 3$. More generally we have that

$$a = qb + r$$

Where a is the *dividend*, b is the *divisor*, q is the *quotient* (the result) and r is the *remainder*. We can also perform *integer division* in most programming languages. In particular, the integer division operator is the operator that gives us the *remainder* of the integer division operation in a/b. In mathematics this is the *modulo* operator and is denoted

$$a \bmod b$$

For example,

$$13 \bmod 5 = 3$$

It is possible that the remainder is zero, for example,

$$10 \bmod 5 = 0$$

Many programming languages support this operation using the percent sign. In C for example,

```
c = a % b;
```

2.3.3. String Concatenation

Strings can also be combined to form new strings. In fact, strings can also be combined with non-string variables to form new strings. You would typically do this in order to

convert a numerical value to a string representation so that it can be output to the user or to a file for longterm storage. The operation of combining strings is referred to as string concatenation. Some languages support this through the same plus operator that is used with addition. For example,

$$message \leftarrow \text{“hello ”} + \text{“world!”}$$

which combines the two strings to form one string containing the characters “hello world!”, storing the value into the *message* variable. For our pseudocode we'll adopt the plus operator for string concatenation.

The string concatenation operator can also sometimes be combined with non-string types; numerical types for example. This allows you to easily convert numbers to a human-readable, base-10 format so that they can be printed to the output. For example suppose that the variable b contains the value 20, then

$$message \leftarrow \text{“the answer is”} + b$$

might result in the string “the answer is 20” being stored in the variable *message*.

Other languages use different symbols to distinguish concatenation and addition. Still yet other languages do not directly support an operator for string concatenation which must instead be done using a function.

2.3.4. Order of Precedence

In mathematics, when you write an expression such as:

$$a + b \cdot c$$

you interpret it as “multiply b and c and then add a.” This is because multiplication has a higher order of precedence than addition. The order of precedence (sometimes referred to as *order of operations*) is a set of rules which define the order in which operations should be evaluated. In this case, multiplication is performed before addition. If, instead, we had written

$$(a + b) \cdot c$$

we would have a different interpretation: “add a and b and then multiply the result by c.” That is, the inclusion of parentheses *changes* the order in which we evaluate the operations. Adding parentheses can have no effect (if we wrote $a + (bc)$ for example), or it can cause operations with a lower order of precedence to be evaluated first as in the example above.

Numerical operators are no different when used in most programming languages. The same order of precedence is used and parentheses can be used to change the order of evaluation.

2.3.5. Common Numerical Errors

When dealing with numeric types its important to know and understand their limitations. In mathematics, the following operations might be considered "invalid"

- Division by zero: $\frac{a}{b}$ where $b = 0$. This is an undefined operation in mathematics and also in programming languages. Depending on the language, any number of things may happen. It may be a fatal error or *exception*; the program may continue executing but give "garbage" results from then on; the result may be a special value such as null, "NaN" (not-a-number) or "INF" (a special representation of infinity). It is best to avoid such an operation entirely using conditionals statements and defensive programming (see Chapter 3)
- Other potentially invalid operations involve common mathematical functions. For example, $\sqrt{-1}$ would be a complex result, i which some languages do support. However, many do not. Similarly, the natural logarithm of zero and negative values is undefined. In either case you could expect a result like "NaN" or "INF".
- Still other operations seem like they should be valid, but because of how numbers are represented in binary, the results are invalid. Recall that for a 32-bit signed, two's complement number, the maximum representable value is 2,147,483,647. Suppose this maximum value is stored in a variable, b. Now suppose we attempt to add one more,

  $$c \leftarrow b + 1$$

  Mathematically we'd expect the result to be 2,147,483,648, but that is more than the maximum representable integer. What happens is something called arithmetic overflow. The actual number stored in binary in memory for 2,147,483,647 is

  $$\texttt{0b0}\underbrace{\texttt{11...11}}_{\text{31 1s}}$$

  When we add 1 to this, it is carried over all the way to the 32nd bit, giving the result

  $$\texttt{0b1}\underbrace{\texttt{00...00}}_{\text{31 0s}}$$

  in binary. However, the 32nd bit is the sign bit, so this is a negative number. In particular, if this is a two's complement integer, it has the decimal value $-2,147,483,648$ which is obviously wrong. Another example would be if we have a "large number, say 2 billion and attempt to double it (multiply by 2). We would expect 4 billion as a result, but again overflow occurs and the result (using 32-bit signed two's complement integers) is $-294,967,296$. do support functions.
- A similar phenomenon can happen with floating point numbers. If an operation (say multiplying two "small" numbers together) results in a number that is smaller than the smallest floating-point number that can be represented, the result is said to have resulted in underflow. The result can essentially be zero, or an error can

be raised to indicate that underflow has occurred. The consequences of underflow can be very complex.

- Floating-point operations can also result in a loss of precision even if no overflow or underflow occurs. For example, when adding a very large number a and a very small number b, the result might be no different from the value of a. This is because (for example) double precision floating-point numbers only have about 16 significant digits of precision with the least significant digits being cutoff in order to preserve the magnitude.

  As another example, suppose we compute $\sqrt{2} = 1.41421356\ldots$. If we squared the result, mathematically we would expect to get 2. However, since we only have a certain number of digits of precision, squaring the result in a computer may result in a value slightly different from 2 (either 1.9999998 or 2.0000001).

2.3.6. Other Operators

Many programming languages support other "convenience" operators that allow you to perform common operations using less code. These operators are generally syntactic sugar the don't add any functionality (the same operation could be achieved using other operators), but they add simpler or more terse syntax for doing so.

Increment Operators

Adding or subtracting one to a variable is a very common operation. So common, that most programming languages define increment operators such as `i++` and `i--` which add one and subtract one from the variables applied. The same effect could be achieved by writing

$$i \leftarrow (i + 1) \quad \text{and } i \leftarrow (i - 1)$$

but the increment operators provide a shorthand way of expressing the operation.

The operators `i++` and `i--` are *postfix* operators: the operator is written *after* (post) the operand. Some languages define similar *prefix* increment operators, `++i` and `--i`. The effect is similar: each adds or subtracts one from the variable `i`. However, the difference is when the operator is used in a larger expression. A postfix operator *retains* the original value for the expression, a prefix operator takes on the new, incremented value in the expression.

To illustrate, suppose the variable `i` has the value 10. In the following line of code, `i` is incremented and used in an expression that adds 5 and stores the result in a variable `x`:

```
x = 5 + (i++);
```

The value of `x` after this code is 15 while the value of `i` is now 11. This is because the postfix operator increments `i`, but `i++` retains the value 10 in the expression. In contrast,

with the line

```
x = 5 + (++i);
```

the variable `i` again now has the value 11, but the value of `x` is 16 since `++i` takes on the new, incremented value of 11.

Appropriately using each can lead to some very concise code, but it is important to remember the difference.

Compound Assignment Operators

If we want to increment or decrement a variable by an amount other than 1 we can do so using *compound assignment* operators that combine an arithmetic operator and an assignment operator into one. For example, `a += 10` would add 10 to the variable `a`. The same could be achieved by coding `a = a + 10`, but the former is a bit shorter as we don't have to repeat the variable.

You can do the same with subtraction, multiplication, and division. More examples using the C programming language can be found in Code Snippet 2.2. It is important to note that these operators are *not*, strictly speaking, equivalent. That is `a += 10` is not equivalent to `a = a + 10`. They have the same effect, but the first involves only *one* operator while the second involves *two* operators.

```c
int a = 10;
a += 5; //adds 5 to a
a -= 3; //subtracts 3 from a
a *= 2; //multiplies a by 2
a /= 4; //divides a by 4

//you can also use compound assignment operators with variables:
int b = 5;
a += b; //adds the value stored in b to a
a -= b; //subtracts the value stored in b from a
a *= b; //multiplies a by b
a /= b; //divides a by b
```

Code Sample 2.2: Compound Assignment Operators in C

2.4. Basic Input/Output

Not all variables can be coded using literals. Sometimes a program needs to read in values as input from a user who can give different values on different runs of a program.

Likewise, a computer program often needs to produce output to the user to be of any use.

The most basic types of programs are interactive programs that interact with a human user. Generally, the program may interactive the user to enter some input value(s) or make some choices. It may then compute some values and respond to the user with some output.

In the following sections we'll overview the various types of input and output or I/O for short that are available.

2.4.1. Standard Input & Output

The standard input (stdin for short), standard output (stdout) and standard error (stderr) are three standard communication *streams* that are defined by most computer systems.

Though perhaps an over simplification, the keyboard usually serves as a standard input device while the monitor (or the system *console* or shell program) serves as a standard output device. The standard error is usually displayed in the same display but may be displayed differently on some systems (it is typeset in red in some consoles that support color to indicate that the output is communicating an error).

As a program is executing, it may prompt a user to enter input. A program may wait (called blocking) until a user has typed whatever input they want to provide. The user typically hits the enter key to indicate their input is done and the program resumes, reading the input provided via the standard input. The program may also produce output which is displayed to the user.

The standard input and output are generally universal: almost any language, and operating system will support them and they are the most basic types of input/output. However, the type of input and output is somewhat limited (usually limited to text-based I/O) and doesn't provide much in the way of input validation. As an example, suppose that a program prompts a user to enter a number. Since the input device (keyboard) is does not really restrict the user, a more obstinate user may enter a non-numeric value, say "hello". The program may crash or provide garbage output with such input.

2.4.2. Graphical User Interfaces

A much more user-oriented way of reading input and displaying output is to use a Graphical User Interface (GUI). GUIs can be implemented as traditional "thick-client" applications (programs that are installed locally on your machine) or as "thin-client" applications such as a web application. They typically support general "widgets" such as input boxes, buttons, sliders, etc. that allow a user to interact with the program in a more visual way. They also allow the programmer to do better input validation. Widgets

Language	Standard Output	String Output
C	`printf`	`sprintf`
Java	`System.out.printf`	`String.format`
PHP	`printf`	`sprintf`

Table 2.5.: `printf`-style Methods in Several Languages. Languages support formatting directly to the Standard Output as well as to strings that can be further used or manipulated. Most languages also support `printf`-style formatting to other output mechanisms (streams, files, etc.).

could be design so that *only* good input is allowed by creating *modal* restrictions: the user is only allowed to select one of several "radio" buttons for example. GUIs also support visual feedback cues to the user: popups, color coding, and other elements can be used to give feedback on errors and indicate invalid selections.

Graphical user interfaces can also make use of more modern input devices: mice, touch screens with gestures, even gaming devices such as the Kinect allow users to use a full body motion as an input mechanism. We discuss GUIs in more detail in Chapter 13. To begin, we'll focus more on plain textual input and output.

2.4.3. Output Using `printf`-style Formatting

Recall that many languages allow you to concatenate a string and a non-string type in order to produce a string that can then, say, be output to the standard output. Concatenation doesn't provide much in the way of customizability when it comes to *formatting* output. We may want to format a floating-point number so that it only prints to two decimal places (as with US currency). We may want to align a column of data so that number places match up. Or we may want to *justify* text either left or right.

Such data formatting can be achieved through the use of a `printf`-style formatting function. The ideas date back to the mid-60s, but the modern `printf` comes from the C programming language. Numerous programming languages support this style of formatted output (`printf` stands for **print f**ormatted). Most support either printing the resulting formatted output to the standard output as well as to strings and other output mechanisms (files, streams, etc.). Table 2.5 contains a small sampling of `printf`-style supported in several languages. We'll illustrate this usage using the C programming language for our examples, but the concepts are generally universal across most languages.

The function works by providing it a number of *arguments*. The first argument is always a string that specifies the formatting of the result using several *placeholders* (flags that begin with a percent sign) which will be replaced with values stored in variables but in a formatted manner. Subsequent arguments to the function are the list of variables to be printed; each argument is delimited by a comma. Figure 2.3 gives an example of of a `printf` statement with two placeholders. The placeholders are ultimately replaced with

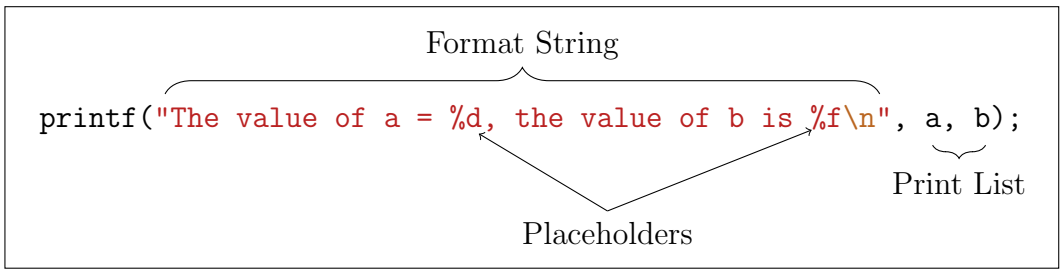


Figure 2.3.: Elements of a `printf` statement in C

the values stored in the provided variables a, b. If a, b held the values 10 and 2.718281, the code would end up printing

```
The value of a = 10, the value of b is 2.718281
```

Though there are dozens of placeholders that are supported, we will focus only on a few:

- `%d` formats an integer variable or literal
- `%f` formats a floating-point variable or literal
- `%c` formats a single character variable or literal
- `%s` formats a string variable or literal

Misuse of placeholders may result in garbage output. For example, using an integer placeholder, `%d`, but providing a string argument; since strings cannot be (directly) converted to integers, the output will not be correct.

In addition to these placeholders, you can also add *modifiers*. A number n between the percent sign and character (`%`n`d`, `%`n`f`, `%`n`s`)) specifies that the result should be formatted with a *minimum* of n columns. If the output takes less than n columns, `printf` will pad out the result with spaces so that there are n columns. If the output takes n or more columns, then the modifier will have no effect (it specifies a *minimum* not a maximum.

Floating-point numbers have a second modifier that allows you to specify the number of digits of precision to be formatted. In particular, you can use the placeholder `%`n`.`m`f` in which n has the same meaning, but m specifies the number of decimals to be displayed. By default, 6 decimals of precision are displayed. If m is greater than the precision of the number, zeros are usually used for subsequent digits; if m is smaller than the precision of the number, rounding *may* occur. Note that the n modifier *includes* the decimal point as a column. Both modifiers are optional.

Finally, each of these modifiers can be made negative (example: `%-20d`) to *left-justify* the result. By default, justification is to the right. Several examples are illustrated in Code Sample 2.3 with the results in Code Sample 2.4.

```c
int a = 4567;
double b = 3.14159265359;

printf("a=%d\n", a);
printf("a=%2d\n", a);
printf("a=%4d\n", a);
printf("a=%8d\n", a);

//by default, prints 6 decimals of precision
printf("b=%f\n", b);
//the .m modifier is optional:
printf("b=%10f\n", b);
//the n modifier is also optional:
printf("b=%.2f\n", b);
//note that this rounds!
printf("b=%10.3f\n", b);
//zeros are added so that 15 decimals are displayed
printf("b=%20.15f\n", b);
```

Code Sample 2.3: `printf` examples in C

2.4.4. Command Line Input

Not all programs are interactive. In fact, the vast majority of software is developed to interact with other software and does not expect that a user is sitting at the console constantly providing it with input. Most languages and operating systems support non-interactive input from the Command Line Interface (CLI). This is input that is provided at the command line when the program is executed. Input provided from the command line are usually referred to as *command line arguments*. For example, if we invoke a program named `myProgram` from the command line prompt using something like the following:

```
~>./myProgram a 10 3.14
```

Then we would have provided 4 command line arguments. The first argument is usually the program's name, all subsequent arguments are separated by whitespace. Command line arguments are provided to the program as strings and it is the program's responsibility to convert them if needed and to validate them to ensure that the correct expected number and type of arguments are were provided.

Within a program, command line arguments are usually referred to as an argument vector (sometimes in a variable named `argv`) and argument count (sometimes in a variable named `argc`). We explore how each language supports this in subsequent sections.

```
a=4567
a=4567
a=4567
a=␣␣␣␣4567
b=3.141593
b=␣␣3.141593
b=3.14
b=␣␣␣␣␣3.142
b=␣␣␣3.141592653590000
```

Code Sample 2.4: Result of Computation in Code Sample 2.3. Spaces are highlighted for clarity.

2.5. Debugging

Making mistakes in programming is inevitable. Even the most expert of software developers make mistakes.[7] An error in computer programs are usually referred to as bugs. The term was popularized by Grace Hopper in 1947 while working on a Mark II Computer at a US Navy research lab. Literally, a moth stuck in the computer was impeding its operation, removing it or "debugging" the computer fixed it. In this section will will identify general types of errors and outline ways to address them.

2.5.1. Types of Errors

When programming there are several types of errors that can occur, some can be easily detected (or even easily fixed) by compilers and other modern code analysis tools such as IDEs.

Syntax Errors

Syntax errors are errors in the usage of a programming language itself. A syntax error can be a failure to adhere to the rules of the language such as misspelling a keyword or forgetting proper "punctuation" (such as missing an ending semicolon). When you have a syntax error, you're essentially not "speaking the same language": You wouldn't be very comprehensible if you started injecting non-sense words or words from different language when speaking to someone in English. Similarly, a computer can't understand what you're trying to say (or what directions you're trying to give it) if you're not speaking the same language.

[7] A severe security bug in the popular unix bash shell utility went undiscovered for 25 years before it was finally fixed in September 2014, missed by thousands of experts and some of the best coders in the world.

Typically syntax errors prevent you from even compiling a program, though syntax errors can be a problem at runtime with interpreted languages. When it encounters a syntax error, a compiler will fail to complete the compilation process and will generally quit. Ideally, the compiler will give reasons for why it was unable to compile and will hopefully identify the line number where the syntax error was encountered with a hint on what was wrong. Unfortunately, many times a compiler's error message isn't too helpful or may indicate a problem on one line where the root cause of the problem is earlier in the program. One cannot expect too much from a compiler after all. If a compiler were able to correctly interpret and fix our errors for us, we'd have "natural language" programming where we could order the computer to execute our commands in plain English. If we had this science fiction-level of computer interaction we wouldn't need programming languages at all.

Fixing syntax errors involves reading and interpreting the compiler error messages, reexamining the program and fixing any and all issues to conform to the syntax of the programming language. Fixing one syntax error may enable the compiler to find additional syntax errors that it had not before. Only once all syntax errors have been resolved can a program actually compile. For interpreted languages, the program may be able to run up to where it encounters a syntax error and then exits with a fatal error. It may take several runs to resolve such errors as well.

Runtime Errors

Once a program is free of syntax errors it can compile and be run. However, that doesn't mean that the program is completely free of bugs, just that it is free of the types of bugs (syntax errors) that the compiler is able to detect. A compiler is not able to predict every action or possible event that could occur when a program is actually run. A runtime error is an error that occurs while a program is being executed. For example, a program could attempt to access a file that does not exist, or attempt to connect to a database, but the computer has lost its network connection, or a user could enter bad data that results in an invalid arithmetic operation, etc.

A compiler cannot be expected to detect such errors because by definition, the conditions under which runtime errors occur occur *at runtime*, not at compile time. One run of a program could execute successfully, while another subsequent run could fail because the system conditions have changed. That doesn't mean that we should mitigate the consequences of runtime errors.

As a programmer it is important to think about the potential problems and runtime errors that could occur and make contingency plans accordingly. We can make reasonable assumptions that certain kinds of errors may occur in the execution of our program and add code to *handle* those errors if the occur. This is known as *error handling* (which we discuss in detail in Chapter 6). For example, we could add code that checks if a user enters bad input and then re-prompt them to enter good data. If a file is missing, we

could add code to create it as needed. By checking for these errors and preventing illegal, potentially fatal operations, we practice defensive programming.

Logic Errors

Other errors may be a result of bad code or bad design. Computer do exactly as they are told to do. Logic errors can occur if we tell the computer to do something that we didn't intend for them to do. For example, if we tell the computer to execute command A under condition X, but we meant to have the computer execute command B under condition Y, we have caused a logical error. The computer will perform the first set of instructions, not the second as we intended. The program may be free of syntax errors and may execute without any problems, but we certainly don't get the *results* that we expected.

Logic errors are generally only detected and addressed by rigorous *software testing*. When developing software, we can also design a collection of *test cases*: a set of inputs along with correct outputs that we would expect the program of code to produce. We can then test the program with these inputs to see if they produce the same output as in the test cases. If they don't, then we've uncovered a logical error that needs to be addressed.

Rigorous testing can be just as complex (or even *more* complex) than writing the program itself. Testing along cannot guarantee that a program is free of bugs (in general, the number of possible inputs is *infinite*; it is *impossible* to test all possibilities). However, the more test cases that we design and pass the higher the confidence we have that the program is correct.

Testing can also be very tedious. Modern software engineering techniques can help streamline the process. Many testing frameworks have been developed and built that attempt to automate the testing process. Test cases can be randomly generated, test suites can be repeatedly run and verified throughout the development process. Frameworks can perform *regression testing* to see if fixing one bug caused another, etc.

2.5.2. Strategies

A common beginner's way of debugging a program is to insert temporary print statements throughout their program to see what values variables have at certain points in an attempt to isolate where an error is occurring. This is an okay strategy for extremely simple programs, but its the "poor man's" way of debugging. As soon as you start writing more complex programs you quickly realize that this strategy is slow, inefficient, and can actually hide the real problems (the standard output is not guaranteed to work as expected if an error has occurred, so print statements may actually mislead you into thinking the problem occurs at one point in the program when it actually occurs in a different part).

Instead, it is much better to use a proper *debugging tool* in order to isolate the problem. A debugger is another program, that allows you to "simulate" an execution of your program. You can set *break points* in your program on certain lines and the debugger will execute your program up to those points. It then pauses and allows you to look at the program's state: you can examine the contents of memory, look at the values stored in certain variables, etc. Debuggers will also allow you to resume the execution of your program to the next break point or allow you to "step" through your program line by line. This allows you to examine the execution of a program at human speed in order to diagnose the exact point in execution where the problem occurs. IDEs allow you to do this visually with a graphical user interface and easy visualization of variables. However, there are command line debuggers such as GDB (GNU's Not Unix! (GNU) Debugger) that you interact with using text commands.

In general, debugging strategies attempt to isolate a problem to as small of a code segment as possible. For this reason, it is good practice to design code into as small of segments as possible using good procedural abstraction and use of functions and methods (see Chapter 5) and creating test cases and suites for these small pieces of code.

It can also help to diagnose a problem by looking at the nature of the failure. If some test cases pass and others fail you can get a hint as to what's wrong by examining the key differences between the test cases. If one value passes and another fails, you can trace that value as it propagates through the execution of your program to see how it affects other value.

In the end, good debugging skills, just like good coding skills, come from experience. A seasoned expert may be able to look at an error message and immediately diagnose the problem. Or, a bug can escape the detection of hundreds of the best developers and software tools and end up costing millions of dollars and thousands of man-hours because of a simple failure to convert from English units to metric (in 1999, the Mars Climate Orbiter broke up in the atmosphere of Mars because one subsystem was computing force using pound-seconds while everything else was computing in newton-seconds leading to a miscalculation of orbital entry [1]).

2.6. Examples

Let's apply these concepts by developing several prompt-and-compute style programs. That is, the programs will prompt the user for input, perform some calculations, and then output a result.

To write these programs, we'll use pseudocode, an informal, abstract description of a program/algorithm. Pseudocode does not use any language-specific syntax. Instead, it describes processes at a high-level, making use of plain English and mathematical notation. This allows us to focus on the actual process/program rather than worrying about the particular syntax of a specific language. Good pseudocode should be easily

translated into any programming language.

2.6.1. Temperature Conversion

Temperature can be measured in several different scales. The most common for everyday use is Celsius and Fahrenheit. Let's write a program to convert from Fahrenheit to Celsius using the following formula:

$$C = \frac{5}{9} \cdot (F - 32)$$

The basic outline of the program will be three simple steps:

1. Read in a Fahrenheit value from the user
2. Compute a Celsius value using the formula above
3. Output the result to the user

This is actually pretty good pseudocode already, but let's be a little more specific using some of the operators and notation we've established above. The full program can be found in Algorithm 2.4.

```
1 prompt the user to enter a temperature in Fahrenheit
2 F ← read input from user
3 C ← 5/9 · (F − 32)
4 Output C to the user
```

Algorithm 2.4: Temperature Conversion Program

2.6.2. Quadratic Roots

A common math exercise is to find the *roots* of a quadratic equation with coefficients, a, b, c,

$$ax^2 + bx + c = 0$$

using the quadratic formula,

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Following the same basic outline, we'll read in the coefficients from the user, compute each of the roots, and output the results to the user. Here, however, we may two computations, one for each of the roots which we label r_1, r_2. The full procedure is presented in Algorithm 2.5.

1 prompt the user to enter a
2 $a \leftarrow$ read input from user
3 prompt the user to enter b
4 $b \leftarrow$ read input from user
5 prompt the user to enter c
6 $c \leftarrow$ read input from user
7 $r_1 \leftarrow \frac{-b+\sqrt{b^2-4ac}}{2a}$
8 $r_2 \leftarrow \frac{-b-\sqrt{b^2-4ac}}{2a}$
9 Output "the roots of $ax^2 + bx + c$ are r_1, r_2"

Algorithm 2.5: Quadratic Roots Program

2.7. Exercises

Exercise 2.1. Write a program that calculates mileage deduction for income tax using the standard rate of $0.575 per mile. Your program will read in a beginning and ending odometer reading and calculate the difference and total deduction. Take care that your output is in whole cents. An example run of the program may look like the following.

```
INCOME TAX MILEAGE CALCULATOR
Enter beginning odometer reading--> 13505.2
Enter ending odometer reading--> 13810.6
You traveled 305.4 miles.  At $.575 per mile,
your reimbursement is $175.61
```

Exercise 2.2. Write a program to compute the total "cost" C of a loan. That is, the total amount of interest paid over the life of a loan. To compute this value, use the following formula.

$$C = \frac{p \cdot i \cdot (1+i)^{12n}}{(1+i)^{12n} - 1} * 12n - p$$

where

- p is the starting principle amount
- $i = \frac{r}{12}$ where r is the APR on the interval $[0, 1]$
- n is the number of years the loan is to be paid back

Exercise 2.3. Write a program to compute the annualized appreciation of an asset (say a house). The program should read in a purchase price p, a sale price s and compute their difference $d = s - p$ (it should support a loss or gain). Then, it should compute an appreciation rate: $r = \frac{d}{p}$ along with an (average) *annualized appreciation rate* (that is,

what was the appreciation rate in each year that the asset was held that compounded):

$$(1+r)^{\frac{1}{y}} - 1$$

Where y is the number of years (possibly fractional) the asset was held (and r is on the scale $[0, 1]$).

Exercise 2.4. The annual percentage yield (APY) is a much more accurate measure of the true cost of a loan or savings account that compounds interest on a monthly or daily basis. For a large enough number of compounding periods, it can be calculated as:

$$APY = e^i - 1$$

where i is the nominal interest rate ($6\% = 0.06$). Write a program that prompts the user for the nominal interest rate and outputs the APY.

Exercise 2.5. Write a program that calculates the speed of sound (v, feet-per-second) in the air of a given temperature T (in Fahrenheit). Use the formula,

$$v = 1086\sqrt{\frac{5T + 297}{247}}$$

Be sure your program does not lose the fractional part of the quotient in the formula shown and format the output to three decimal places.

Exercise 2.6. Write a program to convert from radians to degrees using the formula

$$deg = \frac{180 \cdot rad}{\pi}$$

However, radians are on the scale $[0, 2\pi)$. After reading input from the user be sure to do some error checking and give an error message if their input is invalid.

Exercise 2.7. Write a program to compute the Euclidean Distance between two points, (x_1, y_2) and (x_2, y_2) using the formulate:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Exercise 2.8. Write a program that will compute the value of $\sin(x)$ using the first 4 terms of the Taylor series:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

In addition, your program will compute the *absolute* difference between this calculation and a standard implementation of the sine function supported in your language. Your program should prompt the user for an input value x and display the appropriate output. Your output should looks *something* like the following.

```
Sine Approximation
==================
Enter x: 1.15
Sine approximation: 0.912754
Sine value:         0.912764
Difference:         0.000010
```

Exercise 2.9. Write a program to compute the roots of a quadratic equation:

$$ax^2 + bx + c = 0$$

using the well-known quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Your program will prompt the user for the values, a, b, c and output each real root. However, for "invalid" input ($a = 0$ or values that would result in complex roots), the program will instead output a message that informs the user why that the inputs are invalid (with a specific reason).

Exercise 2.10. One of Ohm's laws can be used to calculate the amount of *power* in Watts (the rate of energy conversion; 1 joule per second) in terms of Amps (a measure of current, 1 amp $= 6.241 \times 10^{18}$ electrons per second) and Ohms (a measure of electrical resistance). Specifically:

$$W = A^2 \cdot O$$

Develop a simple program to read in two of the terms from the user and output the third.

Exercise 2.11. Ohm's Law models the current through a conductor as follows:

$$I = \frac{V}{R}$$

where V is the voltage (in volts), R is the resistance (in Ohms) and I is the current (in amps). Write a program that, given two of these values computes the third using Ohm's Law.

The program should work as follows: it prompts the user for units of the first value: the user should be prompted to enter `V`, `R`, or `I` and should then be prompted for the value. It should then prompt for the second unit (same options) and then the value. The program will then output the third value depending on the input. An example run of the program:

```
Current Calculator
==============
Enter the first unit type (V, R, I): V
Enter the voltage: 25.75
Enter the second unit type (V, R, I): I
Enter the current: 72
The corresponding resistance is 0.358 Ohms
```

Exercise 2.12. Consider the following linear system of equations in two unknowns:

$$\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}$$

Write a program that prompts the user for the coefficients in such a system (prompt for a, b, c, d, e, f). Then output a solution to the system (the values for x, y). Take care to handle situations in which the system is *inconsistent*.

Exercise 2.13. The surface area of a sphere of radius r is

$$4\pi r^2$$

and the volume of a sphere with radius r is

$$\frac{4}{3}\pi r^3$$

Write a program that prompts the user for a radius r and outputs the surface area and volume of the corresponding sphere. If the radius entered is invalid, print an error message and exit. Your output should look something like the following.

```
Sphere Statistics
=================
Enter radius r: 2.5
area: 78.539816
volume: 65.449847
```

Exercise 2.14. Write a program that prompts for the latitude and longitude of two locations (an origin and a destination) on the globe. These numbers are in the range $[-180, 180]$ (negative values correspond to the western and southern hemispheres). Your program should then compute the air distance between the two points using the Spherical Law of Cosines. In particular, the distance d is

$$d = \arccos\left(\sin(\varphi_1)\sin(\varphi_2) + \cos(\varphi_1)\cos(\varphi_2)\cos(\Delta)\right) \cdot R$$

- φ_1 is the latitude of location A, φ_2 is the latitude of location B
- Δ is the difference between location B's longitude and location A's longitude
- R is the (average) radius of the earth, 6,371 kilometers

Note: the formula above assumes that latitude and longitude are measured in radians r, $-\pi \leq r \leq \pi$. See Exercise 2.6 for how to convert between them. Your program output should look something like the following.

```
City Distance
=======================
Enter latitude of origin: 41.9483
Enter longitude of origin: -87.6556
Enter latitude of destination: 40.8206
Enter longitude of destination: -96.7056
Air distance is 764.990931
```

Exercise 2.15. Write a program that prompts the user to enter in a number of days. Your program should then compute the number of years, weeks, and days that number represents. For this exercise, ignore leap years (thus all years are 365 days). Your output should look something like the following.

```
Day Converter
=============
Enter number of days: 1000
That is
  2 years
  38 weeks
  4 days
```

Exercise 2.16. The derivative of a function $f(x)$ can be estimated using the difference function:

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

That is, this gives us an estimate of the slope of the tangent line at the point x. Write a program that prompts the user for an x value and a Δx value and outputs the value of the difference function for all three of the following functions:

$$\begin{aligned} f(x) &= x^2 \\ f(x) &= \sin(x) \\ f(x) &= \ln(x) \end{aligned}$$

Your output should looks something like the following.

```
Derivative Approximation
==================
Enter x: 2
Enter delta-x: 0.1
(x^2)'  ~= 4.100000
sin'(x) ~= -0.460881
ln'x(x) ~= 0.487902
```

In addition, your program should check for invalid inputs: Δx cannot be zero, and $\ln(x)$ is undefined for $x \leq 0$. If given invalid inputs, appropriate error message(s) should be output instead.

Exercise 2.17. Write a program that prompts the user to enter two points in the plane, (x_1, y_1) and (x_2, y_2) which define a line segment ℓ. Your program should then compute and output an equation for the perpendicular line intersecting the *midpoint* of ℓ. You should take care that invalid inputs (horizontal or vertical lines) are handled appropriately. An example run of your program would look something like the following.

```
Perpendicular Line
====================
Enter x1: 2.5
Enter y1: 10
Enter x2: 3.5
Enter y2: 11
Original Line:
  y = 1.0000 x + 7.5000
Perpendicular Line:
  y = -1.0000 x + 13.5000
```

Exercise 2.18. Write a program that computes the total for a bill. The program should prompt the user for a sub-total. It should then prompt whether or not the customer is entitled to an employee discount (of 15%) by having them enter 1 for yes, 2 for no. It should then compute the new sub-total and apply a 7.35% sales tax, and print the receipt details along with the grand total. Take care that you properly round each operation.

An example run of the program should look something like the following.

```
Please enter a sub-total: 100
Apply employee discount (1=yes, 2=no)? 1

Receipt
========================
Sub-Total   $  100.00
Discount    $   15.00
Taxes       $    6.25
Total       $   91.25
```

Exercise 2.19. The ROI (Return On Investment) is computed by the following formula:

$$\text{ROI} = \frac{\text{Gain from Investment} - \text{Cost of Investment}}{\text{Cost of Investment}}$$

Write a program that prompts the user to enter the cost and gain (how much it was sold for) from an investment and computes and outputs the ROI. For example, if the user enters \$100,000 and \$120,000 respectively, the output look similar to the following.

```
Cost of Investment: $100000.00
Gain of Investment: $120000.00
Return on Investment: 20.00%
```

Exercise 2.20. Write a program to compute the real cost of driving. Gas mileage (in the US) is usually measured in miles per gallon but the real cost should be measured in how much it costs to drive a mile, that is, dollars per mile. Write a program to assist a user in figuring out the real cost of driving. Prompt the user for the following inputs.

- Beginning odometer reading

- Ending odometer reading
- Number of gallons it took to fill the tank
- Cost of gas in dollars per gallon

For example, if the user enters 50,125, 50,430, 10 (gallons), and $3.25 (per gallon), then your output should be something like the following.

```
Miles driven: 305
Miles per gallon: 30.50
Cost per mile: $0.11
```

Exercise 2.21. A *bearing* can be measured in degrees on the scale of $[0, 360)$ with 0° being due north, 90° due east, etc. The (initial) directional bearing from location A to location B can be computed using the following formula.

$$\theta = \operatorname{atan2}\big(\sin(\Delta) \cdot \cos(\varphi_2),\, \cos(\varphi_1) \cdot \sin(\varphi_2) - \sin(\varphi_1) \cdot \cos(\varphi_2) \cos(\Delta)\big)$$

Where

- φ_1 is the latitude of location A
- φ_2 is the latitude of location B
- Δ is the difference between location B's longitude and location A's longitude
- atan2 is the two-argument arctangent function

Note: the formula above assumes that latitude and longitude are measured in radians r, $-\pi < r < \pi$. To convert from degrees d ($-180 < d < 180$) to radians r, you can use the simple formula:

$$r = \frac{d}{180}\pi$$

Write a program to prompt a user for a latitude/longitude of two locations (an origin and a destination) and computes the directional bearing (in degrees) from the origin to the destination. For example, if the user enters: $40.8206, -96.7056$ (40.8206° N, 96.7056° W) and $41.9483, -87.6556$ (41.9483° N, 87.6556° W), your program should output something like the following.

```
From (40.8206, -96.7056) to (41.9483, -87.6556):
bearing 77.594671 degrees
```

Exercise 2.22. General relativity tells us that time is relative to your velocity. As you approach the speed of light ($c = 299,792$ km/s), time slows down relative to objects traveling at a slower velocity. This *time dilation* is quantified by the Lorentz equation

$$t' = \frac{t}{\sqrt{1 - \frac{v^2}{c^2}}}$$

Where t is the time duration on the traveling space ship and t' is the time duration on the (say) Earth.

For example, if we were traveling at 50% the speed of light relative to Earth, one hour in our space ship ($t = 1$) would correspond to

$$t' = \frac{1}{\sqrt{1 - (.5)^2}} = 1.1547$$

hours on Earth (about 1 hour, 9.28 minutes).

Write a program that prompts the user for a velocity which represents the *percentage* p of the speed of light (that is, $p = \frac{v}{c}$) and a time duration t in hours and outputs the relative time duration on Earth.

For example, if the user enters 0.5 and 1 respectively as in our example, it should output something *like* the following:

```
Traveling at 1 hour(s) in your space ship at
50.00% the speed of light, your friends on
Earth would experience:
1 hour(s)
9.28 minute(s)
```

Your output should be able to handle years, weeks, days, hours, and minutes. So if the user inputs something like 0.9999 and 168, your output should look something like:

```
Traveling at 168.00 hour(s) in your space ship at
99.99% the speed of light, your friends on
Earth would experience:
1 year(s)
18 week(s)
3 day(s)
17 hour(s)
41.46 minute(s)
```

Exercise 2.23. Radioactive isotopes decay into other isotopes at a rate that is measured by a half-life, H. For example, Strontium-90 has a half-life of 28.79 years. If you started with 10 kilograms of Strontium-90, 28.79 years later you would have only 5 kilograms (with the remaining 5 kilograms being Yttrium-90 and Zirconium-90, Strontium-90's decay products).

Given a mass m of an isotope with half-life H we can determine how much of the isotope remains after y years using the formula,

$$r = m \cdot \left(\frac{1}{2}\right)^{(y/H)}$$

For example, if we have $m = 10$ kilograms of Strontium-90 with $H = 28.79$, after $y = 2$

years we would have

$$r = 10 \cdot \left(\frac{1}{2}\right)^{(2/28.79)} = 9.5298$$

kilograms of Strontium-90 left.

Write a program that prompts the user for an amount m (mass, in kilograms) of an isotope and its half-life H as well as a number of years y and outputs the amount of the isotope remaining after y years. For the example above your output should look something like the following.

```
Starting with 10.00kg of an isotope with half-life
28.79 years, after 2.00 years you would have
9.5298 kilograms left.
```

3. Conditionals

When writing code, its important to be able to distinguish between one or more situations. Based on some *condition* being *true* or *false*, you may want to perform some action if its true, while performing another, different action if it is false. Alternatively, you may simply want to perform one action if and only if the condition is true, and do nothing (move forward in your program) if it is false.

Normally, the *control flow* of a program is *sequential*: each statement is executed top-to-bottom one after the other. A *conditional* statement (sometimes called *selection* control structures) interrupts this normal control flow and executes statements only if some specified condition holds. The usual way of achieving this in a programming languages is through the use conditional statements such as the *if* statement, *if-else* statement, and *if-else-if* statement.

By using conditional statements, we can design more expressive programs whose behavior depends on their *state*: if the value of some variable is greater than some threshold, we can perform action a, otherwise, we can perform action b. You do this on a daily basis as you make decisions for yourself. At a cafe you may want to purchase the grande coffee which costs $2. If you have $2 or more, then you'll buy it. Otherwise, if you have less than $2, you can settle for the normal coffee which costs $1. Yet still, if you have less than $1 you'll not be able to make a purchase. The value of your pocket book determines the decision and subsequent actions that you take.

Similarly, our programs need to be able to "make decisions" based on various conditions (they don't actually make decisions for themselves as computer are not really "intelligent", we are simply specifying what should occur based on the conditions). Conditions in a program are specified by coding logical statements using *logical operators*.

3.1. Logical Operators

In logic, everything is black and white: a logical statement is an assertion that is either *true* or it is *false*. As previously discussed, some programming languages allow you to define and use boolean variables that can be assigned the value *true* or *false*. We can also formulate statements that involve other types of variables whose truth values are determined by the values of the variables at run time.

Psuedocode	Code	Meaning	Type
$<$	`<`	less than	relational
$>$	`>`	greater than	relational
$\leq$	`<=`	less than or equal to	relational
$\geq$	`>=`	greater than or equal to	relational
$=$	`==`	equal to	equality
$\neq$	`!=`	*not* equal to	equality

Table 3.1.: Comparison Operators

3.1.1. Comparison Operators

Suppose we have a variable *age* representing the age of an individual. Suppose we wish to execute some code if the person is an adult, $age \geq 18$ and a different piece of code if they are not an adult, $age < 18$. To achieve this, we need to be able to make *comparisons* between variables, constants, and even more complex expressions. Such logical statements may not have a fixed truth value. That is, they could be *true* or *false* depending on the value of the variables involved when the program is run.

Such comparisons are common in mathematics and likewise in programming languages. Comparison operators are usually *binary operators* in that they are applied to two *operands*: a left operand and a right operand. For example, if a, b are variables (or constants or expressions), then the comparison,

$$a \leq b$$

is *true* if the value stored in a is less than or equal to the value stored in b. Otherwise, if the value stored in b is strictly less than the value stored in a, the expression is *false*. Further, a, b are the *operands* and $\leq$ is the binary operator.

In general, operators do not *commute*. That is,

$$a \leq b \text{ and } b \leq a$$

are not equivalent, just as they are not in mathematics. However,

$$a \leq b \text{ and } b \geq a$$

are equivalent. Thus, the order of operands is important and can change the meaning and truth value of an expression.

A full listing of binary operators can be found in Table 3.1. In this table, we present both the mathematical inspired notation used in our pseudocode examples as well as the most common ways of representing these comparison operators in most programming languages. The need for alternative representations is because some symbols are not part of the ASCII character set common to most keyboards.

When using comparison operators, either operand can be variables, constants, or even more complex expressions. For example, you can make comparisons between two variables,

$$a < b, \quad a > b, \quad a \leq b, \quad a \geq b, \quad a = b, \quad a \neq b$$

or they can be between a variable and a constant

$$a < 10, \quad a > 10, \quad a \leq 10, \quad a \geq 10, \quad a = 10, \quad a \neq 10$$

or

$$10 < b, \quad 10 > b, \quad 10 \leq b, \quad 10 \geq b, \quad 10 = b, \quad 10 \neq b$$

Comparisons can also be used with more complex expressions such as

$$\sqrt{b^2 - 4ac} < 0$$

which could commonly be expressed in code as

```
sqrt(b*b - 4*a*c)   < 0
```

Observe that both operands *could* be constants, such as $5 \leq 10$ but there would be little point. Since both are constants, the truth value of the expression is already determined before the program runs. Such an expression could easily be replaced with a simple *true* or *false* variable. These are referred to as tautologies and contradictions respectively. We'll examine them in more detail below.

Pitfalls

Sometimes you may want to check that a variable falls in a *range*. For example, we may want to test that x lies in the interval $[0, 10]$ (between 0 and 10 inclusive on both ends). Mathematically we could express this as

$$0 \leq x \leq 10$$

and in code, we may try to do something like

```
0 <= x <= 10
```

However, when used in code, the operators `<=` are binary and must be applied to two operands. In a language the first inequality, `0 <= x` would be evaluated and would result in either *true* or *false*. The result is then used in the second comparison which results in a question such as *true* ≤ 10 or *false* ≤ 10.

Some languages would treat this as a syntax error and not allow such an expression to be compiled since you cannot compare a boolean value to a numerical value. However, other languages *may* allow this, typically representing *true* with some nonzero value such as 1 and *false* with 0. In either case, the expression would evaluate to *true* since both $0 \leq 10$ and $1 \leq 10$. However, this is clearly wrong: if x had a value of 20 for example, the

first expression would evaluate to *false*, making the entire expression *true*, but $20 \not\leq 10$. The solution is to use *logical operators* to express the same logic using *two* comparison operators (see Section 3.1.3).

Another common pitfall when programming is to mistake the assignment operator (typically only one equals sign, =) and the equality operator (typically two equal signs, ==). As before, some languages will not allow it. The expression `a = 10` would *not* have a truth value associated with it. Attempts to use the expression in a logical statement would be a syntax error.

Still yet, other languages may permit the expression and would give it a truth value equal to the value of the variable. For example, `a = 10` would take on the value 10 and treated as *true* (nonzero value) while `a = 0` would take on the value 0 and be treated as *false* (zero). In either case, we probably do not get the result that we want. Take care that you use proper equality comparison operator.

Other Considerations

The comparison operators that we've examined are generally used for comparing numerical types. However, sometimes we wish to compare non-numeric types such as single characters or strings. Some languages allow you to use numeric operators with these types as well. We examine specific uses in subsequent chapters.

Some dynamically typed languages (PHP, JavaScript, etc.) have additional rules when comparison operators are used with *mixed* types (that is, we compare a string with a numeric type). They may even have additional "strict" comparison operators such as `(a === b)` and `(a !== b)` which are *true* only if the values *and* types match. So, for example, `(10 == "10")` may be *true* because the values match, but `(10 === "10")` would be *false* since the *types* do not match (one is an integer, the other a string). We discuss specifics in subsequent chapters are they pertain to specific languages.

3.1.2. Negation

The *negation* operator is an operator that "flips" the truth value of the expression that it is applied to. It is very much like the numerical negation operator which when applied to positive numbers results in their negation and vice versa. When the logical negation operator is applied to a variable or statement, it negates its truth value. If the variable or statement was *true*, its negation is *false* and vice versa.

Also like the numerical negation operator, the logical negation operator is a *unary* operator as it applies to only *one* operand. In modern logic, the symbol ¬ is used to

a	$\neg a$
false	*true*
true	*false*

Table 3.2.: Logical Negation, $\neg$ Operator

denote the negation operator[1], examples:

$$\neg p, \quad \neg(a > 10), \quad \neg(a \leq b)$$

We will adopt this notation in our pseudocode, however most programming languages use the exclamation mark, `!` for the negation operator, similar to its usage in the inequality comparison, `!=`.

The negation operator applies to the variable or statement immediately following it, thus

$$\neg(a \leq b) \quad \text{and} \quad \neg a \leq b$$

are not the same thing (indeed, the second expression may not even be valid depending on the language). Further, when used with comparison operators, it is better to use the "opposite" comparison. For example,

$$\neg(a \leq b) \quad \text{and} \quad (a > b)$$

are equivalent, but the second expression is preferred as it is simpler. Likewise,

$$\neg(a = b) \quad \text{and} \quad (a \neq b)$$

are equivalent, but the second expression is preferred.

3.1.3. Logical And

The logical *and* operator (also called a *conjunction*) is a binary operator that is *true* if and only if *both* of its operands is *true*. If one of its operands is *false*, or if both of them are *false*, then the result of the logical and is *false*.

Many programming languages use two ampersands, `a && b` to denote the logical AND operator.[2]. However, for our pseudocode we will adopt the notation AND, thus the logical and can be expressed as a AND b. Table 3.3 contains a truth table representation of the logical AND operator.

[1] This notation was first used by Heyting, 1930 [11]; prior to that the tilde symbol was used ($\sim p$ for example) by Peano [19] and Whitehead & Russell [22]. However, the tilde operator has been adopted to mean *bit-wise* negation in programming languages.

[2] In logic, the "wedge" symbol, $p \wedge q$ is used to denote the logical and. It was first used again by Heyting, 1930 [11] but should not be confused for the keyboard caret, `^`, symbol. Many programming languages do use the caret as an operator, but it is usually the *exclusive-or* operator which is *true* if and only if exactly one of its operands is *true*

a	b	a AND b
false	*false*	*false*
false	*true*	*false*
true	*false*	*false*
true	*true*	*true*

Table 3.3.: Logical AND Operator

The logical AND is used to combine logical statements to form more complex logical statements. Recall that we couldn't directly use two comparison operators to check that a variable falls in a range, $0 \leq x \leq 10$. However, we can now use a logical AND to express this:

$$(0 \leq x) \text{ AND } (x \leq 10)$$

This expression is *true* only if both comparisons are true.

Though the conjunction is a binary operator, we can write statements that involve more than one variable or expression by using multiple instances of the operator. For example,

$$b^2 - 4ac \geq 0 \text{ AND } a \neq 0 \text{ AND } c > 0$$

The above statement would be evaluated left-to-right; the first two operands would be evaluated and the result would be either *true* or *false*. Then the result would be used as the first operand of the second logical AND. In this case, if any of the operands evaluated to *false*, the entire expression would be *false*. Only if all three were *true* would the statement be *true*.

3.1.4. Logical Or

The logical *or* operator is the binary operator that is *true* if *at least one* of its operands is *true*. If both of its operands are *false*, then the logical or is *false*. This is in contrast to what is usually meant in colloquially. If someone says "you can have cake or ice-cream," usually they implicitly also mean, "but not both." With the logical or operator, if both operands are *true*, the result is still *true*.

Many programming languages use two vertical bars (also referred to as *Sheffer strokes*), `||` to denote the logical OR operator.[3]. However, for our pseudocode we will adopt the notation OR, thus the logical or can be expressed as a OR b. Table 3.4 contains a truth table representation of the logical OR operator.

As with the logical AND, the logical OR is used to combine logical statements to make more complex statements. For example,

$$(age \geq 18) \text{ OR } (year = \text{“}senior\text{”})$$

[3]In logic, the "vee" symbol, $p \vee q$ is used to denote the logical OR. It was first used by Russell, 1906 [20].

a	b	a Or b
false	*false*	*false*
false	*true*	*true*
true	*false*	*true*
true	*true*	*true*

Table 3.4.: Logical Or Operator

which is *true* if the individual is aged 18 or older, is a senior, or is both 18 or older *and* a senior. If the individual is aged less than 18 and is not a senior, then the statement would be *false*.

We can also write statements with multiple Or operators,

$$a > b \text{ Or } b > c \text{ Or } a > c$$

which will be evaluated left-to-right. If any of the three operands is *true*, the statement will be *true*. The statement is only *false* when *all three* of the operands is *false*.

3.1.5. Compound Statements

The logical Andand Oroperators can be combined to express even more complex logical statements. For example, you can express the following statements involving *both* of the operators:

$$a \text{ And } (b \text{ Or } c) \quad a \text{ Or } (b \text{ And } c)$$

As an example, consider the problem of deciding whether or not a given year is a leap year. The Gregorian calendar defines a year as a leap year if it is divisible by 4. However, every year that is divisible by 100 is *not* a leap year unless it is also divisible by 400. Thus, 2012 is a leap year (4 goes into 2012 503 times), however, 1900 was *not* a leap year: though it is divisible by 4 ($1900/4 = 475$ with no remainder), it is also divisible by 100. The year 2000 *was* a leap year: it was divisible by 4 and 100 thus it was divisible by 400.

When generalizing these rules into logical statements we can follow a similar process: A *year* is a leap year if it is divisible by 400 or it is divisible by 4 and not by 100. This logic can be modeled with the following expression.

$$year \bmod 400 = 0 \text{ Or } (year \bmod 4 = 0 \text{ And } year \bmod 100 \neq 0)$$

When writing logical statements in programs it is generally best practice to keep things simple. Logical statements should be written in the most simple and succinct (but *correct*) way possible.

Tautologies and Contradictions

Some logical statements have the same meaning regardless of the variables involved. For example,

$$a \text{ OR } \neg a$$

is *always true* regardless of the value of a. To see this, suppose that a is *true*, then the statement becomes

$$a \text{ OR } \neg a = \textit{true} \text{ OR } \textit{false}$$

which is *true*. Now suppose that a is *false*, then the statement is

$$a \text{ OR } \neg a = \textit{false} \text{ OR } \textit{true}$$

which again is *true*. A statement that is always *true* regardless of the truth values of its variables is a tautology.

Similarly, the statement

$$a \text{ AND } \neg a$$

is always *false* (at least one of the operands will always be *false*). A statement that is always *false* regardless of the truth values of its variables is a contradiction.

In most cases, it is pointless to program a conditional statement with tautologies or contradictions: if an if-statement is predicated on a tautology it will *always* be executed. Likewise, an if-statement involved with a contradiction will *never* be executed. In either case, many compilers or code analysis tools may indicate and warn about these situations and encourage you to modify the code or to remove "dead code". Some languages may not even allow you write such statements.

There are always exceptions to the rule. Sometimes you may wish to *intentionally* write an infinite loop (see Section 4.5.2) for example in which case a statement similar to the following may be written.

```
1 WHILE true DO
  |  //some computation
2 END
```

De Morgan's Laws

Another tool to simplify your logic is De Morgan's Laws. When a logical AND statement is negated, it is equivalent to an *unnegated* logical OR statement and vice versa. That is,

$$\neg(a \text{ AND } b) \quad \text{and} \quad \neg a \text{ OR } \neg b$$

Order	Operator
1	$\neg$
2	AND
3	OR

Table 3.5.: Logical Operator Order of Precedence

are equivalent to each other;

$$\neg(a \text{ OR } b) \quad \text{and} \quad \neg a \text{ AND } \neg b$$

are also equivalent to each other. Though equivalent, it is generally preferable to write the simpler statement. From one of our previous examples, we could write

$$\neg\left((0 \leq x) \text{ AND } (x \leq 10)\right)$$

or we could apply De Morgan's Law and simplify this to

$$(0 > x) \text{ OR } (x > 10)$$

which is more concise and arguably more readable.

Order of Precedence

Recall that numerical operators have a well-defined order of precedence that is taken from mathematics (multiplication is performed before addition for example, see Section 2.3.4). When working with logical operators, we also have an order of precedence that somewhat mirrors those of numerical operators. In particular, negations are always applied *first*, followed by AND operators, and then lastly OR operators.

For example, the statement

$$a \text{ OR } b \text{ AND } c$$

is somewhat ambiguous. We don't just read it left-to-right since the AND operator has a higher order of precedence (this is similar to the mathematical expression $a + b \cdot c$ where the multiplication would be evaluated first). Instead, this statement would be evaluated by evaluating the AND operator first and then the result would be applied to the OR operator. Equivalently,

$$a \text{ OR } (b \text{ AND } c)$$

If we had *meant* that the OR operator should be evaluated *first*, then we should have explicitly written parentheses around the operator and its operands like

$$(a \text{ OR } b) \text{ AND } c$$

In fact, its best practice to write parentheses even if it is not necessary. Writing parentheses is often clearer and easier to read and more importantly communicates *intent*. By writing

$$a \text{ Or } (b \text{ And } c)$$

the intent is clear: we want the And operator to be evaluated first. By not writing the parentheses we leave our meaning somewhat ambiguous and force whoever is reading the code to recall the rules for order of precedence. By explicitly writing parentheses, we reduce the chance for error both in writing and in reading. Besides, its not like we're paying by the character.

For similar operators of the same precedence, they are evaluated left-to-right, thus

$$a \text{ Or } b \text{ Or } c \quad \text{is equivalent to} \quad ((a \text{ Or } b) \text{ Or } c)$$

and

$$a \text{ And } b \text{ And } c \quad \text{is equivalent to} \quad ((a \text{ And } b) \text{ And } c)$$

3.1.6. Short Circuiting

Consider the following statement:

$$a \text{ And } b$$

As we evaluate this statement, suppose that we find that a is *false*. Do we need to examine the truth value of b? The answer is no: since a is *false*, regardless of the truth value of b, the statement is false because it is a logical And. Both operands must be *true* for an And to be *true*. Since the first is *false*, the second is irrelevant.

Now imagine evaluating this statement in a computer. If the first operand of an And statement is *false*, we don't need to examine/evaluate the second. This has some potential for improved efficiency: if the second operand does not need to be evaluated, a program could ignore it and save a few CPU cycles. In general, the speed up for most operations would be negligible, but in some cases the second operand could be very "expensive" to compute (it could be a complex function call, require a database query to determine, etc.) in which case it could make a substantial difference.

Historically, avoiding even a few operations in old computers meant a difference on the order of milliseconds or even seconds. Thus, it made sense to avoid unnecessary operations. This is now known as short circuiting and to this day is still supported in many if not most programming languages. Though the differences are less stark in terms of CPU resources, most developers and programmers have come to *expect* the behavior and write statements under the assumption that short-circuiting will occur.

Short circuiting is commonly used to "check" for invalid operations. This is commonly used to prevent invalid operations. For example, consider the following statement:

$$(d \neq 0 \text{ And } 1/d > 1)$$

The first operand is checking to see if d is not zero and the second checks to see if its reciprocal is greater than 1. With short-circuiting, if $d = 0$, then the second operand will not be evaluated and the division by zero will be prevented. If $d \neq 0$ then the first operand is *true* and so the second operand will be evaluated as normal. Without short-circuiting, both operands would be evaluated leading to a division by zero.

There are many other common patterns that rely on short-circuiting to avoid invalid or undefined operations. Checking that a variable is valid (defined or not NULL) before using it to evaluate an expression, or checking that an index variable is within the range of an array's size before accessing its value for example.

Historically, the short-circuited version of the AND operator was known as *McCarthy's sequential conjunction operation* which was formally defined by John McCarthy (1962) as "if p then q, else false", eliminating the evaluation of q if p is *false* [14].

Because of short-circuiting, the logical AND is effectively *not commutative*. An operator is commutative if the order of its operands is irrelevant. For example, addition and multiplication are all commutative,

$$x + y = y + x \quad x \cdot y = y \cdot x$$

but subtraction and division are not,

$$x - y \neq y - x \quad x/y \neq y/x$$

In logic, the AND and OR operators are commutative, but when used in most programming languages they are not,

$$(a \text{ AND } b) \neq (b \text{ AND } a) \quad \text{and} \quad (a \text{ OR } b) \neq (b \text{ OR } a)$$

It is important to emphasize that they are still *logically* equivalent, but they are not *effectively* equivalent: because of short-circuiting, each of these statements have a *potentially* different effect.

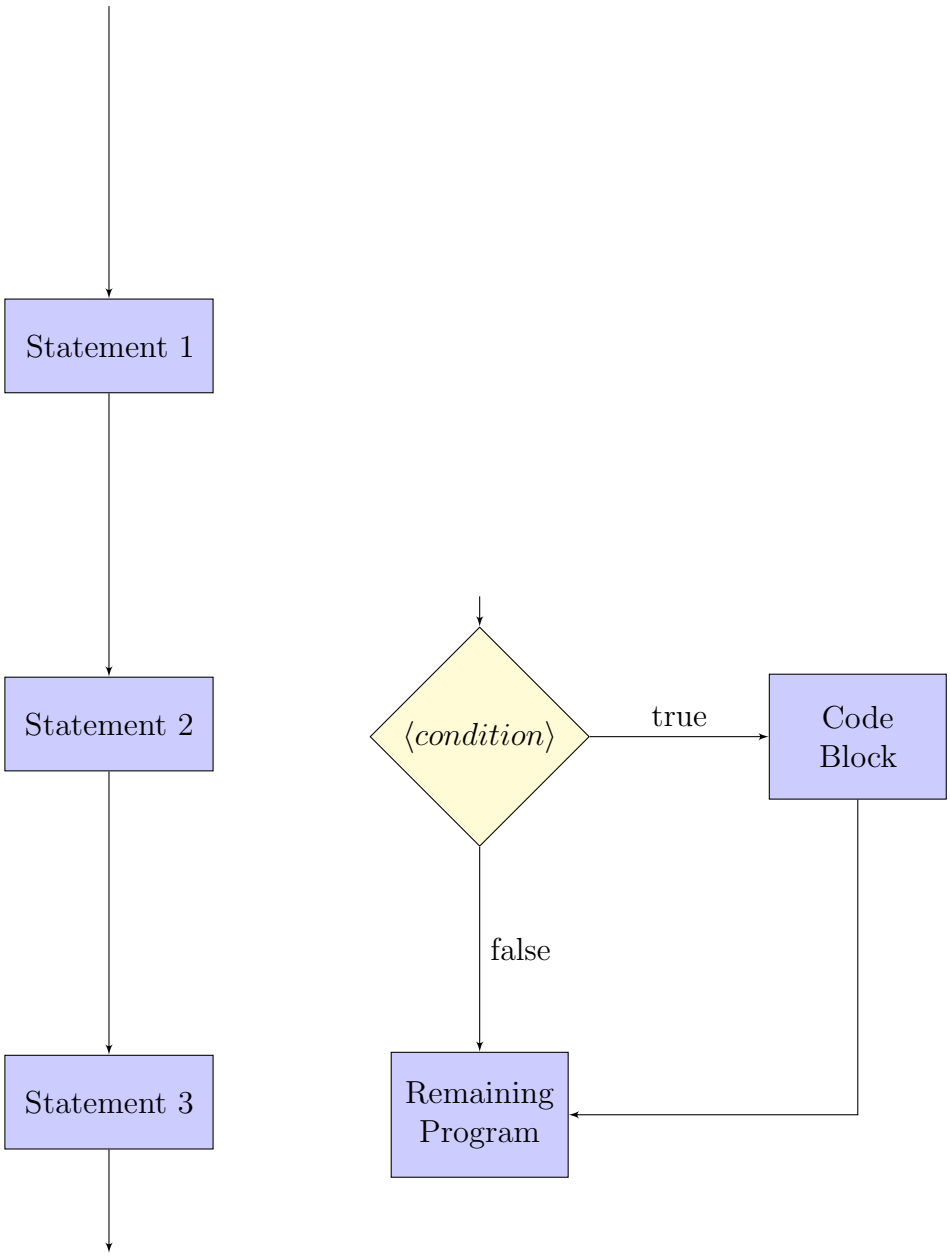
The OR operator is also short-circuited: if the first operand is *true*, then the truth value of the expression is already determined to be *true* and so the second operand will not be evaluated. In the expression,

$$a \text{ OR } b$$

if a evaluates to *true*, then b is not evaluated (since if either operand is *true*, the entire expression is *true*).

3.2. If Statement

Normally, the flow of control (or control flow) in a program is *sequential*. Each instruction is executed, one after the other, top-to-bottom and in individual statements left-to-right



(a) Sequential Flow Chart (b) If-Statement Flow Chart

Figure 3.1.: Control flow diagrams for sequential control flow and an if-statement. In sequential control, statements are executed one after the other as they are written. In an if-statement, the normal flow of control is interrupted and a Code Block is only executed if the given condition is *true*, otherwise it is not. After the if-statement, normal sequential control flow resumes.

just as one reads in English. Moreover, in most programming languages, each statement executes *completely* before the next statement begins. A visualization of this sequential control flow can be found in the control flow diagram in Figure 3.1(a).

However, it is often necessary for a program to "make decisions." Some segments of code may need to be executed only if some condition is satisfied. The *if statement* is a *control structure* that allows us to write a snippet of code predicated on a logical statement. The code executes if the logical statement is *true*, and does *not* execute if the logical statement is *false*. This control flow is featured in Figure 3.1(b)

An example of the syntax using pseudocode can be found in Algorithm 3.1. The use of the keyword "if" is common to most programming languages. The logical statement associated with the if-statement immediately follows the "if" keyword and is usually surrounded by parentheses. The code block immediately following the if-statement is *bound* to the if-statement.

```
1 IF (⟨condition⟩) THEN
2 |  Code Block
3 END
```

Algorithm 3.1: An if-statement

As in the flow chart, if the ⟨*condition*⟩ evaluates to *true*, then the code block bound to the statement executes in its entirety. Otherwise, if the condition evaluates to *false*, the code block bound to the statement is skipped in its entirety.

A simple if-statement can be viewed as a "do this if *and only if* the condition holds." Alternatively, "if this condition holds do this, otherwise don't". In either case, once the if-statement executes, the program returns to the normal sequential control flow.

3.3. If-Else Statement

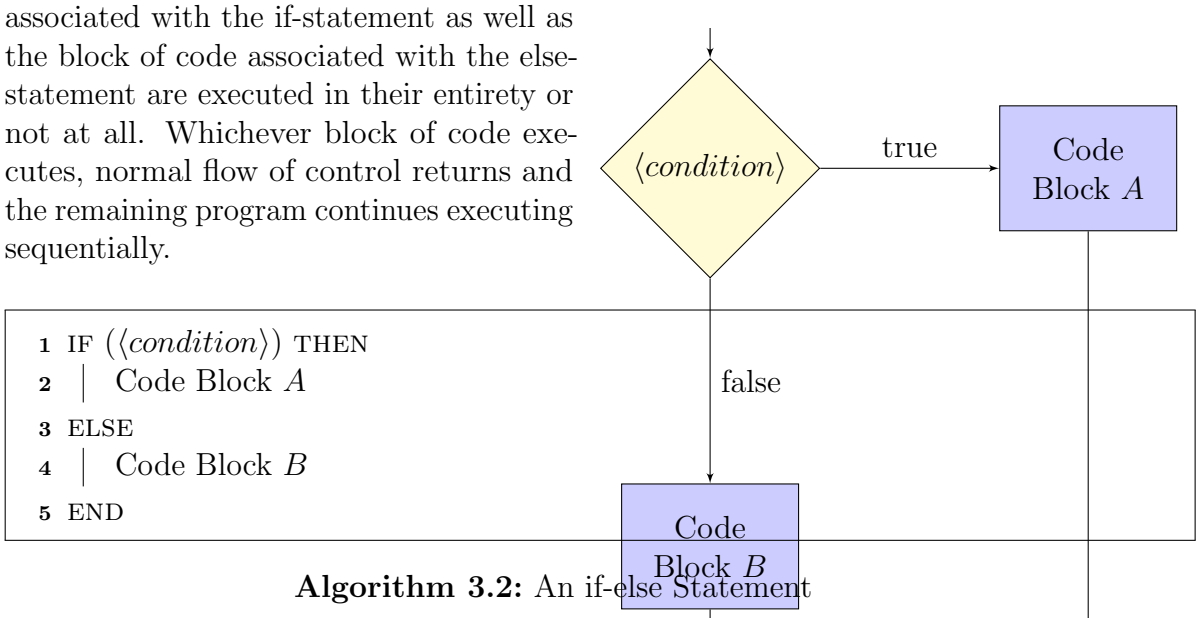
An if-statement allows you to specify a code segment that is executed or is not executed. An if-else statement allows you to specify an alternative. An else-if statement allows you to define a condition such that if the condition is *true*, one code block executes and if the condition is *false*, an entirely different code block executes.

The control flow of an if-else statement is presented in Figure 3.2. Note that Code Block A and Code Block B are *mutually exclusive*. That is, one and only one of them is executed depending on the truth value of the ⟨*condition*⟩. A presentation of a generic if-else statement in our pseudocode can be found in Algorithm 3.2

Just as with an if-statement, the keyword "if" is used. In fact, the if-statement is simply just an if-else statement with the else block omitted (equivalently, we could have defined an *empty* else block, but since it would have no effect, a simple if-statement with no

else block is preferred). It is common to most programming languages to use the "else" keyword to denote the else block of code. Since there is only one ⟨*condition*⟩ to evaluate and it can only be *true* or *false*, it is not necessary to specify the conditions under which the else block executes. It is assumed that if the ⟨*condition*⟩ evaluates to *false*, the else block executes.

As with an if-statement, the block of code associated with the if-statement as well as the block of code associated with the else-statement are executed in their entirety or not at all. Whichever block of code executes, normal flow of control returns and the remaining program continues executing sequentially.



Algorithm 3.2: An if-else Statement

Figure 3.2.: An if-else Flow Chart

3.4. If-Else-If Statement

An if-statement allows you to define a "do this or do not" and an if-else statement allows you to define a "do this or do that" statement. Yet another generalization is an if-else-if statement. Using such a statement you can define any number of mutually exclusive code blocks.

To illustrate, consider the case in which we have exactly three mutually exclusive possibilities. At a particular university, there are three possible semesters depending on the month. January through May is the Spring semester, June/July is the Summer semester, and August through December is the Fall semester. These possibilities are mutually exclusive because it cannot

be *both* Spring and Summer at the same time for example. Suppose we have the current month stored in a variable named *month*. Algorithm 3.3 expresses the logic for determining which semester it is using an if-else-if statement.

```
1 IF (month ≥ January) AND (month ≤ May) THEN
2 |  semester ← "Spring"
3 ELSE IF (month > May) AND (month ≤ July) THEN
4 |  semester ← "Summer"
5 ELSE
6 |  semester ← "Fall"
7 END
```

Algorithm 3.3: Example If-Else-If Statement

Let's understand how this code works. First, the "if" and "else" keywords are used just as the two previous control structures, but we are now also using the "else if" keyword combination to specify an additional condition. Each condition, starting with the condition associated with the if-statement is checked in order. If and when one of the conditions is satisfied (evaluates to *true*), the code block associated with that condition is executed and *all other code blocks are ignored*.

Remember, each of the code blocks in an if-else-if control structure are mutually exclusive. One and *only* one of the code blocks will ever execute. Similar to the sequential control flow, the *first* condition that is satisfied is the one that executes. If *none* of the conditions is satisfied, then the code block associated with the else-statement is the one that is executed.

In our example, we only identified three possibilities. You can generalize an if-else-if statement to specify as many conditions as you like. This generalization is depicted in Algorithm 3.4 and visualized in Figure 3.3. Similar to the if-statement, the else-statement and subsequent code block is actually optional. If omitted, then it may be possible that

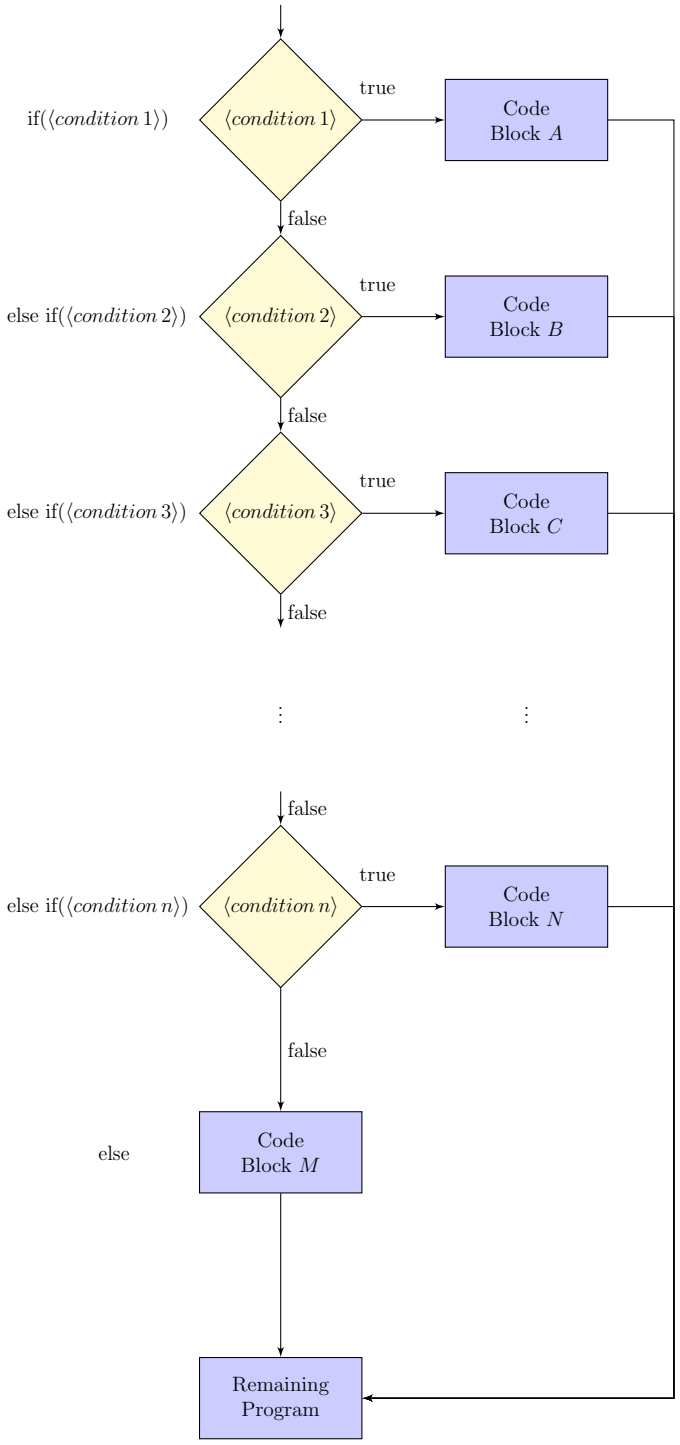


Figure 3.3.: Control Flow for an If-Else-If Statement. Each condition is evaluated in sequence. The first condition that evaluates to *true* results in the corresponding code block being executed. After executing, the program continues. Thus, each code block is *mutually exclusive*: at most *one* of them is executed.

none of the code blocks is executed.

```
1  IF (⟨condition 1⟩) THEN
2  |  Code Block A
3  ELSE IF (⟨condition 2⟩) THEN
4  |  Code Block B
5  ELSE IF (⟨condition 3⟩) THEN
6  |  Code Block C
7  ...
8  ELSE
9  |  Code Block
10 END
```

Algorithm 3.4: General If-Else-If Statement

The design of if-else-if statements must be done with care to ensure that your statements are each mutually exclusive and capture the logic you intend. Since the *first* condition that evaluates to *true* is the one that is executed, the *order* of the conditions is important. A poorly designed if-else-if statement can lead to bugs and logical errors.

As an example, consider describing the loudness of a sound by its *decibel* level in Algorithm 3.5.

```
1  IF decibel ≤ 70 THEN
2  |  comfort ← "intrusive"
3  ELSE IF decibel ≤ 50 THEN
4  |  comfort ← "quiet"
5  ELSE IF decibel ≤ 90 THEN
6  |  comfort ← "annoying"
7  ELSE
8  |  comfort ← "dangerous"
9  END
```

Algorithm 3.5: If-Else-If Statement With a Bug

Suppose that $decibel = 20$ which *should* be described as a "quite" sound. However, in the algorithm, the first condition, $decibel \leq 70$ evaluates to *true* and the sound is categorized as "intrusive". The bug is that the second condition, $decibel \leq 50$ should have come *first* in order to capture all decibel levels less than or equal to 50.

Alternatively, we could have followed the example in Algorithm 3.3 and completely specified both lower bounds and upper bounds in our condition. For example, the condition for "intrusive" could have been

$$(decibel > 50) \text{ AND } (decibel \leq 70)$$

However, doing this is unnecessary if we order our conditions appropriately and we can potentially write simpler conditions if we keep the fact that the if-else-if statement is mutually exclusive.

3.5. Ternary If-Else Operator

Another conditional operator is the ternary if-then-else operator. It is often used to write an expression that can take on one of two values depending on the truth value of a logical expression. Most programming languages support this operator which has the following syntax:

```
E ? X : Y
```

Where `E` is a boolean expression. If `E` evaluates to *true*, the statement takes on the value `X` which does not need to be a boolean value: it can be anything (an integer, string, etc.). If `E` evaluates to *false*, the statement takes on the value `Y`.

A simple usage of this expression is to find the minimum of two values:

```
min = ( (a < b) ? a : b );
```

If $a < b$ is true, then `min` will take on the value a. Otherwise it will take on the value b (in which case $a \geq b$ and so b is minimal).

Most programming languages support this special syntax as it provides a nice convenience (yet another example of syntactic sugar).

3.6. Examples

3.6.1. Meal Discount

Consider the problem of computing a receipt for a meal. Suppose we have the subtotal cost of all items in the meal. Further, suppose that we want to compute a discount (senior citizen discount, student discount, or employee discount, etc.). We can then apply the discount, compute the sales tax, and sum a total, reporting each detail to the user.

To do this, we first prompt the user to enter a subtotal. We can then ask the user if there is a discount to be computed. If the user answers yes, then we again prompt them for an amount (to allow different types of discounts). Otherwise, the discount will be zero. We can then proceed to calculate each of the amounts above. To do this we'll need an if-statement. We could also use a conditional statement to check to see if the input

makes sense: we wouldn't want a discount amount that is greater than 100% after all.

```
1  Prompt the user for a subtotal
2  subTotal ← read input from user
3  discountPercent ← 0
4  Ask the user if they want to apply a discount
5  hasDiscount ← get user input IF hasDiscount = "yes" THEN
6  |  Prompt the user for a discount amount
7  |  discountPercent ← read user input
8  END
9  IF discountPercent > 100 THEN
10 |  Error! Discount cannot be more than 100%
11 END
12 discount ← subTotal × discountPercent
13 discountTotal ← subTotal − discount
14 tax ← taxRate × discountTotal
15 grandTotal ← discountTotal + tax
16 output subTotal, discountTotal, tax, grandTotal to user
```

Algorithm 3.6: A simple receipt program

3.6.2. Look Before You Leap

Recall that dividing by zero is an invalid operation in most programming languages (see Section 2.3.5). Now that we have a means by which numerical values can be checked, we can *prevent* such errors entirely.

Suppose that we were going to compute a quotient of two variables x/y. If $y = 0$, this would be an invalid operation and lead to undefined, unexpected or erroneous behavior. However, if we checked whether or not the denominator is zero *before* we compute the quotient then we could prevent such errors. We present this idea in Algorithm 3.7.

```
1  IF y ≠ 0 THEN
2  |  q ← x/y
3  END
```

Algorithm 3.7: Preventing Division By Zero Using an If Statement

This approach to programming is known as defensive programming. We are essentially checking the conditions for an invalid operation *before* performing that operation. In the example above, we simply chose not to perform the operation. Alternatively, we

could use an if-else statement to perform alternate operations or *handle* the situation differently. Defensive programming is akin to "looking before leaping": before taking another, potentially dangerous step, you look to see if you are at the edge of a cliff, and if so you don't take that dangerous step.

3.6.3. Comparing Elements

Suppose we have two students, student A and student B and we want to compare them: we want to determine which one should be placed first in a list and which should be placed second. For this exercise let's suppose that we want to order them first by their last names (so that Anderson comes before Zadora). What if they have the same last name, like Jane Smith and John Smith? If the last names are equal, then we'll want to order them by their first names (Jane before John). If both their first names and last names are the same, we'll say any order is okay.

Names will likely be represented using strings, so let's say that $<$, $=$ and $>$ apply to strings, ordering them lexicographically (which is consistent with alphabetic ordering). We'll first need to compare their last names. If equal, then we'll need another conditional construct. This is achieved by *nesting* conditional statements as in Algorithm 3.8.

```
1  IF A's last name < B's last name THEN
2  |  output A comes first
3  ELSE IF A's last name > B's last name THEN
4  |  output B comes first
5  ELSE
   |  //last names are equal, so compare their first names
6  |  IF A's first name < B's first name THEN
7  |  |  output A comes first
8  |  ELSE IF A's first name > B's first name THEN
9  |  |  output B comes first
10 |  ELSE
11 |  |  Either ordering is fine
12 |  END
13 END
```

Algorithm 3.8: Comparing Students by Name

3.6.4. Life & Taxes

Another example in which there are several cases that have to be considered is computing the income tax liability using marginal tax brackets. Table 3.6 contains the 2014 US

Federal tax margins and marginal rates for a married couple filing jointly based on the Adjusted Gross Income (income after deductions).

AGI is over	But not over	Tax
0	$18,150	10% of the AGI
$18,150	$73,800	$1,815 plus 15% of the AGI in excess of $18,150
$73,800	$148,850	$10,162.50 plus 25% of the AGI in excess of $73,800
$148,850	$225,850	$28,925 plus 28% of the AGI in excess of $148,850
$225,850	$405,100	$50,765 plus 33% of the AGI in excess of $225,850
$405,100	$457,600	$109,587.50 plus 35% of the AGI in excess of $405,100
$457,600	—	$127,962.50 plus 39.6% of the AGI in excess of $457,600

Table 3.6.: 2014 Tax Brackets for Married Couples Filing Jointly

In addition, one of the tax credits (which offsets tax liability) tax payers can take is the child tax credit. The rules are as follows:

- If the AGI is $110,000 or more, they cannot claim a credit (the credit is $0)
- Each child is worth a $1,000 credit, however at most $3,000 can be claimed
- The credit is not refundable: if the credit results in a negative tax liability, the tax liability is simply $0

As an example: suppose that a couple has $35,000 AGI and has two children. Their tax liability is

$$\$1,815 + 0.15 \times (\$35,000 - \$18,150) = \$4,342.50$$

However, the two children represent a $2,000 refund, so their total tax liability would be $2,342.50.

Let's first design some code that computes the tax liability based on the margins and rates in Table 3.6. We'll assume that the AGI is stored in a variable named *income*. Using a series of if-else-if statements as presented in Algorithm 3.9, the variable *tax* will

contain our initial tax liability.

```
1  IF income ≤ 18,150 THEN
2  |  tax ← .10 · income
3  ELSE IF income > 18,150 AND income ≤ 73,800 THEN
4  |  tax ← 1,815 + .15 · (income − 18,150)
5  ELSE IF income > 73,800 AND income ≤ 148,850 THEN
6  |  tax ← 10,162.50 + .25 · (income − 73,800)
7  ELSE IF income > 148850 AND income ≤ 225,850 THEN
8  |  tax ← 28,925 + .28 · (income − 148,850)
9  ELSE IF income > 225,850 AND income ≤ 405,100 THEN
10 |  tax ← 50,765 + .33 · (income − 225,850)
11 ELSE IF income > 405,100 AND income ≤ 457,600 THEN
12 |  tax ← 109,587.50 + .35 · (income − 405,100)
13 ELSE
14 |  tax ← 127,962.50 + .396 · (income − 457,600)
15 END
```

Algorithm 3.9: Computing Tax Liability with If-Else-If

To compute the amount of a tax credit and adjust the tax accordingly by using similar if-else-if and if-else statements as in Algorithm 3.10.

```
1  IF income ≥ 110000 THEN
2  |  credit ← 0
3  ELSE IF numberOfChildren ≤ 3 THEN
4  |  credit ← numberOfChildren ∗ 1000
5  ELSE
6  |  credit ← 3000
7  END
   //Now adjust the tax, taking care that its a nonrefundable credit
8  IF credit > tax THEN
9  |  tax ← 0
10 ELSE
11 |  tax ← tax − credit
12 END
```

Algorithm 3.10: Computing Tax Credit with If-Else-If

3.7. Exercises

Exercise 3.1. Write a program that prompts the user for an x and a y coordinate in the Cartesian plane and prints out a message indicating if the point (x, y) lies on an axis (x or y axis, or both) or what quadrant it lies in (see Figure 3.4).

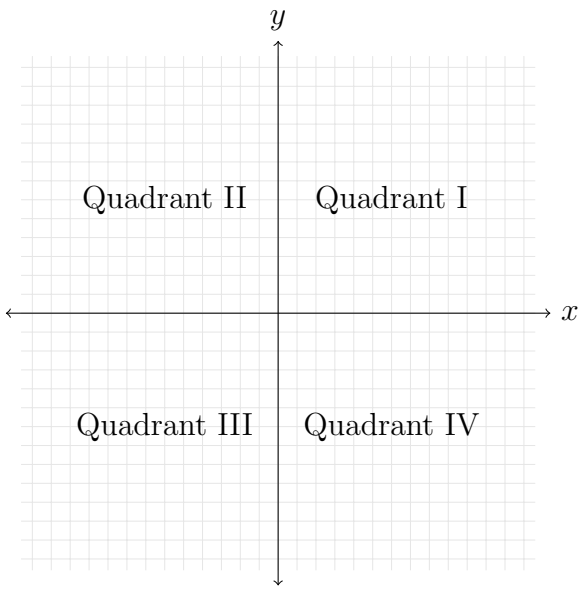


Figure 3.4.: Quadrants of the Cartesian Plane

Exercise 3.2. A BOGO (Buy-One, Get-One) sale is a promotion in which a person buys two items and receives a 50% discount on the less expensive one. Write a program that prompts the user for the cost of two items, computes a 50% discount on the less expensive one, and then computes a grand total.

Exercise 3.3. Write a program that will calculate and print a bill for the city power company. The rates vary depending on whether the use is residential, commercial, or industrial. The program should prompt the user to indicate which type the customer is as well as a total kilowatt hour (kWh) used and compute the total based on the rates in Table 3.7.

Type	Base	Rate
Residential	$24.50	$0.1415 per kWh
Commercial	$75.00	$0.1750 per kWh
Industrial	$245.00	$0.2774 per kWh

Table 3.7.: Rates per kWh for each type of customer

Each customer is assessed a *base* cost of service regardless of how much electricity they use. In addition, if industrial customers use more than 2000 kWh, they are assessed an additional $0.05 per kWh.

Exercise 3.4. Various substances have different boiling points. A selection of substances and their boiling points can be found in Table 3.8. Write a program that prompts the user for the observed boiling point of a substance in degrees Celsius and identifies the substance if the observed boiling point is within 5% of the expected boiling point. If the data input is more than 5% higher or lower than any of the boiling points in the table, it should output `Unknown substance`.

Substance	**Boiling Point (C)**
Methane	-161.7
Butane	-0.5
Water	100
Nonane	150.8
Mercury	357
Copper	1187
Silver	2193
Gold	2660

Table 3.8.: Expected Boiling Points

Exercise 3.5. Electrical resistance in various metals can be measured using nano-ohm metres ($n\Omega \cdot m$). Table 3.9 gives the resistivity of several metals.

Material	Resistivity ($n\Omega \cdot m$)
Copper	16.78
Aluminum	26.50
Beryllium	35.6
Potassium	72.0
Iron	96.10

Table 3.9.: Resistivity of several metals

Write a program that prompts the user for an observed resistivity of an unknown material (as nano-ohm metres) and identifies the substance if the observed resistivity is within $\pm 3\%$ of the known resistivity of any of the materials in Table 3.9. If the input value lies outside the $\pm 3\%$ range, output `Unknown substance`.

Exercise 3.6. The visible light spectrum is measured in nanometer (nm) frequencies. Ranges roughly correspond to visible colors as depicted in Table 3.10.

Write a program that takes an integer corresponding to a wavelength and outputs the corresponding color. If the value lies outside the ranges it should output `Not a visible wavelength`. If a value lies within multiple color ranges it should print all that apply (for example, a wavelength of 495 is "Indigo-green").

Exercise 3.7. A certain production of steel is graded according to the following conditions:

Color	Wave length range (nm)
Violet	380 – 450
Blue	450 – 475
Indigo	476 – 495
Green	495 – 570
Yellow	570 – 590
Orange	590 – 620
Red	620 - 750

Table 3.10.: Visible Light Spectrum Ranges

(i) Hardness must be greater than 50

(ii) Carbon content must be less than 0.7

(iii) Tensile strength must be greater than 5600

A grade of 5 thru 10 is is assigned to the steel according to the conditions in Table 3.11. Write a program that will read in the hardness, carbon content, and tensile strength as

Grade	Conditions
10	All three conditions are met
9	Conditions (i) and (ii) are met
8	Conditions (ii) and (iii) are met
7	Conditions (i) and (iii) are met
6	If only 1 of the three conditions is met
5	If none of the conditions are met

Table 3.11.: Grades of Steel

inputs and output the corresponding grade of the steel.

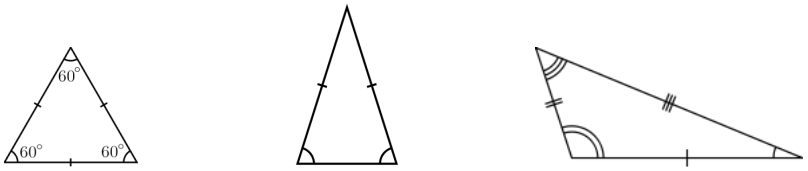
Exercise 3.8. A triangle can be characterized in terms of the length of its three sides. In particular, an *equilateral* triangle is a triangle with all three sides being equal. A triangle such that two sides have the same length is *isosceles* and a triangle with all three sides having a different length is *scalene*. Examples of each can be found in Figure 3.5.

In addition, the three sides of a triangle are *valid* only if the sum of any two sides is strictly greater than the third length.

Write a program to read in three numbers as the three sides of a triangle. If the three sides do not form a valid triangle, you should indicate so. Otherwise, if valid, your program should output whether or not the triangle is equilateral, isosceles or scalene.

Exercise 3.9. Body Mass Index (BMI) is a healthy statistic based on a person's mass and height. For a healthy adult male BMI is calculated as

$$\text{BMI} = \frac{m}{h^2} \cdot 703.069579$$



(a) Equilateral Triangle (b) Isosceles Triangle (c) Scalene Triangle

Figure 3.5.: Three types of triangles

where m is the person's mass (in lbs) and h is the person's height (in whole inches). Write a program that reads in a person's mass and height as input and outputs a characterization of the person's health with respect to the categories in Table 3.12.

Range	Category
BMI < 15	Very severely underweight
$15 \leq$ BMI < 16	Severely underweight
$16 \leq$ BMI < 18.5	Underweight
$18.5 \leq$ BMI < 25	Normal
$25 \leq$ BMI < 30	Overweight
$30 \leq$ BMI < 35	Obese Class I
$35 \leq$ BMI < 40	Obese Class II
BMI ≥ 40	Obese Class III

Table 3.12.: BMI Categories

Exercise 3.10. Let R_1 and R_2 be rectangles in the plane defined as follows. Let (x_1, y_1) be point corresponding to the lower-left corner of R_1 and let (x_2, y_2) be the point of its upper-right corner. Let (x_3, y_3) be point corresponding to the lower-left corner of R_2 and let (x_4, y_4) be the point of its upper-right corner.

Write a program to determine the *intersection* of these two rectangles. In general, the intersection of two rectangles is another rectangle. However, if the two rectangles abut each other, the intersection could be a horizontal or vertical line segment (or even a point). It is also possible that the intersection is *empty*. Your program will need to distinguish between these cases.

If the intersection of R_1, R_2 is a rectangle, R_3, your program should output two points (the lower-left and upper-right corners of R_3) as well as the *area* of R_3. If the intersection is a line segment, your program should output the two *end-points* and whether it is a vertical or horizontal line segment. Finally, if the intersection is empty your program should output "empty intersection". Your program should also be robust enough to check that the input is valid (it should not accept empty or "reversed" rectangles).

Your program should read in $x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4$ from the user and perform the computation above. As an example, the values $2, 1, 6, 7.5, 4, 5.5, 8.5, 8.25$ would correspond

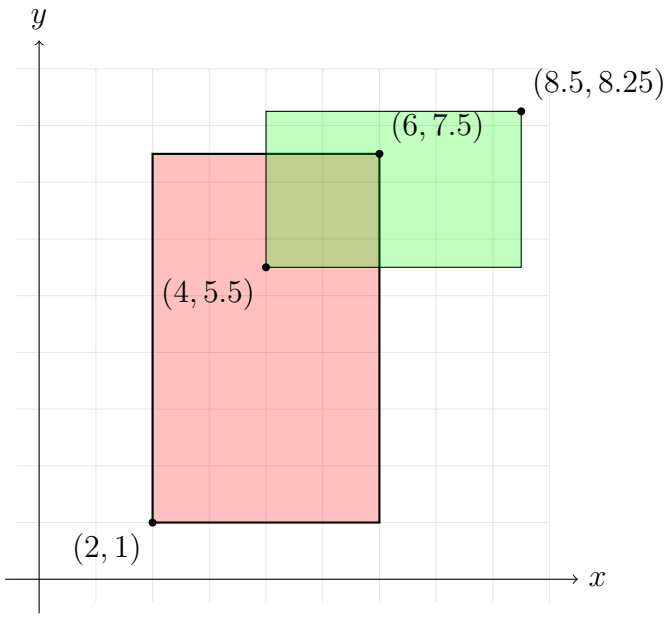


Figure 3.6.: Intersection of Two Rectangles

to the two rectangles in Figure 3.6.

The output for this instance should look something like the following.

```
Intersecting rectangle: (4, 5.5), (6, 7.5)
Area: 4.00
```

Exercise 3.11. Write an app to help people track their cell phone usage. Cell phone plans for this particular company give you a certain number of minutes every 30 days which must be used or they are lost (no rollover). We want to track the average number of minutes used per day and inform the user if they are using too many minutes or can afford to use more.

Write a program that prompts the user to enter the following pieces of data:

- Number of minutes in the plan per 30 day period, m
- The current day in the 30 day period, d
- The total number of minutes used so far u

The program should then compute whether the user is over, under, or right on the average daily usage under the plan. It should also inform them of how many minutes are left and how many, on average, they can use per day for the rest of the month. Of course, if they've run out of minutes, it should inform them of that too.

For example, if the user enters $m = 250$, $d = 10$, and $u = 150$, your program should print out something similar to the following.

```
10 days used, 20 days remaining
Average daily use: 15 min/day

You are EXCEEDING your average daily use (8.33 min/day),
continuing this high usage, you'll exceed your minute plan by
200 minutes.

To stay below your minute plan, use no more than 5 min/day.
```

Of course, if the user is under their average daily use, a different message should be presented. You are allowed/encouraged to compute any other stats for the user that you feel would be useful.

4. Loops

Computers are really good at automation. A key aspect of automation is that we be able to repeat a process over and over on different pieces of data until some condition is met. For example, if we have a collection of numbers and we want to find their sum we would *iterate* over each number, adding it to a total, until we have examined every number. Another example may include sending an email message to each student in a course. To automate the process, we could iterate over each student record and *for each* student we would generate and send the email.

Automated repetition is where *loops* come in handy. Computers are perfectly suited for performing such repetitive tasks. We can write a single block of code that performs some action or processes a single piece of data, then we can write a loop around that block of code to execute it a number of times.

Loops provide a much better alternative than repeating (cut-paste-cut-paste) the same code over and over with different variables. Indeed, we wouldn't even do this in real life. Suppose that you took a 100 mile trip. How would you describe it? Likely, you wouldn't say, "I drove a mile, then I drove a mile, then I drove a mile, ..." repeated 100 times. Instead, you would simply state "I drove 100 miles" or maybe even, "I drove until I reached my destination."

Loops allow us to write concise, repeatable code that can be applied to each element in a collection or perform a task over and over again until some condition is met. When writing a loop, there are three essential components:

- An *initialization* statement that specifies how the loop begins
- A *continuation* (or *termination*) condition that specifies whether the loop should continue to execute or terminate
- An *iteration* statement that makes progress toward the termination condition

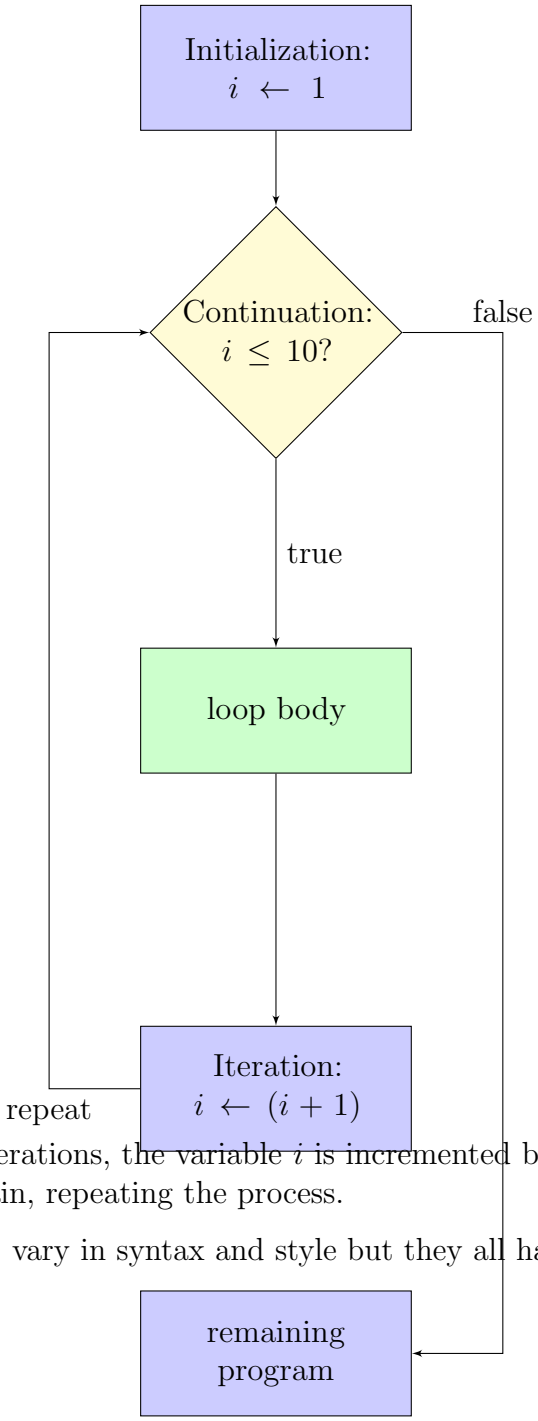
The initialization statement is executed before the loop begins and serves as a way to set the loop up. Typically, the initialization statement involves setting the initial value of some variable.

The continuation statement is a logical statement (that evaluates to *true* or *false*) that specifies if the loop should continue (if the value is *true*) or should terminate (if the value is *false*). Upon termination, code returns to a sequential control flow and the program continues.

The iteration statement is intended to update the state of a program to make progress toward the termination condition. If we didn't make such progress, the loop would continue on forever as the termination condition would never be satisfied. This is known as an *infinite loop* , and results in a program that never terminates.

As a simple example, consider the following outline.

- Initialize the value of a variable i to 1
- While the value of i is less than or equal to 10 . . . (continuation condition)
- Perform some action (this is sometimes referred to as the *loop body*
- Iterate the variable i by adding one to its value



The code outline above specifies that some action is to be performed once for each value: $i = 1, i = 2, \ldots, i = 10$, after which the loop terminates. Overall, the loop executes a total of 10 times. Prior to each of the 10 executions, the value of i is checked; as it is less than or equal to 10, the action is performed. At the end of each of the 10 iterations, the variable i is incremented by 1 and the termination condition is checked again, repeating the process.

There are several different types of loops that vary in syntax and style but they all have the same three basic components.

4.1. While Loops

Figure 4.1.: A Typical Loop Flow Chart

A *while loop* is a type of loop that places the three components in their logical order. The initialization statement is written before the loop code. Typically the keyword *while* is used to specify the continuation/termination condition. Finally, the iteration statement is usually performed at the end of the loop *inside* the code block associated with the loop. A small, counter-controlled while loop is presented in Algorithm 4.1 which illustrates the

previous example of iterating a variable i from 1 to 10.

```
1 i ← 1 //Initialization statement
2 WHILE i ≤ 10 DO
3 |   Perform some action
4 |   i ← (i + 1) //Iteration statement
5 END
```

Algorithm 4.1: Counter-Controlled While Loop

Prior to the WHILE statement, the variable i is initialized to 1. This action is only performed once and it is done so before the loop code. Then, before the loop code is executed, the continuation condition is checked. Since $i = 1 \leq 10$, the condition evaluates to *true* and the loop code block is executed. The last line of the code block is the iteration statement, where i is incremented by 1 and now has a value of 2. The code returns to the *top* of the loop and again evaluates the continuation condition (which is still true as $i = 2 \leq 10$).

On the 10th iteration of the loop when $i = 10$, the loop will execute for the last time. At the end of the loop, i is incremented to 11. The loop *still* returns to the top and the continuation condition is *still* checked one last time. However, since $i = 11 \not\leq 10$, the condition is now *false* and the loop terminates. Regular sequential control flow returns and the program continues executing whatever code is specified after the loop.

4.1.1. Example

In the previous example we knew how many times we wanted the loop to execute. Though you can use a while loop in counter-controlled situations, while loops are typically used in scenarios when you may not know how many iterations you want the loop to execute for. Instead of a straightforward iteration, the loop itself may update a variable in a less-than-predictable manner.

As an example, consider the problem of *normalizing* a number as is typically done in scientific notation. Given a number x (for simplicity, we'll consider $x \geq 0$), we divide it by 10 until its value is in the interval $[0, 10)$, keeping track of how many times we've divided by 10. For example, if we have the number $x = 32,145.234$, we would divide by 10 four times, resulting in 3.2145234 so that we could express it as

$$3.2145234 \times 10^4$$

A simple realization of this process is presented in Algorithm 4.2. The number of times the loop executes depends on how large x is. For the example mentioned, it executes 4 times; for an input of $x = 10,000,000$ it would execute 7 times. A while loop allows us to specify the repetition process without having to know up front how many times it will

execute.

```
   INPUT  : A number x, x ≥ 0
1  k ← 0
2  WHILE x > 10 DO
3  |  x ← (x/10)
4  |  k ← (k + 1)
5  END
6  output x, k
```

Algorithm 4.2: Normalizing a Number With a While Loop

4.2. For Loops

A *for loop* is similar to a while loop but allows you to specify the three components on the same line. In many cases, this results in a loop that is more readable; if the code block in a while loop is long it may be difficult to see the initialization, continuation, and iteration statements clearly. For loops are typically used to iterate over elements stored in a collection such as an array (see Chapter 7).

Usually the keyword *for* is used to identify all three components. A general example is given in Algorithm 4.3.

```
1  FOR (initialization statement); (continuation condition); (iteration statement) DO
2  |  Perform some action
3  END
```

Algorithm 4.3: A General For Loop

Note the additional syntax: in many programming languages, semicolons are used at the end of executable statements. Semicolons are also used to delimit each of the three loop components in a for-loop (otherwise there may be some ambiguity as to where each of the components begins and ends). However, the semicolons are typically only placed after the initialization statement and continuation condition and are *omitted* after the iteration statement. A more concrete example is given in Algorithm 4.4 which represents an equivalent code snippet as the counter-controlled while loop we examined earlier.

Though all three components are written on the same line, the initialization statement is only ever executed once; at the beginning of the loop. The continuation condition is checked prior to each and every execution of the loop. Only if it evaluates to *true* does the loop body execute. The iteration condition is performed *at the end* of each loop iteration.

```
1 FOR i ← 1; i ≤ 10; i ← (i + 1) DO
2 |   Perform some action
3 END
```

Algorithm 4.4: Counter-Controlled For Loop

4.2.1. Example

As a more concrete example, consider Algorithm 4.5 in which we do the same iteration (i will take on the values $1, 2, 3, \ldots, 10$), but in each iteration we add the value of i *for that iteration* to a running total, *sum*.

```
1 sum ← 0
2 FOR i ← 1; i ≤ 10; i ← (i + 1) DO
3 |   sum ← (sum + i)
4 END
```

Algorithm 4.5: Summation of Numbers in a For Loop

Again, the initialization of $i = 1$ is only performed once. On the first iteration of the loop, $i = 1$ and so *sum* will be given the value $sum + i = 0 + 1 = 1$ At the end of the loop, i will be incremented will have a value of 2. The continuation condition is still satisfied, so once again the loop body executes and *sum* will be given the value $sum + i = 1 + 2 = 3$. On the 10th (last) iteration, *sum* will have a value $1 + 2 + 3 + \cdots + 9 = 45$ and $i = 10$. Thus $sum + i = 45 + 10 = 55$ after which i will be incremented to 11. The continuation condition is *still* checked, but since $11 \not\leq 10$, the loop body will not be executed and the loop will terminate.

4.3. Do-While Loops

Yet another type of loop is the *do-while* loop. One major difference between this type of loop and the others is that it is always executed *at least once*. The way that this is achieved is that the continuation condition is checked *at the end* of the loop rather than prior to is execution. The same counter-controlled example can be found in Algorithm 4.6.

In contrast to the previous examples, the loop body is executed on the first iteration without checking the continuation condition. Only after the loop body, including the incrementing of the iteration variable i is the continuation condition checked. If *true*, the loop repeats at the beginning of the loop body.

```
1 i ← 1
2 DO
3 |   Perform some action
4 |   i ← (i + 1)
5 WHILE i ≤ 10
```

Algorithm 4.6: Counter-Controlled Do-While Loop

Do-while loops are typically used in scenarios in which the first iteration is used to "setup" the continuation condition (thus, it needs to be executed at least once). A common example is if the loop body performs an operation that may have result in an error code (or flag) that is either *true* (an error occurred) or *false* (no error occurred).

```
1 DO
2 |   Read some data
3 |   isError ← result of reading
4 WHILE isError
```

Algorithm 4.7: Flag-Controlled Do-While Loop

From this perspective, a do-while loop can also be seen as a do-until loop: perform a task *until* some condition is no longer satisfied. The subtle wording difference implies that we'll perform the action before checking to see if it should be performed again.

4.4. Foreach Loops

Many languages support a special type of loop for iterating over individual elements in a collection (such as a set, list, or array). In general, such loops are referred to as *foreach* loops. These types of loops are essentially syntactic sugar: iterating over a collection could be achieved with a for loop or a while loop, but foreach loops provide a more convenient way to iterate when dealing with collections. We will revisit these loops when we examine arrays in Chapter 7. For now, we look at a simple example in Algorithm 4.8.

```
1 FOREACH element a in the collection A DO
2 |   process the element a
3 END
```

Algorithm 4.8: Example Foreach Loop

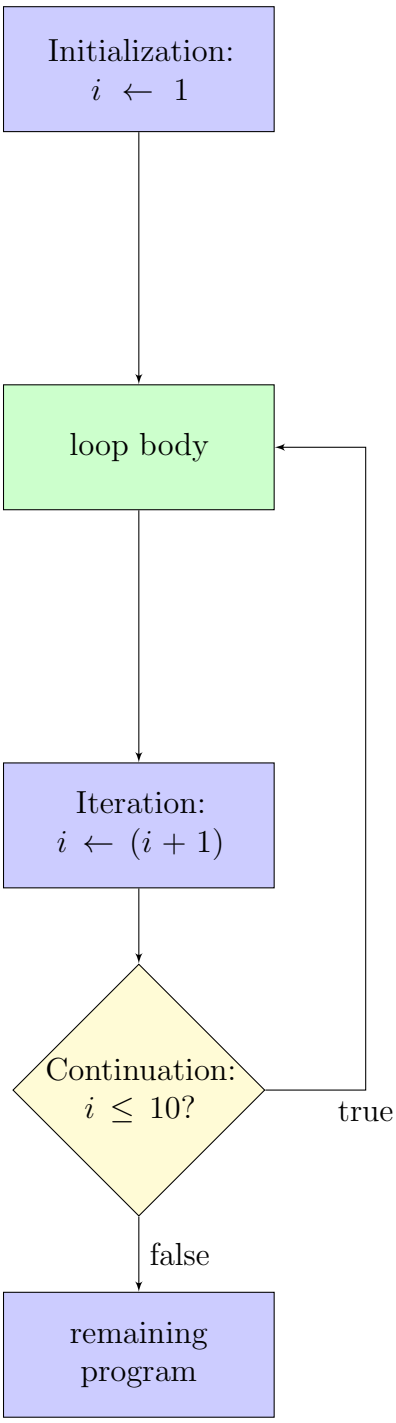


Figure 4.2.: A Do-While Loop Flow Chart. The continuation condition is checked *after* the loop body.

How the elements are stored in the collection and how they are iterated over is not our (primary) concern. We simply want to apply the same block of code to each element, the foreach loop handles the details on how each element is iterated over. The syntax

also provides us a way to refer to each element (the a variable in the algorithm). On each iteration of the loop, the foreach loop updates the reference a to the *next* element in the array. The loop terminates after it has iterated through each and every one of the elements. In this way, a foreach loop simplifies the syntax: we don't have to specify any of the three components ourselves. This is very convenient. As a more concrete example, consider iterating over each student in a course roster. For each student, we wish to compute their grade and then email them the results. The foreach loop allows us to do this without worrying about the iteration details (see Algorithm 4.9).

```
1 FOREACH student s in the class C DO
2 |   g ← compute a's grade
3 |   send a an email informing them of their grade g
4 END
```

Algorithm 4.9: Foreach Loop Computing Grades

4.5. Other Issues

4.5.1. Nested Loops

Just as with conditional statements, we can nest loops within loops to perform more complex processes. Though you can do this with any type of loop, we present a simple example using for loops in Algorithm 4.10.

```
1 n ← 10
2 m ← 20
3 FOR i ← 1; i ≤ m; i ← (i + 1) DO
4 |   FOR j ← 1; j ≤ n; j ← (j + 1) DO
5 |   |   output (i, j)
6 |   END
7 END
```

Algorithm 4.10: Nested For Loops

The outer for loop executes a total of 20 times while the inner for loop executes 10 times. Since the inner for loop is *nested* inside the outer loop, the entire inner loop executes all 10 iterations *for each* of the 20 iterations of the outer loop. Thus, in total the inner most output operation executes $10 \times 20 = 200$ times. Specifically, it outputs $(1,1), (1,2), \ldots, (1,10), (2,1), (2,2), \ldots, (2,10), (3,1), \ldots, (20,10)$.

Some common use cases for nested loops are when you process two dimensional data such as data found in a table that includes rows and columns. Processing matrices is another common use case. Finally, nested loops can be used to process all *pairs* in a collection of elements.

4.5.2. Infinite Loops

Sometimes a simple mistake in the design of a loop can make it execute forever. For example, if we accidentally iterate a variable in the wrong direction or write the *opposite* termination/continuation condition. Such a loop is referred to as an *infinite loop*. As an example, suppose we forgot the increment operation from a previous example.

```
1 sum ← 0
2 i ← 1
3 WHILE i ≤ 10 DO
4 |   sum ← (sum + i)
5 END
```

Algorithm 4.11: Infinite Loop

In Algorithm 4.11 we never make progress toward the terminating condition! Thus, the loop will execute forever, i will continue to have the value 0 and since $0 \leq 10$, the loop body will continue to execute. Care is needed in the design of your loops to ensure that they make progress toward the termination condition.

Most of the time an infinite loop is not something you want and usually you must terminate your buggy program externally (sometimes referred to as "killing" it). However, infinite loops do have their uses. A *poll* loop is a loop that is *intended* to not terminate. At a system level, for example, a computer may *poll* devices (such as input/output devices) one-by-one to see if there is any active input/output request. Instead of terminating, the poll loop simply repeats itself, returning back to the first device. As long as the computer is in operation, we don't want this process to stop. This can be viewed as an infinite loop as it doesn't have any termination condition.

The Zune Bug

Though proper testing and debugging should reduce the likelihood of such bugs, there are several notable instances in which an infinite loop impacted real software. One such instance was the Microsoft *Zune* bug. The Zune was a portable music player, a competitor to the iPod. At about midnight on the night of December 31st, 2008, Zunes everywhere failed to startup properly. A firmware clock driver designed by a 3rd party company contained the following code.

```
1  while(days > 365) {
2    if(IsLeapYear(year)) {
3      if(days > 366) {
4        days -= 366;
5        year += 1;
6     }
7    } else {
8      days -= 365;
9      year += 1;
10   }
11 }
```

Code Sample 4.1: Zune Bug

2008 was a leap year, so the check on line 2 evaluated to *true*. However, though December 31st, 2008 was the 366th day of the year (`days = 366`) the third line evaluated to *false* and the loop was repeated without any of the program state being updated. The problem was "fixed" 24 hours later when it was the 367th day and line 3 worked. The problem was that line 3 *should* have been `days >= 366`).

The failure was that this code was never tested on the "corner cases" that it was designed for. No one thought to test the driver to see if it worked on the last day of a leap year. The code worked the vast majority of the time, but this illustrates the need for rigorous testing.

4.5.3. Common Errors

When writing loops its important to use the proper syntax in the language in which you are coding. Many languages use semicolons to terminate executable statements. However, the `while` statements are not executable: they are part of the control structure of the language and do not have semicolons at the end. A misplaced semicolon could be a syntax error, or it could be syntactically correct but lead to incorrect results. A common error is to place a semicolon at the end of a `while` statement as in

```
while(count <= 10); //WRONG!!!
```

In this example, the `while` loop binds to an *empty* executable statement and results in an infinite loop!

Other common errors are the result of misidentifying either the initialization statement or the continuation condition. Starting a counter at 1 instead of zero, or using a $\leq$ comparison instead of a $<$, etc. These can lead to a loop being *off-by-one* and a resulting error.

Other errors can be avoided by using the proper types of variables. Recall that operations

involving floating-point numbers can have round off and precision errors, $\frac{1}{3} + \frac{1}{3} + \frac{1}{3}$ may not be equal to one for example. It is best to avoid using floating-point numbers or comparisons in the control of your loops. Boolean and integer types are much less error prone.

Finally, you must always ensure that your loops are making progress *toward* the termination condition. A failure to properly increment a counter can lead to incorrect results or even an infinite loop.

4.5.4. Equivalency of Loops

It might not seem obvious at first, but in fact, *any* type of loop can be re-written as another type of loop and perform *equivalent* operations. That is, any while loop can be rewritten as an equivalent for loop. Any do-while loop can be rewritten as an equivalent while loop!

So why do we have different types of loops? The short answer is that we want our programming languages to be as flexible as possible. We could design a language in which every loop *had* to be a while loop for example, but there are some situations in which it would be less "natural" to write code with a while loop. By providing several options, programmers have the choice of which type of loop to write.

In general, there are no "rules" as to which loop to apply to which situation. There are general trends, best practices, and situations where it is more common to use one loop rather than another, but in the end it does come down to personal choice and style. Some software projects or organizations may have established guidelines or accepted style that may need to be followed in the interest of consistency and uniformity.

4.6. Problem Solving With Loops

Loops can be applied to any problem that requires repetition of some sort or two simplify repeated code. When designing loops, it is important to identify the three components by asking the questions:

- Where does the loop start? What variables or other state may need to be initialized or setup prior to the loop beginning?
- What code needs to be repeated? How can it be generalized to depend on loop control variables? This helps you to identify and write the loop body.
- When should the loop end? How many times do we want it to execute? This helps you to identify the continuation and/or termination condition.
- How do we make progress toward the termination condition? What variable(s) need to be incremented and how?

4.7. Examples

4.7.1. For vs While Loop

Recall the classic geometric series,

$$\frac{1}{1-x} = \sum_{k=0}^{\infty} x^k = 1 + x + x^2 + x^3 + \cdots$$

Obviously a computer cannot compute an infinite series as it is required to terminate in a finite number of steps. Thus, we can approach this problem in a number of different ways.

One way we could approximate the series is to compute it out to a fixed, say n, number of terms. To do so, we could initialize a *sum* variable to zero, then iteratively compute and add terms to the *sum* until we have computed n terms. To keep track of the terms, we can define a counter variable, k as in the summation.

Following our strategy, we can identify the initialization: k should start at 0. The iteration is also easy: k should be incremented by 1 each time. The continuation condition should continue the loop until we have computed n terms. However, since k starts at 0, we would want to continue while $k < n$. We would not want to continue the iteration when $k = n$ as that would make $n + 1$ iterations (again since k starts at 0). Further, since we know the number of iterations we want to execute, a for loop is arguably the most appropriate loop for this problem. Such a solution is presented in Algorithm 4.12.

```
   INPUT   : x, n ≥ 0
1  sum ← 0
2  FOR k = 0; k < n; k ← (k + 1) DO
3  |   sum ← (sum + x^k)
4  END
5  output sum
```

Algorithm 4.12: Computing the Geometric Series Using a For Loop

As an alternative consider the following approach: instead of computing a predefined number of terms, what if we computed terms until the difference between the value in the previous iteration and the value in the current iteration is negligible, say less than some small ϵ amount. We could stop our computation because any further iterations would only affect the summation less and less. That is, the current value represents a "good enough" approximation. That way, if someone wanted an even better approximation, they could specify a smaller ϵ.

This approach will be more straightforward with a while loop since the continuation condition will be more along the lines of "while the estimation is not yet good enough,

continue the summation." This approach will also be easier if we keep track of both a *current* and a *previous* value of the summation, then computing and checking the difference will be easier.

```
   INPUT  : x, ε > 0
1  sum_prev ← 0
2  sum_curr ← 1
3  k ← 1
4  WHILE |sum_prev − sum_curr| ≥ ε DO
5  |   sum_prev ← sum_curr
6  |   sum_curr ← (sum_curr + x^k)
7  |   k ← (k + 1)
8  END
9  output sum
```

Algorithm 4.13: Computing the Geometric Series Using a While Loop

4.7.2. Primality Testing

An integer $n > 1$ is called *prime* if the only integers that divide it are 1 and itself. Otherwise it is called *composite*. For example, 30 is composite as it is divisible by 2, 3, and 5 among others. However, 31 is prime as it is only divisible by 1 and 31.

Consider the problem of determining whether or not a given integer n is prime or composite, referred to as *primality testing*. A straightforward way of determining this is to simply try dividing by every integer 2 up to $\sqrt{n}$: if any of these integers divides n, then n is composite. Otherwise, if none of them do, n is prime. Observe that we only need to go up to $\sqrt{n}$ since any prime divisor greater than that will correspond to *some* prime divisor less than $\sqrt{n}$.

A for loop can thus be constructed: our initialization clearly starts at $i = 2$, incrementing by 1 each time until i has exceeded $\sqrt{n}$. This solution is presented in Algorithm 4.14. Of course this is certainly not the most efficient way to solve this problem, but we will not go into more advanced algorithms here.

Now consider this more general problem: given an integer $m > 1$, determine *how many* prime numbers $\leq m$ there are. A key observation is that we've *already solved* part of the problem: determining if a given number is prime in the previous exercise. To solve this problem, we could *reuse* or adapt our previous solution. In particular, we could surround the previous solution in an *outer* loop and iterate over integers from 2 up to m. The inner loop would then determine if the integer is prime and instead of outputting a result, could keep track of a counter of the number of primes. This solution is presented

```
   INPUT  : n > 1
1  FOR i ← 2; i ≤ √n; i ← (i + 1) DO
2  |  IF i divides n THEN
3  |  |  output composite
4  |  END
5  END
6  output prime
```

Algorithm 4.14: Determining if a Number is Prime or Composite

in Algorithm 4.15.

```
   INPUT  : m > 1
1  numberOfPrimes ← 0
2  FOR j = 2; j ≤ m; j ← (j + 1) DO
3  |  isPrime ← true
4  |  FOR i ← 2; i ≤ √j; i ← (i + 1) DO
5  |  |  IF i divides j THEN
6  |  |  |  isPrime ← false
7  |  |  END
8  |  END
9  |  IF isPrime THEN
10 |  |  numberOfPrimes ← (numberOfPrimes + 1)
11 |  END
12 END
13 output numberOfPrimes
```

Algorithm 4.15: Counting the number of primes.

4.7.3. Paying the Piper

Banks issue loans to customers as one lump sum called a *principle* P that the borrower must pay back over a number of *terms*. Usually payments are made on a monthly basis. Further, banks charge an amount of *interest* on a loan measured as an Annual Percentage Rate (APR). Given these conditions, the borrower makes monthly payments determined by the following formula.

$$monthlyPayment = \frac{iP}{1 - (1 + i)^{-n}}$$

Where $i = \frac{apr}{12}$ is the monthly interest rate, and n is the number of terms (in months).

For simplicity, suppose we borrow $P = \$1,000$ at 5% interest ($apr = 0.05$) to be paid back over a term of 2 years ($n = 24$). Our monthly payment would thus be

$$monthlyPayment = \frac{\frac{.05}{12} \cdot 1000}{1 - (1 + \frac{.05}{12})^{-24}} = \$43.87$$

When the borrower makes the first month's payment, some of it goes to interest, some of it goes to paying down the balance. Specifically, one month's interest on $1,000 is

$$\$1,000 \cdot \frac{0.05}{12} = \$4.17$$

and so $\$43.87 - \$4.17 = \$39.70$ goes to the balance, making the new balance $960.30. The next month, this new balance is used to compute the new interest payment,

$$\$960.30 \cdot \frac{0.05}{12} = \$4.00$$

And so on until the balance is fully paid. This process is known as *loan amortization*.

Let's write a program that will calculate a loan amortization schedule given the inputs as described above. To start, we'll need to compute the monthly payment using the formula above and for that we'll need a monthly interest rate. The balance will be updated month-to-month, so we'll use another variable to represent that that gets updated. Finally, we'll want to track the current month in the loan schedule process.

Once we have these variables setup, we can start a loop that will repeat once for each month in the loan schedule. We could do this using either type of loop, but for this exercise, let's use a while loop. Using our *month* variable, we'll start by initializing it to 1 and run the loop through the last month, n.

On each iteration compute that month's interest and principle payments as above, update the balance, and also be sure to update our *month* counter variable to ensure we're making progress toward the termination condition. On each iteration we'll also output each of these variables to the user. The full program can be found in Algorithm 4.16.

If we were to actually implement this we'd need to be more careful. This outlines the basic process, but keep in mind that US dollars are only accurate to cents. A monthly payment can't be $43.871 cents. We'll need to take care to round properly. This introduces another issue: by rounding the final month's payment may not match the expected monthly payment (we may over or under pay in the final month). An actual implementation may need to handle the final month's payment separately with different logic and operations

than are inside the loop.

```
   INPUT  : A principle, P, a number of terms, n, an APR, apr
   OUTPUT : A loan amortization schedule
 1 balance ← P //The initial balance is the principle
 2 i ← apr/12 //monthly interest rate
 3 monthlyPayment ← iP/(1−(1+i)^−n)
 4 month ← 1 //A month counter
 5 WHILE month ≤ n DO
 6 |   monthInterest ← i · balance
 7 |   monthPrinciple ← monthlyPayment − monthInterest
 8 |   balance ← balance − monthPrinciple
 9 |   month = (month + 1)
10 |   output month, monthInterest, monthPrinciple, balance
11 END
```

Algorithm 4.16: Computing a loan amortization schedule

4.8. Exercises

Exercise 4.1. Write a for-loop and a while-loop that accomplishes each of the following.

(a) Prints all integers 1 thru 100 on the same line delimited by a single space

(b) Prints all even integers 0 up to n in reverse order

(c) A list of integers divisible by 3 between a and b where a, b are parameters or inputs

(d) Prints all positive powers of two up to 2^{30} : $1, 2, 4, \ldots, 1073741824$ one value per line (try computing up to 2^{31} and 2^{32} and discern reasons for why it may fail)

(e) Prints all even integers 2 thru 200 on 10 different lines (10 numbers on each line) *in reverse order*

(f) Prints the following pattern of numbers (hint: use two nested loops; the result can be computed using some value of $i + 10j$)

```
 11  21  31  41  51  61  71  81  91 101
 12  22  32  42  52  62  72  82  92 102
 13  23  33  43  53  63  73  83  93 103
 14  24  34  44  54  64  74  84  94 104
 15  25  35  45  55  65  75  85  95 105
 16  26  36  46  56  66  76  86  96 106
 17  27  37  47  57  67  77  87  97 107
 18  28  38  48  58  68  78  88  98 108
 19  29  39  49  59  69  79  89  99 109
 20  30  40  50  60  70  80  90 100 110
```

Exercise 4.2. Civil engineers have come up with two different models on how a city's population will grow over the next several years. The first projection assumes a 10% annual growth rate while the second projection assumes a linear growth rate of 50,000 additional citizens per year. Write a program to project the population growth under both models. Take, as input, the initial population of the city along with a number of years to project the population.

In addition, compute how many years it would take to *double* the population under each model.

Exercise 4.3. Write a loan program similar to the amortization schedule program we developed in Section 4.7.3. However, give the user an option to specify an *extra* monthly payment amount in order to pay off the loan early. Calculate how much quicker the loan gets paid off and how much they save in interest.

Exercise 4.4. The rate of decay of a radioactive isotope is given in terms of its half-life H, the time lapse required for the isotope to decay to one-half of its original mass. For example, the isotope Strontium-90 (^{90}Sr) has a half-life of 28.9 years. If we start with 10kg of Strontium-90 then 28.9 years later you would expect to have only 5kg of Strontium-90 (and 5kg of Yttrium-90 and Zirconium-90, isotopes which Strontium-90 decays into).

Write a program that takes the following input:

- Atomic Number (integer)
- Element Name
- Element Symbol
- H (half-life of the element)
- m, an initial mass in grams

Your program will then produce a table detailing the amount of the element that remains after each year until less than 50% of the original amount remains. This amount can be computed using the following formula:

$$r = m \times \left(\frac{1}{2}\right)^{(y/H)}$$

y is the number of years elapsed, and H is the half-life of the isotope in years.

For example, using your program on Strontium-90 (symbol: Sr, atomic number: 38) with a half-life of 28.9 years and an initial amount of 10 grams would produce a table something like:

```
 Strontium-90 (38-Sr)
Elapsed Years    Amount
------------------------
-                10g
1                9.76g
2                9.53g
3                9.30g
...
28               5.11g
29               4.99g
```

Exercise 4.5. In this exercise, you will develop a program that assists people in saving for a retirement using a tax-deferred 401k program.

Your application will allow a user to enter the following inputs:

- An initial starting balance
- A monthly contribution amount (we'll assume its the same over the life of the savings plan)
- An (average) annual rate of return
- An (average) annual rate of inflation

In addition, your program will allow a user to choose between two different scenarios:

- The first will allow the user to input a number of *years* left until retirement. It will then compute a monthly savings table which will be a projection out to that many years.
- The second will take a *target* dollar amount and compute a monthly savings table until the account balance has reached this target dollar amount.

The monthly interest rate should be inflation-adjusted. The inflation-adjusted rate of return can be computed with the following formula.

$$\frac{1 + \text{rate of return}}{1 + \text{inflation rate}} - 1$$

To get the monthly rate, simply divide by 12. Each month, interest is applied to the balance at this rate along with the monthly contribution amount.

An example: if we start with $10,000 and contribute $500 monthly with a return rate of 9% and an inflation rate of 1.2%, the first few lines of our table would something like the following.

```
Payment     Interest Earned  Contribution   Balance
1           $   64.23        $ 500.00       $ 10564.23
2           $   67.85        $ 500.00       $ 11132.08
3           $   71.50        $ 500.00       $ 11703.58
4           $   75.17        $ 500.00       $ 12278.75
...
```

Exercise 4.6. Write a program that computes various statistics on a collection of numbers that can be read in from the command line, as command line arguments, or via other means. In particular, given n numbers,

$$x_1, x_2, \ldots, x_n$$

your program should compute the following statistics:

- The minimum number
- The maximum number
- The mean,
$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$
- The variance,
$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$
- And the standard deviation,
$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

where n is the number of numbers that was provided. For example, with the numbers,

$$3.14, 2.71, 42, 3, 13$$

your output should look something like:

```
Minimum:      2.71
Maximum:     42.00
Mean:        12.77
Variance:   228.77
Standard
Deviation:   15.13
```

Exercise 4.7. The ancient Greek mathematician Euclid developed a method for finding the greatest common divisor of two positive integers, a and b. His method is as follows:

1. If the remainder of a/b is 0 then b is the greatest common divisor.
2. If it is not 0, then find the remainder r of a/b and assign b to a and the remainder r to b.
3. Return to step (1) and repeat the process.

Write a program that uses a function to perform this procedure. Display the two integers and the greatest common divisor.

Exercise 4.8. Write a program to estimate the value of $e \approx 2.718281\ldots$ using the series:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

Obviously, you will need to restrict the summation to a finite number of n terms.

Exercise 4.9. The value of π can be expressed by the following infinite series:

$$\pi = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \cdots\right)$$

An approximation can be made by taking the first n terms of the series. For $n = 4$, the approximation is

$$\pi \approx 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7}\right) = 2.8952$$

Write a program that takes n as input and outputs an approximation of π according to the series above.

Exercise 4.10. The sine function can be approximated using the following Taylor series.

$$\sin(x) = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$$

Write a function that takes x and n as inputs and approximates $\sin x$ by computing the first n terms in the series above.

Exercise 4.11. One way to compute π is to use Machin's formula:

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}$$

To compute the arctan function, you could use the following series:

$$\arctan x = \frac{x}{1} - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots = \sum_{i=0}^{\infty} \frac{(-1)^i}{2k+1} x^{2i+1}$$

Write a program to estimate π using these formulas but allowing the user to specify how many terms to use in the series to compute it. Compare the estimate with the built-in definition of π in your language.

Exercise 4.12. The *arithmetic-geometric* mean of two numbers x, y, denoted $M(x, y)$ (or $\text{agm}(x, y)$) can be computed iteratively as follows. Initially, $a_1 = \frac{1}{2}(x + y)$ and $g_1 = \sqrt{xy}$ (i.e. the normal arithmetic and geometric means). Then, compute

$$\begin{aligned} a_{n+1} &= \tfrac{1}{2}(a_n + g_n) \\ g_{n+1} &= \sqrt{a_n g_n} \end{aligned}$$

The two sequences will converge to the same number which is the arithmetic-geometric mean of x, y. Obviously we cannot compute an infinite sequence, so we compute until $|a_n - g_n| < \epsilon$ for some small number ϵ.

Exercise 4.13. The integral of a function is a measure of the area under its curve. One numerical method for computing the integral of a function $f(x)$ on an interval $[a, b]$ is the *rectangle* rule. Specifically, an interval $[a, b]$ is split up into n equal subintervals of size $h = \frac{b-a}{n}$. Then the integral is approximated by computing:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(a + ih) \cdot h$$

Write a program to approximate an integral using the rectangle method. For this particular exercise you will integrate the function

$$f(x) = \frac{\sin x}{x}$$

For reference, the function is depicted in Figure 4.3. Write a program that will read the end points a, b and the number of subintervals n and computes the integral of f using the rectangle method. It should then output the approximation.

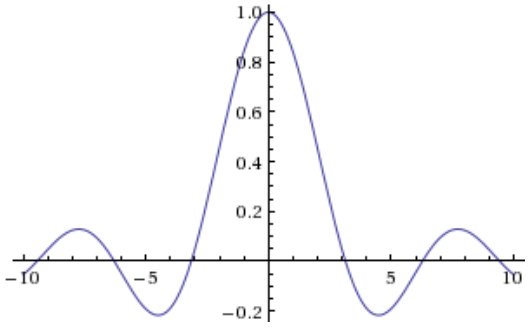


Figure 4.3.: Plot of $f(x) = \frac{\sin x}{x}$

Exercise 4.14. Another way to compute an integral is to a technique called *Monte Carlo Integration*, a randomized numerical integration method.

Given the interval $[a, b]$, we enclose the function in a region of interest with a rectangle of a known area A_r. We then randomly select n points within the rectangle and count

the number of random points that are within the function's curve. If m of the n points are within the curve, we can estimate the integral to be

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{m}{n} A_r$$

Consider again the function $f(x) = \frac{\sin(x)}{x}$. Note that the global maximum and minimum of this function are 1 and ≈ -0.2172 respectively. Therefore, we can also restrict the rectangle along the y-axis from $-.25$ to 1. That is, the lower left of the rectangle will be $(a, -.25)$ and the upper right will be $(b, 1)$ for a known area of

$$A_r = |a - b| \times 1.25$$

Figure 4.4 illustrates the rectangle for the interval $[-5, 5]$.

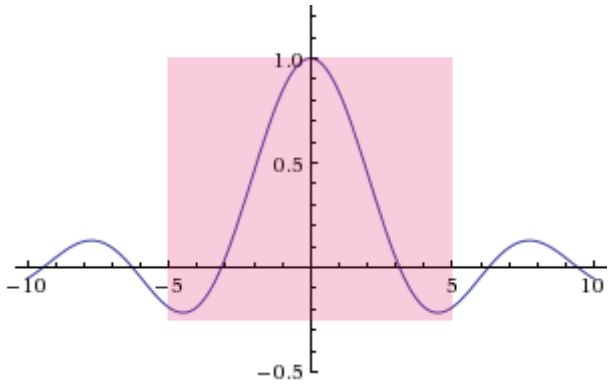


Figure 4.4.: A rectangle for the interval $[-5, 5]$.

Write a program that will takes as input interval values a, b, and an integer n and perform a Monte Carlo estimate of the integral of the function above. Realize that this is just an approximation and it is randomized so your answers may not match exactly and may be different on various executions of your program. Take care that you handle points within the curve but under the x-axis correctly.

Exercise 4.15. Consider a ball trapped in a 2-D box. Suppose that it has an initial position (x, y) within the box (the box's dimensions are specified by its lower left (x_ℓ, y_ℓ) and an upper right (x_r, y_r) points) along with an initial angle of travel θ in the range $[0, 2\pi)$. As the ball travels in this direction it will eventually collide with one of the sides of the box and bounce off. For this model, we will assume no loss of velocity (it keeps going) and its angle of reflection is perfect.

Write a program that takes as input, $x, y, \theta, x_\ell, y_\ell, x_r, y_r$, and an integer n and computes the first $n - 1$ Euclidean points on the box's perimeter that the ball bounces off of in its travel (include the initial point in your printout for a total of n points). You may assume that the input will always be "good" (the ball will always begin somewhere inside the box and the lower left and upper right points will not be reversed).

As an example, consider the inputs:

$$x = 1, y = 1, \theta = .392699, x_\ell = 0, y_\ell = 0, x_r = 4, y_r = 3, n = 20$$

Starting at $(1, 1)$, the ball travels up and to the right bouncing off the right wall. Figure 4.5 illustrates this and the subsequent bounces back and forth.

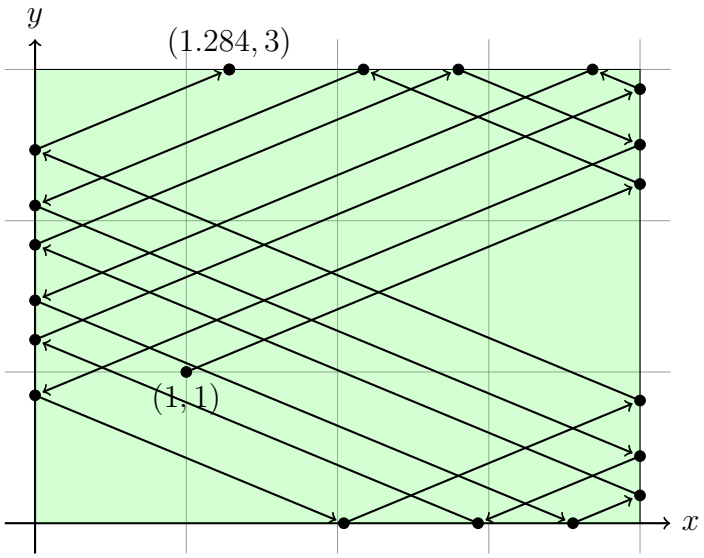


Figure 4.5.: Follow the bouncing ball

Your output should simply be the points and should look something like the following.

```
(1.000000, 1.000000)
(4.000000, 2.242640)
(2.171572, 3.000000)
(0.000000, 2.100506)
(4.000000, 0.443652)
(2.928929, 0.000000)
(0.000000, 1.213202)
(4.000000, 2.870056)
(3.686287, 3.000000)
(0.000000, 1.473090)
(3.556355, 0.000000)
(4.000000, 0.183764)
(0.000000, 1.840617)
(2.798998, 3.000000)
(4.000000, 2.502529)
(0.000000, 0.845675)
(2.041640, 0.000000)
(4.000000, 0.811179)
(0.000000, 2.468033)
(1.284282, 3.000000)
```

Exercise 4.16. An integer $n \geq 2$ is *prime* if its only divisors are 1 and itself, n. For example, $2, 3, 5, 7, 11, \ldots$ are primes. Write a program that outputs all prime numbers 2 up to m where m is read as input.

Exercise 4.17. An integer $n \geq 2$ is *prime* if the only integers that evenly divide it are 1 and n itself, otherwise it is *composite*. The *prime factorization* of an integer is a list of its prime divisors along with their multiplicities. For example, the prime decomposition of $188,760$ is:

$$188,760 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 5 \cdot 11 \cdot 11 \cdot 13$$

Write a program that takes an integer n as input and outputs the prime factorization of n. If n is invalid, an appropriate error message should be displayed instead. Your output should look something like the following.

```
1001 = 7 * 11 * 13
```

Exercise 4.18. One way of estimating π is to randomly sample points within a 2×2 square centered at the origin. If the distance between the randomly chosen point (x, y) and the origin is less than or equal to 1, then the point lies inside the unit circle centered at the origin and we count it. If the point lies outside the circle then we can ignore it. If

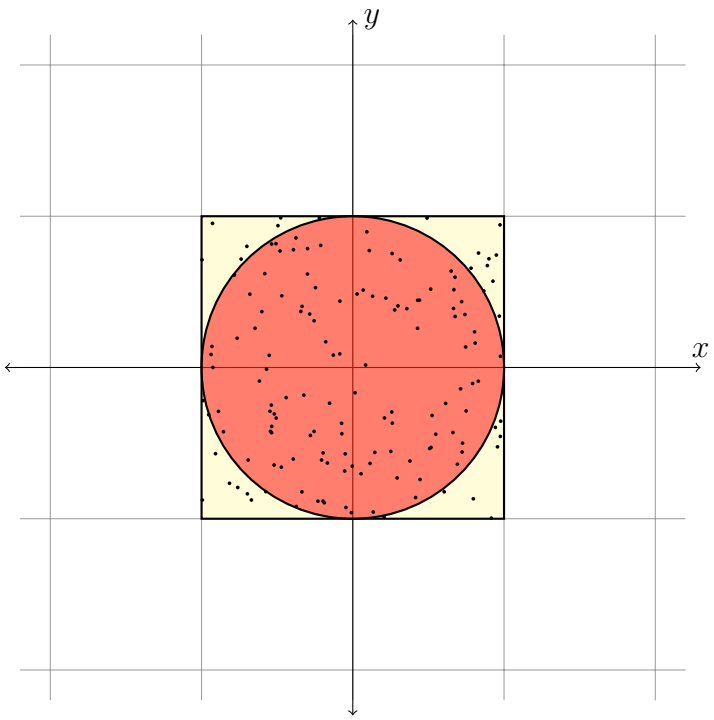


Figure 4.6.: Sampling points in a circle

we sample n points and m of them lie within the circle, then π can be estimated as

$$\pi \approx \frac{4m}{n}$$

Given a point (x, y), its distance from the origin is simply

$$\sqrt{x^2 + y^2}$$

This idea is illustrated in Figure 4.6. Example code is given to randomly generate numbers within a bound. Write a program that takes an integer n as input and randomly samples n points within the 2×2 square and outputs an approximation of π.

Of course, you'll need a way to generate random numbers within the range $[-1, 1]$. Since you are using some randomization, the result is just an *approximation* and may not match exactly or even be the same between two different runs of your program.

Exercise 4.19. A regular polygon is a polygon that is equiangular. That is, it has n sides and n points whose angle from the center are all equal in measure. Examples for $n = 3$ through $n = 8$ can be found in Figure 4.7.

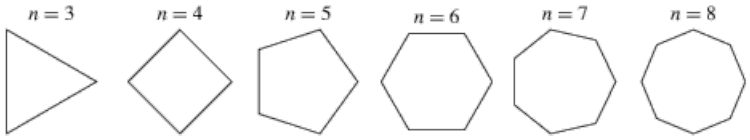


Figure 4.7.: Regular polygons

Write a program that takes n and a radius r as inputs and computes the points of a regular n-sided polygon centered at the origin $(0, 0)$. Each point should be distance r from the origin and the first point should lie on the positive x-axis. Each subsequent point should be at an angle θ equal to $\frac{2\pi}{n}$ from the previous point. Recall that given the polar coordinates θ, r we can convert to cartesian coordinates (x, y) using the following.

$$\begin{aligned} x &= r \cdot \cos\theta \\ y &= r \cdot \sin\theta \end{aligned}$$

Your program should be robust enough to check for invalid inputs. If invalid, an error message should be printed and the program should exit.

For example, running your program with $n = 5, r = 6$ should produce the points of a pentagon with "radius" 6. The output should look something like:

```
Regular 5-sided polygon with radius 6.0:
(6.0000, 0.0000)
(1.8541, 5.7063)
(-4.8541, 3.5267)
(-4.8541, -3.5267)
(1.8541, -5.7063)
```

Exercise 4.20. Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be two points in the cartesian plane which define a *line* segment. Suppose we travel along this line starting at p_1 taking n

steps that are an equal distance apart until we reach p_2. We wish to know which points correspond to each of these steps and which step along this path is *closest* to anther point $p_3 = (x_3, y_3)$. Recall that the distance between two points can be computed using the Euclidean distance formula:

$$\delta = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Write a program that takes three points and an integer n as inputs and outputs a sequence of points along the line defined by p_1, p_2 that are distance $\frac{\delta}{n}$ apart from each other. It should also indicate which of these computed points is the closest to the third point. For example, the execution of your program with inputs $0, 2, -5.5, 7.75, -2, 3, 10$ should produce output that looks something like:

```
(0.00, 2.00) to (-5.50, 7.75) distance: 7.9569
(0.00, 2.00)
(-0.55, 2.58)
(-1.10, 3.15)
(-1.65, 3.72)  <-- Closest point to (-2, 3)
(-2.20, 4.30)
(-2.75, 4.88)
(-3.30, 5.45)
(-3.85, 6.02)
(-4.40, 6.60)
(-4.95, 7.17)
(-5.50, 7.75)
```

Exercise 4.21. The natural log of a number x is usually computed using some numerical approximation method. One such method is to use the following Taylor series.

$$\ln x = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \cdots$$

However, this only works for $|x - 1| \leq 1$ (except for $x = 0$) and diverges otherwise. For x such that $|x| > 1$, we can use the series

$$\ln \frac{y}{y-1} = \frac{1}{y} + \frac{1}{2y^2} + \frac{1}{3y^3} + \cdots$$

where $y = \frac{x}{x-1}$. Of course such an infinite computation cannot be performed by a computer. Instead, we approximate $\ln x$ by computing the series out to a finite number of terms, n. Your program should print an error message and exit for $x \leq 0$; otherwise it should use the first series for $0 < x \leq 1$ and the second for $x > 1$.

Another series that has better convergence properties and works for any range of x is as follows

$$\ln x = \ln \frac{1+y}{1-y} = 2y \left(1 + \frac{1}{3}y^2 + \frac{1}{5}y^4 + \frac{1}{7}y^6 \cdots\right)$$

where $y = \frac{(x-1)}{(x+1)}$.

You will write a program that approximates $\ln x$ using these two methods computed to n terms. You will also compute the error of each method by comparing the approximated value to the standard math library's `log` function.

Your program should accept x and n as inputs. It should be robust enough to reject any invalid inputs ($\ln x$ is not defined for $x = 0$ you may also print an error for any negative value; n must be at least one). It will then compute an approximation using both methods and print the relative error of each method.

For example, the execution of your program with inputs $3.1415, 6$ should produce output that looks something like:

```
Taylor Series: ln(3.1415) ~= 1.11976
  Error: 0.02494
Other Series:  ln(3.1415) ~= 1.14466
  Error: 0.00004
```

Exercise 4.22. There are many different numerical methods to compute the square root of a number. In this exercise, you will implement several of these methods.

(a) The Exponential Identity Method involves the following identity:

$$\sqrt{x} = e^{\frac{1}{2}\ln(x)}$$

Which assumes the use of built-in (or math-library) functions for e and the natural log, ln.

(b) The Babylonian Method involves iteratively computing the following recurrence:

$$a_i = \frac{1}{2}\left(a_{i-1} + \frac{x}{a_{i-1}}\right)$$

where $a_1 = 1.0$. Computation is repeated until $|a_i - a_{i-1}| \leq \delta$ where δ is some small constant value.

(c) A method developed for one of the first electronic computers (EDSAC [18]) involves the following iteration. Let $a_0 = x$, $c_0 = x - 1$. Then compute

$$\begin{aligned} a_{i+1} &= a_i - \frac{a_i c_i}{2} \\ c_{i+1} &= \frac{c_i^2(c_i - 3)}{4} \end{aligned}$$

The iteration is performed for as many iterations as specified (n), or until the change in a is negligible. The resulting value for a is used as an approximation for $\sqrt{x} \approx a$. However, this method only works for values of x such that $0 < x < 3$. We can easily overcome this by *scaling* x by some power of 4 so that the scaled value of x satisfies $\frac{1}{2} \leq x < 2$. After applying the method we can then scale back up by the appropriate value of 2 (since $\sqrt{4} = 2$). Algorithm 4.17 describes how to scale x.

Write a program to compute the square root of an input number using these methods and compare your results.

```
1 power ← 0
2 WHILE x < 1/2 DO
  |  //Scale up
3 |  x ← (x · 4)
4 |  power ← (power − 1)
5 END
6 WHILE x ≥ 2 DO
  |  //Scale down
7 |  x ← x/4
8 |  power ← (power + 1)
9 END
```

Algorithm 4.17: Scaling a value x so that it satisfies $\frac{1}{2} \leq x < 2$. After execution, *power* indicates what power of 2 the value x was scaled by.

Exercise 4.23. There are many different numerical methods to compute the natural logarithm of a number, $\ln x$. In this exercise, you will implement several of these methods.

(a) A formula version approximates the natural logarithm as:

$$\ln(x) \approx \frac{\pi}{2M(1, 4/s)} - m \ln(2)$$

Where $M(a, b)$ is the *arithmetic-geometric* mean and $s = x2^m$. In this formula, m is a parameter (a larger m provides more precision).

(b) The standard Taylor Series for the natural logarithm is:

$$\ln(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n}(x-1)^n$$

As we cannot compute an infinite series, we will simply compute the series to the first m terms. Also note that this series is not convergent for values $x > 1$

(c) Borchardt's algorithm is an iterative method that works as follows. Let

$$a_0 = \frac{1+x}{2} \qquad b_0 = \sqrt{x}$$

Then repeat:

$$\begin{aligned} a_{k+1} &= \tfrac{a_k + b_k}{2} \\ b_{k+1} &= \sqrt{a_{k+1} b_k} \end{aligned}$$

until the absolute difference between a_k, b_k is small; that is $|a_k - b_k| < \epsilon$. Then the logarithm is approximated as

$$\ln(x) \approx 2\frac{x-1}{a_k + b_k}$$

(d) Newton's method works if x is sufficiently close to 1. It works by setting $y_0 = 1$ and then computing

$$y_{n+1} = y_n + 2\frac{x - e^{y_n}}{x + e^{y_n}}$$

The iteration is performed m times.

To ensure that some of the methods above work, you may need to *scale* the number x to be as close as possible to 1. One way to do this is to divide or multiply by e until x is close to 1. Suppose we divided by e k times; that is $x = z \cdot e^k$ where z is close to 1. Then

$$\ln(x) = \ln(z \cdot e^k) = \ln(z) + \ln(e^k) = \ln(z) + k$$

Thus, we can apply the methods above to Newton's method to z and add k to the result to get $\ln(x)$. A similar trick can be used to ensure that the Taylor Series method is convergent.

Exercise 4.24. Consider the following variation on the classical "FizzBuzz" challenge. Write a program that will print out numbers 1 through n where n is provided as a command line argument. However, if the number is a perfect square (that is, the square of some integer; for example $1 = 1^2, 4 = 2^2, 9 = 3^2$, etc.) print "Go Huskers" instead. If the number is a prime $(2, 3, 5, 7$, etc.) print "Go Cubs" instead.

Exercise 4.25. Write a program that takes an integer n and a subsequent list of integers as command line arguments and determines which number(s) between 1 and n are missing from the list. For example, if the following numbers are given to the program: `10 5 2 3 9 2 8 8` your output should look something like:

```
Missing numbers 1 thru 10:
1, 4, 6, 7, 10
```

Exercise 4.26. Write a program that takes a list of pairs of numbers representing latitudes/longitudes (on the scale $[-180, 180]$ (negative values correspond to the southern and western hemispheres). Then, starting with the first pair, calculate the intermediate air distances between each location as well as a final total distance.

To compute air distance from location A to a location B, use the Spherical Law of Cosines:

$$d = \arccos\left(\sin(\varphi_1)\sin(\varphi_2) + \cos(\varphi_1)\cos(\varphi_2)\cos(\Delta)\right) \cdot R$$

where

- φ_1 is the latitude of location A, φ_2 is the latitude of location B

- Δ is the difference between location B's longitude and location A's longitude
- R is the (average) radius of the earth, 6,371 kilometers

Note: the formula above assumes that latitude and longitude are measured in radians r, $-\pi \leq r \leq \pi$. To convert from degrees deg ($-180 \leq deg \leq 180$) to radians r, you can use the simple formula:

$$r = \frac{deg}{180}\pi$$

For example, if the command line arguments were

```
40.8206 -96.756 41.8806 -87.6742 41.9483 -87.6556 28.0222 -81.7329
```

your output should look something like:

```
(40.8206, -96.7560) to (41.8806, -87.6742): 766.8053km
(41.8806, -87.6742) to (41.9483, -87.6556): 7.6836km
(41.9483, -87.6556) to (28.0222, -81.7329): 1638.7151km
Total Distance: 2413.2040
```

Exercise 4.27. A DNA sequence is made up of a sequence of four nucleotide bases, A, C, G, T (adenine, cytosine, guanine, thymine). One particularly interesting statistic of a DNA sequence is finding a *CG island*: a subsequence that contains the highest frequency of guanine and cytosine.

For simplicity, we will be interested in subsequences of a particular length, n that will be provided as part of the input.

Write a program that takes, as command line arguments, an integer n and a DNA sequence. The program should then find all subsequences of the given DNA string of length n with the maximal frequency of C and G in it. For example, if the DNA sequence is

```
ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCCACGGC
```

and the "window" size that we're interested in is $n = 5$ then you would scan the sequence and find every subsequence with the maximum number of C or G bases. Your output should include *all* CG Islands (by indices) in the sequence similar to the following.

```
n = 5
highest frequency: 5 / 5 = 100.00%
CG Islands:
15 thru 20: CCCCC
16 thru 21: CCCCG
17 thru 22: CCCGG
18 thru 23: CCGGC
19 thru 24: CGGCC
42 thru 47: CCGGG
43 thru 48: CGGGG
44 thru 49: GGGGC
45 thru 50: GGGCC
```

Exercise 4.28. Write a program that will assist people in saving for retirement using a tax-deferred 401k program.

Your program will read the following inputs as command line arguments.

- An initial starting balance
- A monthly contribution amount (we'll assume its the same over the life of the savings plan)
- An (average) annual rate of return (on the scale $[0, 1]$)
- An (average) annual rate of inflation (on the scale $[0, 1]$)
- A number of years until retirement

Your program will then compute a monthly savings table detailing the (inflation-adjusted) interest earned each month, contribution, and new balance. The inflation-adjusted rate of return can be computed with the following formula.

$$\frac{1 + \text{rate of return}}{1 + \text{inflation rate}} - 1$$

To get the monthly rate, simply divide by 12. Each month, interest is applied to the balance at this rate (prior to the monthly deposit) and the monthly contribution is added. Thus, the earnings compound month to month.

Be sure that your program handles bad inputs as well as it can. It should also round to the nearest cent for every figure. Finally, as of 2014, annual 401k contributions cannot exceed $17,500. If the user's proposed savings schedule violates this limit, display an error message instead of the savings table.

For inputs `10000 500 0.09 0.012 10` your output should look something like the following:

```
Month    Interest     Balance
    1 $     64.23 $  10564.23
    2 $     67.85 $  11132.08
    3 $     71.50 $  11703.58
    4 $     75.17 $  12278.75
    5 $     78.87 $  12857.62
    6 $     82.58 $  13440.20
    7 $     86.33 $  14026.53
    8 $     90.09 $  14616.62
    9 $     93.88 $  15210.50
...
  116 $    678.19 $ 106767.24
  117 $    685.76 $ 107953.00
  118 $    693.37 $ 109146.37
  119 $    701.04 $ 110347.41
  120 $    708.75 $ 111556.16
Total Interest Earned: $  41556.16
Total Nest Egg: $ 111556.16
```

Exercise 4.29. An *affine cipher* is an encryption scheme that encrypts messages using the following function:

$$e_k(x) = (ax + b) \bmod n$$

Where n is some integer and $0 \leq a, b, x \leq n - 1$. That is, we fix n, which will be used to encode an alphabet as in Table 4.1.

x	character
0	(space)
1	A
2	B
3	C
⋮	⋮
25	Y
26	Z
27	.
28	!

Table 4.1.: Character Mapping for $n = 29$

Then we choose integers a, b to define the encryption function. Suppose $a = 10, b = 13$, then

$$e_k(x) = (10x + 13) \bmod 29$$

So to encrypt "HELLO!" we would encode it as $8, 5, 12, 12, 15, 27$, then encrypt them,

$$e_k(8) = (10 \cdot 8 + 13) \bmod 29 = 6$$

$$e_k(5) = (10 \cdot 5 + 13) \bmod 29 = 5$$
$$e_k(12) = (10 \cdot 12 + 13) \bmod 29 = 17$$
$$e_k(12) = (10 \cdot 12 + 13) \bmod 29 = 17$$
$$e_k(15) = (10 \cdot 15 + 13) \bmod 29 = 18$$
$$e_k(28) = (10 \cdot 28 + 13) \bmod 29 = 3$$

Which, when mapped back to characters using our encoding is "FEQQRC."

To decrypt a message we need to invert the decryption function, that is,

$$d_k(y) = \left(a^{-1} \cdot (y - b)\right) \bmod n$$

where a^{-1} is the inverse of a modulo n. The inverse of an integer a is the value such that

$$(a \cdot a^{-1}) \bmod n = 1$$

so for $a = 10, n = 29$, the inverse, $10^{-1} \bmod 29 = 3$ since $3 \cdot 10 \bmod 29 = 1$. Given a and n, how can we find an inverse, a^{-1}? Obviously it cannot be zero, nor can it be 1 (1 is its own inverse). There is a simple algorithm (the Extended Euclidean Algorithm) that can solve this problem, but $n = 29$ is small enough that a brute-force strategy of testing all possibilities will suffice.

Write a program that takes a, b and an encrypted message as command line arguments and decrypts the message. Your program should print the decrypted message and other cipher information to the standard output. For example:

```
a    = 10
b    = 13
a^-1 = 3
Encrypted Message: FEQQRC
Decrypted Message: HELLO!
```

5. Functions

In mathematics, a function is a mapping from a set of *inputs* to a set of *outputs* such that each input is mapped to exactly one output. For example, the function

$$f(x) = x^2$$

maps numeric values to their squares. The input is a variable x. When we assign an actual value to x and evaluate the function, then the function has a value, its output. For example, setting $x = 2$ as input, the output would be $2^2 = 4$. Mathematical functions can have multiple inputs,

$$f(x, y) = x^2 + y^2 \quad f(\beta, y, z) = 2x + 3y - 4z$$

But will still only ever have *one* output value.

In programming languages, a function (sometimes called subroutine or procedure) can take multiple inputs and produce one output value. We've already seen some examples of these functions. For example, most languages provide a math library that you can use to evaluate the square root or sine of a value x. We've also seen some functions with multiple input values such as the "power" function that allows you to compute $f(x, y) = x^y$. The main entry point to many programs is defined by a main function.

More formally, a function is a sequence of instructions (code) that is packaged into a *unit* that can be reused. A function performs a specific task: given a number of inputs, it performs some sequence of operations (executes some code) and "returns" (outputs) a result. The output can be captured into a variable or other expression by whatever code *invoked* or "called" the function.

Defining and using functions in programming has numerous advantages. The most obvious advantage is that it allows you a way to *organize* code. By separating a program it into distinct units code it is more organized and it is clearer what each piece or segment of code does. This also facilitates top-down design: one way to approach a problem is to split it up into a series of subproblems until each subproblem is either trivial to deal with, or an existing, "off-the-shelf" solution already exists for it. Functions may be the logical unit for each subproblem.

Another advantage is that by organizing code into functions, those functions can be *reused* in multiple places either in your program/project or even in other programs/projects. A prime example of this are the standard libraries available in most programming languages that provide functions to perform standard input/output or mathematical

functions. These standard libraries provide functions that are used by thousands of different programs across multiple different platforms.

Functions also form an *isolated* unit of code. This allows for better and easier *testing*. By isolating pieces of code, we can rigorously test those pieces of code by themselves without worrying about the larger program or contexts.

Finally, functions facilitates procedural abstraction. Placing code into functions allows you to abstract the details of how the function computes its answer. As an example: consider a standard math library's square root function: it may use some interpolation method, a Taylor series, or some other method entirely to compute the square root of a given number. However, by putting this functionality into a function, we, as programmers, do not need to concern ourselves about these details. Instead, we simply *use* the function, allowing us to focus on the larger issues at hand in our program.

5.1. Defining & Using Functions

Like variables, many programming languages may require that you, in some way, declare the function before you can use it. A function declaration may simply include a description of the function's input/output and name. A function declaration may require require you to define the function's body at the same time or separately. Functions can also have scope: some areas of the code may be able to "see" the function or know about it and be able to invoke the function while other areas of the code may *not* be able to see the function and therefore may not be able to invoke it.

Some interpreted programming languages use function hoisting which allows you to use/invoke functions *before* you declare them. This works because the interpreter does an initial scan of the code and identifies all function declarations. Only after it has "hoisted" all functions into scope does it start to execute the program. Thus, a function declaration can appear *after* it has been used and it will still work.

5.1.1. Function Signatures

A function is usually defined by its scope: every function can be identified by its name (also called an *identifier*), its list of *input parameters*, and its *output*. A function signature allows the programming language to uniquely identify each function so that when you

invoke a function there is no ambiguity in which function should be called.

```
1 FUNCTION sum(a, b)
2 |  x ← a + b
3 |  return x
4 END
```

Algorithm 5.1: A function in pseudocode. In this case, the name (identifier) of the function is *sum* and it has two parameters, a and b. Its body is contained in lines 2–3. Its return value is indicated by the return statement on line 3.

A function declaration in pseudocode is presented in Algorithm 5.1. In the pseudocode, explicit variable types are omitted, and thus the return type is inferred from the return statement. In Figure 5.1 we have provided an example of a function declaration in the C programming language with each element labeled.

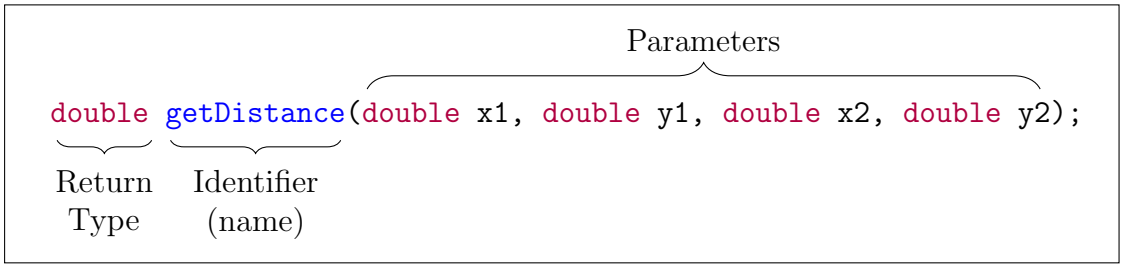


Figure 5.1.: A function declaration (prototype) in the C programming language with the return type, identifier, and parameter list labeled.

Some languages only allow you to use one identifier for one function (like variables) while other languages allow you to define multiple functions with the same identifier as long as the parameter list is different (see Section 5.3.2 below). In general, like variables, function names are case sensitive. Also similar to variables, modern lower camel casing is used with function names.

When defining the parameters to a function (its input), you usually provide a comma delimited list of variable names. In the case of statically typed languages, the types of the variable parameters are also specified. This order is important as when you invoke the function, the number of inputs must match the number of parameters in the function declaration. The variable types may also need to match. In some dynamically typed languages, you may be able to call functions with different types or you may be able to omit some of the parameters (see Section 5.3.4 below).

Similarly, the return type of the function may need to be specified in statically typed languages while with dynamic languages, functions may conditionally return different types. We generally refer to the "return value" or "return type" because when a function

is done executing, it "returns" the control flow back to the line of code that invoked it, returning its computed value.

You can also define functions that may not have any inputs or may not have any output. Some languages use the keyword `void` to indicate no return value and such functions are known as "void functions." When a function doesn't have any input values, its parameter list is usually empty.

The function signature may then be accompanied by the function *body* which contains the actual code that specifies what the function does. Typically the function body is demarcated with opening and closing curly brackets, `{ ... }`. Within the function you can generally write any valid code including declaring variables. When you declare a variable inside a function, however, it is *local* to that function. That is, the variable's scope is only defined within the function. A local variable cannot be accessed outside the function, indeed the local variable does not usually survive when the function ends its execution and returns control back to line of code that called it. Function parameters are essentially locally scoped variables as well and can usually be treated as such.

5.1.2. Calling Functions

When a function has been defined and is in scope, you can *invoke* or "call" the function by coding the function name and providing the input parameters which can typically be either variables or literals. When provided as inputs, parameters are referred to as *arguments* to the function. The arguments are typically provided as a comma delimited list and placed inside parentheses.

Invoking a function changes the usual flow of control. When invoked, control is handed over to the function. When the function finishes executing the code in its body, control flow returns to the point in the code that invoked it. It is common for a program to be written so that a function calls another function and that function calls another. This can form a deep chain of function calls in which the flow of control is transferred multiple times. Upon the completion of each function, control is returned back to the function that called it, referred to as the *calling function*.

If a function returns a value it can either be captured in a variable using an assignment operator or by using it in an expression.

```
1 a ← 10
2 b ← 20
3 c ← sum(a, b)
```

Algorithm 5.2: Using a function. We invoke a function by indicating its name (identifier) and passing it arguments.

5.1.3. Organizing

Functions provide code organization, but functions themselves should also be organized. We've seen this with standard libraries. Functions that provide basic input/output are all grouped together into one library. Functions that involve math functions are grouped together into a math library.

Some languages allow you to define and "import" individual libraries which organize similar functions together. Some languages do this by collecting functions into "utility" classes or *modules*. Only when you import these modules do the functions come into scope and can be used in your code. If you do not import these modules, then the functions are out of scope and cannot be used.

In some languages, functions, once imported, are part of the *global scope* and can be "seen" by any part of the code. This can cause *conflicts*: if you import modules from two different libraries each with different functions that have the same name or signature, then the two function definitions may be in conflict or it may make your code ambiguous as to which function you intend to invoke. This is sometimes referred to as "polluting the namespace." There are several techniques that can avoid this situation. Some languages allow you to place functions into a *namespace* to keep functions with the same name in different "spaces." Other languages allow you to place functions into different classes and then invoke them by explicitly specifying *which* class's function you want to call. Yet other languages don't have great support for function organization and it is the library designer's responsibility to avoid naming conflicts, typically by adding a library-specific prefix to every function.

5.2. How Functions Work

To understand how functions work in practice, it is necessary to understand how a program operates at a lower level. In particular, each program has a program stack (also called a call stack). A stack is a data structure that holds elements in a Last-In First-Out (LIFO) manner. Elements are added to the "top" of the stack in an operation called *push* and elements can be removed from the top of the stack in an operation called *pop*. In general, elements cannot be inserted or removed from the middle or "bottom" of the stack.

In the context of a program, a call stack is used to keep track of the flow of control. Depending on the operating system, compiler and architecture, the details of how elements are stored in the program stack may vary. However, in general when a program begins, the operating system loads it into memory at the bottom of the call stack. Global variables (static data) are stored on top of the main program. Each time a function is called, a new *stack frame* is placed on top of the stack. This frame contains enough space to hold values for the arguments passed to the function, local variables declared and used by the function, as well as a space for a return value and a return *address*. The return

address is a memory location that the program should return to after the execution of the function. That way, when the function finishes its execution, the stack frame can be removed (popped) and the lower stack frame of the calling function is preserved. This is a very efficient way to keep track of the flow of control in a program. As function calls function calls another function, each stack frame is preserved by pushing a new one on top of the program stack.

Each time a function terminates execution and returns, the removal of the stack frame means that all local variables go *out of scope*. Thus, variables that are local to a function are not accessible outside the function.

To illustrate, consider the following snippet of C code. The `main()` function invokes the `average()` function which in turn invokes the `sum()` function. Each invocation creates a new stack frame on top of the last in the program stack which is depicted in Figure 5.2.

```
1  double sum(double a, double b) {
2    double x = a + b;
3    return x;
4  }
5
6  double average(double a, double b) {
7    double y = sum(a, b) / 2.0;
8    return y;
9  }
10
11 int main(int argc, char **argv) {
12   double n = 10.0;
13   double m = 16.0;
14   double ave = average(n, m);
15   printf("average = %f\n", ave);
16   return 0;
17 }
```

5.2.1. Call By Value

When a function is invoked, arguments are passed to it. When you invoke a function you can pass it variables as arguments. However, variables themselves are not passed to the function, but instead the values *stored* in the variables at the time that you call the function are passed to the function. This mechanism is known as call by value and the variable values are *passed by value* to the function.

Recall that the arguments passed to a function are placed in a new stack frame for that function. Thus, in reality *copies* of the values of the variables are passed to the function. Any changes to the parameters inside the function have *no effect* on the original variables that were "passed" to the function when it was invoked.

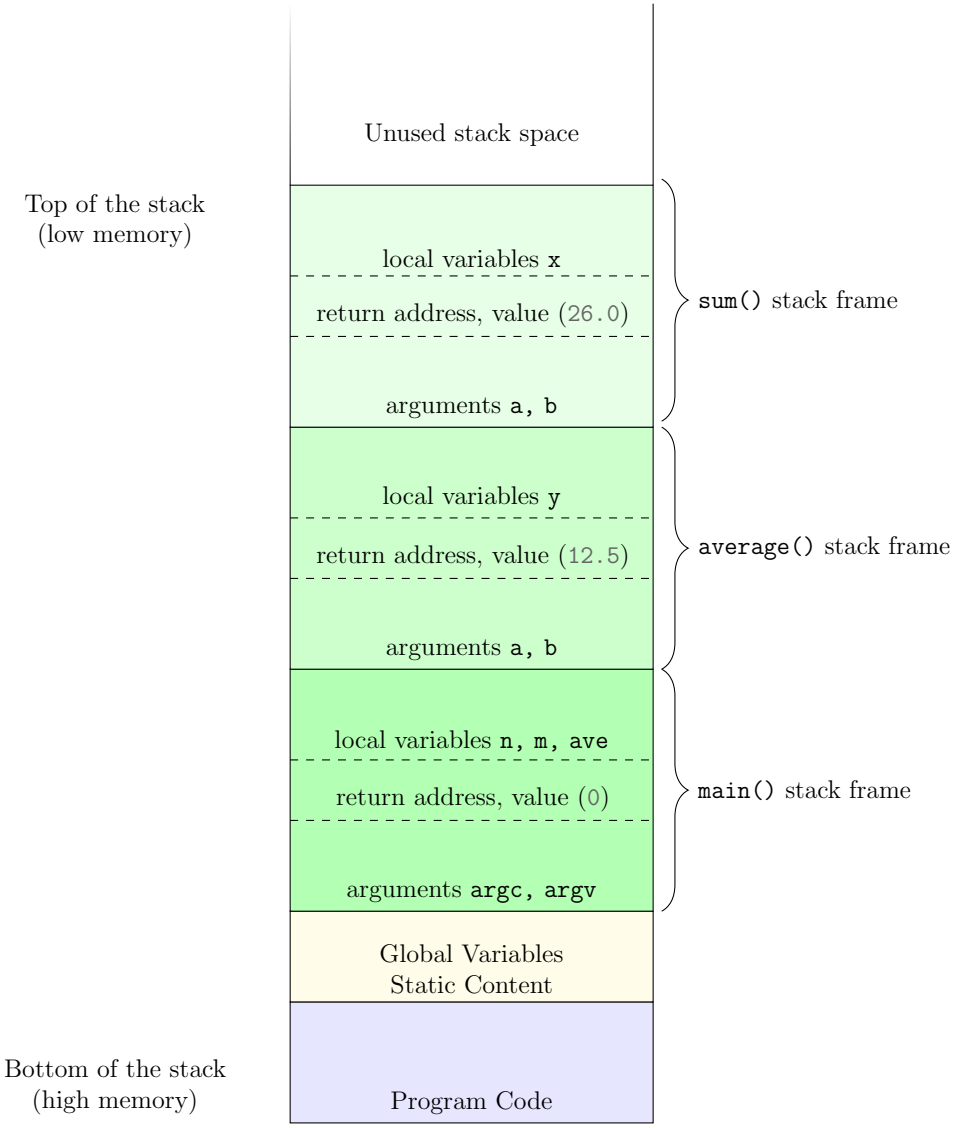


Figure 5.2.: Program Stack. At the bottom we have the program's code, followed by static content such as global variables. Each function call has its own stack frame along with its own arguments and local variables. In particular, the variable arguments `a` and `b` in two different stack frames are completely different variables. Upon returning from the `sum()` function call, the top-most stack frame would be popped and removed, returning to the code for the `average()` function via the return address. The stack is depicted bottom-up with high memory at the bottom and low memory at the top, but this may differ depending on the architecture.

To illustrate, consider the following C code. We have a function `sum` that takes two integer parameters `a` and `b` which are passed by value. Inside `sum`, we create another variable `x` which is the sum of the two passed variables. We then *change* the value of the

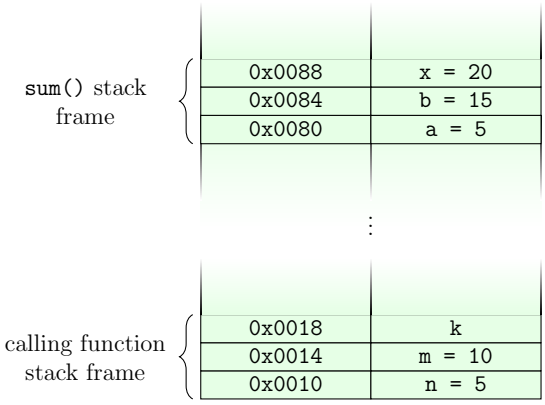
first variable, a to 10. Elsewhere in the code we call sum on two variables, n, m with values 5 and 15 respectively. The invocation of the function sum means that the two values, 5 and 15, stored in the variables are copied into a new stack frame. Thus, changing the value to the first parameter *changes the copy* and has no effect on the variable n. At the end of this code snippet n retains its original value of 5. The program stack frames are depicted in Figure 5.3.

```
1  int sum(int a, int b) {
2    int x = a + b;
3    a = 10;
4    return x;
5  }
6
7  ...
8
9  int n = 5;
10 int m = 15;
11 int k = sum(n, m);
```

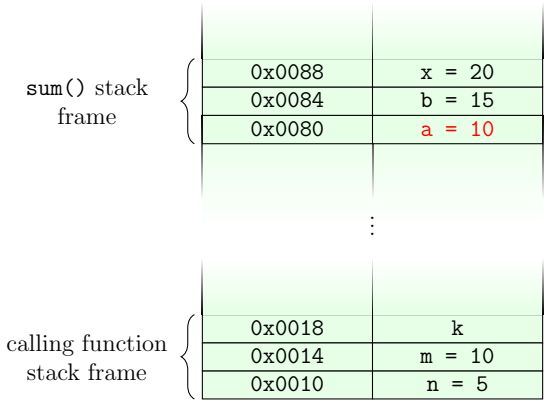
5.2.2. Call By Reference

Some languages allow you to pass a parameter to a function by providing its memory address. Since the memory address is being provided to the function, the function is able to access the original variable and manipulate the contents stored at that memory address. In particular, the function is now able to make changes to the original variable. This mechanism is known as call by reference and the variables are *passed by reference*.

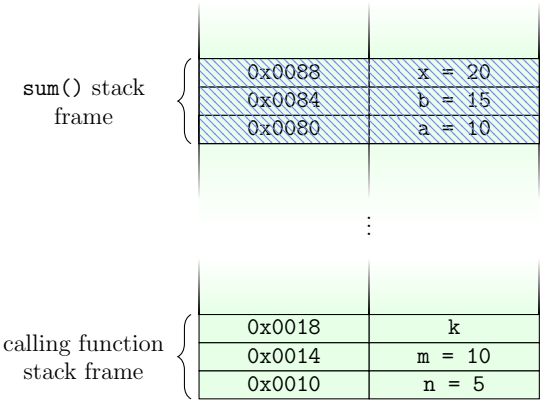
To illustrate consider the following C code. Here, the variable a is passed by reference (b is still passed by value, the *a and &n in the following code are dereferencing and referencing operators respectively. For details, see Section 18.2). Below when we invoke the sum function, we pass not the value stored in n, but the memory address of the variable n. Thus, when we change the value of the variable a in the function, we are actually changing the value of n (since we have access to its memory location. At the conclusion of this snippet of code, the value stored in n has been changed to 10. The program stack frames are depicted in Figure 5.4.



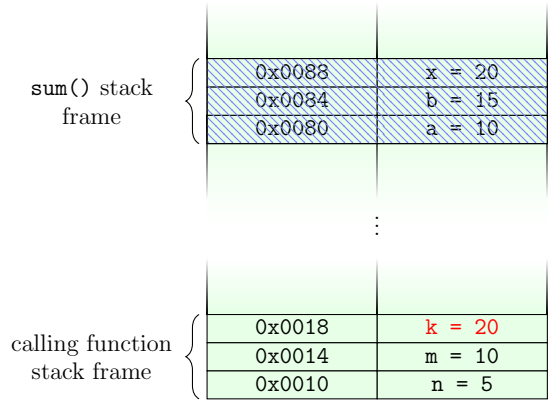
(a) Upon invocation of the `sum` function, a new stack frame is created which holds the parameters and local variable. The parameter variables `a` and `b` are *distinct* from the original argument variables `n` and `m`.



(b) The change to the variable `a` in the `sum()` function changes the parameter variable, but the original variable `n` is unaffected.



(c) When the `sum()` function finishes execution, its stack frame is removed and the variables `a`, `b`, and `x` are no longer valid. The return value 20 is stored in another return value location.



(d) The returned value is stored in the variable `k` and the variable `n` retains its original value.

Figure 5.3.: Demonstration of Pass By Value. Passing variables by value means that *copies* of the values stored in the variables are provided to the function. Changes to parameter variables do not affect the original variables.

```
int sym(int *a, int b) {
  int x = *a + b;
  *a = 10;
  return x;
}

...

int n = 5;
int m = 15;
int k = sum(&n, m);
```

Whether or not a variable is passed by value or by reference depends on the language, type of variable, and the syntax used.

5.3. Other Issues

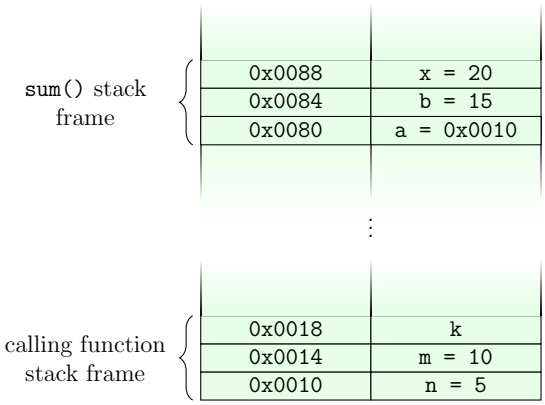
5.3.1. Functions as Entities

In programming languages, any entity that can be stored in variables or passed as an argument to a function or returned as a value from a function are referred to as "first-class citizens." Numerical values for example are usually first-class citizens as they can be stored in variables and passed around in functions.

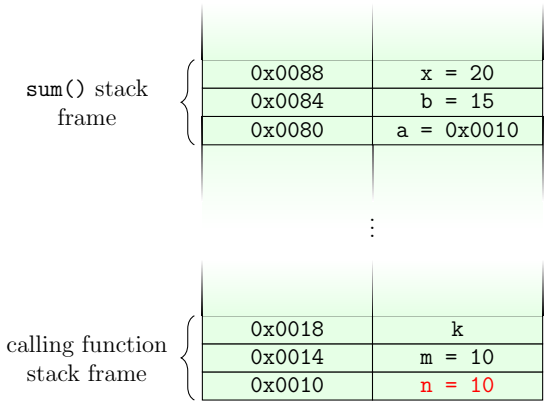
Functional Programming is a programming language paradigm in which functions themselves are first-class citizens. That is, functions can be assigned to variables, functions can be passed to other functions as arguments, and functions can even return functions as a result. This is done as a matter of course in functional programming languages such as Haskell and Clojure, but many programming languages contain some functional aspects.

For example, some languages support the same concept by using function *pointers* which are essentially references to where the function is stored in memory. As a memory location is essentially a number, it can be passed around in functions and be stored in a variable. Purists would argue that this is not sufficient to call a function a "first-class citizen" in such a language. They may argue that a language must be able to create new functions at runtime for it to be considered a language in which functions are "true" first-class citizens.

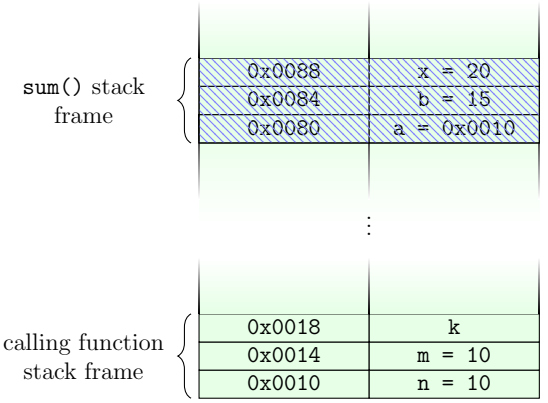
In any case, there are several advantages to being able to pass functions around as arguments or store them in variables. Passing a function to another function as an argument gives you the ability to provide a callback. A callback is simply a function that gets passed to another function as an argument. The idea is that the function that receives the callback will execute or "call back" the passed function at some point.



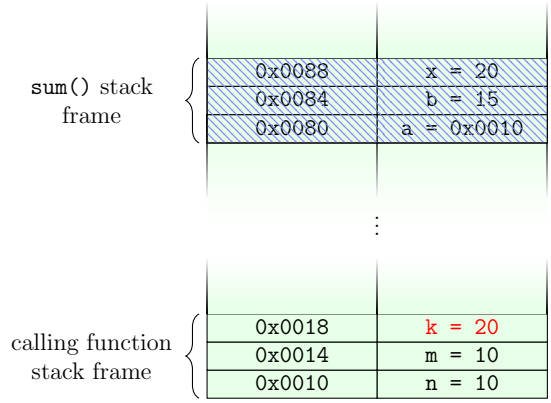
(a) Upon invocation of the `sum` function, a new stack frame is created which holds the parameters and local variable. The parameter variable `a` holds the memory location of the original variable `n`.



(b) The change to the variable `a` in the `sum()` function actually changes what the variable `a` *references*. That is, the original variable `n`.



(c) When the `sum()` function finishes execution, its stack frame is removed and the variables `a`, `b`, and `x` are no longer valid. The return value 20 is stored in another return value location.



(d) The returned value is stored in the variable `k` and the value in the variable `n` has now changed.

Figure 5.4.: Demonstration of Pass By Reference. Passing variables by reference means that the *memory address* of the variables are provided to the function. The function is able to make changes to the original variable because it knows where it is stored.

Using callbacks enables us to program a "generic" function that provides some generalized functionality. Then more specific behavior can be be implemented in the callback function. For example, we could create a generic *sort* function that sorts elements in a collection. We could make the sort function generic so that it could sort any type of data: numbers, strings, objects, etc. A callback would provide more specific behavior on how to *order* individual elements in the sorted array.

As another example, consider GUI Programming in which we want to design a user interface. In particular, we may be able to create a button element in our interface. We need to be able to specify what happens when the user clicks the button. This could be achieved by passing in a function as a callback to "register" it with the click "event."

A related issue is anonymous functions. Typically, we simply want to create a function so that we can pass it as a callback to another function. We may have no intention of actually calling this function directly as it may not be of much use other than passing it as a callback. Some languages allow you to define a function "inline" without an identifier so that it can be passed to another function. Since the function has no name and cannot be invoked by other sections of the code (other than the function we passed it to), it is known as an anonymous function.

5.3.2. Function Overloading

Some languages do not allow you to define more than one function with the same name in the same scope. This is to prevent ambiguity in the code. When you write code to invoke a function and there are several functions with that name, which one are you actually calling?

Some languages *do* allow you to define multiple functions with the same name as long as they differ in either the number of or type of parameters. For example, you could define two absolute value functions with the same name, but one of them takes a floating point number while the other takes an integer as its parameter. This is known as function overloading because you are "overloading" the code by defining multiple functions with the same name.

The ambiguity problem is solved by requiring that each function with the same name differs in their parameters. If you invoke the absolute value function and pass it a floating point number, clearly you meant to call the first version. If you passed it an integer, it is clear that you intended to invoke the second version. Depending on the type and number of arguments you pass to a function, the compiler or interpreter is able to determine *which* version you intend to call and is able to make the right function call. This process is known as static dispatch.

In a language without function overloading, we would be forced to use different names for functions that perform the same operation but on different types.

5.3.3. Variable Argument Functions

Many languages allow you to define special functions that take a *variable number* of parameters. Often they are referred to as "vararg" (short for variable argument) functions. The syntax for doing so varies as does how you can write a function to operate on a variable number of arguments (usually through some array or collection data structure).

The standard `printf` (print formatted) function found in many languages is a good example of a vararg function. The `printf` function allows you to use one function to print any number of variables or values. Without a vararg function, you would have to implement a `printf` version for no arguments, 1 argument, 2 arguments, etc. Even then, you would only be able to support up to n arguments for as many functions as you defined. By using a vararg function, we can write a single function that operates on all of these possibilities.

5.3.4. Optional Parameters & Default Values

Suppose that you define a function which has, say, three parameters. Now suppose you invoke the function but only provide it 2 of the 3 arguments that it expects. Some languages would not allow this and it would be considered a syntax or runtime error. Yet other languages may have very complex rules about what happens when an argument is omitted. Some languages allow you to omit some arguments when calling functions as a feature of the language. That is, the parameters to a function are *optional*.

When a language allows parameters to be optional, it usually also allows you to define *default values* to the parameters if the calling function does not provide them. If a user calls the function without specifying a parameter, it takes on the default value. Alternatively, the default could be a non-value like "null" or "undefined." Inside the function you could implement logic that determined whether or not a parameter was passed to the function and alter the behavior of the function accordingly.

5.4. Exercises

Exercise 5.1. Recall that the *greatest common divisor* (gcd) of two positive integers, a and b is the largest positive integer that divides both a and b. Adapt the solution from Exercise 4.7 into a function. If the language you use supports it, return the gcd via a pass by reference variable.

Exercise 5.2. Write a function that *scales* an input x to to its scientific notation scale so that $1 \leq x < 10$. If you language supports pass by reference, the amount that x is shifted should be stored in a pass-by-reference parameter. For example, a call to this function with $x = 314.15$ should return 3.1415 and the amount it is scaled by is $n = -2$.

Exercise 5.3. Write a function that returns the most significant digit of a floating point number. The function should only return an integer in the range $1 - -9$ (it should return zero only if $x = 0$).

Exercise 5.4. Write a function that, given an integer x, sums the values of its digits. That is, for $x = 29423$ the sum $2 + 9 + 4 + 2 + 3 = 20$.

Exercise 5.5. Write a function to convert radians to degrees using the formula,

$$deg = \frac{180 \cdot rad}{\pi}$$

Write another function to covert degrees to radians.

Exercise 5.6. Write functions to compute the diameter, circumference and area of a circle given its radius. If your language supports pass by reference, compute all three of these with one function.

Exercise 5.7. The *arithmetic-geometric* mean of two numbers x, y, denoted $M(x, y)$ (or $\text{agm}(x, y)$) can be computed iteratively as follows. Initially, $a_1 = \frac{1}{2}(x + y)$ and $g_1 = \sqrt{xy}$ (i.e. the normal arithmetic and geometric means). Then, compute

$$\begin{aligned} a_{n+1} &= \tfrac{1}{2}(a_n + g_n) \\ g_{n+1} &= \sqrt{a_n g_n} \end{aligned}$$

The two sequences will converge to the same number which is the arithmetic-geometric mean of x, y. Obviously we cannot compute an infinite sequence, so we compute until $|a_n - g_n| < \epsilon$ for some small number ϵ.

Exercise 5.8. Write a function to compute the annual percentage yield (APY) given an annual percentage rate (APR) using the formula

$$APY = e^{APR} - 1$$

Exercise 5.9. Write a function that will compute the air distance between two locations given their latitudes and longitudes. Use the formula as in Exercise 2.14.

Exercise 5.10. Write a function to convert a color represented in the RGB (red-green-blue) color model (used in digital monitors) to a CMYK (cyan-magenta-yellow-key) used in printing. RGB values are integers in the range $[0, 255]$ while CMYK are fractional numbers in the range $[0, 1]$. To convert to CMYK, you first need to scale each integer value to the range $[0, 1]$ by simply computing

$$r' = \frac{r}{255}, \quad g' = \frac{g}{255}, \quad b' = \frac{b}{255}$$

and then using the following formulas:

$$\begin{aligned} K &= 1 - \max\{r', g', b'\} \\ C &= (1 - r' - k)/(1 - k) \\ M &= (1 - g' - k)/(1 - k) \\ Y &= (1 - b' - k)/(1 - k) \end{aligned}$$

Exercise 5.11. Write a function to convert from CMYK to RGB using the following formulas.

$$r = 255 \cdot (1 - C) \cdot (1 - K)$$
$$g = 255 \cdot (1 - M) \cdot (1 - K)$$
$$b = 255 \cdot (1 - Y) \cdot (1 - K)$$

Exercise 5.12. Write some functions to convert an RGB color to a gray scale, "removing" the color values. An RGB color value is grayscale if all three components have the same value. To transform a color value to grayscale, there there are several possible techniques. The average method simply sets all three values to the average:

$$\frac{r + g + b}{3}$$

The lightness method averages the most prominent and least prominent colors:

$$\frac{\max\{r, g, b\} + \min\{r, g, b\}}{2}$$

The luminosity technique uses a weighted average to account for a human perceptual preference toward green:

$$0.21r + 0.72g + 0.07b$$

Exercise 5.13. Adapt the methods to compute a square root in Exercise 4.22 into functions.

Exercise 5.14. Adapt the methods to compute the natural logarithm in Exercise 4.23 into functions.

Exercise 5.15. Weight (mass in the presence of gravity) can be measured in several scales: kilogram force (kgf), pounds (lbs), ounces (oz), or Newtons (N). To convert between these scales, you can use the following facts:

- 1 kgf is equal to 2.20462 pounds
- There are 16 ounces in a pound
- 1 kgf is equal to 9.80665 Newtons

Write a collection of functions to convert between these scales.

Exercise 5.16. Length can be measured by several different units. We will concern ourselves with the following scales: kilometer, mile, nautical mile, and furlong. A measure in each one of these scales can be converted to another using the following facts.

- One mile is equivalent to 1.609347219 kilometers
- One nautical mile is equivalent to 1.15078 miles

- A furlong is $\frac{1}{8}$-th of a mile

Write a collection of functions to convert between these scales.

Exercise 5.17. Temperature can be measured in several scales: Celsius, Kelvin, Fahrenheit, and Newton. To convert between these scales, you can use the following conversion table.

From/To	Celsius	Kelvin	Fahrenheit	Newton
Celsius	–	$c + 273.15$	$c\frac{9}{5} + 32$	$c\frac{33}{100}$
Kelvin	$k - 273.15$	–	$\frac{9}{5}k - 459.67$	$.33k - 90.1395$
Fahrenheit	$(f - 32)\frac{5}{9}$	$\frac{5}{9}f + 255.372$	–	$\frac{11}{60}f - \frac{88}{15}$
Newton	$n\frac{100}{33}$	$\frac{100}{33}n + 273.15$	$\frac{60}{11}n + 32$	–

Table 5.1.: Conversion Chart

Write a collection of functions to convert between these scales.

Exercise 5.18. Energy can be measured in several different scales: calories (c), joules (J), ergs (erg) and foot-pound force (ft-lbf) among others. To convert between these scales, you can use the following facts:

- 1 erg equals $1.0 \times 10^{-7} J$
- 1 ft-lbs equals 1.3558 joules
- 1 calorie is equal to 4.184 joules

Write a collection of functions to convert between these scales.

Exercise 5.19. Pressure is a measure of force applied to the surface of an object per unit area. There are several units that can be used to measure pressure:

- Pascal (Pa) which is one Newton per square meter
- Pound-force Per Square Inch (psi)
- Atmosphere (atm) or standard atmospheric pressure
- The torr, an absolute scale for pressure

To convert between these units, you can use the following formulas.

- 1 psi is equal to 6,894.75729 Pascals, 1 psi is equal to 0.06804596 atmospheres
- 1 atmosphere is equal to 101,325 Pascals
- 1 torr is equal to $\frac{1}{760}$ atmosphere and $\frac{101{,}325}{760}$ Pascals

Write a collection of functions to convert between these scales.

6. Error Handling

Writing perfect code is difficult. The more complex a system or code base, the more likely it is to have bugs. That is, flaws or mistakes in a program that result in incorrect behavior or unintended consequences. The term "bug" has been used in engineering for quite a while. The term was popularized in the context of computer systems by Grace Hopper who, when working on the Naval Mark II computer in 1946, tracked a malfunction to a literal bug, a moth, trapped in a relay [2].

Some of the biggest modern engineering failures can be tracked to simple software bugs. For example, on September 26th, 1983 a newly installed Soviet early warning system gave indication that nuclear missiles had been launched on the Soviet Union by the United States. Stanislav Petrov, a lieutenant colonel in the Soviet Air Defense Forces and duty officer at the time, did not trust the new system and did not report the incident to superiors who may have ordered a counter strike. Petrov was correct as the false alarm was caused by sunlight reflections off of high altitude clouds as well as other bugs in the newly deployed system [17].

In September 1999 the Mars Climate Orbiter, a project intended to study the Martian climate and atmosphere was lost after it entered into the upper atmosphere of Mars and disintegrated. The error was due to a subsystem that measured the craft's momentum in non-standard pound force per second when all other systems expected the standard newton second unit [1]. The loss of the project was calculated at over $125 million.

There are numerous other examples, some that have caused inconvenience to users (such as the Zune bug mentioned in Section 4.5.2) to bugs in medical devices that have cost dozens of lives to those resulting in the loss of millions of dollars [6].

In some sense, Software Engineering and programming is unique. If you build a bridge and forget one bolt its likely not going to cause the bridge to collapse. If you draw up plans for a development and the land survey is a few inches off overall, its not a catastrophic problem. However, if you forget one character or are off by one number in a program, it can cause a complete system failure.

There are a variety of reasons for why bugs make it into systems. Bugs could be the result of a fundamental misunderstanding of the problem or requirements. Poorly managed projects and the pressure of time constraints to deliver a project may make developers more careless. A lack of proper testing may mean many more bugs survive the development process than otherwise should have. Even expert programmers can overlook a simple mistake when writing thousands of lines of code.

Given the potential for error, it is important to have good software development methodologies that emphasize testing a system at all levels. Working in teams where regular code reviews are held so that colleagues can examine, critique, and catch potential bugs are essential for writing robust code.

WHERE TO PLACE THIS?

Much of what we now consider Software Engineering was pioneered by people like Margaret Hamilton who was the lead Apollo flight software designer at NASA. During the Apollo 11 Moon landing (1969), an error in one system caused the lander's computer to become overworked with data. However, because the system was designed with a robust architecture, it could detect and handle such situations by prioritizing more important tasks (those related to landing) over lower priority tasks. The resilience that was built into the system is credited with its success [9].

END

Modern coding tools and techniques can also help to improve the robustness of code. For example, debuggers are tools that help a developer debug (that is, find and fix the cause of an error) a program. Debuggers generally allow you to simulate the execution of a program statement by statement and view the current state of the program such as variable values. You can "step through" the execution line by line to find where an error occurs in order to localize an identify a bug.

Other tools allow you to perform static analysis on source code to search for *potential* problems. That is, problems that are not syntax errors and are not necessarily bugs that are causing problems, but instead are anti-patterns or code smells. Anti-patterns are essentially common bad-habits that can be found in code. They are an *attempted* solution to a commonly encountered problem but which don't actually solve the problem or introduces new problems. Code smells are "symptoms" in a source code that indicate a possible deeper design or implementation flaw. Failure to adhere to good programming principles such as properly initializing variables or failure to check for null values are examples of smells. Static analysis tools automatically examine the code base for potential issues like these. For example, a lint (or linter) is a tool that can examine source code suspicious or non-portable code or code that does not comply with generally accepted standards or ways of doing things.

Even if code contains no bugs, it is still susceptible to errors. For example, a program could connect to a remote database to pull and process data. However, if the network connection is temporarily unavailable, the program will not be able to execute properly. Because of the potential of such errors, it is important to write robust and resilient code. We must anticipate possible error conditions and write code to detect, prevent, or recover from such errors. Generally, this is referred to as *error handling*.

6.1. Error Handling

In general, errors are potential conditions or situations that can be readily be anticipated by a developer. For example, if we write code to open and process a file, there are several things that could go wrong. The file may not exist, or we may not have permissions to read it, or the formatting in the file may be corrupted or not as expected. Still yet, everything could be fine with the file, but it may contain erroneous or invalid values.

If an error can be anticipated, we may be able to write code that detects the particular error condition and *handles* it by executing code that may be able to recover from the error condition. In the case of a missing file for example, we may be able to prompt the user for an alternate file.

We may be able to detect but not necessarily recover from certain errors. For example, if the file has been corrupted in example above, there may not be a way to properly "fix" it. If it contains invalid data, we may not even want the program to fix it as it may indicate a bug or other issue that needs to be addressed. Still yet, there may be some error conditions that we cannot recover from at all as they are completely unexpected. In such instances, we may want the error to result in the termination of the program in which case the error is considered *fatal*.

6.2. Error Handling Strategies

There are several general strategies for performing error handling. We'll look at two general methods here: defensive programming and exceptions.

6.2.1. Defensive Programming

Defensive programming is a "look before you leap" strategy. Suppose we have a potentially "dangerous" section of code; that is, a line or block of code whose execution could encounter an error condition. Before we execute the code, we perform a check to see if the error condition is present (usually using a conditional statement). If the error condition does not hold, then we proceed with the code as normal. However, if the error condition does hold, instead of executing the code, we execute alternative code that *handles* the error.

For example, suppose we are about to divide by a number. To prevent a division by zero error, we can check if our denominator is zero or not. If it is, then we raise or handle the error instead of performing the division.

What should be done in such a case? We could, as an alternative, use a predefined value as a result instead. Or we could notify the user and ask for an alternative. Or we could log the error and proceed as normal. Or we could decide that the error is so egregious that it should be fatal and terminate the execution of the program.

Which is the right way to handle this error? It depends on your design requirements really. This raises the question, though: "who" is responsible for making these decisions? Suppose we're designing a function for a library that is for use not just by our project but others as well (as is the case with the standard library functions). Further, the function we're designing could have multiple different error conditions that it checks for. In this scenario there are two entities that could handle the errors: the function itself and the code that invokes the function.

Suppose that we decide to handle the errors inside the function. That is, as designers of the function, we've made the decision to handle the errors *for* the user (the code that invokes our function). Regardless of how we decide to handle the errors, this design decision has essentially taken any decision making ability away from users. This is not very flexible for someone using our code. If they have different design considerations or requirements, they may need or want to handle the errors in a different way than we did.

Now suppose that we decide *not* to handle the errors inside our function. Defensive programming may still be used to prevent the execution of code that results in an error. However, we now need a way to *communicate* the error condition to the calling function so that it can know what type of error happened and handle it appropriately.

Error Codes

One common pattern to communicate errors to a calling function is to use the return type as an *error code*. Usually this is an integer type. By convention 0 is used to indicate "no error" and various other non-zero values are used to indicate various types of errors. Depending on the system and standard used, error codes may have a predefined value or may be specific to an application or library.

One problem with using the return type to indicate errors is that functions are no longer able to use the return type to return an actual computed value. If a language supports pass by reference, then this is not generally a problem. However, even with such languages there are situations where the return type *must* be used to return a value. In such cases, the function can still communicate a general error message by returning some flag value such as null.

Alternatively, a language may support error codes by using a shared global variable that can be set by a function to indicate an error. The calling function can then examine the variable to see if an error occurred during the invocation of the function.

Limitations

Defensive programming has its limitations. Let's return to the example of processing a file. To check for all four of the error conditions we identified, we would need a series of

checks similar to the following.

```
1  IF file does not exists THEN
2  |  return an error code
3  END
4  IF we do not have permissions THEN
5  |  return an error code
6  END
7  IF the file is corrupted THEN
8  |  return an error code
9  END
10 IF the file contains invalid values THEN
11 |  return an error code
12 END
13 process file data
```

A problem arises when an error condition is checked and does not hold. Then, later in the execution, circumstances change and the error condition then holds. However, since it was already checked for, the program remains under the assumption that the error condition does not hold. For example, suppose that another process or program deletes the file that we wish to process after its existence has been checked but before we start processing it.

Because of the sequential nature of our program, this type of error checking is susceptible to these issues.

6.2.2. Exceptions

An exception is an event or occurrence of an anomalous, erroneous or "exceptional" condition that requires special handling. Exceptions interrupt the normal flow of control in a program by handing the flow of control over to *exception handlers*.

Languages usually support exception handling using a try-catch control structure such as the following.

```
try {
  //potentially dangerous code here
} catch(Exception e) {
  //exception handling code here
}
```

The `try` is used to encapsulate potentially dangerous code, or simply code that would fail if an error condition occurs. If an error occurs at some point within the `try` block,

control flow is immediately transferred to the `catch` block. The `catch` block is where you specify *how* to handle the exception. If the code in the `try` block does not result in an exception, then control flow will skip over the `catch` statement and resume normally after.

It is important to understand how exceptions interrupt the normal control flow. For example, consider the following pseudocode

```
try {
  statement1;
  statement2;
  statement3;
} catch(Exception e) {
  //exception handling code here
}
```

Suppose `statement1` executes with no error but that when `statement2` executes, it results an exception. Control flow is then transferred to the `catch` block, skipping `statement3` entirely. In general, there may not be a mechanism for your `catch` block to recover and execute `statement3`. Therefore, maybe necessary to make your `try-catch` blocks fine-grained, perhaps having only a single statement within the `try` statement.

Some languages only support a generic `Exception` and the type of error may need to be communicated through other means such as a string error message. Still other languages may support many different types of exceptions and you may be able to provide *multiple* `catch` statements to handle each one differently. In such languages, the order in which you place your `catch` statements may be important as similar to an `if-else-if` statement, the first one that matches will be the one that executes. Thus, it is best practice to order your `catch` blocks from the most specific to the most general.

Some languages also support a third `finally` control statement as in the following example.

```
try {
  //potentially dangerous code here
} catch(Exception e) {
  //exception handling code here
} finally {
  //unconditionally executed code here
}
```

The `try-catch` block operates as previously described. However, the `finally` block will execute *regardless* of whether or not an exception was raised. If no exception was raised, then the `try` block will fully execute and the `finally` block will execute immediately after. If an exception was raised, control flow will be transferred to the `catch` block. After the `catch` block has executed, the `finally` block will execute.

`finally` blocks are generally used to handle resources that need to be "cleaned up"

whether or not an exception occurs. For example, opening a connection to a database to retrieve and process data. Whether or not an exception occurs during this process the connection will need to be properly closed as it represents a substantial amount of resources (a network connection, memory and processing time on both the server and client machines, etc.). Failure to properly close the connection may result in wasted resources. By placing the clean up code inside a `finally` statement, we can be assured that it will execute regardless of an error or exception.

In addition to handling exceptions, a language may allow you to "throw" usually by using the keyword `throw`. In this way you can also practice defensive programming. You could write a conditional statement to check for an error condition and then `throw` and exception.

6.3. Exercises

Exercise 6.1. Rewrite the function to compute the GCD in Exercise 5.1 to handle invalid inputs.

Exercise 6.2. Rewrite the function to compute statistics of a circle in Exercise 5.6 to handle invalid input (negative radius).

Exercise 6.3. Rewrite the function to compute the annual percentage yield in Exercise 5.8 to handle invalid input.

Exercise 6.4. Rewrite the function to compute air distance in Exercise 5.9 to handle invalid input (latitude/longitude values outside the range $[-180, 180]$).

Exercise 6.5. Rewrite the function to convert from RGB to CMYK in Exercise 5.10 to handle invalid inputs (values outside the range $[0, 255]$).

Exercise 6.6. Rewrite the function to convert from CMYK to RGB in Exercise 5.11 to handle invalid inputs.

Exercise 6.7. Rewrite the square root functions from Exercise 5.13 to handle invalid inputs.

Exercise 6.8. Rewrite the natural logarithm functions from Exercise 5.14 to handle invalid inputs.

Exercise 6.9. Rewrite the weight conversion functions from Exercise 5.15 to handle invalid inputs.

Exercise 6.10. Rewrite the length conversion functions from Exercise 5.16 to handle invalid inputs.

Exercise 6.11. Rewrite the temperature conversion functions from Exercise 5.17 to handle invalid inputs.

Exercise 6.12. Rewrite the energy conversion functions from Exercise 5.18 to handle invalid inputs.

Exercise 6.13. Rewrite the pressure conversion functions from Exercise 5.19 to handle invalid inputs.

7. Arrays, Collections & Dynamic Memory

TODO

8. Strings

TODO

9. File Input/Output

TODO

10. Encapsulation: Objects & Structures

TODO

11. Recursion

TODO

12. Searching & Sorting

TODO

13. Graphical User Interfaces & Event Driven Programming

TODO

14. Introduction to Databases & Database Connectivity

TODO

Part I.

The C Programming Language

15. Basics

The C programming language is a relatively old language, but still widely used to this day. It is nearly universal in that nearly every system, platform, and operating system has a C compiler that produces machine code for that system. C is used extensively in systems programming for operating system kernels, embedded systems, microcontrollers, and supercomputers. It is generally an "imperative" language which is a paradigm that characterizes computation in terms of executable statements and functions that change a program's state (variable values). C has been highly influential in the design of other languages including C++, Objective-C, C#, Java, PHP, Python, among many others. Many languages have adopted the basic syntactic elements and structured programming approach of the C language.

C was originally developed by Dennis Ritchie while at AT&T Bell Labs 1969–1972. C was born out of the need for a new language for the PDP-11 minicomputer that used the Unix operating system (written by Ken Thompson). From its inception, C has had a close relation to Unix; in fact the operating system was subsequently rewritten in C, making it the first OS to be written in a language other than assembly. The language was dubbed "C" as its predecessor was named "B", a simplified version of BCPL (Basic Combined Programming Language). The first formal specification was published as *The C Programming Language* by Kernighan and Ritchie (1978) [16] often referred to as "The K&R Book" which would later become the American National Standards Institute (ANSI) C standard.

C gained in popularity and directly influenced object-oriented variations of it. Bjarne Stroustrup developed C++ while at Bell Labs during 1979–1983. Brad Cox and Tom Love developed Objective-C during 1981–1983 at their company Stepstone. Subsequent standards of the C language have added and extended features. In 1990, the International Organization for Standardization (ISO)/IEC 9899:1990 standard, referred to as C89 or C90, was adopted. About every 10 years since, a new standard has been adopted; ISO/IEC 9899:1999 (referred to as C99) in 1999 and ISO/IEC 9899:2011 (C11) in 2011.

15.1. Getting Started: Hello World

The hallmark of the introduction of programming languages is the *Hello World!* program. It consists of a simple program whose only purpose is to print out the message "Hello World!" to the user in some manner. The simplicity of the program allows the focus

to be on the basic syntax of the language. It is also typically used to ensure that your development environment, compiler, runtime environment, etc. are functioning properly with a minimal example. The Hello World! program is generally attributed to Brian Kernighan who used it as an example of programming in C in 1974 [15]. A basic Hello World! program in C can be found in Code Sample 15.1.

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

/**
 * Basic Hello World program in C
 * Prints "Hello World" to the standard output and exits
 */
int main(int argc, char **argv) {

  printf("Hello World\n");

  return 0;
}
```

Code Sample 15.1: Hello World Program in C

We will not focus on any particular development environment, code editor, or any particular operating system, compiler, or ancillary standards in our presentation. However, as a first step, you should be able to write, compile, and run the above program on the environment you intend to use for the rest of this book. This may require that you download and install a basic C compiler/development environment (such as GCC, the GNU Compiler Collection on OSX/Unix/Linux, cygwin or MinGW for Windows) or a full IDE (such as Xcode for OSX, or Code::Blocks, http://www.codeblocks.org/ for Windows).

15.2. Basic Elements

Using the Hello World! program as a starting point, we will now examine the basic elements of the C language.

15.2.1. Basic Syntax Rules

C is a highly influential programming language. Many modern programming languages have adopted syntactic elements that originated in C. Usually such languages are referred to as "C-style syntax" languages. These elements include the following.

- C is a statically typed language so variables must be declared along with their types before using them.
- Strings are delimited with double quotes. Single characters, including special escaped characters are delimited by single quotes; `"this is a string"`, and these are characters: `'A'`, `'4'`, `'$'` and `'\n'`
- Executable statements are terminated by a semicolon, `;`
- Code blocks are defined by using opening and closing curly brackets, `{ ... }`. Moreover, code blocks can be *nested*: code blocks can be defined within other code blocks.
- Variables are scoped to the code block in which they are declared and are only valid within that code block.
- In general, whitespace between coding elements is ignored.

Though not a syntactic requirement, the proper use of whitespace is important for good, readable code. Code inside code blocks is indented at the same indentation. Nested code blocks are indented further. Think of a typical table of contents or the outline of a formal paper or essay. Sections and subsections or points and points all follow proper indentation with elements at the same level at the same indentation. This convention is used to organize code and make it more readable.

15.2.2. Preprocessor Directives

The lines,

```
1 #include<stdlib.h>
2 #include<stdio.h>
```

are *preprocessor directives*. Preprocessor directives are instructions to the compiler to *modify* the source code before it starts to compile it. These particular lines are "including" standard libraries of functions so that the program can use the functionality that has already been implemented for us.

The first, `stdlib.h` represents the C standard (`std`) library (`lib`). This library is so essential that many compilers will automatically include it even if you do not explicitly do so in your program. Still, it is best practice to include it in your code.

The second, `stdio.h` is the standard (`std`) input/output (`io`) library which contains basic I/O functions that we can use. In particular, the standard output function `printf` is part of this library.

Failure to include a library means that you will not be able to use the functions it provides in your program. Using the functions without including the library may result in a compiler error. The `.h` in the library names stands for "header"; function declarations

Function	Description
`abs(x)`	Absolute value for `int` variables, $\lvert x \rvert$[a]
`fabs(x)`	Absolute value for `double` variables
`ceil(x)`	Ceiling function, $\lceil 46.3 \rceil = 47.0$
`floor(x)`	Floor function, $\lfloor 46.3 \rfloor = 46.0$
`cos(x)`	Cosine function[b]
`sin(x)`	Sine function[b]
`tan(x)`	Tangent function[b]
`exp(x)`	Exponential function, e^x, $e = 2.71828\ldots$
`log(x)`	Natural logarithm, $\ln(x)$[c]
`log10(x)`	Logarithm base 10, $\log_{10}(x)$[c]
`pow(x,y)`	The power function, computes x^y
`sqrt(x)`	Square root function[c]

Table 15.1.: Several functions defined in the C standard math library. [a]The absolute value function is actually in the standard library, `stdlib.h`. [b]all trigonometric functions assume input is in *radians*, **not** degrees. [c]Input is assumed to be positive, $x > 0$.

are typically contained in a header file while their definitions are placed in a source file of the same name. We'll explore this convention in detail when we look at functions in C (Chapter 18.

There are many other important standard libraries that we'll touch on as needed, but another one that may be of immediate interest is the standard mathematics library, `math.h`. It includes many useful functions to compute common mathematical functions such as the square root and natural logarithm. Table 15.1 highlights several of these functions. To use them you'd include the math library in your source file `#include<math.h>` and then "call" them by providing input and getting the output. For example:

```
1 double x = 1.5;
2 double y, z;
3 y = sqrt(x); //y now has the value √x = √1.5
4 z = sin(x); //z now has the value sin(x) = sin(1.5)
```

In both of the function calls above, the value of the variable `x` is "passed" to the math function which computes and "returns" the result which then gets assigned to another variable.

Macros

Another preprocessor directive establishes *macros* using the `#define` keyword. A macro is a single instruction that specifies a more complex set of instructions. The macro can be used to define constants to be used throughout your program. To illustrate, consider the following example.

```
#define MILES_PER_KM 1.609
```

The macro defines an "alias" for the `MILES_PER_KM` identifier as the value 1.60934. Essentially, the C preprocessor will go through the code and any instance of `MILES_PER_KM` will be replaced with `1.609`. The advantage of using a macro like this is that we can use the identifier `MILES_PER_KM` throughout our program instead of mysterious numbers whose meaning and intent may not be immediately clear. Moreover, if we want to change the definition (say make it more precise, 1.60934) then we only need to change the macro instead of making the same change throughout our program.

As a stylistic note: macro constants in C are usually associated with uppercase underscore casing as in our example. Also, the math standard library defines several macros for common mathematical constants such as π, e, and $\sqrt{2}$ (`M_PI`, `M_E`, and `M_SQRT2` respectively) among others.

15.2.3. Comments

Comments can be written in a C program either as a single line using two forward slashes, `//comment` or as a multiline comment using a combination of forward slash and asterisk: `/* comment */`. With a single line comment, everything on the line *after* the forward slashes is ignored. With a multiline comment, everything in between the forward slash/asterisk is ignored. Comments are ultimately ignored by the compiler so the amount of comments do not have an effect on the final executable code. Consider the following example.

```
//this is a single line comment
int x;  //this is also a single line comment, but after some code

/*
  This is a comment that can
  span multiple lines to format the comment
  message more clearly
*/
double y;
```

Most code editors and IDEs will present comments in a special color or font to distinguish them from the rest of the code (just as our example above does). Failure to close a multiline comment will likely result in a compiler error but with color-coded comments

its easy to see the mistake visually.

15.2.4. The `main` Function

Every executable program has to have a beginning: a point at which the program starts to execute. In C, the starting point is the `main` function. When a program is compiled to an executable and the program is invoked, the code in the `main` function starts executing. In our example, the code associated with the `main` function placed in between the two curly brackets is part of the `main` function. At the end of the function we have a `return` statement. We'll examine `return` statements in detail when we examine functions, for now though, the program is "returning" a zero value to the operating system. The convention in C is that zero indicates "no error occurred" while a non-zero value is used to indicate that "some" error occurred (the value used is determined by system standards such as the Portable Operating System Interface (POSIX) standard).

In addition, our `main` function has two *arguments*: `argc` and `argv` which serve to communicate any command line arguments provided to the program (review Section 2.4.4 for details). The first, `argc` is an integer that indicates the *number* of arguments provided *including* the executable file name itself. The second, `argv` actually stores the arguments as strings. We'll be able to understand the syntax later on, but for now we can at least understand how we might convert these arguments to different types such as integers and floating-point numbers.

First, recall that `argv` is the argument *vector*: it is an array (see Chapter 20) of the command line arguments. To access them, you can *index* them starting at zero, the first being `argv[0]`, the second `argv[1]`, etc. (the last one of course would be at `argv[argc-1]`). The first one is always the name of the executable file being run. The remaining are the command line arguments provided by the user.

To convert them you can use two different functions, `atoi` and `atof` which are short for **a**lphanumeric **to** **i**nteger and **f**loafing-point number respectively. An example:

```
1 //prints the first command line argument:
2 printf("%s\n", argv[0]);
3 //converts the "second" command line argument to an integer
4 int x = atoi(argv[1]);
5 //converts the "third" command line argument to a double:
6 double y = atof(argv[2]);
```

15.3. Variables

As previewed, the three main primitive types supported in C are `int`, `double`, and `char` which support integers, floating-point numbers, and single ASCII characters.

Integer (`int`) types are only guaranteed to "be at least" 16 bytes by the C standard but are usually 32-bit signed integers on most modern systems[1]. With a 32-bit signed `int` we can represent integers between $-2,147,483,648$ and 2,147,483,647.

Doubles (`double`) types are usually double-precision floating-point numbers as per the IEEE 754 standard and provide about 16 digits of precision.

Though C does provide a `float` (single precision floating-point number) type and there are various modifiers such as `short`, `long`, `unsigned` and `signed` that can be used, but these are either system-dependent or rely on later versions of the C standard (such as C99). We will restrict our focus to more portable, interoperable code and stick with the basic two types in most of our code.

Finally, the `char` type is typically a single byte that represents a single ASCII character. For all intents and purposes a `char` can be treated as an integer in the range 0 to 127 (or 255) as defined by the ASCII text table (see Table 2.4).

15.3.1. Declaration & Assignment

C is a statically typed language meaning that all variables must be declared before you can use them or refer to them. In addition, when declaring a variable, you must specify both its type and its identifier. For example:

```
1 int numUnits;
2 double costPerUnit;
3 char firstInitial;
```

Each declaration specifies the variables type followed by the identifier and ending with a semicolon. The identifier rules are fairly standard: a name can consist of lower and uppercase alphabetic characters, numbers, and underscores but may *not* begin with a numeric character. We adopt the modern camelCasing naming convention for variables in our code.

The assignment operator is a single equal sign, `=` and is a right-to-left assignment. That is, the variable that we wish to assign the value to appears on the left-hand-side while the value (literal, variable or expression) is on the right-hand-size. Using our variables from before, we can assign them values:

```
1 numUnits = 42;
2 costPerUnit = 32.79;
3 firstInitial = 'C';
```

An important thing to understand and to keep in mind is: if you declare a variable but do not assign it a value, its value is *undefined*. That is, if we code something like `int a;`, the value of the variable `a` is *not* necessarily zero; depending on the system, it could

[1] You may have to deal with 16-bit `int` types in legacy systems/compilers or in modern embedded systems.

contain a special value that indicates "uninitialized memory" or it could contain garbage, or it *could* have the value zero. The C standard does *not* specify default values for variables. The default value of variables is highly system dependent–on the compiler, the libraries, and even the operating system. Do not make any assumptions on the initial or default values of variables. If you need such assumptions, then values must be assigned.

For brevity, C allows you to declare a variable and immediately assign it a value on the same line. So these two code blocks could have been more compactly written as:

```
int numUnits = 42;
double costPerUnit = 32.79;
char firstInitial = 'C';
```

As another shorthand, we can declare multiple variables on the same line by delimiting them with a comma. However, they *must* be of the same type. We can also use an assignment with them.

```
int numOrders, numUnits = 42, numCustomers = 10, numItems;
double costPerUnit = 32.79, salesTaxRate;
```

Another convenient keyword is `const`, short for "constant". We can apply it to any variable to indicate that it is a read-only variable. Of course, we *must* assign it a value at declaration. For example:

```
const int secret = 42;
const double salesTaxRate = 0.075;
```

Any attempt to reassign the values of `const` variables will result in a compiler error.

15.4. Operators

C supports the standard arithmetic operators for addition, subtraction, multiplication, and division using `+`, `-`, `*`, and `/` respectively. Each of these operators is a binary operator that acts on two operands which can either be literals or other variables and follow the usually rules of arithmetic when it comes to order of precedence (multiplication and division before addition and subtraction).

```
int a = 10, b = 20, c = 30, d;
d = a + 5;
d = a + b;
d = a - b;
d = a + b * c;
d = a * b;
d = a / b; //integer division and truncation!  See below

double x = 1.5, y = 3.4, z = 10.5, w;
w = x + 5.0;
w = x + y;
w = x - y;
w = x + y * z;
w = x * y;
w = x / y;

//you can do arithmetic with both types:
w = a + x;
d = b + y; //truncation also occurs here!
```

Special care must be taken when dealing with `int` types. For all four operators, if both operands are integers, the result will be an integer. For addition, subtraction, and multiplication this isn't a big deal, but for division it means that when we divide, say `10 / 20`, the result is not 0.5 as expected. The number 0.5 is a floating-point number. As such, the fractional part gets truncationtruncated (cut off and thrown out) leaving only zero. In the code above, `d = a / b;` the variable `d` ends up getting the value zero because of this.

Similarly, attempting to assign a floating-point number to an integer also results in truncation because an `int` type cannot handle the fractional part. In the line `d = b + y` above, `b + y` is correctly $20 + 3.4 = 23.4$, but when assigned to the `int` variable `d` the .4 gets truncated and `d` has the value 23.

Assigning an `int` value to a `double` variable is not a problem as the integer 2 becomes the floating-point number 2.0.

A solution to this problem is to use explicit type casting to force at least one of the operands in an integer division to become a `double` type. For example:

```
int a = 10, b = 20;
double x;

x = (double) a / b;
```

results in `x` getting the "correct" value of 0.5. This works because the `(double)` code forces the `int` variable `a` to *temporarily* be treated as a double variable (in this case 10.0) for the purposes of division (so that truncation does *not* occur).

C also supports the integer remainder operator using the % symbol. This operator gives the remainder of the result of dividing two integers. Examples:

```
int x;

x = 10 % 5; //x is 0
x = 10 % 3; //x is 1
x = 29 % 5; //x is 4
```

15.5. Basic I/O

The C Standard I/O library (`stdio.h`) contains many functions that facilitate input and output including `printf` for standard output and `scanf` (**scan f**ormatted input) for standard input.

The `printf` function works exactly as discussed in Section 2.4.3. The `scanf` function works using similar placeholders as `printf`. To illustrate how it works, consider the following lines of code:

```
int a;
printf("Please enter a number: ");
scanf("%d", &a);
```

The `printf` statement prompts the user for an input. The `scanf` then executes and the program *waits* for the user to enter input. The user is free to start typing. When the user is done, they hit the enter key at which point the program resumes and reads the input from the standard input buffer, converts the value entered by the user into an integer and places the result in the variable `a` where it can now be used by the remainder of the program.

A few points of interest. First, the same placeholder as `printf` was used, `%d` for `int` values. However, when we "passed" the variable `a` to `scanf` we placed an ampersand, `&` in front of it. This is passing the variable *by reference* and we'll explore that concept further in Chapter 18, but for now just know that when using variables with `scanf`, an ampersand is required. Failure to place an ampersand in front of a variable with `scanf` will likely result in a *segmentation fault* (an illegal memory access).

You can use the same placeholder, `%c` with `scanf` to read in single characters as well. However, for floating-point numbers, in particular `double` types, the placeholder `%lf` must be used (which stands for long float, a double precision number). Failure to use the correct placeholder may result in garbage results as the input will be interpreted incorrectly. Another example:

```
double x;
printf("Please enter a fractional number: ");
scanf("%lf", &x);
```

Another potential problem is that `scanf` expects a certain *format* (thus its name). If we prompt the user for a number but they just start mashing the keyboard giving non-numerical input, we may get incorrect results. `scanf` will likely interpret the input as zero. It may be very difficult to distinguish between the case of a user actually entering in zero as a legitimate input versus bad input. In general, `scanf` is not a good mechanism for reading input (and in fact can be very dangerous), but it is a good starting point.

15.6. Examples

15.6.1. Converting Units

Let's start with a simple task: let's write a program that will prompt the user to enter a temperature in degrees Fahrenheit and convert it to degrees Celsius using the formula

$$C = (F - 32) \cdot \frac{5}{9}$$

We begin with the basic program outline which will include preprocessor directives to bring in the standard library and the standard input/output library (after all, we'll need to prompt for input and print the result as output to the user). Further, we want our program to be executable, so we need to put our code into the main method. Finally, we'll document our program to indicate its purpose.

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

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */
int main(int argc, char **argv) {

  //TODO: implement this

  return 0;
}
```

It is common for programmers to use a comment along with a `TODO` note to themselves as a reminder of things that they still need to do with the program.

Let's first outline the basic steps that our program will go through:

1. We'll first prompt the user for input, asking them for a temperature in Fahrenheit
2. Next we'll read the user's input, likely into a floating-point number as degrees can be fractional

3. Once we have the input, we can calculate the degrees Celsius by using the formula above
4. Lastly, we will want to print the result to the user to inform them of the value

Sometimes its helpful to write an outline of such a program directly in the code using comments to provide a step-by-step process. For example:

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

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */
int main(int argc, char **argv) {

  //TODO: implement this
  //1. Prompt the user for input in Fahrenheit
  //2. Read the Fahrenheit value from the standard input
  //3. Compute the degrees Celsius
  //4. Print the result to the user

  return 0;
}
```

As we read each step it becomes apparent that we'll need a couple of variables: one to hold the Fahrenheit (input) value and one for the Celsius (output) value. It also makes sense that each of these should be `double` variables as we want to support fractional values. So at the top of our `main` function, we'll add the variable declarations:

```c
double fahrenheit, celsius;
```

Each of the steps is now straightforward; we'll use a `printf` statement in the first step to prompt the user for input:

```c
printf("Please enter degrees in Fahrenheit: ");
```

In the second step, we'll use the standard input to read the `fahrenheit` variable value from the user. Recall that we use the placeholder `%lf` for reading `double` values and use an ampersand when using `scanf`:

```c
scanf("%lf", &fahrenheit);
```

We can now compute `celsius` using the formula provided:

```c
celsius = (fahrenheit - 32) * (5 / 9);
```

Finally, we use `printf` again to output the result to the user:

```
printf("%f Fahrenheit is %f Celsius\n", fahrenheit, celsius);
```

Try typing and running the program as defined above and you'll find that you don't get correct answers. In fact, you'll find that no matter what values you enter, you get zero. This is because of the calculation using `5 / 9`: recall what happens with integer division: truncation! This will *always* end up being zero.

One way we could fix it would be to pull out our calculators and find that $\frac{5}{9} = 0.55555\ldots$ and replace `5/9` with `0.555555`. But, how many fives? It may be difficult to tell how accurate we can make this floating-point number by hardcoding it ourselves. A much better approach would be to let the compiler take care of the optimal computation for us by making at least one of the numbers a `double` to prevent integer truncation. That is, we should instead use `5.0 / 9`.

The full program can be found in Code Sample 15.2.

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

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */
int main(int argc, char **argv) {

  double fahrenheit, celsius;

  //1. Prompt the user for input in Fahrenheit
  printf("Please enter degrees in Fahrenheit: ");

  //2. Read the Fahrenheit value from the standard input
  scanf("%lf", &fahrenheit);

  //3. Compute the degrees Celsius
  celsius = (fahrenheit - 32) * 5.0/9;

  //4. Print the result to the user
  printf("%f Fahrenheit is %f Celsius\n", fahrenheit, celsius);

  return 0;
}
```

Code Sample 15.2: Fahrenheit-to-Celsius Conversion Program in C

15.6.2. Computing Quadratic Roots

Some programs require the user to enter multiple inputs. The prompt-input process can be repeated. In this example, consider asking the user for the coefficients, a, b, c to a quadratic polynomial,

$$ax^2 + bx + c$$

and computing its roots using the quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

As before, we can create a basic program with a `main` function and start filling in the details. In particular, we'll need to prompt for the input a, then read it in; then prompt for b, read it in and repeat for c. We'll also need several variables: three for the coefficients a, b, c and *two* more; one for each root. Thus, we have

```
1 double a, b, c, root1, root2;
2
3 printf("Please enter a: ");
4 scanf("%lf", &a);
5 printf("Please enter b: ");
6 scanf("%lf", &b);
7 printf("Please enter c: ");
8 scanf("%lf", &c);
```

Now to compute the roots: we need to take care that we correctly adapt the formula so it accurately reflects the order of operations. We also need to use the standard math library's square root function (unless you want to write your own![2] Carefully adapting the formula leads to

```
1 root1 = (-b + sqrt(b*b - 4*a*c) ) / (2*a);
2 root1 = (-b - sqrt(b*b - 4*a*c) ) / (2*a);
```

Finally, we print the output using `printf`. The full program can be found in Code Sample 15.3.

This program was interactive. As an alternative, we could have read all three of the inputs as command line arguments, taking care that we need to convert them to floating-point numbers. Lines 12–17 in the program could have been changed to

```
1 a = atof(argv[1]);
2 b = atof(argv[2]);
3 c = atof(argv[3]);
```

Finally, think about the possible inputs a user could provide that may cause problems for this program. For example:

[2] We *will* write several square root methods as exercises later!

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

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */
int main(int argc, char **argv) {

  double a, b, c, root1, root2;

  printf("Please enter a: ");
  scanf("%lf", &a);
  printf("Please enter b: ");
  scanf("%lf", &b);
  printf("Please enter c: ");
  scanf("%lf", &c);

  root1 = (-b + sqrt(b*b - 4*a*c) ) / (2*a);
  root2 = (-b - sqrt(b*b - 4*a*c) ) / (2*a);

  printf("The roots of %fx^2 + %fx + %f are: \n", a, b, c);
  printf("  root1 = %f\n", root1);
  printf("  root2 = %f\n", root2);
  return 0;
}
```

Code Sample 15.3: Quadratic Roots Program in C

- What if the user entered zero for a?
- What if the user entered some combination such that $b^2 < 4ac$?
- What if the user entered non-numeric values?
- For the command line argument version, what if the user provided less than three argument? Or more?

How might we prevent the consequences of such bad inputs? That is, how might we handle the even that a users enters those bad inputs and how do we communicate these errors to the user? To do so we'll need conditionals.

16. Conditionals

C supports the basic if, if-else, and if-else-if conditional structures as well as switch statements. Logical statements are built using the standard logical operators for numeric comparisons as well as logical operators such as negations, AND, and OR. However, there are a few idiosyncrasies that need to be understood.

16.1. Logical Operators

C has no built-in Boolean type, nor does it have any keywords associated with *true* and *false*. Instead, C uses numeric types as implicit Boolean types with the convention that zero is associated with *false* and *any* non-zero value is associated with *true*. So the values 1, 2.5, and even negatives, -1, -123.2421, etc. are all interpreted as *true* when used in logical statements.

The standard numeric comparison operators are also supported. Consider the following code snippet:

```
1 int a = 10;
2 int b = 20;
3 int c = 10;
4 int d = 0;
```

The six standard comparison operators are presented in Table 16.1 using these variables as examples. The comparison operators are the same when used with `double` types as well and `int` types and `double` types can be compared with each other without type casting.

The three basic logical operators are also supported as described in Table 16.2 using the same code snippet variable values as examples.

16.1.1. Order of Precedence

At this point it is worth summarizing the order of precedence of all the operators that we've seen so far including assignment, arithmetic, comparison, and logical. Since all of these operators could be used in one statement, for example,

```
(b*b < 4*a*c || a == 0 || argc != 4)
```

Name	Operator Syntax	Examples	Value
Equals	`==`	`a == 10`	*true*
		`b == 10`	*false*
		`a == b`	*false*
		`a == c`	*true*
Not Equals	`!=`	`a != 10`	*false*
		`b != 10`	*true*
		`a != b`	*true*
		`a != c`	*false*
Strictly Less Than	`<`	`a < 15`	*true*
		`a < 5`	*false*
		`a < b`	*true*
		`a < c`	*false*
Less Than Or Equal To	`<=`	`a <= 15`	*true*
		`a <= 5`	*false*
		`a <= b`	*true*
		`a <= c`	*true*
Strictly Greater Than	`>`	`a > 15`	*false*
		`a > 5`	*true*
		`a > b`	*false*
		`a > c`	*false*
Greater Than Or Equal To	`>=`	`a >= 15`	*false*
		`a >= 5`	*true*
		`a >= b`	*false*
		`a >= c`	*true*

Table 16.1.: Comparison Operators in C

Operator	Operator Syntax	Examples	Values
Negation	`!`	`!a`	*false*
		`!d`	*true*
And	`&&`	`a && b`	*true*
		`a && d`	*false*
Or	`||`	`a || b`	*false*
		`a || d`	*true*

Table 16.2.: Logical Operators in C

	Operator(s)	Associativity	Notes
Highest	`++`, `--`	left-to-right	increment operators
	`-`, `!`	right-to-left	unary negation operator, logical not
	`*`, `/`, `%`	left-to-right	
	`+`, `-`	left-to-right	addition, subtraction
	`<`, `<=`, `>`, `>=`	left-to-right	comparison
	`==`, `!=`	left-to-right	equality, inequality
	`&&`	left-to-right	logical AND
	`||`	left-to-right	logical OR
Lowest	`=`, `+=`, `-=`, `*=`, `/=`	right-to-left	assignment and compound assignment operators

Table 16.3.: Operator Order of Precedence in C. Operators on the same level have equivalent order and are performed in the associative order specified.

it is important to understand the order in which each one gets evaluated. Table 16.3 summarizes the order of precedence for the operators seen so far. This is not an exhaustive list of C operators.

16.1.2. Comparing Strings and Characters

The comparison operators in Table 16.1 can also be used for *single characters* because of the nature of the ASCII text table (see Table 2.4). Each alphanumeric character, including the various symbols and whitespace characters, is associated with an integer 0–127. We can therefore write statements like `('A' < 'a')`, which is *true* since uppercase letters are ordered before lowercase letters in the ASCII table (`'A'` is 65 and `'a'` is 97 and so $65 < 97$ is *true*). Several more examples can be found in Table 16.4.

Comparison Example	Result
`('A' < 'a')`	*true*
`('A' == 'a')`	*false*
`('A' < 'Z')`	*true*
`('0' < '9')`	*true*
`('\n' < 'A')`	*true*
`(' ' < '\n')`	*false*

Table 16.4.: Character comparisons in C

Numeric comparison operators *cannot* be used to compare strings in C. For example, if we could write `("aardvark" < "zebra")` which would be valid C, and it would even have a result. However, that result wouldn't necessarily be *true* or *false*. The reason for this is that strings in C are actually represented as arrays, which in turn are represented as *memory locations*. We'll explore these issues in greater depth later on, but for now

understand that you can write this code, it will compile, and it will even run. However, the results will not be as expected.

16.2. If, If-Else, If-Else-If Statements

Conditional statements in C utilize the key words `if`, `else`, and `else if`. Conditions are placed inside parentheses immediately after the `if` and `else if` keywords. Examples of all three can be found in Code Sample 16.1.

```
1  //example of an if statement:
2  if(x < 10) {
3    printf("x is less than 10\n");
4  }
5
6  //example of an if-else statement:
7  if(x < 10) {
8    printf("x is less than 10\n");
9  } else {
10   printf("x is 10 or more \n");
11 }
12
13 //example of an if-else-if statement:
14 if(x < 10) {
15   printf("x is less than 10\n");
16 } else if(x == 10) {
17   printf("x is equal to ten\n");
18 } else {
19   printf("x is greater than 10\n");
20 }
```

Code Sample 16.1: Examples of Conditional Statements in C

Some observations about the syntax: the statement, `if(x < 10)` does not have a semicolon at the end. This is because it is a conditional statement that determines the flow of control and *not* an executable statement. Therefore, no semicolon is used. Suppose we made a mistake and *did* include a semicolon:

```
1  int x = 15;
2  if(x < 10); {
3    printf("x is less than 10\n");
4  }
```

Some compilers may give a warning, but this is valid C; it will compile and it will run. However, it will end up printing `x is less than 10`, even though $x = 15$! Recall that

a conditional statement *binds* to the executable statement or code block *immediately* following it. In this case, we've provided an *empty* executable statement ended by the semicolon. The code is essentially equivalent to

```
int x = 15;
if(x < 10) {
}
printf("x is less than 10\n");
```

Which is obviously not what we wanted. The semicolon ended up binding to the empty executable statement, and the code block containing the print statement immediately followed, but was *not* bound to the conditional statement which is why the print statement executed regardless of the value of x.

Another convention that we've used in our code is where we have placed the curly brackets. First, if a conditional statement is bound to only one statement, the curly brackets are not necessary. However, it is best practice to include them even if they are not necessary and we'll follow this convention. Second, the opening curly bracket is on the same line as the conditional statement while the closing curly bracket is indented to the same level as the start of the conditional statement. Moreover, the code inside the code block is indented. If there were more statements in the block, they would have all been at the same indentation level.

16.3. Examples

16.3.1. Computing a Logarithm

The logarithm of x is the exponent that some *base* must be raised to get x. The most common logarithm is the natural logarithm, $\ln(x)$ which is base $e = 2.71828\ldots$. But logarithms can be in any base $b > 1$[1] What if we wanted to compute $\log_2(x)$? Or $\log_\pi(x)$? Let's write a program that will prompt the user for a number x and a base b and computes $\log_b(x)$.

Arbitrary bases can be computed using the change of base formula:

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$$

If we can compute *some* base a, then we can compute any base b. Fortunately we have such a solution. Recall that the standard library provides a function to compute the natural logarithm, `log()`). This is one of the fundamentals of problems solving: if a solution already exists, use it. In this case, a solution exists for a different, but similar problem (computing the natural logarithm), but we can *adapt* the solution using the

[1] Bases can also be $0 < b < 1$, but we'll restrict our attention to increasing functions only.

change of base formula. In particular, if we have variables `b` (base) and `x`, we can compute $\log_b(x)$ using

```
log(x) / log(b)
```

But wait: we have a problem similar to the examples in the previous section. The user could enter invalid values such as $b = -10$ or $x = -2.54$ (logarithms are undefined for non-positive values in any base). We want to ensure that $b > 1$ and $x > 0$. With conditionals, we can now do this. Once we have read in the input from the user we can make a check for good input using an `if` statement.

```
1 if(x <= 0 || b <= 1) {
2   printf("Error: bad input!\n");
3   exit(1);
4 }
```

This code has something new: `exit(1)`. The `exit` function immediately terminates the program regardless of the rest of the code that may remain. The argument passed to `exit` is an integer that represents an *error code*. The convention is that zero indicates "no error" while non-zero values indicate some error. This is a simple way of performing *error handling*: if the user provides bad input, we inform them and quit the program, forcing them to run it again and provide good input. By prematurely terminating the program we avoid any illegal operation that would give a bad result.

Alternatively, we could have split the conditions into two statements and given a more descriptive error message. We use this design in the full program which can be found in Code Sample 16.2. The program also takes the input as command line arguments. Now that we have conditionals, we can actually check that the correct number of arguments was provided by the user and quit in the event that they don't provide the correct number.

16.3.2. Life & Taxes

Let's adapt the conditional statements we developed in Section 3.6.4 into a full C program. The first thing we need to do is establish the variables we'll need and read them in from the user. At the same time we can check for bad input (negative values) for both the inputs.

```
double income, tax, numChildren, credit;

printf("Please enter your Adjusted Gross Income: ");
scanf("%lf", &income);

printf("How many children do you have?");
scanf("%d", &numChildren);

if(income < 0 || numChildren < 0) {
  printf("Invalid inputs");
  exit(1);
}
```

Next, we can code a series of if-else-if statements for the income range. By placing the ranges in increasing order, we only need to check the upper bounds just as in the original example.

```
if(income <= 18150) {
  baseTax = income * .10;
} else if(income <= 73800) {
  baseTax = 1815 + (income - 18150) * .15;
} else if(income <= 148850) {
...
} else {
  baseTax = 127962.50 + (income - 457600) * .396;
}
```

Next we compute the child tax credit, taking care that it does not exceed $3,000. A conditional based on the number of children should suffice as at this point in the program we already know it is zero or greater.

```
if(numChildren <= 3) {
  credit = numChildren * 1000;
} else {
  credit = 3000;
}
```

Finally, we need to ensure that the credit does not exceed the total tax liability (the credit is non-refundable, so if the credit is greater, the tax should only be zero, not negative).

```
if(baseTax - credit >= 0) {
  totalTax = baseTax - credit;
} else {
  totalTax = 0;
}
```

The full program is presented in Code Sample 16.3.

16.3.3. Quadratic Roots Revisited

Let's return to the quadratic roots program we previously designed that uses the quadratic equation to compute the roots of a quadratic polynomial by reading coefficients a, b, c in from the user. One of the problems we had previously identified is if the user enters "bad" input: if $a = 0$, we would end up dividing by zero; if $b^2 - 4ac < 0$ then we would have complex roots. With conditionals, we can now check for these issues and exit with an error message.

Another potential case we might want to handle differently is when there is only one distinct root ($b^2 - 4ac = 0$). In that case, the quadratic formula simplifies to $\frac{-b}{2a}$ and we can print a different, more specific message to the user. The full program can be found in Code Sample 16.4.

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

/**
 * This program computes the logarithm base b (b > 1)
 * of a given number x > 0
 */
int main(int argc, char **argv) {

  double b, x, result;
  if(argc != 3) {
    printf("Usage: %s b x \n", argv[0]);
    exit(1);
  }

  b = atof(argv[1]);
  x = atof(argv[2]);

  if(x <= 0) {
    printf("Error: x must be greater than zero\n");
    exit(1);
  }
  if(b <= 1) {
    printf("Error: base must be greater than one\n");
    exit(1);
  }

  result = log(x) / log(b);
  printf("log_(%f)(%f) = %f\n", b, x, result);
  return 0;
}
```

Code Sample 16.2: Logarithm Calculator Program in C

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

int main(int argc, char **argv) {

  double income, baseTax, credit, totalTax;
  int numChildren;

  //prompt for income from the user
  printf("Please enter your Adjusted Gross Income: ");
  scanf("%lf", &income);

  //prompt for children
  printf("How many children do you have?");
  scanf("%d", &numChildren);

  if(income < 0 || numChildren < 0) {
    printf("Invalid inputs");
    exit(1);
  }

  if(income <= 18150) {
    baseTax = income * .10;
  } else if(income <= 73800) {
    baseTax = 1815 + (income  -18150) * .15;
  } else if(income <= 148850) {
    baseTax = 10162.50 + (income - 73800) * .25;
  } else if(income <= 225850) {
    baseTax = 28925.00 + (income - 148850) * .28;
  } else if(income <= 405100) {
    baseTax = 50765.00 + (income - 225850) * .33;
  } else if(income <= 457600) {
    baseTax = 109587.50 + (income - 405100) * .35;
  } else {
    baseTax = 127962.50 + (income - 457600) * .396;
  }

  if(numChildren <= 3) {
    credit = numChildren * 1000;
  } else {
    credit = 3000;
  }

  if(baseTax - credit >= 0) {
    totalTax = baseTax - credit;
  } else {
    totalTax = 0;
  }

  printf("AGI:           $%10.2f\n", income);
  printf("Tax:           $%10.2f\n", baseTax);
  printf("Credit:        $%10.2f\n", credit);
  printf("Tax Liability: $%10.2f\n", totalTax);

  return 0;
}
```


Code Sample 16.3: Tax Program in C

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

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */
int main(int argc, char **argv) {

  double a, b, c, root1, root2;

  if(argc !=4) {
    printf("Usage: %s a b c\n", argv[0]);
    exit(1);
  }

  a = atof(argv[1]);
  b = atof(argv[2]);
  c = atof(argv[3]);

  if(a == 0) {
    printf("Error: a cannot be zero\n");
    exit(1);
  } else if(b*b < 4*a*c) {
    printf("Error: cannot handle complex roots\n");
    exit(1);
  } else if(b*b == 4*a*c) {
    root1 = -b / (2*a);
    printf("Only one distinct root: %f\n", root1);
  } else {
    root1 = (-b + sqrt(b*b - 4*a*c) ) / (2*a);
    root2 = (-b - sqrt(b*b - 4*a*c) ) / (2*a);

    printf("The roots of %fx^2 + %fx + %f are: \n", a, b, c);
    printf("  root1 = %f\n", root1);
    printf("  root2 = %f\n", root2);
  }
    return 0;
}
```

Code Sample 16.4: Quadratic Roots Program in C With Error Checking

17. Loops

C supports while loops, for loops, and do-while loops using the keywords `while`, `for`, and `do` (along with another `while`). Continuation conditions for loops are enclosed in parentheses, `(...)` and the blocks of code associated with the loop are enclosed in curly brackets.

17.1. While Loops

Code Sample 17.1 contains an example of a basic while loop in C. Just as with conditional statements, our code styling places the opening curly bracket on the same line as the `while` keyword and continuation condition. The inner block of code is also indented and all lines in the block are indented to the same level.

```
1 int i = 1; //Initialization
2 while(i <= 10) {  //continuation condition
3   //perform some action
4   i++; //iteration
5 }
```

Code Sample 17.1: While Loop in C

In addition, the continuation condition does *not* contain a semicolon since it is not an executable statement. Just as with an if-statement, if we *had* placed a semicolon it would have led to unintended results. Consider the following:

```
1 while(i <= 10); {
2   //perform some action
3   i++; //iteration
4 }
```

A similar problem occurs: the `while` keyword and continuation condition bind to the next executable statement or code block. As a consequence of the semicolon, the executable statement that gets bound to the while loop is *empty*. What happens is even worse: the program will enter an infinite loop. To see this, the code is essentially equivalent to the following:

```
while(i <= 10) {
}
{
  //perform some action
  i++; //iteration
}
```

In the while loop, we never increment the counter variable `i`, the loop does nothing, and so the computation will continue on forever! Some compilers will warn you about this, others will not. It is valid C and it will compile and run, but obviously won't work as intended. Avoid this problem by using proper syntax.

Another common use case for a while loop is a flag-controlled loop in which we use a Boolean flag rather than an expression to determine if a loop should continue or not. Recall that in C, zero is treated as *false* and any non-zero numeric value is treated as *true*. We can thus create an implicit Boolean flag by using an integer variable and setting it to 1 for *true* and 0 for *false* (when we want the loop to terminate. An example can be found in Code Sample 17.2.

```
int i = 1;
int flag = 1;
while(flag) {
  //perform some action
  i++; //iteration
  if(i>10) {
    flag = 0;
  }
}
```

Code Sample 17.2: Flag-controlled While Loop in C

17.2. For Loops

For loops in C use the familiar syntax of placing the initialization, continuation condition, and iteration on the same line as the keyword `for`. An example can be found in Code Sample 17.3.

```
int i;
for(i=1; i<=10; i++) {
  //perform some action
}
```

Code Sample 17.3: For Loop in C

Again, note the syntax: semicolons are placed at the end of the initialization and continuation condition, but *not* the iteration statement. Just as with while loops, the opening curly bracket is placed on the same line as the `for` keyword. Code within the loop body is indented, all at the same indentation level.

Another observation is that we declared the index variable `i` *prior* to the for loop. Some languages allow you to declare the index variable in the initialization statement, for example `for(int i=1; i<=10; i++)`. Doing so *scopes* the index variable to the loop and so `i` would be out-of-scope before and after the loop body. This is a nice convenience and is generally good practice. However, C89 and prior standards do not allow you to do this; the variable must be declared prior to the loop structure. C99 and newer standards do allow you to do this and some compilers will be somewhat forgiving when you use the newer syntax (by supporting their own non-standard extensions to C). For maximum portability, we'll follow the older convention.

17.3. Do-While Loops

Finally, C does support do-while loops. Recall that the difference between a while loop and a do-while loop is when the continuation condition is checked. For a while loop it is *prior* to the beginning of the loop body and in a do-while loop it is at the *end* of the loop. This means that a do-while always executes *at least once*. An example can be found in Code Sample 17.4.

```
1 int i;
2 do {
3   //perform some action
4   i++;
5 } while(i<=10);
```

Code Sample 17.4: Do-While Loop in C

Note the syntax and style: the opening curly bracket is again on the same line as the keyword `do`. The `while` keyword and continuation condition are on the same line as the closing curly bracket. In a slight departure from consistent syntax, a semicolon *does* appear at the end of the continuation condition even though it is not an executable statement.

17.4. Other Issues

C does not support a traditional foreach loop. When iterating over a collection like an array, you iterate a traditional index variable, typically with a for loop. However, there

are some syntactic tricks that you can use to get the same effect. Some of this will be a preview of Chapter 20 where we discuss arrays in C, but in short, you can assign a variable to an element in an array in iteration statement:

```
double arr[] = {1.41, 2.71, 3.14};
int n = 3;
int i = 0;
double x;
for(x=arr[i]; i<n; x=arr[++i]) {
  //x now holds the i-th element in arr
}
```

The initialization sets the variable `x` to the first element in the array, `arr[0]`. The loop continues for as many elements as there are in the array, `n`. The iteration does two things: it assigns `x` to the next element in the array while at the same time incrementing the index variable using the prefix increment operator (see Section 2.3.6).

17.5. Examples

17.5.1. Normalizing a Number

Let's revisit the example from Section 4.1.1 in which we *normalize* a number by continually dividing it by 10 until it is less than 10. The code in Code Sample 17.5 specifically refers to the value 32145.234 but would work equally well with any value of `x`.

```
double x = 32145.234;
int k = 0;
while(x > 10) {
  x = x / 10; //or: x /= 10;
  k++;
}
```

Code Sample 17.5: Normalizing a Number with a While Loop in C

17.5.2. Summation

Let's revisit the example from Section 4.2.1 in which we computed the sum of integers $1 + 2 + \cdots + 10$. The code is presented in Code Sample 17.6

```c
int i;
int sum = 0;
for(i=1; i<=10; i++) {
  sum += i;
}
```

Code Sample 17.6: Summation of Numbers using a For Loop in C

Of course we could easily have generalized the code somewhat. Instead of computing a sum up to a particular number, we could have written it to sum up to another variable `n`, in which case the for loop would instead look like the following.

```c
for(i=1; i<=n; i++) {
  sum += i;
}
```

17.5.3. Nested Loops

Recall that you can write loops within loops. The inner loop will execute fully *for each* iteration of the outer loop. An example of two nested of loops in C can be found in Code Sample 17.7.

```c
int i, j;
int n = 10;
int m = 20;
for(i=0; i<n; i++) {
  for(j=0; j<m; j++) {
    printf("(i, j) = (%d, %d)\n", i, j);
  }
}
```

Code Sample 17.7: Nested For Loops in C

The inner loop execute for $j = 0, 1, 2, \ldots, 19 < m = 20$ for a total of 20 times. However, it executes 20 times *for each* iteration of the outer loop. Since the outer loop execute for $i = 0, 1, 2, \ldots, 9 < n = 10$, the total number of times the `printf` statement execute is $10 \times 20 = 200$. In this example, the sequence $(0, 0), (0, 1), (0, 2), \ldots, (0, 19), (1, 0), \ldots, (9, 19)$ will be printed.

17.5.4. Paying the Piper

Let's adapt the solution for the loan amortization schedule we developed in Section 4.7.3. First, we'll read the principle, terms, and interest as command line inputs.

Adapting the formula for the monthly payment and using the standard math library's `pow` function, we get

```
double monthlyPayment = (monthlyInterestRate * principle) / (1 - pow( (1 + monthl
```

However, recall that we may have problems due to accuracy. The monthly payment could come out to be a fraction of a cent, say $43.871. For accuracy, we need to ensure that all of the figures for currency are rounded to the nearest cent. The standard math library does have a `round` function, but it only rounds to the nearest whole number, not the nearest 100th.

However, we can *adapt* the "off-the-shelf" solution to fit our needs. If we take the number, multiply it by 100, we get (say) 4387.1 which we can now round to the nearest whole number, giving us 4387. We can then divide by 100 to get a number that has been rounded to the nearest 100th! In C, we could simply do the following.

```
monthlyPayment = round(monthlyPayment * 100.0) / 100.0;
```

We can use the same trick to round the monthly interest payment and any other number expected to be whole cents. To output our numbers, we use `printf` and take care to align our columns to make make it look nice. To finish our adaptation, we handle the final month separately to account for an over/under payment due to rounding. The full solution can be found in Code Sample 17.8.

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

int main(int argc, char **argv) {

  if(argc != 4) {
    printf("Usage: %s principle apr terms\n", argv[0]);
    exit(1);
  }

  double principle = atof(argv[1]);
  double apr = atof(argv[2]);
  int n = atoi(argv[3]);

  double balance = principle;
  double monthlyInterestRate = apr / 12.0;
  int i;

  //monthly payment
  double monthlyPayment = (monthlyInterestRate * principle) /
          (1 - pow( (1 + monthlyInterestRate), -n));
  //round to the nearest cent
  monthlyPayment = round(monthlyPayment * 100.0) / 100.0;

  printf("Principle: $%.2f\n", principle);
  printf("APR: %.4f%%\n", apr*100.0);
  printf("Months: %d\n", n);
  printf("Monthly Payment: $%.2f\n", monthlyPayment);

  //for the first n-1 payments in a loop:
  for(i=1; i<n; i++) {
    //  compute the monthly interest, rounded:
    double monthlyInterest =
            round( (balance * monthlyInterestRate) * 100.0) / 100.0;
    //  compute the monthly principle payment
    double monthlyPrinciplePayment = monthlyPayment - monthlyInterest;
    //  update the balance
    balance = balance - monthlyPrinciplePayment;
    //  print i, monthly interest, monthly principle, new balance
    printf("%d\t$%10.2f  $%10.2f  $%10.2f\n", i, monthlyInterest,
            monthlyPrinciplePayment, balance);
  }

  //handle the last month and last payment separately
  double lastInterest = round( (balance * monthlyInterestRate) * 100.0) / 100.0;
  double lastPayment = balance + lastInterest;

  printf("Last payment = $%.2f\n", lastPayment);

  return 0;
}
```

Code Sample 17.8: Loan Amortization Program in C

18. Functions

As a procedural-style language, functions are essential in C programming. As we've already seen, C provides a large library of standard functions to perform basic input/output, math, and many other functions. C also provides the ability to define and use your own functions.

When you define functions in C, careful thought must be made as to the naming of your functions. This is because C does *not* support function overloading. When you name a function, that is the *only* function that can have that name. Consequently, you cannot, in general, use the same function names as defined in the standard libraries or any other 3rd party library that you would like to use in your programs.

C supports both call by value and call by reference using *pointers* (see Section 18.2). C also supports vararg functions (`printf()` being a prime example) and allows you to define vararg functions, but we will not cover them in depth here. Finally, parameters are not, in general, optional. For modern versions of C the omission of parameters is a syntax error. For older versions, complex rules dictate what happens when arguments are omitted, but doing so usually results in garbage.

18.1. Defining & Using Functions

For modern C, defining functions is a two step process. First, you *declare* the function by define the function signature using a *prototype*. Then later, you *define* the function by providing a function body that defines what the function does.

18.1.1. Declaration: Prototypes

Just as with variables, functions in C must be *declared* before they can be used. The modern way to declare a function in C is to use a *prototype* declaration which specifies the function's signature including its return type, identifier, and parameters. However, a prototype does *not* include the actual body of function. Instead, the function's *definition*, which includes the function body is included later in the program. Consequently, a prototype always ends with a semicolon.

Typically, the documentation for functions is included with the prototype but is *not* repeated with the function definition. This is a principle known as Don't Repeat Yourself

(DRY). Consider the following examples. In these examples we use a commenting style known as "doc comments." This style was originally developed for Java but has since been adopted by many other languages.

```
/**
 * Computes the sum of the two arguments.
 */
int sum(int a, int b);

/**
 * Computes the Euclidean distance between the 2-D points,
 * (x1,y1) and (x2,y2).
 */
double getDistance(double x1, double y1, double x2, double y2);

/**
 * Computes a monthly payment for a loan with the given
 * principle at the given APR (annual percentage rate) which
 * is to be repaid over the given number of terms (usually
 * months).
 */
double getMonthlyPayment(double principle, double apr, int terms);
```

In each of these, the return type is the first thing specified. The function identifier (name) is then specified. Function names must follow the same naming rules as variables: they must begin with an alphabetic character and may contain alphanumeric characters as well as underscores. However, using modern coding conventions we usually name functions using lower camel casing.

Note again, each prototype ends with a semicolon. Further, prototypes do *not* specify what the function does, they only specify its signature. Later in the program, we can provide the actual definition of each function by using the following syntax. We repeat the signature, but instead of using a semicolon, we provide a code block, enclosed using opening/closing curly brackets, that specifies the function body. Here are the definitions from the prototype examples above:

```
int sum(int a, int b) {
  return (a + b);
}

double getDistance(double x1, double y1, double x2, double y2) {
  double xDiff = (x1-x2);
  double yDiff = (y1-y2);
  return sqrt( xDiff * xDiff + yDiff * yDiff);
}

double getMonthlyPayment(double principle, double apr, int terms) {
  double rate = (apr / 12.0);
  double payment = (principle * rate) / (1-pow(1+rate, -terms));
  return payment;
}
```

The keyword `return` is used to specify the value that is returned to the calling function.

18.1.2. Void Functions

The keyword `void` can be used in C to indicate a function does *not* return a value, in which case it is called a "void function." Though it is not necessary, it is still good practice to include a `return` statement.

```
//prototype:
void printCopyright();

//definition:
void printCopyright() {
  printf("(c) Bourke 2015\n");
}
```

In the example above, we've also illustrated how to define a function that has no inputs. Some sources may include an explicit `void` keyword as a parameter to indicate the function takes no parameters as in `void printCopyright(void);`.

18.1.3. Organizing Functions

The separation of a declaration (prototype) and a function definition provides a natural way to organize functions in C. We place prototypes into a *header* file which has a file extension `.h` and then place the corresponding function definitions into a *source* file with the usual file extension, `.c`.

We've seen this before with the standard libraries: we use `#include<math.h>` to "include"

the math library's header file in our code. This essentially brings in the math library function prototypes so that we can write calls to, say, `sqrt()` or `sin()`. Only when we compile do we actually need to *link* our code to the function definitions.

When we separate prototypes into header files and definitions into source files we also need to "include" the prototypes in our source file just as we would need to include them in any other file in which we use one of the functions. Suppose our functions above have their prototypes in a file named `utils.h` and their definitions in a file named `utils.c`. In the `utils.c` source file we would typically use the following syntax to include the header file:

```
#include "utils.h"
```

We use the double quote syntax with user-defined libraries while the usual less-than/greater-than syntax is used with standard libraries. With the less-than/greater-than syntax, the compiler will usually attempt to look for the header file(s) in a specified system directory which it will fail to find if it is a user-defined library.

Furthermore, other elements are usually included in header files such as preprocessor directives and other declarations (such as enumerated types and structures which we introduce later).

18.1.4. Calling Functions

Once a function has been defined, or at least a prototype has been brought into scope via an `#include` statement, you can write code to call your function(s). The syntax for doing so it to simply provide the function name followed by parentheses containing values or variables to pass to the function. Some examples:

```
1  int a = 10, b = 20;
2  int c = sum(a, b); //c contains the value 30
3
4  //invoke a function with literal values:
5  double dist = getDistance(0.0, 0.0, 10.0, 20.0);
6
7  //invoke a function with a combination:
8  double p = 1500.0;
9  double r = 0.05;
10 double monthlyPayment = getMonthlyPayment(p, r, 60);
```

By default, all primitive types including `int`, `double`, and `char` are passed by valuefunction arguments are passed by value. To be able to pass arguments by reference, we need to use *pointers*.

18.2. Pointers

Consider the following line of C code.

```
int a = 10;
```

This line creates an integer variable and sets it equal to 10. In more detail, this line creates a spot in memory (typically 32 bits) and stores a binary representation of the value 10 at that location. In many instances, we don't care *where* the variable is stored in memory. However, we may have need to communicate that memory location to other functions. To do so, we can use *pointers*.

A pointer in C is a reference to a memory location. Because different types (`int`, `double`, `char`) take a different amount of memory, it is necessary to have a pointer for each type. That is, a pointer that points to a memory location that stores an `int` or a pointer that points to a memory location that stores a `double`, etc.

The syntax for declaring a pointer is to use an asterisk.

```
1 //regular variable declarations
2 int a;
3 double b;
4
5 //pointer variable declarations
6 int *ptrA;
7 double *ptrB;
```

If `ptrA` represents a memory location, what values can it take on? At the end of the day, a memory location is just a number, so you *could* do something like the following.

```
ptrA = 10;
```

Though syntactically this makes sense (and generally the compiler will let you do this with perhaps at most a warning), its not really what you want. This assigns to the pointer variable `ptrA` the value 10, which will be interpreted as the *memory address* 10. This memory address may not belong to your program, or it may not even exist as a valid memory address. Attempts to access the value stored at an arbitrary memory location may be illegal and may result in the operating system killing the program with a segmentation fault or similar error.

There are many reasons why a program should not be allowed access to arbitrary memory locations, but one of the prime reasons is security. Imagine if the operating system allowed a program access to any part of memory; in particular memory that contained sensitive information such as passwords or secret Secure Sockets Layer (SSL) keys. To prevent this, operating systems generally only allow a program to access its own memory.

Referencing Operator

So how do we assign a valid value to a pointer? We do so using a *referencing operator*. Given a normal variable as above, we place an ampersand in front of it to get the memory address of the variable. For example:

```
ptrA = &a;
ptrB = &b;
```

The operation `&a` gets the memory address of the variable `a` and we can assign it to a pointer value. Again, note that the pointer type and variable type should match. making a `double` pointer point to an `int` variable type such as

`ptrB = &a;`

is valid syntax, but since the two types use different amounts of memory you may get garbage results.

There is a special value used in C called `NULL` which is a (case-sensitive) keyword used for uninitialized, undefined, empty or otherwise invalid or meaningless value. In the context of memory locations, `NULL` "points" to nothing. As with regular variables, its best practice to initialize pointer values to `NULL`. For example,

`int *ptrA = NULL;`

Without an initialization, the pointer may point to a random memory address which may be dangerous to attempt to access. You can also test whether or not a pointer points to `NULL` using the usual equality operator.

```
int *ptrA = NULL;
...
if(ptrA == NULL) {
  printf("Error: invalid memory location\n");
}
```

Dereferencing Operator

Once we have a valid pointer to a memory location, we may want to manipulate the contents of the memory it references. To do this we use the inverse operator, the *dereferencing operator* which again uses an asterisk. Given a pointer variable `ptrA`, we apply an asterisk in front of it to turn it into a regular variable. Consider the following example.

```c
//declare a normal integer variable
int a = 10;
//declare a pointer and initialize it to NULL
int *ptrA = &a;

//point ptrA to a's memory location
ptrA = &a;

//change the value of the variable a using its pointer
*ptrA = 20;

//now the variable a has a value of 20:
printf("a = %d\n", a); //prints "a = 20"
```

How each of these lines of code operate in memory is depicted in Figure 18.1.

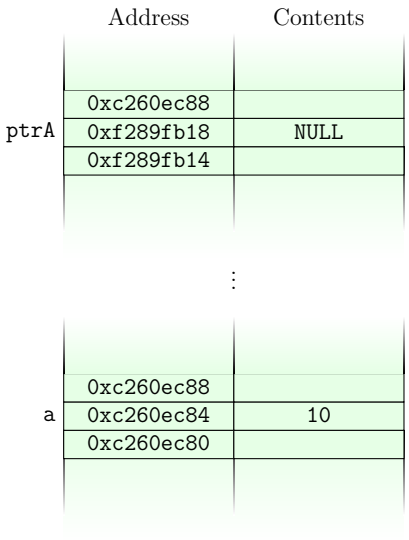
18.2.1. Passing By Reference

Now that we have the ability to reference a memory location using pointers, we can write functions that pass variables by reference. To do so, we use the same asterisk syntax used with pointer variables.

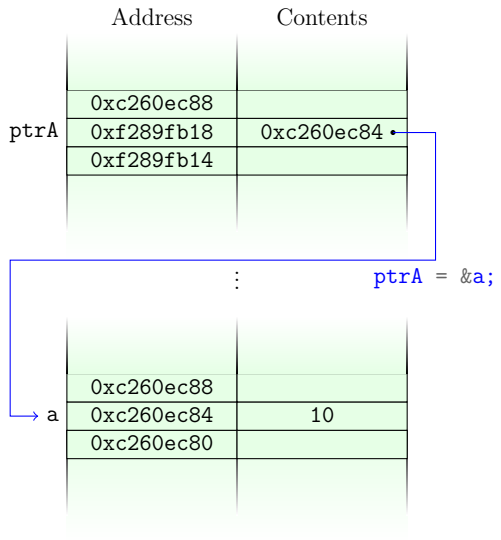
```c
//prototypes
/**
 * This function sums the first two variables (passed by
 * value) and places the result into the third variable
 * (passed by reference).
 */
void sum(int a, int b, int *c);

/**
 * This function swaps the values stored in the
 * two variables passed by reference.
 */
void swap(int *a, int *b);
```

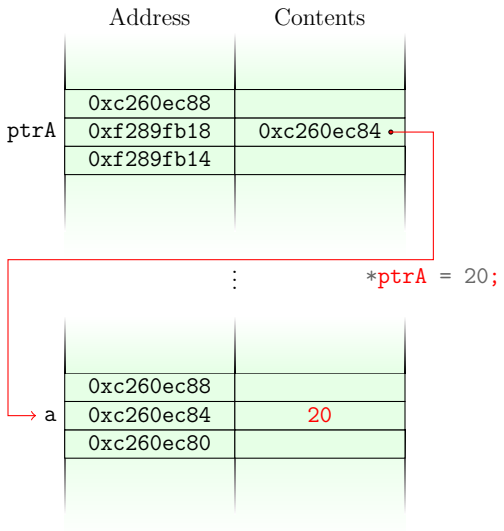
In the function definitions, we can use the dereferencing operator to access or modify the value stored in the variable pointed to by the pointer.



(a) After the first two lines memory has been dedicated for the variable `a` and the pointer variable `ptrA` and their values have been initialized.



(b) Making `ptrA` point to the variable `a`'s memory location. The value stored in the variable `ptrA` is a memory address.



(c) Dereferencing `ptrA` and assigning a value changes the value stored at what it *points* to.

Figure 18.1.: Pointer Operations. Pointers can be made to point to other variable's memory locations. You can manipulate/access values of variables via their pointers using dereferencing.

```
void sum(int a, int b, int *c) {
  int x = a + b;
  *c = x;
  return;
}

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  return;
}
```

To invoke these functions, we need to pass pointers to the functions appropriately. We could do this by creating pointer variables or using the referencing operator directly.

```
int x = 10;
int y = 20;
int c;
int *ptrC = &c;
sum(x, y, ptrC);
//at this point c contains the value 30

swap(&x, &y);
//at this point, the values in x and y have been swapped
// x contains 20 and y contains 10
```

This should look familiar. We saw this same syntax when we used `scanf()` to read input from the standard input. We needed to place an ampersand in front of each variable in order to pass the variable by reference so that `scanf()` could place the results into the respective memory locations. If the variables had been passed by value, then `scanf()` would not have been able to manipulate their values.

You can also specify functions to *return* pointers which we discuss in detail in Chapter 20.

18.2.2. Function Pointers

Functions are just pieces of code that reside somewhere in memory just as variables do. Since we can create pointers that point to variables, it makes sense to be able to create variables that point to functions too! These are referred to as *function pointers*.

The syntax for declaring function pointers is similar to variable pointers. However, since a function's signature involves a return type and parameter list, these need to be specified. For example, suppose we wanted to create a function pointer that could point to the

math library's `sqrt()` function which takes a single `double` parameter and returns a `double` value.

```
double (*ptrToSqrt)(double) = NULL;
```

The above line creates a function pointer that can point to any function that takes a single `double` parameter and returns a `double` value. As is good practice, we've initialized it to point to `NULL`. The function pointer itself is named `ptrToSqrt`. To make it point to the `sqrt()` function we can use the following syntax.

```
ptrToSqrt = sqrt;
```

This is because a function's identifier acts as a pointer as well! Once we have a pointer to a function, we can invoke the function via its pointer as we would any other function call.

```
double x = ptrToSqrt(2.0);
```

Some more examples:

```
//this pointer can point any function that takes
//three arguments: an int, double, and a char
//and returns an int value
int (*ptrToFunc)(int, double, char)= NULL;

double x;
double (*ptr)(double) = NULL;
//we can make it point to sqrt:
ptr = sqrt;
x = ptr(2.0); //x contains 1.4142...
//or we can make it point to fabs
ptr = fabs;
x = ptr(-10.5); //x contains 10.5
```

You generally want to create and use function pointers when passing and returning functions as arguments to other functions as callbacks. We discuss this in further detail in Chapter **??**.

18.3. Examples

18.3.1. Generalized Rounding

Recall that the standard math library provides a `round()` function that rounds a number to the nearest whole number. Often, we've had need to round to cents as well. We now have the ability to write a function to do this for us. Before we do, however, let's think more generally. What if we wanted to round to the nearest tenth? Or what if we

wanted to round to the nearest 10s or 100s place? Let's write a general purpose rounding function that allows us to specify *which* decimal place to round.

The most natural input values would be to specify the place using an integer exponent. That is, if we wanted to round to the nearest tenth, then we would pass it -1 as $0.1 = 10^{-1}$, -2 if we wanted to round to the nearest 100th, etc. On the positive end passing in 0 would correspond to the usual round function, 1 to the nearest 10s spot, and so on.

Moreover, we could demonstrate good code reuse (as well as procedural abstraction) by *scaling* the input value and reusing the functionality already provided in the math library's `round()` function. We could further define a `roundToCents()` function that used our generalized round function.

Let's also think about organization. We could place the prototypes into a `round.h` header file and the corresponding definitions in a `round.c` source file. The contents of these two files are presented here:

```
/**
 * Rounds to the nearest digit specified by the place
 * argument.  In particular to the (10^place)-th digit
 */
double roundToPlace(double x, int place);

/**
 * Rounds to the nearest cent
 */
double roundToCents(double x);
```

```
#include<math.h>
#include "round.h"

double roundToPlace(double x, int place) {
  double scale = pow(10, -place);
  double rounded = round(x * scale) / scale;
  return rounded;
}

double roundToCents(double x) {
  return roundToPlace(x, -2);
}
```

Observe that neither of these files contains a `main()` function. By themselves they would not be able to be compiled into an executable program. We've essentially built a small *library* of rounding functions. We could compile them though into a binary *object* file using gcc (something like `gcc -c round.c`). We could then link into the object file when compiling an executable program that uses these functions.

18.3.2. Quadratic Roots

Another advantage of passing variables by reference is that we can "return" multiple values with one function call. Functions are limited in that they can only return at most one value. But if we pass multiple parameters by reference, the function can manipulate the contents of them, thereby communicating (though not strictly returning) multiple values.

Consider again the problem of computing the roots of a quadratic equation,

$$ax^2 + bx + c = 0$$

using the quadratic formula,

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Since there are two roots, we may have to write two functions, one for the "plus" root and one for the "minus" root both of which take the coefficients, a, b, c as arguments. However, if we wrote a single function that took the coefficients as parameters by value as well as two other parameters by reference, we could place *both* root values, one in each of the by-reference variables.

```
1 void quadraticRoots(double a, double b, double c,
2                      double *root1, double *root2) {
3   double discriminant = sqrt(b*b - 4*a*c);
4   *root1 = (-b + discriminant) / (2*a);
5   *root2 = (-b - discriminant) / (2*a);
6   return;
7 }
```

By using pass by reference variables, we avoid multiple functions. We also note that the return value in this case is unused since we are "returning" the root values in the two pass-by-reference variables. This frees up the return value to be used to communicate *errors* to the calling function. Recall that there could be several "bad" inputs to this function. The roots could be complex values, the coefficient a could be zero, etc. And now that we are dealing with pointers, the pointers could be invalid (point to `NULL`). In the next chapter, we examine how we can use the return value to communicate different errors to the calling function, letting it *handle* those errors.

19. Error Handling

The C language does not support exceptions or exception handling. Instead, the usual method of error handling is done through defensive programming. As a user, it is your responsibility to write code that checks for invalid or unsafe operations before executing them and handle the error appropriately.

One way that we've already seen to handle errors in C is to use the `exit()` function in the standard library. This function immediately terminates the execution of our program and "returns" an integer valued "exit code." This exit code can be used to communicate errors between different processes. The process that executed the program can use it to determine if the program exited with or without an error. By convention, 0 indicates an error while a non-zero value indicates an error. Using the `exit()` function should only be done when an error should be considered *fatal* or severe enough that the program should immediately terminate.

19.1. Language Supported Error Codes

In the standard libraries, some functions are designed to indicate a n error by returning a special "flag" value such as -1 or `NULL`. In general, though, the standard library functions communicate an error code represented as an integer value. As previously mentioned, C uses the convention that 0 indicates "no error" while a non-zero value indicates an error.

Error codes are communicated through a special global variable called `errno` which is defined in the `errno.h` standard header file. This header file also defines several macros that are identified with various error codes. The C standard actually only requires the following three error codes to be supported.

- `EDOM` indicates an error in the *domain* of a function; that is, an error with respect to the function's input value(s). For example, calling the math library's `sqrt(-1)` on a negative value results in an `EDOM` error as C does not support complex numbers.
- `ERANGE` indicates an error in the *range* of a function; that is, an error with respect to the function's output value(s). For example, calling the math library's `log(0)` with a zero value results in an `ERANGE` error as C does not support $-\infty$ as an actual number.
- `EILSEQ` indicates an illegal byte sequences in characters on systems that use UTF-8.

All three of these are defined in the `errno.h` header file. Depending on the system, additional error codes may also be defined and supported (see POSIX Error Codes below).

When an error occurs, a function will set the global variable `errno` to one of these error code values. Upon returning from a function, you can check for these error codes. Since these error codes are represented as integers, you simply use the numerical comparison operator, `==`. You can check for no error by making a comparison to zero.

In addition, the standard string library, `string.h` provides a function, `char * strerror(errno)` that can be used to map the value in `errno` to a human-readable error message. We discuss strings in detail later on, but we can see how to use this function in the following demonstration (see below). The output of this program is as follows.

```
result: 1.4142, error: 0
result: -nan, error: 33
it was an EDOM error
Error Message: Numerical argument out of domain
result: -inf, error: 34
it was an ERANGE error
Error Message: Numerical result out of range
```

For this particular system, the `EDOM` and `ERANGE` error codes were associated with the integer values 33 and 34 respectively. These numbers are not necessarily the same on all systems so comparisons must be made against the macro names for portability.

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include<errno.h>

int main(int argc, char **argv) {

  double a = -1, b = 2, c = 0.0;
  double x;

  //okay
  x = sqrt(b);
  printf("result: %.4f, error: %d\n", x, errno);

  //NaN and EDOM error...
  x = sqrt(a);
  printf("result: %.4f, error: %d\n", x, errno);

  //make a comparison
  if(errno == EDOM) {
    printf("it was an EDOM error\n");
  }

  //error messages can be accessed via:
  char *errMsg = strerror(errno);
  printf("Error Message: %s\n", errMsg);

  //ERANGE error
  x = log(c);
  printf("result: %.4f, error: %d\n", x, errno);

  if (errno == ERANGE) {
    printf("it was an ERANGE error\n");
  }

  //error messages can be accessed via:
  errMsg = strerror(errno);
  printf("Error Message: %s\n", errMsg);

  return 0;

}
```

19.1.1. POSIX Error Codes

POSIX is an IEEE set of standards for maintaining compatibility between various operating systems. It defines several rules and expectations that operating systems must adhere to in order to be POSIX *compliant*. Such standards allow developers to develop code that should be interoperable among a collection of different operating systems, reducing the need for rewrites and reimplementations for different systems.

In particular, POSIX compliant systems define many other error codes (see [3]) that can be used beyond the three mentioned above. For example, the `ENOENT` error code corresponds to "No such file or directory" and `EACCES` corresponds to a "Permission denied" error.

19.2. Error Handling By Design

In our own code we *could* communicate errors to calling functions by setting `errno`, however, we may run into compatibility issues with the standard error codes or POSIX error codes. Instead, it may be more appropriate to do error handling by utilizing the return value of a function to communicate an error code.

That is, to do error handling we could design our functions to always return an `int` value to indicate an error: 0 for no error and some non-zero value to indicate various different types of errors. Of course, as a consequence any value that needs to be "returned" to the calling function would need to be done so via a pass by reference variable.

As an example, let's revisit the quadratic roots example in Section 18.3.2. By returning the two output values via pass by reference variables, we freed up the return value. We can now modify our function to return an integer indicating an error code instead.

Previously we had identified several different types of errors: division by zero (if $a = 0$), complex roots (if $b^2 - 4ac < 0$) and a `NULL` pointer error if the variables passed by reference were `NULL`. We can now modify our function to check for these errors and return an appropriate error code. We return zero in the event that no error was encountered.

```
int quadraticRoots(double a, double b, double c,
                   double *root1, double *root2) {
  if(a == 0) {
    return 1;
  } else if( b*b - 4*a*c < 0 ) {
    return 2;
  } else if(root1 == NULL || root2 == NULL) {
    return 3;
  }
  double discriminant = sqrt(b*b - 4*a*c);
  *root1 = (-b + discriminant) / (2*a);
  *root2 = (-b - discriminant) / (2*a);
  return 0;
}
```

Now when a function invokes our `quadraticRoots()` it can check to see what kind of error code it returned and handle the error in whatever way it wants.

There is still an issue, however. The usage of the integers 1, 2, 3 to indicate the various errors was arbitrary. These are essentially magic numbers that the calling function would have to deal with by making comparisons with various integers. The numbers themselves are meaningless and someone using them would need to constantly refer to documentation to understand which integer corresponded to which error condition.

It would be much better if we could follow the strategy of the `errno` and define human-readable identifiers for each error code. We could accomplish this by defining macros, but another solution is to use an *enumerated type*.

19.3. Enumerated Types

C allows you to define enumerated types which allow you to define a fixed list of possible values. For example, the days of the week or months of the year are possible values used in a date. However, they are more-or-less fixed (no one will be adding a new day of the week or month any time soon). An enumerated type allows us to define these values using human-readable keywords (much like the `#define` macro).

To declare an enumerated type we use the keywords `typedef enum` (short for type definition and enumeration). We then provide a comma delimited list of keywords inside a code block. At the end of the code block we provide the type's identifier. That is, the name of the type itself. For example, the names of integer and floating-point types in C are `int` and `double` respectively. This identifier gives our type a name that we can later use to declare variables of that type. Consider the following example.

```
typedef enum {
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY
} DayOfWeek;
```

In this example we've defined an enumeration of the days of the week. The name of the type itself is `DayOfWeek` and we can now declare variables of this type. The possible *values* it can take are `SUNDAY`, `MONDAY`, etc. and we can use these keywords in our program. For example,

```
DayOfWeek today = MONDAY;

if(today == SUNDAY || today == SATURDAY) {
  printf("Its the weekend!\n");
}
```

Note the modern naming conventions: the type identifier uses upper camel casing while the enumerated values follow an upper case underscore convention. Though our example does not contain a value with multiple words, if it had, we would have used an underscore to separate them. Furthermore, enumerated type declarations are usually placed in separate header files along with function prototype declarations.

Care must be taken when using enumerated types, however. Internally, C simply associates integers with the values. Thus, in our example, `SUNDAY` is actually `0`, `MONDAY` is `1`, and `SATURDAY` is `6`. When we do assignments or equality comparisons, we're actually just comparing integers.

Consequently, a `DayOfWeek` variable *may* be assigned values that do not correspond to our enumeration. For example,

```
DayOfWeek today = 1000;
```

is valid code and will not (in general) result in any compiler errors or warnings, even though it is assigning an invalid value to the variable. Care must be taken to only assign valid values to an enumerated type variable. Proper error checking should also be done.

Despite this limitation, using enumerated types in C provides an obvious advantage. Without an enumerated type we'd be forced to use a collection of magic numbers to indicate values. Even for something as simple as the days of the week we'd be constantly trying to remember: which day is Wednesday again? I forget, do our week start with Monday or Sunday? Etc. By using an enumerated type these questions are mostly moot as we can use the more human-readable keywords and eliminate the guess work.

19.4. Using Enumerated Types for Error Codes

Let's apply enumerated types to our `quadraticRoots()` function example from before. First, we define our enumerated type which includes all types of errors including a `NO_ERROR` type. Note that we start with `NO_ERROR` as its value will be zero following our convention.

```
typedef enum {
  NO_ERROR,
  DIV_BY_ZERO_ERROR,
  COMPLEX_ROOT_ERROR,
  NULL_POINTER_ERROR
} ErrorCode;
```

Now in the `quadraticRoots()` function, we can return the appropriate error code.

```
ErrorCode quadraticRoots(double a, double b, double c,
                         double *root1, double *root2) {
  if(a == 0) {
    return DIV_BY_ZERO_ERROR;
  } else if( b*b - 4*a*c < 0 ) {
    return COMPLEX_ROOT_ERROR;
  } else if(root1 == NULL || root2 == NULL) {
    return NULL_POINTER_ERROR;
  }
  double discriminant = sqrt(b*b - 4*a*c);
  *root1 = (-b + discriminant) / (2*a);
  *root2 = (-b - discriminant) / (2*a);
  return NO_ERROR;
}
```

20. Arrays

TODO

21. Strings

TODO

Part II.

The Java Programming Language

22. Basics

The Java programming language was developed in the early 1990s at Sun Microsystems by James Gosling, Mike Sheridan, and Patrick Naughton. Its original intention was to enable cable box sets to be more interactive. By the mid-90s, Java was retargeted toward the World Wide Web (WWW). The first public release came on May 23, 1995 with the first Java Development Kit (JDK), Java 1.0 on January 23rd, 1996. A new, updated release has come about every other year. As of 2014, Java 8 is the current stable version.

Today, Java is one of the most popular programming languages, consistently ranked as one of the top 2 languages (see `http://www.tiobe.com`). It is now owned and maintained by Oracle, but there are many open source tools, compilers and runtime environments available. Java is used in everything from mobile devices (Android) and desktop applications to enterprise-level servers.

From its inception, Java was designed with 5 basic principles:

1. Simple, Object-oriented, familiar
2. Robust and secure
3. Architecture-neutral and portable
4. High performance
5. Interpreted, threaded and dynamic

Java offers many key features that have made it popular. It is unique in that it is not entirely compiled nor interpreted. Instead, Java source code is compiled into an intermediate form, called Java bytecode. This bytecode is not directly runnable on a processor. Instead, a JVM, an application that was written and compiled a particular processor, interprets the bytecode and runs the application. This added layer of abstraction means that Java source code can be written once (and compiled once) and then run anywhere on any device that has a JVM. The added layer of abstraction makes development easier, but does come at a cost in performance. However, the most recent JVMs have offered performance that is comparable to native machine code in many applications.

Another key feature is that Java has its own automated garbage collection. Some languages require manual memory management, meaning that requesting, managing, and freeing up memory is part of the code that you write as a developer. Failure to handle memory management properly can lead to wasted resources (memory leaks), poor or

unstable performance, and even more serious security issues (buffer overflows). In Java, there is no manual memory management. The JVM handles the allocation and clean up of memory automatically.

In following with the five design principles, Java is similar in syntax to C (called "C-style syntax"). Executable statements are terminated by semicolons, code blocks are defined by opening/closing curly brackets, etc. Java is also fundamentally an Object-Oriented Programming (OOP) language. With the exception of a few primitive types, in Java, everything is an object or belongs to an object.

22.1. Getting Started: Hello World

The hallmark of the introduction of programming languages is the *Hello World!* program. It consists of a simple program whose only purpose is to print out the message "Hello World!" to the user in some manner. The simplicity of the program allows the focus to be on the basic syntax of the language. It is also typically used to ensure that your development environment, compiler, runtime environment, etc. are functioning properly with a minimal example. A basic Hello World! program in Java can be found in Code Sample 22.1.

```
1  package unl.cse;  //package declaration
2
3  //imports would go here
4
5  /**
6   * A basic hello  world program in Java
7   */
8  public class HelloWorld {
9
10   //static main method
11   public static void main(String args[]) {
12     System.out.println("Hello World!");
13   }
14
15 }
```

Code Sample 22.1: Hello World Program in Java

We will not focus on any particular development environment, code editor, or any particular operating system, compiler, or ancillary standards in our presentation. However, as a first step, you should be able to write, compile, and run the above program on the environment you intend to use for the rest of this book. This may require that you download and install a JDK and IDE. Eclipse (`http://eclipse.org/`) is the

industry standard, though IntelliJ (https://www.jetbrains.com/idea/) and NetBeans (https://netbeans.org/) are also popular among many others.

22.2. Basic Elements

Using the Hello World! program as a starting point, we will now examine the basic elements of the Java language.

22.2.1. Basic Syntax Rules

Java's syntax is adopted from C, referred to as "C-style syntax." These elements include the following.

- Java is a statically typed language so variables must be declared along with their types before using them.
- Strings are delimited with double quotes. Single characters, including special escaped characters are delimited by single quotes; `"this is a string"`, and these are characters: `'A'`, `'4'`, `'$'` and `'\n'`
- In addition, Java uses Unicode (UTF-16 encoding) to represent characters. This is fully back-compatible with ASCII, but also allows you to specify Unicode characters using special escape sequences and hexadecimal encodings. For example, `'\u4FFA'` represents a Japanese character:

  俺

  The string `"\u4FFA\u306F \u6700\u9AD8\u3060\u305C\uFF01"` represents the phrase

  俺は 最高だぜ！

- Executable statements are terminated by a semicolon, `;`
- Code blocks are defined by using opening and closing curly brackets, `{ ... }`. Moreover, code blocks can be *nested*: code blocks can be defined within other code blocks.
- Variables are scoped to the code block in which they are declared and are only valid within that code block.
- In general, whitespace between coding elements is ignored.

Though not a syntactic requirement, the proper use of whitespace is important for good, readable code. Code inside code blocks is indented at the same indentation. Nested

code blocks are indented further. Think of a typical table of contents or the outline of a formal paper or essay. Sections and subsections or points and points all follow proper indentation with elements at the same level at the same indentation. This convention is used to organize code and make it more readable.

22.2.2. Program Structure

Classes

In Java, everything is a *class* or belongs to a class. A class is an extensible program or blueprint for creating objects. Objects are an integral part of OOP that we'll explore later. However, to start out our programs will be simple enough that the can be contained in a single class.

To declare a class, you use the following syntax.

```
public class HelloWorld { ... }
```

where the contents of the class are placed between the opening/closing curly brackets. In addition, a class must be placed in a Java source file with the same name. In our previous example, the class must be in a file named `HelloWorld.java`.

When naming classes, the most commonly accepted naming convention is to use uppercase camel casing (also called PascalCase) in which each word in the name is capitalized including the first. Examples: `Employee`, `SavingsAccount`, `ImageFile`, etc. Class names (as well as their source file names) are case sensitive.

Packages

Java code is organized into modules called *packages*. Packages are essentially directories (or folders) which follow a directory tree structure which allows subdirectories and separate directories at the same level. It all starts at the *root* directory called the "default" package.

Within a source file, we declare which package the file belongs to using the keyword `package` followed by a "fully qualified package declaration" which is essentially just the names of the directories that the file is located in, separated by a period. The declaration is terminated by a semicolon. For example, the package declaration,

```
package unl.cse;
```

would indicate that the file belongs in the directory `cse` which is a subdirectory of the directory `unl`. The absence of a package declaration will mean that the file is associated with the default directory.

Packages allow you to organize source files and code functionality. Mathematics related

classes can be placed in one package while image related classes can be placed in another, etc. It also provides separate name spaces for classes. There could be 3 or 4 different classes named `List` for example. They could not be located in the same directory, however: if you wanted to use one which one would you be referring to? By separating them into packages, they can all exist without conflict. When we want to use a particular one, we *import* that class with its fully qualified package name.

Imports

An `import` statement essentially "brings in" another class so that its methods and functionality can be used. For example, there is a class named `Scanner` (located in the package `java.util`) that makes it easy to read input from the standard input. To include it in our program so that we can use its functionality, we would need[1] to import it:

```
import java.util.Scanner;
```

Classes in the package `java.lang` (such as `String` and `Math`) are considered standard and are imported by default without an explicit `import` statement.

You may see some code that uses a *wildcard* like `import java.util.*;` which ends up importing every class in that package. This is generally considered bad practice. In general, code should be intentional and specific, importing every class even if they are not used goes against this principle.

When naming packages, you must follow the general naming rules for identifiers (see below). Package names cannot begin with a number, no whitespace, etc. Moreover, the general convention for package names is to use lowercase underscore casing, `here_is_an_example`. Moreover, packages and subpackages follow the same convention as directories: the top most directory is the most general and subdirectories are more and more specific.

In many of our examples we'll use `unl.cse` (UNL, University of Nebraska–Lincoln; CSE, Department of Computer Science & Engineering) which illustrates this general-to-specific organization.

22.2.3. The `main` Method

Every executable program has to have a beginning: a point at which the program starts to execute. In Java, a class may contain many variables and methods, but a class is only *executable* if it contains a `main` method. When a Java class is compiled and the JVM is started, the JVM loads the class into memory and starts executing code contained in the `main` method.

[1] Strictly, speaking you can still use it without importing it, but you'd need to use a fully qualified path name at declaration/instantiation.

In addition, our `main` method takes an *array* of `String` types which which serve to communicate any command line arguments provided to the program (review Section 2.4.4 for details). The array, `args` stores the arguments as strings. The number of arguments provided can be determined using the `length` property of the array. Specifically, `args.length` is an integer indicating how many arguments were provided. This does *not* include the name of the program (class) itself as that is already be known to the programmer.

To access any one argument, it will be necessary to *index*. The index for the first argument is zero, thus the first argument is `args[0]`, the second is `args[1]`, etc. The last one would be at `args[args.length-1]`.

If a user is expected to provide numbers as input, they'll need to be converted as the `args` array are only `String` types. To convert the arguments you can use parsing methods provided by the `Integer` and `Double` classes. An example:

```
1 //converts the "first" command line argument to an integer
2 int x = Integer.parseInt(args[0]);
3 //converts the "third" command line argument to a double:
4 double y = Double.parseDouble(args[1]);
```

22.2.4. Comments

Comments can be written in a Java program either as a single line using two forward slashes, `//comment` or as a multiline comment using a combination of forward slash and asterisk: `/* comment */`. With a single line comment, everything on the line *after* the forward slashes is ignored. With a multiline comment, everything in between the forward slash/asterisk is ignored. Comments are ultimately ignored by the compiler so the amount of comments do not have an effect on the final executable code. Consider the following example.

```
1 //this is a single line comment
2 int x;  //this is also a single line comment, but after some code
3
4 /*
5   This is a comment that can
6   span multiple lines to format the comment
7   message more clearly
8 */
9 double y;
```

Most code editors and IDEs will present comments in a special color or font to distinguish them from the rest of the code (just as our example above does). Failure to close a multiline comment will likely result in a compiler error but with color-coded comments its easy to see the mistake visually.

Type	Description	Wrapper Class
`byte`	8-bit signed 2s complement integer	`Byte`
`short`	16-bit signed 2s complement integer	`Short`
`int`	32-bit signed 2s complement integer	`Integer`
`long`	64-bit signed 2s complement integer	`Long`
`float`	32-bit IEEE 754 floating point number	`Float`
`double`	64-bit floating point number	`Double`
`boolean`	may be set to `true` or `false`	`Boolean`
`char`	16-bit Unicode (UTF-16) character	`Character`

Table 22.1.: Primitive types in Java

Another common comment style convention is the Javadoc (Java Documentation) style of comments. Javadoc style comments are multiline comments that begin with `/**` (that is, two asterisks). The Javadoc framework allows you to *markup* your comments with tags and links so that documentation can be automatically generated and published. We will sometimes use this style, but we will not cover the details. For documentation, see Oracle's website, `http://www.oracle.com/`.

22.3. Variables

Java has 8 built-in primitive types supporting numbers (integers and floating-point numbers), Booleans, and characters. Table 22.1 contains a complete description of these types. Each of these primitive types also has a corresponding *wrapper* class defined in the `java.lang` package. Wrapper classes provide *object* versions of each of these classes. The object versions have many utility methods that can be used in relation to their type. For example, the aforementioned `Integer.parseInt()` method is part of the `Integer` wrapper class.

The wrapper classes, however, are different. These are objects, so when a reference is declared for them, by default, that reference refers to `null`. The keyword `null` is used to indicate a special memory address that represents "nothing". In fact, the default value for *any* object type is `null`. Care must be taken when mixing primitive types and their wrapper classes (see below) as `null` references may result in a `NullPointerException`. Finally, instances of the wrapper classes are immutable. Once they are created, they cannot be changed. References can be made to reference a different object, but the object's value cannot be changed.

22.3.1. Declaration & Assignment

Java is a statically typed language meaning that all variables must be declared before you can use them or refer to them. In addition, when declaring a variable, you must specify both its type and its identifier. For example:

```
int numUnits;
double costPerUnit;
char firstInitial;
boolean isStudent;
```

Each declaration specifies the variables type followed by the identifier and ending with a semicolon. The identifier rules are fairly standard: a name can consist of lower and uppercase alphabetic characters, numbers, and underscores but may *not* begin with a numeric character. We adopt the modern camelCasing naming convention for variables in our code.

The Java specification provides default values for each of the 8 primitives if they are not immediately assigned a value. For all numeric types, zero is the default value. For the `boolean` type, `false` is the default, and the default `char` value is `\0`, the null-terminating character (zero in the ASCII table).

The assignment operator is a single equal sign, `=` and is a right-to-left assignment. That is, the variable that we wish to assign the value to appears on the left-hand-side while the value (literal, variable or expression) is on the right-hand-size. Using our variables from before, we can assign them values:

```
numUnits = 42;
costPerUnit = 32.79;
firstInitial = 'C';
isStudent = true;
```

For brevity, Java allows you to declare a variable and immediately assign it a value on the same line. So these two code blocks could have been more compactly written as:

```
int numUnits = 42;
double costPerUnit = 32.79;
char firstInitial = 'C';
boolean isStudent = true;
```

As another shorthand, we can declare multiple variables on the same line by delimiting them with a comma. However, they *must* be of the same type. We can also use an assignment with them.

```
int numOrders, numUnits = 42, numCustomers = 10, numItems;
double costPerUnit = 32.79, salesTaxRate;
```

Another convenient keyword is `final`. Though it has several uses, when applied to a variable declaration, it makes it a read-only variable. After a value has been assigned to

a `final` variable, its value cannot be changed.

```
final int secret = 42;
final double salesTaxRate = 0.075;
```

Any attempt to reassign the values of `final` variables will result in a compiler error.

22.4. Operators

Java supports the standard arithmetic operators for addition, subtraction, multiplication, and division using `+`, `-`, `*`, and `/` respectively. Each of these operators is a binary operator that acts on two operands which can either be literals or other variables and follow the usually rules of arithmetic when it comes to order of precedence (multiplication and division before addition and subtraction).

```
int a = 10, b = 20, c = 30, d;
d = a + 5;
d = a + b;
d = a - b;
d = a + b * c;
d = a * b;
d = a / b; //integer division and truncation!  See below

double x = 1.5, y = 3.4, z = 10.5, w;
w = x + 5.0;
w = x + y;
w = x - y;
w = x + y * z;
w = x * y;
w = x / y;

//you can do arithmetic with both types:
w = a + x;

//however you CANNOT assign a double to an integer:
d = b + y; //compilation error
//but you can do so with an explicit type cast:
d = (int) (b + y);  //though truncation occurs, d is 23
```

In addition, you can *mix* the wrapper classes with their primitive types. You must be careful though. The wrapper classes are object references which can be `null`. If a `null` reference is used in an arithmetic expression, it will result in a `NullPointerException` which can be caught and handled. If not caught, it will end up being a fatal error. Some examples:

```
int a = 10, c;
Integer b = 20;

//int and Integer can be mixed:
c = a + b;

double x = 3.14, z;
Double y = 2.71;
//double and Double can be mixed:
z = x + y;

//all types can be mixed:
double u = a + x + b + y;

//Be careful:
Integer d = null;
c = a + d; //NullPointerException
```

This works because of a mechanism called *autoboxing* (or *autounboxing* in this case). The wrapper class is acting like a "box": its an object that stores the value of a primitive type. When it gets used in an arithmetic expression, it gets "unboxed" and converted to a primitive type so that the arithmetic operation is performed on compatible primitive types. This is all done by the compiler and is completely transparent to us. However, that is the reason that we may get a `NullPointerExcpetion`. Our code actually gets converted from `c = a + d;` to `c = a +d.doubleValue();`. The `doubleValue` method returns a `double` primitive value. However, if `d` is `null`, you can't call a method on it; thus the `NullPointerException`.

Special care must be taken when dealing with `int` types. For all four operators, if both operands are integers, the result will be an integer. For addition, subtraction, and multiplication this isn't an issue, but for division it means that when we divide, say `(10 / 20)`, the result is not 0.5 as expected. The number 0.5 is a floating-point number. As such, the fractional part gets truncationtruncated (cut off and thrown out) leaving only zero. In the code above, `d = a / b;` the variable `d` ends up getting the value zero because of this.

A solution to this problem is to use explicit type casting to force at least one of the operands in an integer division to become a `double` type. For example:

```
int a = 10, b = 20;
double x;

x = (double) a / b;
```

Assigning a floating-point number to an integer is not allowed in Java and attempting to do so will be treated as a compiler error. This is because Java does not support *implicit*

type casts. However, you *can* do so if you provide an *explicit* type cast as in the code above,

```
d = (int) (b + y);
```

In this code, `b + y` is correctly computed as $20 + 3.4 = 23.4$, but the explicit type cast (down to an integer) results in truncation. The .4 gets cutoff and `d` gets the value 23.

Assigning an `int` value to a `double` variable is not a problem as the integer 2 becomes the floating-point number 2.0.

Java also supports the integer remainder operator using the `%` symbol. This operator gives the remainder of the result of dividing two integers. Examples:

```
int x;

x = 10 % 5; //x is 0
x = 10 % 3; //x is 1
x = 29 % 5; //x is 4
```

22.5. Basic I/O

Java provides several ways to perform input and output operations on the standard input/output as part of the `System` class. The `System` class contains a standard output stream which can be accessed using `System.out`. It also has a standard input stream which can be accessed using `System.in`.

For output, there are several methods that can be used but we'll focus on `println` and `printf`. The first is for printing strings on a single line. The `ln` at the end indicates that the method will insert an endline character, `\n` for you. The second is a `printf`-style output method that takes placeholders like `%d` and `%f` (review Section 2.4.3 for details).

The easiest way to read input is through the `Scanner` class. You can create a new instance of the scanner class and associate it with the standard input using the following code.

```
Scanner s = new Scanner(System.in);
```

The variable `s` is now active and can be read from. You can get specific values from the `Scanner` by calling various methods such as `s.nextInt()` to get an `int`, `s.nextDouble()` to get a `double`, etc. When these methods are called, the program *blocks* until the user enters her input and presses the enter/return key. The conversion to the type you requested is automatic. A full example is depicted in Code Sample 22.2.

One potential problem with using `Scanner` is that the methods cannot force a user to enter good input. In the example above, if the user, instead of entering a number, entered `"Hello"`, the conversion to a number would fail. This would result in a

```
Scanner s = new Scanner(System.in);
int a;
System.out.println("Please enter a number: ");
a = s.nextInt();
System.out.printf("Great, you entered %d\n", a);
```

Code Sample 22.2: Basic Input/Output in Java

`InputMismatchException`.

22.6. Examples

22.6.1. Converting Units

Let's start with a simple task: let's write a program that will prompt the user to enter a temperature in degrees Fahrenheit and convert it to degrees Celsius using the formula

$$C = (F - 32) \cdot \frac{5}{9}$$

We begin with the basic program outline which will include a package and class declaration. We'll also need to read from the standard input, so we'll import the `Scanner` class. We'll want want our class to be executable, so we need to put a `main` method in our class. Finally, we'll document our program to indicate its purpose.

```
package unl.cse;

import java.util.Scanner;

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */
public class TemperatureConverter {

  public static void main(String args[]) {

    //TODO: implement this

  }
}
```

It is common for programmers to use a comment along with a `TODO` note to themselves as a reminder of things that they still need to do with the program.

Let's first outline the basic steps that our program will go through:

1. We'll first prompt the user for input, asking them for a temperature in Fahrenheit
2. Next we'll read the user's input, likely into a floating-point number as degrees can be fractional
3. Once we have the input, we can calculate the degrees Celsius by using the formula above
4. Lastly, we will want to print the result to the user to inform them of the value

Sometimes its helpful to write an outline of such a program directly in the code using comments to provide a step-by-step process. For example:

```
1  package unl.cse;
2
3  import java.util.Scanner;
4
5  /**
6   * This program converts Fahrenheit temperatures to
7   * Celsius
8   */
9  public class TemperatureConverter {
10
11   public static void main(String args[]) {
12
13     //TODO: implement this
14     //1. Prompt the user for input in Fahrenheit
15     //2. Read the Fahrenheit value from the standard input
16     //3. Compute the degrees Celsius
17     //4. Print the result to the user
18
19   }
20 }
```

As we read each step it becomes apparent that we'll need a couple of variables: one to hold the Fahrenheit (input) value and one for the Celsius (output) value. It also makes sense that each of these should be `double` variables as we want to support fractional values. So at the top of our `main` method, we'll add the variable declarations:

```
double fahrenheit, celsius;
```

We'll also need a scanner, initialized to read from the standard input:

```
Scanner s = new Scanner(System.in);
```

Each of the steps is now straightforward; we'll use a `Sytem.out.println` statement in the first step to prompt the user for input:

```
System.out.println("Please enter degrees in Fahrenheit: ");
```

In the second step, we'll use our `Scanner` to read in a value from the user for the `fahrenheit` variable. Recall that we use the method `s.nextDouble()` to read a `double` value from the user.

```
fahrenheit = s.nextDouble();
```

We can now compute `celsius` using the formula provided:

```
celsius = (fahrenheit - 32) * (5 / 9);
```

Finally, we use `System.out.printf` to output the result to the user:

```
System.out.printf("%f Fahrenheit is %f Celsius\n", fahrenheit, celsius);
```

Try typing and running the program as defined above and you'll find that you don't get correct answers. In fact, you'll find that no matter what values you enter, you get zero. This is because of the calculation using `(5 / 9)`: recall what happens with integer division: truncation! This will *always* end up being zero.

One way we could fix it would be to pull out our calculators and find that $\frac{5}{9} = 0.55555\ldots$ and replace `(5 / 9)` with `0.555555`. But, how many fives? It may be difficult to tell how accurate we can make this floating-point number by hardcoding it ourselves. A much better approach would be to let the compiler take care of the optimal computation for us by making at least one of the numbers a `double` to prevent integer truncation. That is, we should instead use `5.0 / 9`.

The full program can be found in Code Sample 22.3.

```java
package unl.cse;

import java.util.Scanner;

public class TemperatureConverter {

  public static void main(String args[]) {

    double fahrenheit, celsius;
    Scanner s = new Scanner(System.in);

    //1. Prompt the user for input in Fahrenheit
    System.out.println("Please enter degrees in Fahrenheit: ");

    //2. Read the Fahrenheit value from the standard input
    fahrenheit = s.nextDouble();

    //3. Compute the degrees Celsius
    celsius = (fahrenheit - 32) * 5.0 / 9;

    //4. Print the result to the user
    System.out.printf("%f Fahrenheit is %f Celsius\n",
            fahrenheit, celsius);

  }
}
```

Code Sample 22.3: Fahrenheit-to-Celsius Conversion Program in Java

22.6.2. Computing Quadratic Roots

Some programs require the user to enter multiple inputs. The prompt-input process can be repeated. In this example, consider asking the user for the coefficients, a, b, c to a quadratic polynomial,

$$ax^2 + bx + c$$

and computing its roots using the quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

As before, we can create a basic program with a `main` method and start filling in the details. In particular, we'll need to prompt for the input a, then read it in; then prompt for b, read it in and repeat for c. We'll also need several variables: three for the coefficients a, b, c and *two* more; one for each root. Thus, we have

```
double a, b, c, root1, root2;
Scanner s = new Scanner(System.in);

System.out.println("Please enter a: ");
a = s.nextDouble();
System.out.println("Please enter b: ");
b = s.nextDouble();
System.out.println("Please enter c: ");
c = s.nextDouble();
```

Now to compute the roots: we need to take care that we correctly adapt the formula so it accurately reflects the order of operations. We also need to use the standard math library's square root function (unless you want to write your own! Carefully adapting the formula leads to

```
root1 = (-b + Math.sqrt(b*b - 4*a*c) ) / (2*a);
root2 = (-b - Math.sqrt(b*b - 4*a*c) ) / (2*a);
```

Finally, we print the output using `System.out.printf`. The full program can be found in Code Sample 22.4.

This program was interactive. As an alternative, we could have read all three of the inputs as command line arguments, taking care that we need to convert them to floating-point numbers. Lines 16–21 in the program could have been changed to

```
a = Double.parseDouble(args[0]);
b = Double.parseDouble(args[1]);
c = Double.parseDouble(args[2]);
```

Finally, think about the possible inputs a user could provide that may cause problems for this program. For example:

- What if the user entered zero for a?
- What if the user entered some combination such that $b^2 < 4ac$?
- What if the user entered non-numeric values?
- For the command line argument version, what if the user provided less than three argument? Or more?

How might we prevent the consequences of such bad inputs? That is, how might we handle the even that a users enters those bad inputs and how do we communicate these errors to the user? To do so we'll need conditionals.

```java
package unl.cse;

import java.util.Scanner;

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */
public class QuadraticRoots {

  public static void main(String args[]) {

    double a, b, c, root1, root2;
    Scanner s = new Scanner(System.in);

    System.out.println("Please enter a: ");
    a = s.nextDouble();
    System.out.println("Please enter b: ");
    b = s.nextDouble();
    System.out.println("Please enter c: ");
    c = s.nextDouble();

    root1 = (-b + Math.sqrt(b*b - 4*a*c) ) / (2*a);
    root2 = (-b - Math.sqrt(b*b - 4*a*c) ) / (2*a);

    System.out.printf("The roots of %fx^2 + %fx + %f are: \n",
            a, b, c);
    System.out.printf("  root1 = %f\n", root1);
    System.out.printf("  root2 = %f\n", root2);

  }

}
```

Code Sample 22.4: Quadratic Roots Program in Java

23. Conditionals

Java supports the basic if, if-else, and if-else-if conditional structures as well as switch statements. Java has Boolean types and logical statements are built using the standard logical operators for numeric comparisons as well as logical operators such as negations, And, and Orthat can be used with Boolean types.

23.1. Logical Operators

Recall that Java has Boolean types built-in to the language using either the primitive type, `boolean` or its wrapper class `Boolean`. Moreover, the keywords `true` and `false` can be used to assign and check values. Because Java has Boolean types, it does not allow you to mix logical operators with numeric types. That is, code like the following is invalid.

```
1 int a = 10;
2 boolean b = true;
3 boolean result = (a || b); //compilation error
```

The standard numeric comparison operators are also supported. Consider the following code snippet:

```
1 int a = 10;
2 int b = 20;
3 int c = 10;
4 boolean x = true;
5 boolean y = false;
```

The six standard comparison operators are presented in Table 23.1 using these variables as examples. The comparison operators are the same when used with `double` types as well and `int` types and `double` types can be compared with each other without type casting.

Furthermore, because of autoboxing and unboxing, the wrapper classes for numeric types can be compared using the same operators. For example:

Name	Operator Syntax	Examples	Value
Equals	`==`	`a == 10`	*true*
		`b == 10`	*false*
		`a == b`	*false*
		`a == c`	*true*
Not Equals	`!=`	`a != 10`	*false*
		`b != 10`	*true*
		`a != b`	*true*
		`a != c`	*false*
Strictly Less Than	`<`	`a < 15`	*true*
		`a < 5`	*false*
		`a < b`	*true*
		`a < c`	*false*
Less Than Or Equal To	`<=`	`a <= 15`	*true*
		`a <= 5`	*false*
		`a <= b`	*true*
		`a <= c`	*true*
Strictly Greater Than	`>`	`a > 15`	*false*
		`a > 5`	*true*
		`a > b`	*false*
		`a > c`	*false*
Greater Than Or Equal To	`>=`	`a >= 15`	*false*
		`a >= 5`	*true*
		`a >= b`	*false*
		`a >= c`	*true*

Table 23.1.: Comparison Operators in Java

Operator	Operator Syntax	Examples	Values
Negation	!	`!x`	*false*
		`!y`	*true*
And	&&	`x && true`	*true*
		`x && y`	*false*
Or	||	`x || false`	*true*
		`x || y`	*false*

Table 23.2.: Logical Operators in Java

```
1 int a = 10;
2 Integer b = 20;
3 Double x = 3.14;
4 boolean r;
5 r = (a < b);
6 r = (a >= b);
7 r = (x == 2.71);
```

The three basic logical operators are also supported as described in Table 23.2 using the same code snippet variable values as examples.

23.1.1. Order of Precedence

At this point it is worth summarizing the order of precedence of all the operators that we've seen so far including assignment, arithmetic, comparison, and logical. Since all of these operators could be used in one statement, for example,

```
(b*b < 4*a*c || a == 0 || args.length != 4)
```

it is important to understand the order in which each one gets evaluated. Table 23.3 summarizes the order of precedence for the operators seen so far. This is not an exhaustive list of Java operators.

23.1.2. Comparing Strings and Characters

The comparison operators in Table 23.1 can also be used for *single characters* because of the nature of the ASCII text table (see Table 2.4). Each alphanumeric character, including the various symbols and whitespace characters, is associated with an integer 0–127. We can therefore write statements like `('A' < 'a')`, which is *true* since uppercase letters are ordered before lowercase letters in the ASCII table (`'A'` is 65 and `'a'` is 97 and so $65 < 97$ is *true*). Several more examples can be found in Table 23.4.

Numeric comparison operators *cannot* be used to compare strings in Java. For example, we could *not* code something like `("aardvark" < "zebra")`. The Java compiler would

	Operator(s)	Associativity	Notes
Highest	`++`, `--`	left-to-right	postfix increment operators
	`-`, `!`	right-to-left	unary negation operator, logical not
	`*`, `/`, `%`	left-to-right	
	`+`, `-`	left-to-right	addition, subtraction
	`<`, `<=`, `>`, `>=`	left-to-right	comparison
	`==`, `!=`	left-to-right	equality, inequality
	`&&`	left-to-right	logical AND
	`||`	left-to-right	logical OR
Lowest	`=`, `+=`, `-=`, `*=`, `/=`	right-to-left	assignment and compound assignment operators

Table 23.3.: Operator Order of Precedence in Java. Operators on the same level have equivalent order and are performed in the associative order specified.

Comparison Example	Result
`('A' < 'a')`	*true*
`('A' == 'a')`	*false*
`('A' < 'Z')`	*true*
`('0' < '9')`	*true*
`('\n' < 'A')`	*true*
`(' ' < '\n')`	*false*

Table 23.4.: Character comparisons in Java

not allow you to do this because the comparison operator is for numeric types *only*. However, the following code *would* compile and run:

```
1 String s = "aardvark";
2 String t = "zebra";
3 boolean b = (s == t);
```

but it wouldn't necessarily give you what you want. To understand why this is okay, recall that a `String` is an object; the `s` and `t` variables are *references* to that object in memory. When we use the equality comparison, `==` we're asking if `s` and `t` are the *same memory address*. In this case, likely they are not and so the result is `false`. However, similar code, for example,

```
1 String s = new String("liger");
2 String t = new String("liger");
3 boolean b = (s == t);
```

would also result in `false` because `s` and `t` represent different strings in memory, even though they have the same sequence of characters. We'll explore how to properly compare strings later. For now, avoid using the comparison operators with strings.

23.2. If, If-Else, If-Else-If Statements

Conditional statements in Java utilize the keywords `if`, `else`, and `else if`. Conditions are placed inside parentheses immediately after the `if` and `else if` keywords. Examples of all three can be found in Code Sample 23.1.

```
1  //example of an if statement:
2  if(x < 10) {
3    System.out.println("x is less than 10");
4  }
5
6  //example of an if-else statement:
7  if(x < 10) {
8    System.out.println("x is less than 10");
9  } else {
10   System.out.println("x is 10 or more");
11 }
12
13 //example of an if-else-if statement:
14 if(x < 10) {
15   System.out.println("x is less than 10");
16 } else if(x == 10) {
17   System.out.println("x is equal to ten");
18 } else {
19   System.out.println("x is greater than 10");
20 }
```

Code Sample 23.1: Examples of Conditional Statements in Java

Some observations about the syntax: the statement, `if(x < 10)` does not have a semicolon at the end. This is because it is a conditional statement that determines the flow of control and *not* an executable statement. Therefore, no semicolon is used. Suppose we made a mistake and *did* include a semicolon:

```
1  int x = 15;
2  if(x < 10); {
3    System.out.println("x is less than 10");
4  }
```

Some compilers may give a warning, but this is valid Java; it will compile and it will run. However, it will end up printing `x is less than 10`, even though $x = 15$! Recall that a conditional statement *binds* to the executable statement or code block *immediately* following it. In this case, we've provided an *empty* executable statement ended by the semicolon. The code is essentially equivalent to

```
int x = 15;
if(x < 10) {
}
System.out.println("x is less than 10");
```

Which is obviously not what we wanted. The semicolon ended up binding to the empty executable statement, and the code block containing the print statement immediately followed, but was *not* bound to the conditional statement which is why the print statement executed regardless of the value of x.

Another convention that we've used in our code is where we have placed the curly brackets. First, if a conditional statement is bound to only one statement, the curly brackets are not necessary. However, it is best practice to include them even if they are not necessary and we'll follow this convention. Second, the opening curly bracket is on the same line as the conditional statement while the closing curly bracket is indented to the same level as the start of the conditional statement. Moreover, the code inside the code block is indented. If there were more statements in the block, they would have all been at the same indentation level.

23.3. Examples

23.3.1. Computing a Logarithm

The logarithm of x is the exponent that some *base* must be raised to get x. The most common logarithm is the natural logarithm, $\ln(x)$ which is base $e = 2.71828\ldots$. But logarithms can be in any base $b > 1$[1] What if we wanted to compute $\log_2(x)$? Or $\log_\pi(x)$? Let's write a program that will prompt the user for a number x and a base b and computes $\log_b(x)$.

Arbitrary bases can be computed using the change of base formula:

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$$

If we can compute *some* base a, then we can compute any base b. Fortunately we have such a solution. Recall that the standard library provides a function to compute the natural logarithm, `Math.log()`). This is one of the fundamentals of problems solving: if a solution already exists, use it. In this case, a solution exists for a different, but similar problem (computing the natural logarithm), but we can *adapt* the solution using the change of base formula. In particular, if we have variables `b` (base) and `x`, we can compute $\log_b(x)$ using

```
Math.log(x) / Math.log(b)
```

[1] Bases can also be $0 < b < 1$, but we'll restrict our attention to increasing functions only.

But wait: we have a problem similar to the examples in the previous section. The user could enter invalid values such as $b = -10$ or $x = -2.54$ (logarithms are undefined for non-positive values in any base). We want to ensure that $b > 1$ and $x > 0$. With conditionals, we can now do this. Once we have read in the input from the user we can make a check for good input using an `if` statement.

```
if(x <= 0 || b <= 1) {
  System.out.println("Error: bad input!");
  System.exit(1);
}
```

This code has something new: `System.exit(1)`. The `exit` function immediately terminates the program regardless of the rest of the code that may remain. The argument passed to `exit` is an integer that represents an *error code*. The convention is that zero indicates "no error" while non-zero values indicate some error. This is a simple way of performing *error handling*: if the user provides bad input, we inform them and quit the program, forcing them to run it again and provide good input. By prematurely terminating the program we avoid any illegal operation that would give a bad result.

Alternatively, we could have split the conditions into two statements and given a more descriptive error message. We use this design in the full program which can be found in Code Sample 23.2. The program also takes the input as command line arguments. Now that we have conditionals, we can actually check that the correct number of arguments was provided by the user and quit in the event that they don't provide the correct number.

23.3.2. Life & Taxes

Let's adapt the conditional statements we developed in Section 3.6.4 into a full Java program. The first thing we need to do is establish the variables we'll need and read them in from the user. At the same time we can check for bad input (negative values) for both the inputs.

```
Scanner s = new Scanner(System.in);
double income, tax, numChildren, credit;

System.out.println("Please enter your Adjusted Gross Income: ");
income = s.nextDouble();

System.out.println("How many children do you have?");
numChildren = s.nextDouble();

if(income < 0 || numChildren < 0) {
  System.out.println("Invalid inputs");
  System.exit(1);
}
```

Next, we can code a series of if-else-if statements for the income range. By placing the ranges in increasing order, we only need to check the upper bounds just as in the original example.

```
if(income <= 18150) {
  baseTax = income * .10;
} else if(income <= 73800) {
  baseTax = 1815 + (income - 18150) * .15;
} else if(income <= 148850) {
...
} else {
  baseTax = 127962.50 + (income - 457600) * .396;
}
```

Next we compute the child tax credit, taking care that it does not exceed $3,000. A conditional based on the number of children should suffice as at this point in the program we already know it is zero or greater.

```
if(numChildren <= 3) {
  credit = numChildren * 1000;
} else {
  credit = 3000;
}
```

Finally, we need to ensure that the credit does not exceed the total tax liability (the credit is non-refundable, so if the credit is greater, the tax should only be zero, not negative).

```
if(baseTax - credit >= 0) {
  totalTax = baseTax - credit;
} else {
  totalTax = 0;
}
```

The full program is presented in Code Sample 23.3.

23.3.3. Quadratic Roots Revisited

Let's return to the quadratic roots program we previously designed that uses the quadratic equation to compute the roots of a quadratic polynomial by reading coefficients a, b, c in from the user. One of the problems we had previously identified is if the user enters "bad" input: if $a = 0$, we would end up dividing by zero; if $b^2 - 4ac < 0$ then we would have complex roots. With conditionals, we can now check for these issues and exit with an error message.

Another potential case we might want to handle differently is when there is only one distinct root ($b^2 - 4ac = 0$). In that case, the quadratic formula simplifies to $\frac{-b}{2a}$ and we can print a different, more specific message to the user. The full program can be found in Code Sample 23.4.

```java
/**
 * This program computes the logarithm base b (b > 1)
 * of a given number x > 0
 */
public class Logarithm {

  public static void main(String args[]) {

    double b, x, result;
    if(args.length != 2) {
      System.out.println("Usage: b x");
      System.exit(1);
    }

    b = Double.parseDouble(args[0]);
    x = Integer.parseInt(args[1]);

    if(x <= 0) {
      System.out.println("Error: x must be greater than zero");
      System.exit(1);
    }
    if(b <= 1) {
      System.out.println("Error: base must be greater than one");
      System.exit(1);
    }

    result = Math.log(x) / Math.log(b);
    System.out.printf("log_(%f)(%f) = %f\n", b, x, result);

  }

}
```

Code Sample 23.2: Logarithm Calculator Program in Java

```java
import java.util.Scanner;

public class Taxes {

  public static void main(String args[]) {

    Scanner s = new Scanner(System.in);
    double income, baseTax, totalTax, numChildren, credit;

    System.out.println("Please enter your Adjusted Gross Income: ");
    income = s.nextDouble();

    System.out.println("How many children do you have?");
    numChildren = s.nextDouble();

    if(income < 0 || numChildren < 0) {
      System.out.println("Invalid inputs");
      System.exit(1);
    }

    if(income <= 18150) {
      baseTax = income * .10;
    } else if(income <= 73800) {
      baseTax = 1815 + (income  -18150) * .15;
    } else if(income <= 148850) {
      baseTax = 10162.50 + (income - 73800) * .25;
    } else if(income <= 225850) {
      baseTax = 28925.00 + (income - 148850) * .28;
    } else if(income <= 405100) {
      baseTax = 50765.00 + (income - 225850) * .33;
    } else if(income <= 457600) {
      baseTax = 109587.50 + (income - 405100) * .35;
    } else {
      baseTax = 127962.50 + (income - 457600) * .396;
    }

    if(numChildren <= 3) {
      credit = numChildren * 1000;
    } else {
      credit = 3000;
    }

    if(baseTax - credit >= 0) {
      totalTax = baseTax - credit;
    } else {
      totalTax = 0;
    }

    System.out.printf("AGI:           $%10.2f\n", income);
    System.out.printf("Tax:           $%10.2f\n", baseTax);
    System.out.printf("Credit:        $%10.2f\n", credit);
    System.out.printf("Tax Liability: $%10.2f\n", totalTax);

  }

}
```

Code Sample 23.3: Tax Program in Java

```java
/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */
public class Roots {

  public static void main(String args[]) {
    double a, b, c, root1, root2;

    if(args.length != 3) {
      System.err.println("Usage: a b c\n");
      System.exit(1);
    }

    a = Double.parseDouble(args[0]);
    b = Double.parseDouble(args[1]);
    c = Double.parseDouble(args[2]);

    if(a == 0) {
      System.err.println("Error: a cannot be zero");
      System.exit(1);
    } else if(b*b < 4*a*c) {
      System.err.println("Error: cannot handle complex roots\n");
      System.exit(1);
    } else if(b*b == 4*a*c) {
      root1 = -b / (2*a);
      System.out.printf("Only one distinct root: %f\n", root1);
    } else {
      root1 = (-b + Math.sqrt(b*b - 4*a*c) ) / (2*a);
      root2 = (-b - Math.sqrt(b*b - 4*a*c) ) / (2*a);

      System.out.printf("The roots of %fx^2 + %fx + %f are: \n",
              a, b, c);
      System.out.printf("  root1 = %f\n", root1);
      System.out.printf("  root2 = %f\n", root2);
    }
  }

}
```

Code Sample 23.4: Quadratic Roots Program in Java With Error Checking

24. Loops

Java supports while loops, for loops, and do-while loops using the keywords `while`, `for`, and `do` (along with another `while`). Continuation conditions for loops are enclosed in parentheses, `(...)` and the blocks of code associated with the loop are enclosed in curly brackets.

24.1. While Loops

Code Sample 24.1 contains an example of a basic while loop in C. Just as with conditional statements, our code styling places the opening curly bracket on the same line as the `while` keyword and continuation condition. The inner block of code is also indented and all lines in the block are indented to the same level.

```
1 int i = 1; //Initialization
2 while(i <= 10) {  //continuation condition
3   //perform some action
4   i++; //iteration
5 }
```

Code Sample 24.1: While Loop in Java

In addition, the continuation condition does *not* contain a semicolon since it is not an executable statement. Just as with an if-statement, if we *had* placed a semicolon it would have led to unintended results. Consider the following:

```
1 while(i <= 10); {
2   //perform some action
3   i++; //iteration
4 }
```

A similar problem occurs: the `while` keyword and continuation condition bind to the next executable statement or code block. As a consequence of the semicolon, the executable statement that gets bound to the while loop is *empty*. What happens is even worse: the program will enter an infinite loop. To see this, the code is essentially equivalent to the following:

```
while(i <= 10) {
}
{
  //perform some action
  i++; //iteration
}
```

In the while loop, we never increment the counter variable `i`, the loop does nothing, and so the computation will continue on forever! Some compilers will warn you about this, others will not. It is valid Java and it will compile and run, but obviously won't work as intended. Avoid this problem by using proper syntax.

Another common use case for a while loop is a flag-controlled loop in which we use a Boolean flag rather than an expression to determine if a loop should continue or not. An example can be found in Code Sample 24.2.

```
int i = 1;
boolean flag = true;
while(flag) {
  //perform some action
  i++; //iteration
  if(i>10) {
    flag = false;
  }
}
```

Code Sample 24.2: Flag-controlled While Loop in Java

24.2. For Loops

For loops in Java use the familiar syntax of placing the initialization, continuation condition, and iteration on the same line as the keyword `for`. An example can be found in Code Sample 24.3.

```
for(int i=1; i<=10; i++) {
  //perform some action
}
```

Code Sample 24.3: For Loop in Java

Again, note the syntax: semicolons are placed at the end of the initialization and continuation condition, but *not* the iteration statement. Just as with while loops, the

opening curly bracket is placed on the same line as the `for` keyword. Code within the loop body is indented, all at the same indentation level.

Another observation: the declaration of the counter variable `i` was done in the initialization statement. This scopes the variable to the loop itself. The variable `i` is valid inside the loop body, but will be out-of-scope *after* the loop body. It is possible to declare the variable prior to the loop, but the variable `i` would have a much larger scope. It is best practice to limit the scope of variables only to where they are needed. Thus, we will write our loops as above.

24.3. Do-While Loops

Finally, Java does support do-while loops. Recall that the difference between a while loop and a do-while loop is when the continuation condition is checked. For a while loop it is *prior* to the beginning of the loop body and in a do-while loop it is at the *end* of the loop. This means that a do-while always executes *at least once*. An example can be found in Code Sample 24.4.

```
1 int i;
2 do {
3   //perform some action
4   i++;
5 } while(i <= 10);
```

Code Sample 24.4: Do-While Loop in Java

Note the syntax and style: the opening curly bracket is again on the same line as the keyword `do`. The `while` keyword and continuation condition are on the same line as the closing curly bracket. In a slight departure from consistent syntax, a semicolon *does* appear at the end of the continuation condition even though it is not an executable statement.

24.4. Enhanced For Loops

Java also supports foreach loops (which were introduced in JDK 1.5.0) which Java refers to as "Enhanced For Loops". Foreach loops allow you to iterate over each element in a collection without having to define an index variable or otherwise "get" each element. We'll revisit these concepts in detail in Chapter 27, but let's take a look at a couple of examples.

An enhanced for loop in Java still uses the keyword `for` but uses different syntax for its control. The example in Code Sample 24.5 illustrates this syntax: `(int a : arr)`. The

last element of this syntax is a reference to the collection that we want to iterate over. The first part is the *type* and local reference variable that the loop will use.

```
int arr[] = {10, 20, 8, 42};
int sum = 0;
for(int a : arr) {
  sum += a;
}
```

Code Sample 24.5: Enhanced For Loops in Java Example 1

The code `(int a : arr)` should be read as "for each integer element `a` in the collection `arr`..." Within the enhanced for loop, the variable `a` will be automatically updated for you on each iteration. Outside the loop body, the variable `a` is out-of-scope.

Java allows you to use an enhanced for loop with any array or collection (technically, anything that implements the `Iterable` interface). One example is a `List`, an ordered collection of elements. Code Sample 24.6 contains an example.

```
List<Integer> list = Arrays.asList(10, 20, 8, 42);
int sum = 0;
for(Integer a : list) {
  sum += a;
}
```

Code Sample 24.6: Enhanced For Loops in Java Example 2

24.5. Examples

24.5.1. Normalizing a Number

Let's revisit the example from Section 4.1.1 in which we *normalize* a number by continually dividing it by 10 until it is less than 10. The code in Code Sample 24.7 specifically refers to the value 32145.234 but would work equally well with any value of `x`.

```
double x = 32145.234;
int k = 0;
while(x > 10) {
  x = x / 10; //or: x /= 10;
  k++;
}
```

Code Sample 24.7: Normalizing a Number with a While Loop in Java

24.5.2. Summation

Let's revisit the example from Section 4.2.1 in which we computed the sum of integers $1 + 2 + \cdots + 10$. The code is presented in Code Sample 24.8

```
int i;
int sum = 0;
for(i=1; i<=10; i++) {
  sum += i;
}
```

Code Sample 24.8: Summation of Numbers using a For Loop in Java

Of course we could easily have generalized the code somewhat. Instead of computing a sum up to a particular number, we could have written it to sum up to another variable `n`, in which case the for loop would instead look like the following.

```
for(i=1; i<=n; i++) {
  sum += i;
}
```

24.5.3. Nested Loops

Recall that you can write loops within loops. The inner loop will execute fully *for each* iteration of the outer loop. An example of two nested of loops in Java can be found in Code Sample 24.9.

```
int i, j;
int n = 10;
int m = 20;
for(i=0; i<n; i++) {
  for(j=0; j<m; j++) {
    System.out.printf("(i, j) = (%d, %d)\n", i, j);
  }
}
```

Code Sample 24.9: Nested For Loops in Java

The inner loop execute for $j = 0, 1, 2, \ldots, 19 < m = 20$ for a total of 20 times. However, it executes 20 times *for each* iteration of the outer loop. Since the outer loop execute for $i = 0, 1, 2, \ldots, 9 < n = 10$, the total number of times the `System.out.printf` statement execute is $10 \times 20 = 200$. In this example, the sequence $(0, 0), (0, 1), (0, 2), \ldots, (0, 19), (1, 0), \ldots, (9, 19)$ will be printed.

24.5.4. Paying the Piper

Let's adapt the solution for the loan amortization schedule we developed in Section 4.7.3. First, we'll read the principle, terms, and interest as command line inputs.

Adapting the formula for the monthly payment and using the math library's `Math.pow()` function, we get

```
1 double monthlyPayment = (monthlyInterestRate * principle) /
2   (1 - pow( (1 + monthlyInterestRate), -n));
```

However, recall that we may have problems due to accuracy. The monthly payment could come out to be a fraction of a cent, say $43.871. For accuracy, we need to ensure that all of the figures for currency are rounded to the nearest cent. The standard math library does have a `Math.round()` function, but it only rounds to the nearest whole number, not the nearest 100th.

However, we can *adapt* the "off-the-shelf" solution to fit our needs. If we take the number, multiply it by 100, we get (say) 4387.1 which we can now round to the nearest whole number, giving us 4387. We can then divide by 100 to get a number that has been rounded to the nearest 100th! In Java, we could simply do the following.

```
monthlyPayment = Math.round(monthlyPayment * 100.0) / 100.0;
```

We can use the same trick to round the monthly interest payment and any other number expected to be whole cents. To output our numbers, we use `System.out.printf` and take care to align our columns to make make it look nice. To finish our adaptation, we handle the final month separately to account for an over/under payment due to rounding. The full solution can be found in Code Sample 24.10.

```java
public class LoanAmortization {

  public static void main(String args[]) {

    if(args.length != 4) {
      System.err.println("Usage: principle apr terms");
      System.exit(1);
    }

    double principle = Double.parseDouble(args[0]);
    double apr = Double.parseDouble(args[1]);
    int n = Integer.parseInt(args[2]);

    double balance = principle;
    double monthlyInterestRate = apr / 12.0;

    //monthly payment
    double monthlyPayment = (monthlyInterestRate * principle) /
      (1 - Math.pow( (1 + monthlyInterestRate), -n));
    //round to the nearest cent
    monthlyPayment = Math.round(monthlyPayment * 100.0) / 100.0;

    System.out.printf("Principle: $%.2f\n", principle);
    System.out.printf("APR: %.4f%%\n", apr*100.0);
    System.out.printf("Months: %d\n", n);
    System.out.printf("Monthly Payment: $%.2f\n", monthlyPayment);

    //for the first n-1 payments in a loop:
    for(int i=1; i<n; i++) {
      //  compute the monthly interest, rounded:
      double monthlyInterest =
        Math.round( (balance * monthlyInterestRate) * 100.0) / 100.0;
      //  compute the monthly principle payment
      double monthlyPrinciplePayment = monthlyPayment - monthlyInterest;
      //  update the balance
      balance = balance - monthlyPrinciplePayment;
      //  print i, monthly interest, monthly principle, new balance
      System.out.printf("%d\t$%10.2f  $%10.2f  $%10.2f\n", i, monthlyInterest,
        monthlyPrinciplePayment, balance);
    }

    //handle the last month and last payment separately
    double lastInterest = Math.round(
            (balance * monthlyInterestRate) * 100.0) / 100.0;
    double lastPayment = balance + lastInterest;

    System.out.printf("Last payment = $%.2f\n", lastPayment);

  }

}
```

Code Sample 24.10: Loan Amortization Program in Java

25. Methods

As an object-oriented programming language, functions in Java are usually referred to as *methods* and are essential to writing programs. The distinction is that a function is usually a standalone element while methods are functions that are members of a class. In Java, since everything is a class or belongs to a class, standalone functions cannot be defined.

In Java you can define your own methods, but they need to be placed within a class. Usually methods that act on data in the class (or instances of the class, see Chapter **??**) or have common functionality are placed into one class. For example, all the basic math methods are all part of the `java.lang.Math` class. It is not uncommon to place similar methods together into one "utility" class.

Java supports method overloading, so within the same class you can define multiple methods with the same name as long as they differ in either the number of type of parameters. For example, in the `java.lang.Math` class, there are 3 versions of the absolute value method, `abs()`, one that takes/returns an `int`, one that takes/returns a `double` and one for `float` types. Naming conflicts can easily be solved by ensuring that you place your methods in a class/package that is unique to your application.

In Java, the 8 primitive types (`int`, `double`, `char`, `boolean`, etc.) are passed by value. All object types, however, such as the wrapper classes `Integer`, `Double` as well as `String`, etc. are passed by reference. That is, the memory address in the JVM is passed to the method. This is done for efficiency, for objects that are "large" it would be inefficient to copy the entire object into the call stack in order to pass it to a method.

However, though object types are passed by reference, the method cannot necessarily change them. Recall that the wrapper classes `Integer`, `Double` and the `String` class are all immutable, meaning that once created they cannot be modified. Thus, even though they are passed by reference, the method that receives them cannot change them.

There are many *mutable* objects in Java. The `StringBuilder` class for example is a mutable object. If you pass a `StringBuilder` instance to a method, that method is free to invoke mutator methods (any methods that *change* the object's state). Since it is the *same* object as in the calling method, the calling method can "see" those changes.

As of Java 5, you can write and use vararg methods. The `System.out.printf()` method is a prime example of this. However, we will not discuss in detail how to do this. Instead, refer to standard Java documentation. Finally, parameters are not optional in Java. This is because Java supports method overloading. You can write multiple versions of

the same method that each take a different number of arguments. You can even design them so that the more specific versions (with fewer arguments) invoke the more general versions (with more arguments), passing in sensible "defaults." when doing so.

25.1. Defining Methods

Defining methods is fairly straightforward. First you create class to place them in. Then you provide the method signature along with the body of the method. In addition, there are several *modifiers* that you can place in the method signature to specify its *visibility* and whether or not the method "belongs" to the class or to instances of the class. This is a concept we'll explore in Chapter **??**. For now, we'll only focus on what is needed to get started.

Typically, the documentation for methods is included with the method definition using "Javadoc" style comments. Consider the following examples.

```java
/**
 * Computes the sum of the two arguments.
 * @param a
 * @param b
 * @return the sum, <code>a + b</code>
 */
public static int sum(int a, int b) {
  return (a + b);
}

/**
 * Computes the Euclidean distance between the 2-D points,
 * (x1,y1) and (x2,y2).
 * @param x1
 * @param y1
 * @param x2
 * @param y2
 * @return
 */
public static double getDistance(double x1, double y1,
                         double x2, double y2) {
  double xDiff = (x1-x2);
  double yDiff = (y1-y2);
  return Math.sqrt( xDiff * xDiff + yDiff * yDiff);
}

/**
 * Computes a monthly payment for a loan with the given
 * principle at the given APR (annual percentage rate) which
 * is to be repaid over the given number of terms.
 * @param principle - the amount borrowed
 * @param apr - the annual percentage rate
 * @param terms - number of terms (usually months)
 * @return
 */
public static double getMonthlyPayment(double principle,
                           double apr, int terms) {
  double rate = (apr / 12.0);
  double payment = (principle * rate) / (1-Math.pow(1+rate, -terms));
  return payment;
}
```

In each of the examples above, the first modifier keyword we used was `public`. This makes the method visible to all other parts of the code base. Any other piece of code

can invoke the method and take advantage of the functionality it provides. Alternatively, we could have used the keywords `private`, to make it only visible to other methods in the same class, `protected` or "package protected" by omitting the modifier altogether. We'll discuss these in detail later on. We'll mostly want our methods to be available, so we'll make most of them `public`.

The second modifier is `static` which makes it so that the method belongs to the class itself rather than instances of the class. We'll discuss objects and instances in detail later on. For now, we'll simply make all of our methods `static`.

After the modifiers, we provide the method signature including the return type, its identifier (name), and its parameter list. Method names must follow the same naming rules as variables: they must begin with an alphabetic character and may contain alphanumeric characters as well as underscores. However, using modern coding conventions we usually name methods using lower camel casing.

Immediately after the signature we provide a method body which contains the code that will be run upon invocation of the method. The method body is enclosed using opening/closing curly brackets.

25.1.1. Void Methods

The keyword `void` can be used in Java to indicate a method does *not* return a value, in which case it is called a "void method." Though it is not necessary, it is still good practice to include a `return` statement.

```
1 public static void printCopyright()
2   System.out.println("(c) Bourke 2015");
3 }
```

In the example above, we've also illustrated how to define a method that has no inputs.

25.1.2. Using Methods

Once a method has been defined in a class, you can make use of the method as follows. First, you may need to import the class itself depending on where it is. For example, suppose that the examples we've presented so far are contained in a class named `Utils` (short for "utilities") which is in a package named `unl.cse`. Then in the class in which we want to call some of these functions we would import it using

```
import unl.cse.Utils;
```

prior to the class declaration. Once the class has been imported, we can invoke a method in the class by first referencing the class and using the *dot operator* to access one of its methods. For example,

```
int a = 10, b = 20;
int c = Utils.sum(a, b); //c contains the value 30

//invoke a method with literal values:
double dist = Utils.getDistance(0.0, 0.0, 10.0, 20.0);

//invoke a method with a combination:
double p = 1500.0;
double r = 0.05;
double monthlyPayment = Utils.getMonthlyPayment(p, r, 60);
```

The `Utils.methodName()` syntax is used because the methods are `static`–they belong to the class and so must be invoked *through* the class using the class's name. We've previously seen this syntax when using `System.` or `Math.` with the standard JDK library functions.

25.1.3. Passing By Reference

Java does not allow you the ability to specify if a variable is passed by reference or by value. Instead, all primitive types are passed by value while all object types are passed by reference. Moreover, most of the built-in types such as `Integer` and `String` are immutable, even though they are passed by reference, any method that receives them cannot change them. Only if the passed object is *mutable* can the method make changes to it (by invoking its methods).

As an example, consider the following piece of code. The `StringBuilder` class is a mutable string object. You can *change* the string contents stored in a `StringBuilder` by calling one of its many methods such as `append()`, which will add whatever string you give it to the end.

In the main method, we create two objects, a `String` and a `StringBuilder` and pass it to a method that makes changes to both by appending `" world!"` to them. Understand what happens here though. The first line in `change()` actually creates a *new* string and then changes what the parameter variable `s` references. The reference to the original string, `"Hello"` is lost and replaced with the new string. In contrast, the `StringBuilder` instance is actually changed via its `append()` method but is *still the same* object.

```
1  public class Mutability {
2
3    public static void change(String s, StringBuilder sb) {
4      s = s + " world!";
5      sb.append(" world!");
6
7      System.out.println("change: s  = " + s);
8      System.out.println("change: sb = " + sb);
9    }
10
11   public static void main(String args[]) {
12     String a = "Hello";
13     StringBuilder b = new StringBuilder("Hello");
14
15     System.out.println("main: s = " + a);
16     System.out.println("main: b = " + b);
17
18     change(a, b);
19
20     System.out.println("main after: s = " + a);
21     System.out.println("main after: b = " + b);
22
23   }
24 }
```

To see this, observe the following output. When we return to the main method, the original string `s` is unchanged (since it was immutable). However, the `StringBuilder` has been changed by the method.

TODO: figure?

```
main: s = Hello
main: b = Hello
change: s  = Hello world!
change: sb = Hello world!
main after: s = Hello
main after: b = Hello world!
```

25.2. Examples

25.2.1. Generalized Rounding

Recall that the standard math library provides a `Math.round()` method that rounds a number to the nearest whole number. Often, we've had need to round to cents as well.

We now have the ability to write a method to do this for us. Before we do, however, let's think more generally. What if we wanted to round to the nearest tenth? Or what if we wanted to round to the nearest 10s or 100s place? Let's write a general purpose rounding method that allows us to specify *which* decimal place to round to.

The most natural input values would be to specify the place using an integer exponent. That is, if we wanted to round to the nearest tenth, then we would pass it -1 as $0.1 = 10^{-1}$, -2 if we wanted to round to the nearest 100th, etc. On the positive end passing in 0 would correspond to the usual round function, 1 to the nearest 10s spot, and so on.

Moreover, we could demonstrate good code reuse (as well as procedural abstraction) by *scaling* the input value and reusing the functionality already provided in the math library's `Math.round()` method. We could further define a `roundToCents()` method that used our generalized round method. Finally, we could place all of these methods into `RoundUtils` Java class for good organization.

```java
package unl.cse;

/**
 * A collection of rounding utilities
 *
 */
public class RoundUtils {

  /**
   * Rounds to the nearest digit specified by the place
   * argument.  In particular to the (10^place)-th digit
   *
   * @param x the number to be rounded
   * @param place the place to be rounded to
   * @return
   */
  public static double roundToPlace(double x, int place) {
    double scale = Math.pow(10, -place);
    double rounded = Math.round(x * scale) / scale;
    return rounded;
  }

  /**
   * Rounds to the nearest cent (100th place)
   *
   * @param x
   * @return
   */
  public static double roundToCents(double x) {
    return RoundUtils.roundToPlace(x, -2);
  }

}
```

Observe that this class does not contain a `main()` method. That means that this class is *not* executable itself. It only provides functionality to other classes in the code base.

26. Error Handling & Exceptions

Java supports error handling through the use of exceptions. Java has many different predefined types of exceptions that you can freely use in your own code. It also allows you to define your own exception types by creating new classes that *inherit* from the predefined classes. Java uses the standard `try-catch-finally` control structure to handle exceptions and allows you to `throw` your own exceptions.

26.1. Exceptions

Java defines a base class named `Throwable` that an object type that can be thrown using the keyword `throw`. There are two majors subtypes of `Throwable`: `Error` and `Exception`. The `Error` class is used primarily for *fatal* errors such as the JVM running out of memory or some other extreme case that your code cannot reasonably be expected to recover from.

There are dozens of types of exceptions that are subclasses of the standard Java `Exception` class defined by the JDK including `IOException` (and its subclasses such as `FileNotFoundException`) or `SQLException` (when working with Structured Query Language (SQL) databases).

An important subclass of `Exception` is `RuntimeException` which represent *unchecked* exceptions that do *not* need to be explicitly caught (see Section 26.1.4 below for further details). We'll mostly focus on this type of exception.

26.1.1. Catching Exceptions

To catch an exception in Java you can use the standard `try-catch` control block (and optionally use the `finally` block to clean up any resources). Let's take, for example, the simple task of reading input from a user using `Scanner` and manually parsing its value into an integer. If the user enters a non-numeric value, parsing will fail and result in a `NumberFormatException` that we can then catch and handle. For example,

```
Scanner s = new Scanner(System.in);
int n = 0;

try {
  String input = s.next();
  n = Integer.parseInt(input);
} catch (NumberFormatException nfe) {
  System.err.println("You entered invalid data!");
  System.exit(1);
}
```

In this example, we've simply displayed an error message to the standard error output and exited the program. That is, we've made the design decision that this error should be fatal. We could have chosen to handle this error differently in the `catch` block.

The code above could have resulted in other exceptions. For example if the `Scanner` failed to read the next token from the standard input, it would have thrown a `NoSuchElementException`. We can add as many `catch` blocks as we want to handle each exception differently.

```
Scanner s = new Scanner(System.in);
int n = 0;

try {
  String input = s.next();
  n = Integer.parseInt(input);
} catch (NumberFormatException nfe) {
  System.err.println("You entered invalid data!");
  System.exit(1);
} catch (NoSuchElementException nsee) {
  System.err.println("Input reading failed, using default...");
  n = 20; //a default value
} catch(Exception e) {
  System.err.println("A general exception occurred");
  e.printStackTrace();
  System.exit(1);
}
```

Each `catch` block catches a different type of exception. Thus, the name of the variable that holds each exception must be different in the chain of `catch` blocks; `nfe`, `nsee`, `e`.

Note that the last `catch` block was written to catch a generic `Exception`. This last block will essentially catch any other type of exception. Much like an `if-else-if` statement, the first type of exception that is caught is the block that will be executed and they are all mutually exclusive. Thus, a "catch all' block like this should always be the last `catch` block. The most specific types of exceptions should be caught first and the most general types should be caught last.

26.1.2. Throwing Exceptions

We can also manually `throw` an exception if we need to. For example, we can `throw` a generic `RuntimeException` using the following.

```
throw new RuntimeException("Something went wrong");
```

By using a generic `RuntimeException`, we can only attach a message to the exception (which can be printed by code that catches the exception). If we want more fine-grained control over the type of exceptions, we need to define our own exceptions.

26.1.3. Creating Custom Exceptions

To create your own exceptions, you need to create a new class to represent the exception and make it *extends* `RuntimeException` by using the keyword `extends`. This makes your exception a subclass or *subtype* of the `RuntimeException`. This is a concept known as *inheritance* in OOP.

Consider the example in the previous chapter of computing the roots of a quadratic polynomial. One possible error situation is when the roots are complex numbers. We could define a new Java class as follows.

```
public class ComplexRootException extends RuntimeException {

  /**
   * Constructor that takes an error message
   */
  public ComplexRootException(String errorMessage) {
    super(errorMessage);
  }
}
```

Now in our code we can catch and even throw this new type of exception.

```
//throw this exception:
if( b*b - 4*a*c < 0) {
  throw new ComplexRootException("Cannot Handle complex roots");
}
```

```
try {
  r1 = getComplexRoot01(a, b, c);
} catch(ComplexRootException cre) {
  //handle the exception here...
}
```

26.1.4. Checked Exceptions

A *checked* exception in Java is an exception that *must* be explicitly caught and handled. For example, the generic `Exception` class is a checked exception (others include `IOException`, `SQLException`, etc.). If a checked exception is thrown within a block of code such as a method then it must either be caught and handled within that block of code or the (say) method must be specified to explicitly throw the exception. For example, the method

```
1 public static void processFile() {
2   Scanner s = new Scanner(new File("data.csv"));
3 }
```

would not actually compile because `Scanner` throws a checked `FileNotFoundException`. Either we would have to explicitly `catch` the exception:

```
1 public static void processFile() {
2   Scanner s = null;
3   try {
4     s = new Scanner(new File("data.csv"));
5   } catch (FileNotFoundException e) {
6     //handle the exception here
7   }
8 }
```

or we would need to specify that the method `processFile()` explicitly `throws` the exception:

```
1 public static void processFile() throws FileNotFoundException {
2   Scanner s = new Scanner(new File("data.csv"));
3 }
```

Doing this, however, would force any code that called the `processFile()` method to surround it in a `try-catch` block and explicitly handle it (or once again, throw it back to the calling method).

The point of a checked exception is to force code to deal with potential issues that can be reasonably anticipated (such as the unavailability of a file). However, from another point of view checked exceptions represent the exact opposite goal of error handling. Namely, that a function or code block can and should *inform* the calling function that an error has occurred, but *not* explicitly make a decision on how to handle the error. A checked exception doesn't make the full decision for the calling function, but it does eliminate *ignoring* the error as an option from the calling function.

Java also supports *unchecked* exceptions which do *not* need to be explicitly caught. For example, `NumberFormatException` or `NullPointerException` are unchecked exceptions. If an unchecked exception is thrown and not caught, it bubbles up through the call stack until some piece of code does catch it. If no code catches it, it results in a fatal error and

terminates the execution of the JVM.

The `RuntimeException` class and any of its subclasses are unchecked exceptions. In our `ComplexRootException` example above, because we extended `RuntimeException` we made it an unchecked exception, allowing the calling function to decide not only *how* to handle it, but also whether or not to handle it *at all*. If we had instead decided to extend `Exception` we would have made our exception a checked exception.

There is considerable debate as to whether or not checked exceptions are a good thing (and as to whether or not unchecked exceptions are a good thing). Many feel (the author included) that checked exceptions were a mistake and their usage should be avoided. The rationale behind checked exceptions is summed up in the following quote from the Java documentation [7].

> Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception

The problem is that the JDK's own design violates this principle. For example, `FileNotFound` is a checked exception; the reasoning being that a program could *re-prompt* the user for a different file. The problem is the assumption that the program we are writing is always interactive. In fact most software is *not* interactive and is instead designed to interact with other software. Reprompting is *not* an option in the vast majority of cases.

As another example, consider Java's SQL library which allows you to programmatically connect to an SQL database. Nearly every method in the API explicitly throws a checked `SQLException`. It stretches the imagination to understand how our code or even a user would be able to recover (programmatically at least) from a lost internet connection or a bad password, etc.

In general and in the opinion of this author, you should use unchecked exceptions.

26.2. Enumerated Types

TODO: move this to a miscellaneous section?

Java allows you to define a special class called an enumerated type which allow you to define a fixed list of possible values. For example, the days of the week or months of the year are possible values used in a date. However, they are more-or-less fixed (no one will be adding a new day of the week or month any time soon). An enumerated class allows us to define these values using human-readable keywords.

To create an enumerated type class we use the keywords `enum` (short for enumeration). Since an enumerated type is a class, it must follow the same rules. It must be in a `.java` source file of the same name. Inside the class we provide a comma-delimited list

of keywords to define our enumeration. Consider the following example.

```
public enum Day {
  SUNDAY,
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY;
}
```

In the example, since the name of the enumeration is `Day` this declaration must be in a source file named `Day.java`. We can now declare variables of this type. The possible *values* it can take are restricted to `SUNDAY`, `MONDAY`, etc. and we can use these keywords in our program. However these values belong to the class `Day` and must be accessed statically. For example,

```
Day today = Day.MONDAY;

if(today == Day.SUNDAY || today == Day.SATURDAY) {
  System.out.println("Its the weekend!");
}
```

Note the naming conventions: the name of the enumerated type follows the same upper camel casing that all classes use while the enumerated values follow an upper case underscore convention. Though our example does not contain a value with multiple words, if it had, we would have used an underscore to separate them.

Using an enumerated type has two advantages. First, it enforces that only a particular set of predefined values can be used. You would not be able to assign a value to a `Day` variable that was not already defined by the `Day` enumeration.

Second, without an enumerated type we'd be forced to use a collection of magic numbers to indicate values. Even for something as simple as the days of the week we'd be constantly trying to remember: which day is Wednesday again? I forget, do our weeks start with Monday or Sunday? Etc. By using an enumerated type these questions are mostly moot as we can use the more human-readable keywords and eliminate the guess work.

26.2.1. More Tricks

Every `enum` type has some additional built-in methods that you can use. For example, there is a `values()` method that can be called on the `enum` that returns an array of the `enum`'s values. You can use an enhanced for loop to iterate over them, for example,

```
for(Day d : Day.values() {
  System.out.println(d.name());
}
```

In the example above, we used another feature: each `enum` value has a `name()` method that returns the value as a `String`. This example would end up printing the following.

```
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
```

Of course, we may want more human-oriented representations. To do this we could override the class's `toString()` method to return a better string representation. For example:

```
public String toString() {
  if(this == SUNDAY) {
    return "Sunday";
  } else if(this == MONDAY) {
    return "Monday";
  }
  ...
  } else {
    return "Saturday";
  }
}
```

Because `enum` types are full classes in Java, many more tricks can be used that leverage the power of classes including using additional state and constructors. We will cover these topics later.

27. Arrays

TODO

28. Strings

TODO

Part III.

The PHP Programming Language

29. Basics

Back in the mid-1990s the world-wide web was in its infancy but becoming more and more popular. For the most part, web pages contained static content: articles and text that was "just-there." Web pages were far from the fully interactive and dynamic applications that they've become. Rasmus Lerdorf had a home page containing his resume and he wanted to track how many visitors were coming to his page. With purely static pages, this was not possible. So, in 1994 he developed PHP/FI which stood for Personal Home Page tools and Forms Interpreter. This was a series of binary tools written in C that operated through a web server's Common Gateway Interface. When a user visited a webpage, instead of just retrieving static content, a script was invoked: it could do a number of operations and send back HyperText Markup Language (HTML) formatted as a response. This made web pages much more dynamic: by serving it through a script, a counter could be maintained that tracked how many people had visited the site. Lerdorf eventually released his source code in 1995 and it became widely used.

Today, PHP is used in a substantial number of pages on the web and is used in many Content Management System (CMS) applications such as Drupal and WordPress. Because of its history, much of the syntax and aspects of PHP (now referred to as "PHP: Hypertext Preprocessor") are influenced or inspired by C. In fact, many of the standard functions available in the language are directly from the C language.

As a scripting language, PHP is not compiled: instead, PHP code is *interpreted* by a PHP interpreter. Though there are several interpreters available, the de facto PHP interpreter is the Zend Engine, a free and open source project that is widely available on many platforms, operating systems, and web servers.

Because it was originally intended to serve web pages, PHP code can be *interleaved* with static HTML tags. This allows PHP code to be embedded in web pages and dynamically interpreted/rendered when a user requests a webpage through a web browser. Though rendering web pages is its primary purpose, PHP can be used as a general scripting language from the command line (which is how we'll present it here).

29.1. Getting Started: Hello World

The hallmark of the introduction of programming languages is the *Hello World!* program. It consists of a simple program whose only purpose is to print out the message "Hello World!" to the user in some manner. The simplicity of the program allows the focus

to be on the basic syntax of the language. It is also typically used to ensure that your development environment, compiler, runtime environment, etc. are functioning properly with a minimal example. A basic Hello World! program in PHP can be found in Code Sample 29.1. A version in which the PHP code is interleaved in HTML is presented in Code Sample 29.2.

```
1 <?php
2
3   printf("Hello World\n");
4
5 ?>
```

Code Sample 29.1: Hello World Program in PHP

```
1  <html>
2    <head>
3      <title>Hello World PHP Page</title>
4    </head>
5    <body>
6      <h1>A Simple PHP Script</h1>
7
8      <?php printf("<p>Hello World</p>");  ?>
9
10   </body>
11 </html>
```

Code Sample 29.2: Hello World Program in PHP with HTML

We will not focus on any particular development environment, code editor, or any particular operating system, compiler, or ancillary standards in our presentation. However, as a first step, you should be able to write and run the above program on the environment you intend to use for the rest of this book. This may require that you download and install a basic PHP interpreter/development environment (such as a standard LAMP, WAMP or MAMP technology stack) or a full IDE (such as Eclipse, PhpStorm, etc.).

29.2. Basic Elements

Using the Hello World! program as a starting point, we will now examine the basic elements of the PHP language.

29.2.1. Basic Syntax Rules

PHP has adopted many aspects of the C programming language (the interpreter is written in C). However, there are some major aspects in which it differs from C. The major aspects and syntactic elements of PHP include the following.

- PHP is an interpreted language: it is not compiled. Instead it is interpreted through an interpreter that parses commands in a script file line-by-line. Any syntax errors, therefore, become runtime errors.
- PHP is *dynamically typed*: you do *not* declare variables or specify a variable's type. Instead, when you assign a variable a value, the variable's type dynamically changes to accommodate the assigned value. Variable names *always* begin with a dollar sign, `$`.
- Strings and characters are essentially the same thing (characters are strings of length 1). Strings and characters can be delimited by *either* single quotes *or* double quotes. `"this is a string"`, `'this is also a string'`; `'A'` and `"a"` are both single-character strings.
- Executable statements are terminated by a semicolon, `;`
- Code blocks are defined by using opening and closing curly brackets, `{ ... }`. Moreover, code blocks can be *nested*: code blocks can be defined within other code blocks.
- A complete list of reserved and keywords can be found in the PHP Manual: `http://php.net/manual/en/reserved.php` and `http://php.net/manual/en/reserved.keywords.php`

PHP also supports aspects of OOP including classes and inheritance. There is also no memory management in PHP: it has automated garbage collection.

Though not a syntactic requirement, the proper use of whitespace is important for good, readable code. Code inside code blocks is indented at the same indentation. Nested code blocks are indented further. Think of a typical table of contents or the outline of a formal paper or essay. Sections and subsections or points and points all follow proper indentation with elements at the same level at the same indentation. This convention is used to organize code and make it more readable.

29.2.2. PHP Tags

PHP code can be interleaved with static HTML or text. Because of this, we need a way to indicate what should be interpreted as PHP and what should be left alone. We can do this using PHP *tags*: the opening tag is `<?php` and the closing tag is `?>`. Anything placed between these tags will be interpreted as PHP code which must adhere to the syntax rules of the language.

A file can contain multiple opening/closing PHP tags to allow you to interleave multiple sections of HTML or text. When an interpreter runs a PHP script, it will start processing the script file. Whenever it sees the opening PHP tag, it begins to execute the commands and stops executing commands when it sees the closing tag. The text outside the PHP tags is treated as raw data; the interpreter includes it as part of its output without modifying or processing it.

29.2.3. Libraries

PHP has many built-in functions that you can use. These standard libraries are loaded and available to use without any special command to import or include them. Full documentation on each of these functions is maintained in the PHP manual, available online at `http://php.net/manual/en/index.php`. The manual's web pages also contain many curated comments from PHP developers which contain further explanations, tips & tricks, suggestions, and sample code to provide further assistance.

There are many useful functions that we'll mention as we progress through each topic. One especially useful collection of functions is the math library. This library is directly adapted from C's standard math library so all the function names are the same. It provides many common mathematical functions such as the square root and natural logarithm. Table 29.1 highlights several of these functions; full documentation can be found in the PHP manual (`http://php.net/manual/en/ref.math.php`). To use these, you simply "call" them by providing input and getting the output. For example:

```
1 $x = 1.5;
2 $y = sqrt($x); //y now has the value √x = √1.5
3 $z = sin($x); //z now has the value sin(x) = sin(1.5)
```

In both of the function calls above, the value of the variable `$x` is "passed" to the math function which computes and "returns" the result which then gets assigned to another variable.

29.2.4. Comments

Comments can be written in PHP code either as a single line using two forward slashes, `//comment` or as a multiline comment using a combination of forward slash and asterisk: `/* comment */`. With a single line comment, everything on the line *after* the forward slashes is ignored. With a multiline comment, everything in between the forward slash/asterisk is ignored. Comments are ultimately ignored by the compiler so the amount of comments do not have an effect on the final executable code. Consider the following example.

Function	Description
`abs($x)`	Absolute value, $\lvert x \rvert$
`ceil($x)`	Ceiling function, $\lceil 46.3 \rceil = 47.0$
`floor($x)`	Floor function, $\lfloor 46.3 \rfloor = 46.0$
`cos($x)`	Cosine function[a]
`sin($x)`	Sine function[a]
`tan($x)`	Tangent function[a]
`exp($x)`	Exponential function, e^x, $e = 2.71828\ldots$
`log($x)`	Natural logarithm, $\ln(x)$[b]
`log10($x)`	Logarithm base 10, $\log_{10}(x)$[b]
`pow($x,$y)`	The power function, computes x^{y}[c]
`sqrt($x)`	Square root function[b]

Table 29.1.: Several functions defined in the PHP math library. [a]all trigonometric functions assume input is in *radians*, **not** degrees. [b]Input is assumed to be positive, $x > 0$. [c]alternatively, PHP supports exponentiation by using `x ** y`.

```
1 //this is a single line comment
2 $x = 10;  //this is also a single line comment, but after some code
3
4 /*
5   This is a comment that can
6   span multiple lines to format the comment
7   message more clearly
8 */
9 $y = 3.14;
```

Most code editors and IDEs will present comments in a special color or font to distinguish them from the rest of the code (just as our example above does). Failure to close a multiline comment will likely result in a compiler error but with color-coded comments its easy to see the mistake visually.

29.2.5. Entry Point & Command Line Arguments

Every PHP script begins executing at the top of the script file and proceed in a linear, sequential manner top-to-bottom. In addition, you can provide a PHP script with inputs when you run it from the command line. These *command line arguments* are stored in two variables, `$argc` and `$argv`. The first variable, `$argc` is an integer that indicates

the *number* of arguments provided *including* the name of the script file being executed. The second, `$argv` actually stores the arguments as strings. We'll be able to understand the syntax later on, but for now we can at least understand how we can access these arguments.

The variable `$argv` is an array (see Section 34) consisting of the command line arguments. To access them, you can *index* them starting at zero, the first being `$argv[0]`, the second `$argv[1]`, etc. (the last one of course would be at `$argv[$argc-1]`). The first one is always the name of the script file being run. The remaining are the command line arguments provided by the user when the script is executed. We'll see several examples later.

29.3. Variables

PHP is a dynamically typed language. As a consequence, you do not declare variables before you start using them. If you need to store a value into a variable, you simply name the variable and use an assignment operator to assign the value. Since you do not declare variables, you also do not specify a variable's type. If you assign a string to a variable, its type becomes a string. If you assign an integer to a variable, its type becomes an integer. If you reassign the value of a variable to a value with a different type, the variable's type also changes.

Internally, however, PHP does support several different types: Booleans, integers, floating-point numbers, strings, arrays, and objects.

The way that integers are represented may be platform dependent, but are usually 32-bit signed two's complement integers, able to represent integers between $-2,147,483,648$ and 2,147,483,647.

Floating-point numbers are also platform-dependent, but are usually 64-bit double precision numbers are defined by the IEEE 754 standard, providing about 16 digits of precision.

Strings and single characters are the same thing in PHP. Strings are represented as sequences of characters from the extended ASCII text table (see Table 2.4) which includes all characters in the range 0–255. PHP does not have native Unicode support for international characters.

29.3.1. Using Variables

To use a variable in PHP, you simply need to assign a value to a named variable identifier and the variable comes into scope. Variable names *always* begin with a single dollar sign, `$`. The assignment operator is a single equal sign, `=` and is a right-to-left assignment. That is, the variable that we wish to assign the value to appears on the left-hand-side

while the value (literal, variable or expression) is on the right-hand-size. For example:

```
$numUnits = 42;
$costPerUnit = 32.79;
$firstInitial = "C";
```

Each assignment also implicitly changes the variable's type. Each of the variables above becomes an integer, floating-point number, and string respectively. Assignment statements are terminated by a semicolon like most executable statements in PHP. The identifier rules are fairly standard: a variable's name can consist of lower and uppercase alphabetic characters, numbers, and underscores. You can also use the extended ASCII character set in variable names but it is not recommended (umlauts and other diacritics can easily be confused). Variable names are *case sensitive*. As previously mentioned, variable names must *always* begin with a dollar sign, `$`. Stylistically, we adopt the modern camelCasing naming convention for variables in our code.

If you do not assign a value to a variable, that variable remains undefined or "unset". Undefined variables are treated as `null` in PHP. The concept of "null" refers to uninitialized, undefined, empty, missing, or meaningless values. In PHP the keyword `null` is used which is case insensitive (`null`, `Null` and `NULL` are all the same), but for consistency, we'll use `null`. When `null` values are used in arithmetic expressions, `null` is treated as zero. So, `(10 + null)` is equal to 10. When `null` is used in the context of strings, it is treated as an empty string and ignored. When used in a Boolean expression or conditional, `null` is treated as `false`.

PHP also allows you to define *constants*: values that cannot be changed once set. To define a constant, you invoke a function named `define` and providing a name and value. Examples:

```
define("PI", 3.14159);
define("INSTITUTION", "University of Nebraska-Lincoln");
define("COST_PER_UNIT", 2.50);
```

Constant names are case sensitive. By convention, we use uppercase underscore casing. An attempt to redefine a constant value will raise a script warning, but will ultimately have no effect. When referring to constants later on in the script, you use the constant's name. You do not treat it as a string, nor do you use a dollar sign. For example:

```
$area = $r * $r * PI;
```

29.4. Operators

PHP supports the standard arithmetic operators for addition, subtraction, multiplication, and division using `+`, `-`, `*`, and `/` respectively. Each of these operators is a binary operator that acts on two operands which can either be literals or other variables and follow the usually rules of arithmetic when it comes to order of precedence (multiplication and

division before addition and subtraction).

```
$a = 10, $b = 20, $c = 30;
$d = $a + 5;
$d = $a + $b;
$d = $a - $b;
$d = $a + $b * $c;
$d = $a * $b; //d becomes a floating-point number with a value .5

$x = 1.5, $y = 3.4, $z = 10.5;
$w = $x + 5.0;
$w = $x + $y;
$w = $x - $y;
$w = $x + $y * $z;
$w = $x * $y;
$w = $x / $y;

//mixing integers and floating-point numbers is no problem
$w = $a + $x;
```

PHP also supports the integer remainder operator using the % symbol. This operator gives the remainder of the result of dividing two integers. Examples:

```
$x = 10 % 5; //x is 0
$x = 10 % 3; //x is 1
$x = 29 % 5; //x is 4
```

29.4.1. Type Juggling

The expectations of an arithmetic expression involving two variables that are either integers or floating-point numbers are straightforward. We expect the sum/product/etc. as a result. However, since PHP is dynamically typed, a variable involved in an arithmetic expression could be anything, including a Boolean, object, array, or string. When a Boolean is involved in an arithmetic expression, `true` is treated as 1 and `false` is treated as zero. If an array is involved in an expression, it is usually a fatal error.[1] Non-null objects are treated as 1 and `null` is treated as 0.

However, when string values are involved in an arithmetic expression, PHP does something called *type juggling*. When juggled, an attempt is made to convert the string variable into a numeric value by parsing it. The parsing goes over each numeric character, converting the value to a numeric type (either an integer or floating-point number). The first type a non-numeric character is encountered, the parsing stops and the value parsed *so far* is the value used in the expression.

[1] PHP does allow you to "add" two arrays together which results in their union.

Consider the following examples in Code Sample 29.3. In the first block, $a is type juggled to the value 10. when added to 5. In the second example, $a represents a floating-point number, and is converted to 3.14, the result of adding to 5 is thus 8.14. In the third example, the string does not contain any numerical values. In this case, the parsing stops at the first character and what has been parsed *so far* is zero! Finally, in the last example, the first two characters in $a are numeric, so the parsing ends at the third character, and what has been parsed *so far* is 10.

```
$a = "10";
$b = 5 + $a;
print $b; //b = 15

$a = "3.14";
$b = 5 + $a;
print "b = $b"; //b = 8.14

$a = "ten";
$b = 5 + $a;
print "b = $b"; //b = 5

//partial conversions also occur:
$a = "10ten";
$b = 5 + $a;
print "b = $b"; //b = 15
```

Code Sample 29.3: Type Juggling in PHP

Relying on type juggling to convert values can be ugly and error prone. You can write much more intentional code by using the several conversion functions provided by PHP. For example:

```
$a = intval("10");
$b = floatval("3.14");
$c = intval("ten"); //c has the value zero
```

In all three of the examples above, the strings are converted just as they are when type juggled. However, the variables are guaranteed to have the type indicated (integer or floating-point number).

There are several utility functions that can be used to help determine the type of variable. The function `is_numeric($x)` returns `true` if $x is a numeric (integer or floating-point number) or represents a *pure* numeric string. The functions `is_int($x)` and `is_float($x)` each return `true` or `false` depending on whether or not $x is of that type. For example:

```php
$a = 10;
$b = "10";
$c = 3.14;
$d = "3.14";
$e = "hello";
$f = "10foo";

is_numeric($a); //true
is_numeric($b); //true
is_numeric($c); //true
is_numeric($d); //true
is_numeric($e); //false
is_numeric($f); //false

is_int($a); //true
is_int($b); //false
is_int($c); //false
is_int($d); //false
is_int($e); //false
is_int($f); //false

is_float($a); //false
is_float($b); //false
is_float($c); //true
is_float($d); //false
is_float($e); //false
is_float($f); //false
```

A more general way to determine the type of a variable is to use the function `gettype($x)` which returns a string representation of the type of the variable `$x`. The string returned by this function is one of the following depending on the type of `$x`: `"boolean"`, `"integer"`, `"double"`, `"string"`, `"array"`, `"object"`, `"resource"`, `"NULL"`, or `"unknown type"`.

Other checker functions allow you to determine if a variable has been set, if its null, "empty" etc. For example, `is_null($x)` returns `true` if `$x` is not set or is set, but has been set to `null`. The function `isset($x)` returns `true` only if `$x` is set and it is not `null`. The function `empty($x)` returns `true` if `$x` represents an empty entity: an empty string, `false`, an empty array, `null`, `"0"`, 0, or an unset variable. Several examples are presented in Table 29.2.

29.4.2. String Concatenation

Strings in PHP can be concatenated (combined) in several different ways. One way you can combine strings is by using the concatenation operator which in PHP is the period.

Value of $var	isset($var)	empty($var)	is_null($var)
42	bool(true)	bool(false)	bool(false)
"" (an empty string)	bool(true)	bool(true)	bool(false)
" " (space)	bool(true)	bool(false)	bool(false)
false	bool(true)	bool(true)	bool(false)
true	bool(true)	bool(false)	bool(false)
array() (an empty array)	bool(true)	bool(true)	bool(false)
null	bool(false)	bool(true)	bool(true)
"0" (0 as a string)	bool(true)	bool(true)	bool(false)
0 (0 as an integer)	bool(true)	bool(true)	bool(false)
0.0 (0 as a float)	bool(true)	bool(true)	bool(false)
var $var; (declared with no value)	bool(false)	bool(true)	bool(true)
NULL byte ("\0")	bool(true)	bool(false)	bool(false)

Table 29.2.: Results for various variable values

Some examples:

```
1 $s = "Hello";
2 $t = "World!";
3 $msg = $s . " " . $t; //msg contains "Hello World!"
```

Another way you can combine strings is by placing variable values directly in a string. The variables inside the string are replaced with the variable's values. Example:

```
1 $x = 13;
2 $name = "Starlin";
3 $msg = "Hello, $name, your number is $x";
4 //msg contains the string "Hello, Starlin, your number is 13"
```

29.5. Basic I/O

Recall the main purpose of PHP is as a scripting language to serve dynamic webpages. However, it does support a CLI and so it does support input and output from the standard input/output. There are several keywords that allow you to print output to the standard output. The keywords print and echo (which are essentially aliases of each other) allow you to print any variable or string. PHP also has the standard printf function to allow formatted output. Some examples:

```
1  $a = 10;
2  $b = 3.14;
3
4  print $a;
5  print "The value of a is $a\n";
6
7  echo $a;
8  echo "The value of a is $a\n";
9
10 printf("The value of a is %d, and b is %f\n", $a, $b);
```

There are also several ways to perform standard input, but the easiest is to use `fgets` (short for **f**lie **get** **s**tring) using the keyword `STDIN` (Standard Input). This function will return, as a string, everything the user enters up to *and including* the enter key (interpreted as the endline character, `\n`. To remove the endline character, you can use another function, `trim` which removes leading and trailing whitespace from a string. A full example:

```
1  //prompt the user to enter input
2  printf("Please enter a number: ");
3  $a = fgets(STDIN);
4  $a = trim($a);
```

Alternatively, lines 3–4 could be combined into one:

```
$a = trim(fgets(STDIN));
```

The call to `fgets` *waits* (referred to as "blocking") for the user to enter input. The user is free to start typing. When the user i s done, they hit the enter key at which point the program resumes and reads the input from the standard input buffer, and returns it as a string value which we assign to the variable `$a`.

The standard input is unstructured. The user is free to type whatever they want. If we prompt the user for a number but they just start mashing the keyboard giving non-numerical input, we may get incorrect results. We can use the conversion functions mentioned above to attempt to properly convert the values. However, this only guarantees that the resulting variable is of the type we want (integer or floating-point value for example). A non numerical input may be treated as zero in the end. The standard input is not a good mechanism for reading input, but it provides a good starting point.

29.6. Examples

29.6.1. Converting Units

Let's start with a simple task: let's write a program that will prompt the user to enter a temperature in degrees Fahrenheit and convert it to degrees Celsius using the formula

$$C = (F - 32) \cdot \frac{5}{9}$$

We begin with the basic script shell with the opening and closing PHP tags and some comments documenting the purpose of our script.

```
<?php

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */

//TODO: implement this

?>
```

It is common for programmers to use a comment along with a `TODO` note to themselves as a reminder of things that they still need to do with the program.

Let's first outline the basic steps that our program will go through:

1. We'll first prompt the user for input, asking them for a temperature in Fahrenheit
2. Next we'll read the user's input, likely into a floating-point number as degrees can be fractional
3. Once we have the input, we can calculate the degrees Celsius by using the formula above
4. Lastly, we will want to print the result to the user to inform them of the value

Sometimes its helpful to write an outline of such a program directly in the code using comments to provide a step-by-step process. For example:

```php
<?php

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */

//TODO: implement this

//1. Prompt the user for input in Fahrenheit
//2. Read the Fahrenheit value from the standard input
//3. Compute the degrees Celsius
//4. Print the result to the user

?>
```

As we read each step it becomes apparent that we'll need a couple of variables: one to hold the Fahrenheit (input) value and one for the Celsius (output) value. We'll want to ensure that these are floating-point numbers which we can do by making some explicit conversion.

We'll use a `printf` statement in the first step to prompt the user for input:

```php
printf("Please enter degrees in Fahrenheit: ");
```

In the second step, we'll use the standard input to read the `$fahrenheit` variable value from the user. Recall that we can use `fgets` to read from the standard input, but may have to `trim` the trailing whitespace.

```php
$fahrenheit = trim(fgets(STDIN));
```

If we want to ensure that the variable `$fahrenheit` is a floating-point value, we can further use `floatval()`:

```php
$fahrenheit = floatval($fahrenheit);
```

We can now compute `$celsius` using the formula provided:

```php
$celsius = ($fahrenheit - 32) * (5 / 9);
```

Finally, we use `printf` again to output the result to the user:

```php
printf("%f Fahrenheit is %f Celsius\n", $fahrenheit, $celsius);
```

The full program can be found in Code Sample 29.4.

```php
<?php

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */

//1. Prompt the user for input in Fahrenheit
printf("Please enter degrees in Fahrenheit: ");

//2. Read the Fahrenheit value from the standard input
$fahrenheit = trim(fgets(STDIN));
$fahrenheit = floatval($fahrenheit);

//3. Compute the degrees Celsius
$celsius = ($fahrenheit - 32) * (5/9);

//4. Print the result to the user
printf("%f Fahrenheit is %f Celsius\n", $fahrenheit, $celsius);

?>
```

Code Sample 29.4: Fahrenheit-to-Celsius Conversion Program in PHP

29.6.2. Computing Quadratic Roots

Some programs require the user to enter multiple inputs. The prompt-input process can be repeated. In this example, consider asking the user for the coefficients, a, b, c to a quadratic polynomial,

$$ax^2 + bx + c$$

and computing its roots using the quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

As before, we can create a basic program with PHP tags and start filling in the details. In particular, we'll need to prompt for the input a, then read it in; then prompt for b, read it in and repeat for c. Thus, we have

```php
printf("Please enter a: ");
$a = floatval(trim(fgets(STDIN)));
printf("Please enter b: ");
$b = floatval(trim(fgets(STDIN)));
printf("Please enter c: ");
$c = floatval(trim(fgets(STDIN)));
```

Now to compute the roots: we need to take care that we correctly adapt the formula so it accurately reflects the order of operations. We also need to use the math library's square root function (unless you want to write your own! Carefully adapting the formula leads to

```
$root1 = (-$b + sqrt($b*$b - 4*$a*$c) ) / (2*$a);
$root2 = (-$b - sqrt($b*$b - 4*$a*$c) ) / (2*$a);
```

Finally, we print the output using `printf`. The full program can be found in Code Sample 29.5.

```
<?php

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */

printf("Please enter a: ");
$a = floatval(trim(fgets(STDIN)));
printf("Please enter b: ");
$b = floatval(trim(fgets(STDIN)));
printf("Please enter c: ");
$c = floatval(trim(fgets(STDIN)));

$root1 = (-$b + sqrt($b*$b - 4*$a*$c) ) / (2*$a);
$root2 = (-$b - sqrt($b*$b - 4*$a*$c) ) / (2*$a);

printf("The roots of %fx^2 + %fx + %f are: \n", $a, $b, $c);
printf("  root1 = %f\n", $root1);
printf("  root2 = %f\n", $root2);

?>
```

Code Sample 29.5: Quadratic Roots Program in PHP

This program was interactive. As an alternative, we could have read all three of the inputs as command line arguments, taking care that we need to convert them to floating-point numbers. Lines 8–13 in the program could have been changed to

```
$a = floatval($argv[1]);
$b = floatval($argv[2]);
$c = floatval($argv[3]);
```

Finally, think about the possible inputs a user could provide that may cause problems for this program. For example:

- What if the user entered zero for a?
- What if the user entered some combination such that $b^2 < 4ac$?
- What if the user entered non-numeric values?
- For the command line argument version, what if the user provided less than three argument? Or more?

How might we prevent the consequences of such bad inputs? That is, how might we handle the even that a users enters those bad inputs and how do we communicate these errors to the user? To do so we'll need conditionals.

30. Conditionals

PHP supports the basic if, if-else, and if-else-if conditional structures as well as switch statements. Logical statements are built using the standard logical operators for numeric comparisons as well as logical operators such as negations, AND, and OR.

30.1. Logical Operators

PHP has a built-in Boolean type and supports the keywords `true` and `false`. However, *any* variable can be treated as a Boolean if used in a logical expression. Depending on the variable, it could evaluate to *true* or *false*! For example, an empty string, `""`, `null`, or a numeric value of zero, `0` are all considered *false*. A non-empty string, a non-zero numeric value, or a non-empty array all evaluate to *true*. It is best to avoid these issues by writing clean code that uses clear, explicit statements.

Because PHP is dynamically typed, comparison operators work differently depending on how they are used. First, let's consider the four basic inequality operators, `<`, `<=`, `>`, and `>=`. When used to compare numeric types to numeric types, these operators work as expected and the value of the numbers are compared and the result is either *true* or *false*. Examples:

```
1  $a = 10;
2  $b = 20;
3  $c = 20;
4
5  $r = ($a < $b);  //true
6  $r = ($a <= $b); //false
7  $r = ($b <= $c); //true
8  $r = ($a > $b);  //false
9  $r = ($a >= $b); //false
10 $r = ($b >= $c); //true
```

When these operators are used to compare strings to strings, the strings are compared lexicographically according to the standard ASCII text table. Some examples follow, but it is better to use a function (in particular `strcmp` which we discuss later) to do string comparisons.

```
$s = "aardvark";
$t = "zebra";

$r = ($s < $t);  //true
$r = ($s <= $t); //true
$r = ($s >= $t); //false
$r = ($s > $t);   //false
```

However, when these operators are used to compare strings to numeric types, the strings are converted to numbers using the same type juggling that happens when strings are mixed with arithmetic operators. In the following example, `$b` gets converted to a numeric type when compared to `$a` which give the results indicated in the comments.

```
$a = 10;
$b = "10";

$r = ($a <= $b);  //true
$r = ($a < $b);   //false
$r = ($a >= $b);  //true
$r = ($a > $b);   //false
```

With the equality operators, `==` and `!=`, something similar happens. When the types of the two operands match, the expected comparison is made: when numbers are compared to numbers their values are compared; when strings are compared to strings, their content is compared (case sensitively). However, when the types are different, again, type juggling happens and strings are converted to numbers for the purpose of comparison. Thus, a comparison like `(10 == "10")` ends up being *true*! The operators are `==` and `!=` are referred to as *loose* comparison operators because of this.

What if we want to ensure that we're comparing apples to apples? To rectify this, PHP offers another set of comparison operators, *strict* comparison operators, `===` and `!==` (the same, but with an extra equals sign, `=`). These operators will make a comparison *without* type juggling either operand first. Now a similar comparison, `(10 === "10")` ends up evaluating to *false*. The operator `===` will only evaluate to *true* if the both the operands' type *and* value are the same.

```
$a = 10;
$b = "10";

$r = ($a == $b);  //true
$r = ($a != $b);  //false
$r = ($a === $b); //false
$r = ($a !== $b); //true
```

The three basic logical operators, not `!`, And `&&`, and Or `||` are also supported. The operands applied to these operators will be converted to Boolean values according to the rules described in Table **??**.

	Operator(s)	Associativity	Notes
Highest	++, --	left-to-right	increment operators
	-, !	right-to-left	unary negation operator, logical not
	*, /, %	left-to-right	
	+, -	left-to-right	addition, subtraction
	<, <=, >, >=	left-to-right	comparison
	==, !=, ===, !==	left-to-right	equality, inequality
	&&	left-to-right	logical And
	||	left-to-right	logical Or
Lowest	=, +=, -=, *=, /=	right-to-left	assignment and compound assignment operators

Table 30.1.: Operator Order of Precedence in PHP. Operators on the same level have equivalent order and are performed in the associative order specified.

30.1.1. Order of Precedence

At this point it is worth summarizing the order of precedence of all the operators that we've seen so far including assignment, arithmetic, comparison, and logical. Since all of these operators could be used in one statement, for example,

```
($b*$b < 4*$a*$c || $a === 0 || $argc != 4)
```

it is important to understand the order in which each one gets evaluated. Table 30.1 summarizes the order of precedence for the operators seen so far. This is not an exhaustive list of PHP operators.

30.2. If, If-Else, If-Else-If Statements

Conditional statements in PHP utilize the key words `if`, `else`, and `else if`. Conditions are placed inside parentheses immediately after the `if` and `else if` keywords. Examples of all three can be found in Code Sample 30.1.

Some observations about the syntax: the statement, `if($x < 10)` does not have a semicolon at the end. This is because it is a conditional statement that determines the flow of control and *not* an executable statement. Therefore, no semicolon is used. Suppose we made a mistake and *did* include a semicolon:

```
1 $x = 15;
2 if($x < 10); {
3   printf("x is less than 10\n");
4 }
```

This PHP code will run without error or warning. However, it will end up printing

```php
//example of an if statement:
if($x < 10) {
  printf("x is less than 10\n");
}

//example of an if-else statement:
if($x < 10) {
  printf("x is less than 10\n");
} else {
  printf("x is 10 or more \n");
}

//example of an if-else-if statement:
if($x < 10) {
  printf("x is less than 10\n");
} else if($x === 10) {
  printf("x is equal to ten\n");
} else {
  printf("x is greater than 10\n");
}
```

Code Sample 30.1: Examples of Conditional Statements in PHP

`x is less than` 10, even though $x = 15$! Recall that a conditional statement *binds* to the executable statement or code block *immediately* following it. In this case, we've provided an *empty* executable statement ended by the semicolon. The code is essentially equivalent to

```php
$x = 15;
if($x < 10) {
}
printf("x is less than 10\n");
```

Which is obviously not what we wanted. The semicolon ended up binding to the empty executable statement, and the code block containing the print statement immediately followed, but was *not* bound to the conditional statement which is why the print statement executed regardless of the value of x.

Another convention that we've used in our code is where we have placed the curly brackets. First, if a conditional statement is bound to only one statement, the curly brackets are not necessary. However, it is best practice to include them even if they are not necessary and we'll follow this convention. Second, the opening curly bracket is on the same line as the conditional statement while the closing curly bracket is indented to the same level as the start of the conditional statement. Moreover, the code inside the code block is indented. If there were more statements in the block, they would have all been at the

same indentation level.

30.3. Examples

30.3.1. Computing a Logarithm

The logarithm of x is the exponent that some *base* must be raised to get x. The most common logarithm is the natural logarithm, $\ln(x)$ which is base $e = 2.71828\ldots$. But logarithms can be in any base $b > 1$[1] What if we wanted to compute $\log_2(x)$? Or $\log_\pi(x)$? Let's write a program that will prompt the user for a number x and a base b and computes $\log_b(x)$.

Arbitrary bases can be computed using the change of base formula:

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$$

If we can compute *some* base a, then we can compute any base b. Fortunately we have such a solution. Recall that the standard library provides a function to compute the natural logarithm, `log()`). This is one of the fundamentals of problems solving: if a solution already exists, use it. In this case, a solution exists for a different, but similar problem (computing the natural logarithm), but we can *adapt* the solution using the change of base formula. In particular, if we have variables `b` (base) and `x`, we can compute $\log_b(x)$ using

```
log(x) / log(b)
```

But wait: we have a problem similar to the examples in the previous section. The user could enter invalid values such as $b = -10$ or $x = -2.54$ (logarithms are undefined for non-positive values in any base). We want to ensure that $b > 1$ and $x > 0$. With conditionals, we can now do this. Once we have read in the input from the user we can make a check for good input using an `if` statement.

```
1 if($x <= 0 || $b <= 1) {
2   printf("Error: bad input!\n");
3   exit(1);
4 }
```

This code has something new: `exit(1)`. The `exit` function immediately terminates the script regardless of the rest of the code that may remain. The argument passed to `exit` is an integer that represents an *error code*. The convention is that zero indicates "no error" while non-zero values indicate some error. This is a simple way of performing *error handling*: if the user provides bad input, we inform them and quit the program,

[1] Bases can also be $0 < b < 1$, but we'll restrict our attention to increasing functions only.

forcing them to run it again and provide good input. By prematurely terminating the program we avoid any illegal operation that would give a bad result.

Alternatively, we could have split the conditions into two statements and given a more descriptive error message. We use this design in the full program which can be found in Code Sample 30.2. The program also takes the input as command line arguments. Now that we have conditionals, we can actually check that the correct number of arguments was provided by the user and quit in the event that they don't provide the correct number.

```
<?php

/**
 * This program computes the logarithm base b (b > 1)
 * of a given number x > 0
 */

if($argc != 3) {
  printf("Usage: %s b x \n", $argv[0]);
  exit(1);
}

$b = floatval($argv[1]);
$x = floatval($argv[2]);

if($x <= 0) {
  printf("Error: x must be greater than zero\n");
  exit(1);
}
if($b <= 1) {
  printf("Error: base must be greater than one\n");
  exit(1);
}

$result = log($x) / log($b);
printf("log_(%f)(%f) = %f\n", $b, $x, $result);

?>
```

Code Sample 30.2: Logarithm Calculator Program in C

30.3.2. Life & Taxes

Let's adapt the conditional statements we developed in Section 3.6.4 into a full PHP script. The first thing we need to do is establish the variables we'll need and read them in from the user. At the same time we can check for bad input (negative values) for both the inputs.

```
1  //prompt for income from the user
2  printf("Please enter your Adjusted Gross Income: ");
3
4  $income = floatval(trim(fgets(STDIN)));
5
6  //prompt for children
7  printf("How many children do you have? ");
8  $numChildren = intval(trim(fgets(STDIN)));
9
10 if($income < 0 || $numChildren < 0) {
11   printf("Invalid inputs");
12   exit(1);
13 }
```

Next, we can code a series of if-else-if statements for the income range. By placing the ranges in increasing order, we only need to check the upper bounds just as in the original example.

```
1  if($income <= 18150) {
2    $baseTax = $income * .10;
3  } else if($income <= 73800) {
4    $baseTax = 1815 + ($income  -18150) * .15;
5  } else if($income <= 148850) {
6  ...
7  } else {
8    $baseTax = 127962.50 + ($income - 457600) * .396;
9  }
```

Next we compute the child tax credit, taking care that it does not exceed $3,000. A conditional based on the number of children should suffice as at this point in the program we already know it is zero or greater.

```
1  if($numChildren <= 3) {
2    $credit = $numChildren * 1000;
3  } else {
4    $credit = 3000;
5  }
```

Finally, we need to ensure that the credit does not exceed the total tax liability (the credit is non-refundable, so if the credit is greater, the tax should only be zero, not

negative).

```
if($baseTax - $credit >= 0) {
  $totalTax = $baseTax - $credit;
} else {
  $totalTax = 0;
}
```

The full program is presented in Code Sample 30.3.

30.3.3. Quadratic Roots Revisited

Let's return to the quadratic roots program we previously designed that uses the quadratic equation to compute the roots of a quadratic polynomial by reading coefficients a, b, c in from the user. One of the problems we had previously identified is if the user enters "bad" input: if $a = 0$, we would end up dividing by zero; if $b^2 - 4ac < 0$ then we would have complex roots. With conditionals, we can now check for these issues and exit with an error message.

Another potential case we might want to handle differently is when there is only one distinct root ($b^2 - 4ac = 0$). In that case, the quadratic formula simplifies to $\frac{-b}{2a}$ and we can print a different, more specific message to the user. The full program can be found in Code Sample 30.4.

```php
<?php
//prompt for income from the user
printf("Please enter your Adjusted Gross Income: ");

$income = floatval(trim(fgets(STDIN)));

//prompt for children
printf("How many children do you have? ");
$numChildren = intval(trim(fgets(STDIN)));

if($income < 0 || $numChildren < 0) {
  printf("Invalid inputs");
  exit(1);
}

if($income <= 18150) {
  $baseTax = $income * .10;
} else if($income <= 73800) {
  $baseTax = 1815 + ($income  -18150) * .15;
} else if($income <= 148850) {
  $baseTax = 10162.50 + ($income - 73800) * .25;
} else if($income <= 225850) {
  $baseTax = 28925.00 + ($income - 148850) * .28;
} else if($income <= 405100) {
  $baseTax = 50765.00 + ($income - 225850) * .33;
} else if($income <= 457600) {
  $baseTax = 109587.50 + ($income - 405100) * .35;
} else {
  $baseTax = 127962.50 + ($income - 457600) * .396;
}

if($numChildren <= 3) {
  $credit = $numChildren * 1000;
} else {
  $credit = 3000;
}

if($baseTax - $credit >= 0) {
  $totalTax = $baseTax - $credit;
} else {
  $totalTax = 0;
}

printf("AGI:           $%10.2f\n", $income);
printf("Tax:           $%10.2f\n", $baseTax);
printf("Credit:        $%10.2f\n", $credit);
printf("Tax Liability: $%10.2f\n", $totalTax);

?>
```

Code Sample 30.3: Tax Program in PHP

```php
<?php

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */

if($argc != 4) {
  printf("Usage: %s a b c\n", $argv[0]);
  exit(1);
}

$a = floatval($argv[1]);
$b = floatval($argv[2]);
$c = floatval($argv[3]);

if($a === 0) {
  printf("Error: a cannot be zero\n");
  exit(1);
} else if($b*$b < 4*$a*$c) {
  printf("Error: cannot handle complex roots\n");
  exit(1);
} else if($b*$b === 4*$a*$c) {
  $root1 = -$b / (2*$a);
  printf("Only one distinct root: %f\n", $root1);
} else {
  $root1 = (-$b + sqrt($b*$b - 4*$a*$c) ) / (2*$a);
  $root2 = (-$b - sqrt($b*$b - 4*$a*$c) ) / (2*$a);

  printf("The roots of %fx^2 + %fx + %f are: \n", $a, $b, $c);
  printf("  root1 = %f\n", $root1);
  printf("  root2 = %f\n", $root2);
}

?>
```

Code Sample 30.4: Quadratic Roots Program in PHP With Error Checking

31. Loops

PHP supports while loops, for loops, and do-while loops using the keywords `while`, `for`, and `do` (along with another `while`). Continuation conditions for loops are enclosed in parentheses, `(...)` and the blocks of code associated with the loop are enclosed in curly brackets.

31.1. While Loops

Code Sample 31.1 contains an example of a basic while loop in PHP. Just as with conditional statements, our code styling places the opening curly bracket on the same line as the `while` keyword and continuation condition. The inner block of code is also indented and all lines in the block are indented to the same level.

```
1 $i = 1; //Initialization
2 while($i <= 10) {  //continuation condition
3   //perform some action
4   $i++; //iteration
5 }
```

Code Sample 31.1: While Loop in PHP

In addition, the continuation condition does *not* contain a semicolon since it is not an executable statement. Just as with an if-statement, if we *had* placed a semicolon it would have led to unintended results. Consider the following:

```
1 while($i <= 10); {
2   //perform some action
3   $i++; //iteration
4 }
```

A similar problem occurs: the `while` keyword and continuation condition bind to the next executable statement or code block. As a consequence of the semicolon, the executable statement that gets bound to the while loop is *empty*. What happens is even worse: the program will enter an infinite loop. To see this, the code is essentially equivalent to the following:

```
while($i <= 10) {
}
{
  //perform some action
  $i++; //iteration
}
```

In the while loop, we never increment the counter variable `$i`, the loop does nothing, and so the computation will continue on forever! Some compilers will warn you about this, others will not. It is valid PHP and will run, but obviously won't work as intended. Avoid this problem by using proper syntax.

Another common use case for a while loop is a flag-controlled loop in which we use a Boolean flag rather than an expression to determine if a loop should continue or not. Since PHP has built-in Boolean types, we can use a variable along with the keywords `true` and `false` appropriately. An example can be found in Code Sample 31.2.

```
$i = 1;
$flag = true;
while($flag) {
  //perform some action
  $i++; //iteration
  if($i>10) {
    $flag = false;
  }
}
```

Code Sample 31.2: Flag-controlled While Loop in PHP

31.2. For Loops

For loops in PHP use the familiar syntax of placing the initialization, continuation condition, and iteration on the same line as the keyword `for`. An example can be found in Code Sample 31.3.

```
$i;
for($i=1; $i<=10; $i++) {
  //perform some action
}
```

Code Sample 31.3: For Loop in PHP

Again, note the syntax: semicolons are placed at the end of the initialization and continuation condition, but *not* the iteration statement. Just as with while loops, the

opening curly bracket is placed on the same line as the `for` keyword. Code within the loop body is indented, all at the same indentation level.

31.3. Do-While Loops

PHP also supports do-while loops. Recall that the difference between a while loop and a do-while loop is when the continuation condition is checked. For a while loop it is *prior* to the beginning of the loop body and in a do-while loop it is at the *end* of the loop. This means that a do-while always executes *at least once*. An example can be found in Code Sample 31.4.

```
$i;
do {
  //perform some action
  $i++;
} while($i <= 10);
```

Code Sample 31.4: Do-While Loop in PHP

Note the syntax and style: the opening curly bracket is again on the same line as the keyword `do`. The `while` keyword and continuation condition are on the same line as the closing curly bracket. In a slight departure from consistent syntax, a semicolon *does* appear at the end of the continuation condition even though it is not an executable statement.

31.4. Foreach Loops

Finally, PHP does support foreach loops using the keyword `foreach`. Some of this will be a preview of Section 34 where we discuss arrays in PHP[1], but in short you can iterate over the elements of an array as follows.

```
$arr = array(1.41, 2.71, 3.14);
foreach($arr as $x) {
  //$x now holds the "current" element in $arr
}
```

In the `foreach` syntax we specify the array we want to iterate over, `$arr` and use the keyword `as`. The last element in the statement is the variable name that we want to use within the loop. This should be read as "`foreach` element `$x` in the array `$arr`...". Inside the loop, the variable `$x` will be automatically updated on each iteration to the next element in `$arr`.

[1] Actually, PHP supports *associative* arrays, which are not the same thing as traditional arrays.

31.5. Examples

31.5.1. Normalizing a Number

Let's revisit the example from Section 4.1.1 in which we *normalize* a number by continually dividing it by 10 until it is less than 10. The code in Code Sample 31.5 specifically refers to the value 32145.234 but would work equally well with any value of `$x`.

```
$x = 32145.234;
$k = 0;
while($x > 10) {
  $x = $x / 10;
  $k++;
}
```

Code Sample 31.5: Normalizing a Number with a While Loop in PHP

31.5.2. Summation

Let's revisit the example from Section 4.2.1 in which we computed the sum of integers $1 + 2 + \cdots + 10$. The code is presented in Code Sample 31.6

```
$sum = 0;
for($i=1; $i<=10; $i++) {
  $sum += $i;
}
```

Code Sample 31.6: Summation of Numbers using a For Loop in PHP

Of course we could easily have generalized the code somewhat. Instead of computing a sum up to a particular number, we could have written it to sum up to another variable `$n`, in which case the for loop would instead look like the following.

```
for($i=1; $i<=n; $i++) {
  $sum += $i;
}
```

31.5.3. Nested Loops

Recall that you can write loops within loops. The inner loop will execute fully *for each* iteration of the outer loop. An example of two nested of loops in PHP can be found in Code Sample 31.7.

```php
$n = 10;
$m = 20;
for($i=0; $i<$n; $i++) {
  for($j=0; $j<$m; $j++) {
    printf("(i, j) = (%d, %d)\n", $i, $j);
  }
}
```

Code Sample 31.7: Nested For Loops in PHP

The inner loop execute for $j = 0, 1, 2, \ldots, 19 < m = 20$ for a total of 20 times. However, it executes 20 times *for each* iteration of the outer loop. Since the outer loop execute for $i = 0, 1, 2, \ldots, 9 < n = 10$, the total number of times the `printf` statement execute is $10 \times 20 = 200$. In this example, the sequence $(0, 0), (0, 1), (0, 2), \ldots, (0, 19), (1, 0), \ldots, (9, 19)$ will be printed.

31.5.4. Paying the Piper

Let's adapt the solution for the loan amortization schedule we developed in Section 4.7.3. First, we'll read the principle, terms, and interest as command line inputs.

Adapting the formula for the monthly payment and using the standard math library's `pow` function, we get

```php
$monthlyPayment = ($monthlyInterestRate * $principle) /
          (1 - pow( (1 + $monthlyInterestRate), -$n));
```

However, recall that we may have problems due to accuracy. The monthly payment could come out to be a fraction of a cent, say $43.871. For accuracy, we need to ensure that all of the figures for currency are rounded to the nearest cent. The standard math library does have a `round` function, but it only rounds to the nearest whole number, not the nearest 100th.

However, we can *adapt* the "off-the-shelf" solution to fit our needs. If we take the number, multiply it by 100, we get (say) 4387.1 which we can now round to the nearest whole number, giving us 4387. We can then divide by 100 to get a number that has been rounded to the nearest 100th! In PHP, we could simply do the following.

```php
$monthlyPayment = round($monthlyPayment * 100) / 100;
```

We can use the same trick to round the monthly interest payment and any other number expected to be whole cents. To output our numbers, we use `printf` and take care to align our columns to make make it look nice. To finish our adaptation, we handle the final month separately to account for an over/under payment due to rounding. The full solution can be found in Code Sample 31.8.

```php
<?php
  if($argc != 4) {
    printf("Usage: %s principle apr terms\n", $argv[0]);
    exit(1);
  }

  $principle = floatval($argv[1]);
  $apr = floatval($argv[2]);
  $n = intval($argv[3]);

  $balance = $principle;
  $monthlyInterestRate = $apr / 12;

  //monthly payment
  $monthlyPayment = ($monthlyInterestRate * $principle) /
         (1 - pow( (1 + $monthlyInterestRate), -$n));
  //round to the nearest cent
  $monthlyPayment = round($monthlyPayment * 100) / 100;

  printf("Principle: $%.2f\n", $principle);
  printf("APR: %.4f%%\n", $apr * 100.0);
  printf("Months: %d\n", $n);
  printf("Monthly Payment: $%.2f\n", $monthlyPayment);

  //for the first n-1 payments in a loop:
  for($i=1; $i<$n; $i++) {
    //  compute the monthly interest, rounded:
    $monthlyInterest =
      round( ($balance * $monthlyInterestRate) * 100) / 100;
    //  compute the monthly principle payment
    $monthlyPrincipalPayment = $monthlyPayment - $monthlyInterest;
    //  update the balance
    $balance = $balance - $monthlyPrincipalPayment;
    //  print i, monthly interest, monthly principle, new balance
    printf("%d\t$%10.2f  $%10.2f  $%10.2f\n", $i, $monthlyInterest,
                          $monthlyPrincipalPayment, $balance);
  }

  //handle the last month and last payment separately
  $lastInterest = round( ($balance * $monthlyInterestRate) * 100) / 100;
  $lastPayment = $balance + $lastInterest;

  printf("Last payment = $%.2f\n", $lastPayment);
?>
```

Code Sample 31.8: Loan Amortization Program in PHP

32. Functions

Functions are essential in PHP programming. As we've already seen, PHP provides a large library of standard functions to perform basic input/output, math, and many other functions. PHP also provides the ability to define and use your own functions.

PHP does not support function overloading, so when you define a function and give it a name, that name cannot be in conflict with any other function name in the standard library or any other code that you might use. Therefore, careful though should go into the design of your functions.

PHP supports both call by value and call by reference. As of PHP 5.6, vararg functions are also supported (though earlier versions supported some vararg-like functions such as `printf()`). However, we will not go into detail here. Finally, another feature of PHP is that function parameters are all optional. You may invoke a function with a subset of the parameters; depending on your PHP setup, it may be issue a warning that a parameter was omitted. However, PHP allows you to define default values for optional parameters.

32.1. Defining & Using Functions

In general, you can define functions anywhere in your PHP script or codebase. They can even appear *after* code that invokes them because PHP essentially *hoists* the function definitions by doing two passes of the script. However, it is good style to include function definitions at the top of your script or in a separate PHP file for organization.

32.1.1. Declaring Functions

In PHP, to declare a function you use the keyword `function`. Because PHP is dynamically typed, a function can return *any* type. Therefore, you do not declare the return type (just as you do not declare a variable's type). After the `function` keyword you do provide an identifier and parameters as the function signature. Immediately following, you provide the function body enclosed with opening/closing curly brackets.

Typically, the documentation for functions is included with its declaration. Consider the following examples. In these examples we use a commenting style known as "doc comments." This style was originally developed for Java but has since been adopted by many other languages.

```php
/**
 * Computes the sum of the two arguments.
 */
function sum($a, $b) {
  return ($a + $b);
}

/**
 * Computes the Euclidean distance between the 2-D points,
 * (x1,y1) and (x2,y2).
 */
function getDistance($x1, $y1, $x2, $y2) {
  $xDiff = ($x1-$x2);
  $yDiff = ($y1-$y2);
  return sqrt( $xDiff * $xDiff + $yDiff * $yDiff);
}

/**
 * Computes a monthly payment for a loan with the given
 * principle at the given APR (annual percentage rate) which
 * is to be repaid over the given number of terms (usually
 * months).
 */
function getMonthlyPayment($principle, $apr, $terms) {
  $rate = ($apr / 12.0);
  $payment = ($principle * $rate) / (1-pow(1+$rate, -$terms));
  return $payment;
}
```

Function identifiers (names) follow similar naming rules as variables, however they do not begin with a dollar sign. Function names must begin with an alphabetic character and may contain alphanumeric characters as well as underscores. However, using modern coding conventions we usually name functions using lower camel casing. Another quirk of PHP is that function names are *case insensitive*. Though we declared a function, `getDistance()` above, it could be invoked with either `getdistance()`, `GETDISTANCE` or any other combination of capital/lower case letters. However, good code will use consistent naming and your function calls *should* match their declaration.

The keyword `return` is used to specify the value that is returned to the calling function. Whatever value you end up returning is the return type of the function. Since you do not specify variable or return types, functions are usually referred to as returning a "mixed" type. You could design a function that, given one set of inputs, returns a number while another set of inputs ends up returning a string.

You can use the syntax `return;` to return no value (you do not use the keyword `void`).

In practice, however, the function ends up returning `null` when doing this.

32.1.2. Organizing Functions

There are many coding standards that guide how PHP code should be organized. We'll only discuss a simple mechanism here. One way to organize functions is to collect functions with similar functionality into separate PHP source files.

Suppose the functions above are in a PHP source file named `utils.php`. We could include them in another source file (our "main" source file) using an `include_once` function invocation. An example:

```
1  <?php
2
3  include_once("utils.php");
4
5  //we can now use the functions in utils.php:
6  $p = getMonthlyPayment(1000, 0.05, 12);
```

The `include_once` function essentially loads and evaluates the given PHP source file at the point in the code in which it is invoked. The "once" in the function refers to the fact that if the source file was already included in the script/code before, it will not be included a second time. This allows you to include the same source file in multiple source files without a conflict.

32.1.3. Calling Functions

The syntax for calling a function is to simply provide the function name followed by parentheses containing values or variables to pass to the function. Some examples:

```
1  $a = 10, $b = 20;
2  $c = sum($a, $b); //c contains the value 30
3
4  //invoke a function with literal values:
5  $dist = getDistance(0.0, 0.0, 10.0, 20.0);
6
7  //invoke a function with a combination:
8  $p = 1500.0;
9  $r = 0.05;
10 $monthlyPayment = getMonthlyPayment($p, $r, 60);
```

32.1.4. Passing By Reference

By default, all types (including numbers, strings, etc.) are passed by value. To be able to pass arguments by reference, we need to use slightly different syntax when defining our functions.

To specify that a parameter is to be passed by reference, we place an ampersand, `&` in front of it in the function signature. [1] No other syntax is necessary and when you call the function, PHP automatically takes care of the referencing/dereferencing for you. Consider the following examples.

```
1  <?php
2
3  function swap($a, $b) {
4    $t = $a;
5    $a = $b;
6    $b = $t;
7  }
8
9  function swapByRef(&$a, &$b) {
10   $t = $a;
11   $a = $b;
12   $b = $t;
13 }
14
15 $x = 10;
16 $y = 20;
17
18 printf("x = %d, y = %d\n", $x, $y);
19 swap($x, $y);
20 printf("x = %d, y = %d\n", $x, $y);
21 swapByRef($x, $y);
22 printf("x = %d, y = %d\n", $x, $y);
23
24 ?>
```

The first function, `swap()` passes both variables by value. Swapping the values only affects the *copies* of the parameters. The original variables `$x` and `$y` will be unaffected. In the second function, `swap2()`, both variables are passed by reference as there are ampersands in front of them. Swapping them inside the function, swaps the original variables. The output to this code is as follows.

[1] Those familiar with pointers in C will note that this is the exact *opposite* of the C operator.

```
x = 10, y = 20
x = 10, y = 20
x = 20, y = 10
```

Observe that when we invoked the function, `swapByRef($x, $y);` we used the same syntax as the pass by value version. The only syntax needed to pass by reference is in the function signature itself.

32.1.5. Function Pointers

Functions are just pieces of code that reside somewhere in memory just as variables do. Since we can pass variables by reference, it also makes sense that we would do the same with functions.

In PHP, functions are first-class citizens[2] meaning that you can assign a function to a variable just as you would a numeric value. For example, you can do the following.

```
1 $func = swapByRef;
2
3 $func($x, $y);
```

In the example above, we assigned the function `swapByRef()` to the variable `$func` by using its identifier. The variable essentially holds a reference to the `swapByRef()` function. Since it refers to a function, we can also invoke the function using the variable as in the last line. This allows you to treat functions as callbacks to other functions. We will revisit this concept in Chapter **??**.

32.2. Examples

32.2.1. Generalized Rounding

Recall that the standard math library provides a `round()` function that rounds a number to the nearest whole number. Often, we've had need to round to cents as well. We now have the ability to write a function to do this for us. Before we do, however, let's think more generally. What if we wanted to round to the nearest tenth? Or what if we wanted to round to the nearest 10s or 100s place? Let's write a general purpose rounding function that allows us to specify *which* decimal place to round.

The most natural input values would be to specify the place using an integer exponent. That is, if we wanted to round to the nearest tenth, then we would pass it -1 as $0.1 = 10^{-1}$, -2 if we wanted to round to the nearest 100th, etc. On the positive end

[2] Some would use a much more restrictive definition of first-class and would *not* consider them first-class citizens in this sense

passing in 0 would correspond to the usual round function, 1 to the nearest 10s spot, and so on.

Moreover, we could demonstrate good code reuse (as well as procedural abstraction) by *scaling* the input value and reusing the functionality already provided in the math library's `round()` function. We could further define a `roundToCents()` function that used our generalized round function. Consider the following.

```
<?php

/**
 * Rounds to the nearest digit specified by the place
 * argument.  In particular to the (10^place)-th digit
 */
function roundToPlace($x, $place) {
  $scale = pow(10, -$place);
  $rounded = round(x * $scale) / $scale;
  return $rounded;
}

/**
 * Rounds to the nearest cent
 */
function roundToCents($x) {
  return roundToPlace($x, -2);
}

?>
```

We could place these functions into a file named `round.php` and include them in another PHP source file.

32.2.2. Quadratic Roots

Another advantage of passing variables by reference is that we can "return" multiple values with one function call. Functions are limited in that they can only return at most one value. But if we pass multiple parameters by reference, the function can manipulate the contents of them, thereby communicating (though not strictly returning) multiple values.

Consider again the problem of computing the roots of a quadratic equation,

$$ax^2 + bx + c = 0$$

using the quadratic formula,

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Since there are two roots, we may have to write two functions, one for the "plus" root and one for the "minus" root both of which take the coefficients, a, b, c as arguments. However, if we wrote a single function that took the coefficients as parameters by value as well as two other parameters by reference, we could place *both* root values, one in each of the by-reference variables.

```
function quadraticRoots($a, $b, $c, &$root1, &$root2) {
  $discriminant = sqrt($b*$b - 4*$a*$c);
  $root1 = (-$b + $discriminant) / (2*$a);
  $root2 = (-$b - $discriminant) / (2*$a);
  return;
}
```

By using pass by reference variables, we avoid multiple functions. Recall that there could be several "bad" inputs to this function. The roots could be complex values, the coefficient a could be zero, etc. In the next chapter, we examine how we can *handle* these errors.

33. Error Handling & Exceptions

Modern versions of PHP support error handling through the use of exceptions. PHP has several different predefined types of exceptions and also allows you to define your own exception types by creating new classes that *inherit* from the generic `Exception` class. PHP uses the standard `try-catch-finally` control structure to handle exceptions and allows you to `throw` your own exceptions.

33.1. Throwing Exceptions

Though PHP defines several different types of exceptions, we'll only look at the generic `Exception` class. We can `throw` an exception in PHP by using the keyword `throw` and creating a new `Exception` with an error message.

```
1 throw new Exception("Something went wrong");
```

By using a generic `Exception`, we can only attach a message to the exception (which can be printed by code that catches the exception). If we want more fine-grained control over the type of exceptions, we need to define our own exceptions.

33.2. Catching Exceptions

To catch an exception in PHP you can use the standard `try-catch` control block. Optionally (and as of PHP version 5.5.0) you can use the `finally` block to clean up any resources or execute code regardless of whether or not an exception was raised. Let's take, for example, the simple task of reading input from a user and manually parsing its value into an integer. If the user enters a non-numeric value, parsing will fail and we should instead throw an exception. Consider the following function.

```
function readNumber() {
  $input = readline("Please enter a number: ");
  if( is_numeric($input) ) {
    $value = floatval($input);
  } else {
    throw new Exception("Invalid input!");
  }
}
```

Elsewhere in the code, we can surround a call to `readNumber()` in a `try-catch` statement.

```
try {
  readNumber();
} catch(Exception $e) {
  printf("Error: exception encountered: " . $e->getMessage());
  exit(1);
}
```

In this example, we've simply displayed an error message to the standard error output and exited the program. That is, we've made the design decision that this error should be fatal. We could have chosen to handle this error differently in the `catch` block. The `$e->getMessage()` prints the message that the exception was created with. In this case, `"Invalid input!"`.

33.3. Creating Custom Exceptions

As of PHP 5.3.0 it is possible to define custom exceptions by *extending* the `Exception` class. To do this, you need to declare a new class. We will cover the details of classes later on. For now, we simply look at an example.

Consider the example in the previous chapter of computing the roots of a quadratic polynomial. One possible error situation is when the roots are complex numbers. We could define a new PHP class as follows.

```php
/**
 * Defines a ComplexRoot exception class
 */
class ComplexRootException extends Exception
{
    public function __construct($message = null,
                                $code = 0, Exception $previous = null) {
        // simply call the parent constructor
        parent::__construct($message, $code, $previous);
    }

    // custom string representation of object
    public function __toString() {
        return __CLASS__ . ": [{$this->code}]: {$this->message}\n";
    }
}
```

Now in our code we can catch and even throw this new type of exception.

```php
if( $b*$b - 4*$a*$c < 0) {
  throw new ComplexRootException("Cannot Handle complex roots");
}
```

```php
try {
  $r1 = getRoot($a, $b, $c);
} catch(ComplexRootException $e) {
  //handle here
} catch(Exception $e) {
  //handle all other types of exceptions here
}
```

In the code above we had *two* `catch` blocks. Since we can have multiple types of exceptions, we can also catch each different type and handle them differently if we choose.

Each `catch` block catches a different type of exception. The last `catch` block was written to catch a generic `Exception`. This last block will essentially catch any other type of exception. Much like an `if-else-if` statement, the first type of exception that is caught is the block that will be executed and they are all mutually exclusive. Thus, a "catch all' block like this should always be the last `catch` block. The most specific types of exceptions should be caught first and the most general types should be caught last.

34. Arrays

TODO

35. Strings

TODO

Glossary

algorithm a process or method that consists of a specified step-by-step set of operations. 15

anonymous function a function that has no identifier or name, typically created so that it can be passed as an argument to another function as a callback. 130

anti-pattern a common software pattern that is used as a solution to recurring problems that is usually ineffective in solving the problem or introduces risks and other problems; a technical term for common "bad-habits" that can be found in software. 136

assignment operator an operator that allows a user to assign a value to a variable. 31

backward compatible a program, code, library, or standard that is compatible with previous versions so that current and older versions of it can coexist and successfully operate without breaking anything. 27

bit the basic unit of information in a digital computer. A bit can be either 1 or 0 (alternatively, *true*/*false*, on/off, high voltage/low voltage, etc.). Originally a portmanteau (mash up) of **b**inary dig**it**. 4, 20

Boolean a data type that represents the truth value of a logical statement. Booleans typically have only two values: *true* or *false*. 27

bug A flaw or mistake in a computer program that results in incorrect behavior that may have unintended such as errors or failure. The term predates modern computer systems but was popularized by Grace Hopper who, when working with the Mark II computer in 1946 traced a system failure to a moth stuck in a relay. 43, 73, 135

byte a unit of information in a digital computer consisting of 8 bits. 4

call by reference when a variable's memory address is passed as a parameter to a function, enabling the function to manipulate the contents of the memory address and change the original variable's value. 126

call by value when a *copy* of a variable's value is passed as a parameter to a function; the function has no reference to the original variable and thus changes to the copy inside the function have no effect on the original variable. 124

callback a function or executable unit of code that is passed as an argument to another

function with the intention that the function that it is passed to will execute or "call back" the passed function at some point. 128

case sensitive a language is case sensitive if it recognizes differences between lower and upper case characters in identifier names. A language is case insensitive if it does not. 17

code smell a symptom or common pattern in source code that is usually indicative of a deeper problem or design flaw; smells are usually not bugs and may not cause problems in and of themselves, but instead indicate a pattern of carelessness or low quality of software design or implementation. 136

constant a variable whose value cannot be changed once set.

contradiction a logical statement that is always *false* regardless of the truth values of the statement's variables. 64

control flow the order in which individual statements in a program are executed or evaluated. 67

cruft anything that is left over, redundant or getting in the way; in the context of code cruft is code that is no longer needed, legacy or simply poorly written source code.

dead code a code segment that has no effect on a program either because it is unused or unreachable (the conditions involving the code will never be satisfied). 64

debug the process of analyzing a program to find a fault or error with the code that leads to bad or unexpected results. 136

debugger a software tool that facilitates debugging; usually a debugger simulates the execution of a program allowing a developer to view the contents of a program as it executes and to "walk" through the execution step by step. 136

defensive programming an approach to programming in which error conditions are checked and handled, preventing undefined or erroneous operations from happening in a program. 36, 45, 75

dynamic typing a variable whose type can change during runtime based on the value it is assigned. 28,

encapsulation the grouping and protection of data together into one logical entity along with the functionality (functions or methods) that act on that data. 28

enumerated type a data type (usually user defined) that consists of a list of named values. 213, 273

exception an event or occurrence of an erroneous or "exceptional" condition that interrupts the normal flow of control in a program, handing control over to exception handler(s). 139

expression a combination of values, constants, literals, variables, operators and possibly function calls such that when evaluated, produce a resulting value. 31

flowcharts a diagram that represents an algorithm or process, showing steps as boxes connected by arrows which establish an order or flow. 15

function a sequence of program instructions that perform a specific task, packaged as a unit, also known as a *subroutine*. 119

function overloading the ability to define multiple functions with the same name but with with a different number of or different types of parameters. 130

garbage collection automated memory management in which a garbage collector attempts to reclaim memory (garbage) that is no longer being used by a program so that it can be reallocated for other purposes. 223

global scope a variable, function, or other element in a program has global scope if it is visible or has effect throughout the entire program. 30,

hoisting usually used in interpreted languages, hoisting involves processing code to find variable or function declarations and processing them before actually executing the code or script. 120

identifier a symbol, token, or label that is used to refer to a variable. Essentially, a variable's name. 16

immutable an object whose internal state cannot be changed once created, alternatively, one whose internal state cannot be *observably* changed once created. 229, 261

input data or information that is provided to a computer program for processing. 38

interactive a program that is designed to interface with humans by prompting them for input and displaying output directly to them. 39

interactive an informal, abstract, high-level description of a process or algorithm. 39

keyword a word in a programming language with a special meaning in a particular context. In contrast to a reserved word, a keyword *may* be used for an identifier (variable or function name) but it is strongly discouraged to do so as the keyword already has an intended meaning.

kilobyte a unit of information in a digital computer consisting of 1024 bytes (equivalently, 2^{10} bytes), KB for short.

lint (or linter) a static code analysis tool that analyzes code for suspicious or error-prone code that is likely to cause problems. 136

literal in a programming language, a literal is notation for specifying a value such as a number of string that can be directly assigned to a variable. 27, 31

magic number a value used in a program with unexplained, undocumented, or ambiguous meaning, usually making the code less understandable. 213, 214, 274

mantissa the part of a floating-point number consisting of its significant digits (called a significand in scientific notation). 23

naming convention a set of guidelines for choosing identifier names for variables, functions, etc. in a programming language. Conventions may be generally accepted by all developers of a particular language or they may be established for use in a particular library, framework, or organization. 17

operand the arguments that an operator applies to. 30

operator a symbol used to denote some transformation that combines or changes the operands it is applied to to produce a new value. 30

order of precedence the order in which operators are evaluated, multiplication is performed before addition for example. 35

output data or information that is produced as the result of the execution of a program. 39

overflow when an arithmetic operation results in a number that is larger than the specified type can represent overflow occurs resulting in an invalid result. 36

pointer a reference to a particular memory location in a computer. 28, 201

primitive a basic data type that is defined and provided by a programming language. Typically numeric and character types are primitive types in a language for example. Generally, the user doesn't need to define the operations involving primitive types as they are defined by the language. Primitive data types are used as the basic building blocks in a program and used to *compose* more complex user-defined types. 28, 229

procedural abstraction the concept that a procedure or sequence of operations can be encapsulated into one logical unit (function, subroutine, etc.) so that a user need not concern themselves with the low-level details of how it operates. 120

program stack also referred to as a *call stack*, it is an area of memory where stack frames are stored for each function call containing memory for arguments, local variables and return values/addresses. 123

pseudocode the act of a program asking a user to enter input and subsequently waiting for the user to enter data. 12

queue a data structure that store elements in a FIFO (First-In First-Out) manner; elements can be added to the end of a queue by an *enqueue* operation and removed from the start of a queue by a *dequeue* operation.

radix the base of a number system. Binary, octal, decimal, hexadecimal would be base 2, 8, 10, and 16 respectively.

refactor the process of modifying, updating or restructuring code without changing its external behavior; refactoring may be done to make code more efficient, more readable, more reliable, or simply to bring it into compliance with style or coding conventions.

reference a reference in a computer program is a variable that refers to an object or function in memory. 28

reserved word a word or identifier in a language that has a special meaning to the syntax of the language and therefore cannot be used as an identifier in variables, functions, etc.. 17

scope the *scope* of a variable, method, or other entity in a program is the part of the program in which the name or reference of the entity is bound. That is, the part of the program that "knows" about the variable in which the variable can be accessed, changed, or used. 29, 163, 225

scope a function signature is how a function is uniquely identified. A signature includes the name (identifier) of the function, its parameter list (and maybe types) and the return type. 120

segmentation fault a fault or error that arises when a program attempts to access a segment of memory that it is not allowed access to, usually resulting in the program being terminated by the operating system. 201

short circuiting the process by which the second operand in a logical statement is not evaluated if the value of the expression is determined by the first operand. 66

spaghetti code a negative term used for code that is overly complex, disorganized or unstructured code.

stack a data structure that stores elements in a LIFO (last-in first-out) manner; elements can be added to a stack via a *push* operation which places the element on the "top" of the stack; elements can be removed from the top of the stack via a *pop* operation. 123

static analysis the analysis of software that is performed on source (or object) code without actually running or compiling a program usually by using an automated tool that can detect actual or potential problems with the source code (other than syntactic problems that could easily be found by a compiler). 136

static dispatch when function overloading is supported in a language, this is the mecha-

nism by which the compiler determines *which* function should be called based on the number and type of arguments passed to the function when it is called. 130

static typing a variable whose type is specified when it is created (declared) and does not change while the variable remains in scope. 28,

string a data type that consists of a sequence of characters which are encoded under some encoding standard such as ASCII or Unicode. 24

string concatenation an operation by which a string and another data type are combined to form a new string. 35

syntactic sugar syntax in a language or program that is not absolutely necessary (that is, the same thing can be achieved using other syntax), but may be shorter, more convenient, or easier to read/write. In general, such syntax makes the language "sweeter" for the humans reading and writing it. 37, 74, 90

tautology a logical statement that is always *true* regardless of the truth values of the statement's variables. 64

top-down design an approach to problem solving where a problem is broken down into smaller parts. 3, 119

truncation removing the fractional part of a floating-point number to make it an integer. Truncation is *not* a rounding operation. 33, 169, 232

two's complement A way of representing signed (positive and negative) integers using the first bit as a sign bit (0 for positive, 1 for negative) and where negative numbers are represented as the complement with respect to 2^n (the result of subtracting the number from 2^n) . 22

type a variable's type is the classification of the data it represents which could be numeric, string, boolean, or a user defined type. 20

type casting converting or variable's type into another type, for example, converting an integer into a more general floating-point number, or converting a floating-point number into an integer, truncating and losing the fractional part. 34, 169, 232

underflow when an arithmetic operation involving floating-point numbers results in a number that is smaller than the smallest representable number underflow occurs resulting in an invalid result. 36

validation the process of verifying that data is correct or conforms to certain expectations including formatting, type, range of values, represents a valid value, etc.. 39

variable a memory location which stores a value that may be set using an assignment operator. Typically a variable is referred to using a name or *identifier*. 16

Acronyms

ACM Association for Computing Machinery.

ALU Arithmetic and Logic Unit. 4

ANSI American National Standards Institute. 161

API Application Programmer Interface. 14, 273

ASCII American Standard Code for Information Interchange. 25, 58, 166, 225, 230

CLI Command Line Interface. 42, 293

CMS Content Management System. 283

CPU Central Processing Unit. 4, 66

CSS Cascading Style Sheets.

DRY Don't Repeat Yourself. 197

EB Exabyte.

ECMA European Computer Manufacturers Association.

FIFO First-In First-Out.

FOSS Free and Open Source Software.

GB Gigabyte.

GCC GNU Compiler Collection.

GDB GNU Debugger.

GIMP GNU Image Manipulation Program.

GIS Geographic Information System.

GNU GNU's Not Unix!. 46

GUI Graphical User Interface. 39, 130

HTML HyperText Markup Language. 283

IDE Integrated Development Environment. 5, 19, 43, 46, 165, 224, 228, 287

IEC International Electrotechnical Commission. 23, 161

IEEE Institute of Electrical and Electronics Engineers. 23, 24, 167, 212, 288

ISO International Organization for Standardization. 161

JDBC Java Database Connectivity.

JDK Java Development Kit. 223, 224, 265, 269, 273

JEE Java Enterprise Edition.

JIT Just In Time. 9

JPEG Joint Photographic Experts Group.

JRE Java Runtime Environment.

JVM Java Virtual Machine. 7, 9, 223, 224, 261, 269, 273

KB Kilobyte.

LIFO Last-In First-Out. 123

MB Megabyte.

NIST National Institute of Standards and Technology.

ODBC Open Database Connectivity.

OEM Original Equipment Manufacturer.

OOP Object-Oriented Programming. 224, 226, 271, 285

PB Petabyte.

POJO Plain Old Java Object.

POSIX Portable Operating System Interface. 166, 212

RAM Random Access Memory. 5

REPL Read-Eval-Print Loop.

ROM Read-Only Memory.

SQL Structured Query Language. 269, 273

SSL Secure Sockets Layer. 201

STEAM Science, Technology, Engineering, Art, and Math.

STEM Science, Technology, Engineering, and Math.

TB Terabyte.

UTF-8 Universal (Character Set) Transformation Format–8-bit.

VLSI Very Large Scale Integration. 4

W3C World Wide Web Consortium.

WWW World Wide Web. 223

XML Extensible Markup Language. 18

Index

Bibliography

[1] Mars climate orbiter. http://mars.jpl.nasa.gov/msp98/orbiter/, 1999. [Online; accessed 17-March-2015].

[2] Moth in the machine: Debugging the origins of 'bug'. Computer World Magazine, September 2011.

[3] errno.h: system error numbers - base definitions reference. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/errno.h.html, 2013. [Online; accessed 13-September-2015].

[4] ISO/IEC 9899 - Programming Languages - C. http://www.open-std.org/JTC1/SC22/WG14/www/standards, 2013.

[5] Java platform standard edition 7. http://docs.oracle.com/javase/7/docs/api/, 2015. [Online; accessed 10-February-2015].

[6] List of software bugs. https://en.wikipedia.org/wiki/List_of_software_bugs, 2015. [Online; accessed 12-September-2015].

[7] Unchecked exceptions — the controversy. https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html, 2015. [Online; accessed 15-September-2015].

[8] Douglas Adams. *Dirk Gently's Holistic Detective Agency*. Pocket Books, 1987.

[9] Michael Braukus. NASA honors apollo engineer. NASA News, September 2003.

[10] Bruce Eckel. *Thinking in Java*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2005.

[11] Arend Heyting. *Die formalen Regeln der intuitionistischen Logik*. Berlin, 1930. First use of the notation ¬ as a negation operator.

[12] IEC. *IEC 60559 (1989-01): Binary floating-point arithmetic for microprocessor systems*. 1989. This Standard was formerly known as IEEE 754.

[13] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[14] John McCarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[15] Brian W. Kernighan. *Programming in C – A Tutorial*. Bell Laboratories, Murray

Hill, New Jersey, 1974.

[16] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[17] Tony Long. The man who saved the world by doing ... nothing. Wired, September 2007.

[18] M. V. Wilkes, D. J. Wheeler and S. Gill. *The preparation of programs for an electronic digital computer, with special reference to the EDSAC and the use of a library of subroutines*. Addison-Wesley Press, Cambridge, Mass., 1951.

[19] Giuseppe Peano. Studii de Logica Matematica. In *Atti della Reale Accademia delle scienze di Torino*, volume 32 of *Classe di Scienze Fisiche Matematiche e Naturali*, pages 565–583. Accademia delle Scienze di Torino, Torino, April 1897.

[20] Bertrand Russell. The theory of implication. *American Journal of Mathematics*, 28:159–202, 1906.

[21] Bruce A. Tate. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. Pragmatic Bookshelf, 1st edition, 2010.

[22] Alfred North Whitehead and Bertrand Arthur William Russell. *Principia mathematica; vol. 1*. Cambridge Univ. Press, Cambridge, 1910.