All of Programming

Edition 0

Andrew Hilton
Anne Bracy
Table of Contents

| Detailed Table of Contents                  | iii |
| List of Figures                             | v   |
| List of Videos                              | vii |
| Preface                                     | viii|

I Introduction to Programming in C

| 1 Introduction                             | 2   |
| 2 Reading Code                             | 22  |
| 3 Types                                     | 44  |
| 4 Writing Code                             | 65  |

Appendices

| Why Expert Tools?                          | 82  |
| A UNIX Basics                              | 83  |

Index

100
# Detailed Table of Contents

Detailed Table of Contents  iii  
List of Figures  v  
List of Videos  vii  
Preface  viii  

## I Introduction to Programming in C  

### 1 Introduction 2  
1.1 How to Write a Program ................................................. 3  
1.2 Algorithms ........................................................................ 4  
1.3 Step 1: Work an Example Yourself  ...................................... 7  
1.4 Step 2: Write Down What You Just Did  ................................. 8  
1.5 Step 3: Generalize Your Steps ............................................. 8  
1.6 Step 4: Test Your Algorithm .............................................. 9  
1.7 Some Examples .................................................................. 11  
1.8 Next Steps ......................................................................... 17  
1.9 Practice Exercises ............................................................ 18  

### 2 Reading Code 22  
2.1 Variables .......................................................................... 23  
2.2 Expressions ....................................................................... 24  
2.3 Functions .......................................................................... 25  
2.4 Conditional Statements ..................................................... 31  
2.5 Shorthand .......................................................................... 34  
2.6 Loops ................................................................................ 35  
2.7 Higher-level Meaning ........................................................ 38  
2.8 Practice Exercises ............................................................ 39  

### 3 Types 44  
3.1 Hardware Representations ................................................... 44  
3.2 Basic Data Types ................................................................ 46  
3.3 Printing Redux ................................................................... 51  
3.4 Expressions Have Types ..................................................... 51  
3.5 “Non-Numbers” .............................................................. 56  
3.6 Complex, Custom Data Types ............................................ 58
3.7 Practice Exercises ........................................................................ 63

4 Writing Code ............................................................................... 65
  4.1 Step 1: Work an example yourself ............................................. 66
  4.2 Step 2: Write down what you just did ......................................... 67
  4.3 Step 3: Generalize your steps ..................................................... 68
  4.4 Step 4: Test Your Algorithm ..................................................... 72
  4.5 Step 5: Translation to Code ...................................................... 73
  4.6 A Complete Example ................................................................. 78
  4.7 Next Steps: Compiling, Running, Testing and Debugging .......... 78
  4.8 Practice Exercises .................................................................. 78

Appendices ...................................................................................... 82

Why Expert Tools? ........................................................................... 82

A UNIX Basics .................................................................................. 83
  A.1 In the Beginning Was the Command Line ................................. 84
  A.2 Getting Help: man and help .................................................... 85
  A.3 Directories ............................................................................. 87
  A.4 Displaying Files .................................................................... 90
  A.5 Moving, Copying, and Deleting ............................................... 91
  A.6 Pattern Expansion: Globbing and Braces ................................. 91
  A.7 Redirection and Pipes ............................................................ 92
  A.8 Searching ............................................................................ 94
  A.9 Command Shells .................................................................... 95
  A.10 Scripting ............................................................................... 96
  A.11 Environment Variables .......................................................... 98
  A.12 Remote Login: ssh ............................................................... 99

Index ............................................................................................... 100

All of Programming, http://aop.cs.cornell.edu
# List of Figures

## Chapter 1 Introduction

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>6</td>
</tr>
</tbody>
</table>

## Chapter 2 Reading Code

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>26</td>
</tr>
<tr>
<td>2.3</td>
<td>26</td>
</tr>
<tr>
<td>2.5</td>
<td>28</td>
</tr>
<tr>
<td>2.6</td>
<td>29</td>
</tr>
<tr>
<td>2.7</td>
<td>32</td>
</tr>
<tr>
<td>2.8</td>
<td>33</td>
</tr>
<tr>
<td>2.9</td>
<td>35</td>
</tr>
<tr>
<td>2.10</td>
<td>35</td>
</tr>
<tr>
<td>2.11</td>
<td>37</td>
</tr>
</tbody>
</table>

## Chapter 3 Types

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>47</td>
</tr>
<tr>
<td>3.4</td>
<td>47</td>
</tr>
<tr>
<td>3.5</td>
<td>48</td>
</tr>
<tr>
<td>3.6</td>
<td>49</td>
</tr>
<tr>
<td>3.7</td>
<td>50</td>
</tr>
<tr>
<td>3.8</td>
<td>50</td>
</tr>
<tr>
<td>3.9</td>
<td>51</td>
</tr>
<tr>
<td>3.10</td>
<td>56</td>
</tr>
<tr>
<td>3.11</td>
<td>58</td>
</tr>
<tr>
<td>3.12</td>
<td>59</td>
</tr>
<tr>
<td>3.13</td>
<td>61</td>
</tr>
<tr>
<td>3.14</td>
<td>62</td>
</tr>
</tbody>
</table>

## Chapter 4 Writing Code

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>66</td>
</tr>
</tbody>
</table>
4.2 Another instance of the rectangle problem, with the Cartesian grid removed 67
4.3 Rectangles demonstrating problem with algorithm 69

Appendix Intro 82
4 Tools for novices vs tools for experts 82

Appendix A UNIX Basics 83
A.1 The command prompt 84
A.2 Display of man page for printf 86
A.3 Examples of the cd and ls commands 89
# List of Videos

## Chapter 1 Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Testing our algorithm</td>
<td>13</td>
</tr>
<tr>
<td>1.2</td>
<td>Devising the algorithm for the pattern on the grids.</td>
<td>15</td>
</tr>
<tr>
<td>1.3</td>
<td>Devising the algorithm for filling in a rectangle</td>
<td>16</td>
</tr>
<tr>
<td>1.4</td>
<td>Devising an algorithm to find the closest point in a set $S$ to another point $P$</td>
<td>17</td>
</tr>
</tbody>
</table>

## Chapter 2 Reading Code

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Variable declarations</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>Declarations and Assignments</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>Assignment statements with more complex right-hand sides</td>
<td>25</td>
</tr>
<tr>
<td>2.4</td>
<td>Execution of function calls</td>
<td>27</td>
</tr>
<tr>
<td>2.5</td>
<td>An example with printed output</td>
<td>30</td>
</tr>
<tr>
<td>2.6</td>
<td>Execution of if/else</td>
<td>33</td>
</tr>
<tr>
<td>2.7</td>
<td>Execution of <code>switch/case</code></td>
<td>34</td>
</tr>
<tr>
<td>2.8</td>
<td>Execution of a while loop</td>
<td>36</td>
</tr>
<tr>
<td>2.9</td>
<td>Execution of a nested loops and ifs</td>
<td>38</td>
</tr>
<tr>
<td>2.10</td>
<td>Execution of <code>continue</code> in a <code>for</code> loop</td>
<td>39</td>
</tr>
</tbody>
</table>

## Chapter 3 Types

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Execution of our naïve code for computing the hours in a 7 day work week.</td>
<td>54</td>
</tr>
<tr>
<td>3.2</td>
<td>Execution of our fixed code for computing the hours in a 7 day work week</td>
<td>55</td>
</tr>
<tr>
<td>3.3</td>
<td>Declaration and use of a <code>struct</code> for a rectangle</td>
<td>60</td>
</tr>
<tr>
<td>3.4</td>
<td>Executing code which uses an enumerated type</td>
<td>63</td>
</tr>
</tbody>
</table>

## Chapter 4 Writing Code

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Writing the <code>isPrime</code> function</td>
<td>78</td>
</tr>
</tbody>
</table>
Preface

Programming is an increasingly popular skill—whether for those who want to be professional software developers, or those who want to write programs to analyze and manipulate data or automate tasks in some other field. Programming course enrollment is soaring, and a plethora of online options are springing up to provide instruction in the field. However, experience shows that many courses (of either form) which aim to teach introductory programming do not actually teach how to program.

In writing this book, we set out to provide a platform for instructors to design courses which properly place their focus on the core fundamentals of programming, or to let a motivated student learn these skills independently. We do not waste any time or space on the specifics of the buzzword-of-the-day technology. Instead, we dedicate all of our efforts to core concepts and techniques of programming. A student who masters the material in this book will not just be a competent C programmer, but also a competent programmer, as the title of the book would suggest.\footnote{Our title is also a playful nod to Larry Wasserman’s seminal book “All of Statistics.”}

Some people may question the language choice of this book: “Why C?” “Isn’t C hard for beginners?” “Everyone loves language X, why not do this in X?” At some level, the answer is “it does not matter.” We are teaching programming not a particular language. We just need a language so that students can implement their programs in a way that a computer can understand. We note that we do briefly discuss other languages in four chapters, beginning with Chapter ??.

On another level, C (and C++) are excellent choices for a variety of reasons. Perhaps most importantly, we can introduce ideas in a natural and logical fashion without “just do this because you have to, but cannot understand it yet”—such a practice is harmful to teaching any programmer, who should fully understand any code she writes. Furthermore, C and C++ provide a more complete picture of programming concepts. Many other language choices would require omitting some core concept which that language does not have. Such an omission would require the student to learn an entirely new concept to switch languages. As the “icing on the cake,” C and C++ have a long history, and still have a wide-spread (and well-paid!) presence in industry.

We note that this book is quite large: over 30 chapters, and 6 appendices, spanning over 700 pages and 7.5 hours of video. Covering all of this material in a single semester is quite an aggressive pace—approximately one chapter per class day. Such a pace is possible, but requires heavily motivated students who are willing to put in significant effort. Generally that pace would only be appropriate to a Masters level “ramp up” course for students switching disciplines from one with no programming background into one where many other classes expect near-professional level programming.

For an undergraduate course, a more appropriate pace would be to use Part I (Introduction to Programming in C) as a “CS 1” course (likely with heavy reference to the appendices on programmers’ tools and editors). Such a pace would result in approximately one chapter per week. A “CS 2” course could then be constructed from Parts II (C++) and III (Data Structures and Algorithms) also at approximately one chapter per week. Part IV’s material could be placed in
later courses that are intended only for more serious programmers.

We further recommend using this book in a “flipped classroom” model—in which students’ primary intake of material is done out of class (i.e., by reading this book), and in class time is spent on activities. These activities should primarily be formed from programming, or programming-related (e.g., executing code by hand) topics. Students can then perform the most important tasks—doing programming—with expert help and guidance available.

On the http://aop.cs.cornell.edu, we provide some questions and problems for each chapter (in Parts I, II, and III) to help you check your understanding of the material. Some of these problems ask you to explain the basic concepts in the chapter. Others ask you to perform the skills you should be learning (reading and writing code). If you are teaching a class with this book, we encourage you to create some larger, more sophisticated problems for students to do in class—possibly providing some infrastructure to allow students to do write “cool and exciting” programs. Some practice problems have sample answers provided.

We will also note that this book has embedded videos, which are an integral part of its design. You should watch the videos as you work your way through this book, as they convey important material—a lot of things in programming happen actively, and are much better conveyed to you, the learner, through animations rather than static figures. Videos should look generally like this:

![Video](image)

You will notice that the video has relatively standard play controls. You can click the video to play/pause it, as well as use the time-position slider at the bottom to jump backwards or forwards in the video. If you do not understand something, you may want to jump back and rewatch it!

Not all e-readers support videos. If you cannot view the videos, you should try a different e-reader.

Finally, we will note that this is the first version of the book. Accordingly, we would be shocked if there is not at least one typo, somewhere in the book. If you discover a problem, please check our website http://aop.cs.cornell.edu/ to see if we are already aware of it. If not, please report the problem to us there. We will post a correction and fix it in the next edition. If you need to contact us, you can email us at aop@cs.cornell.edu.
We hope you enjoy the book and learn a lot—happy hacking!

All of Programming, http://aop.cs.cornell.edu
Acknowledgements

We would like to take a moment to thank the many people who made this book possible. We are both deeply grateful to the many wonderful teachers of computer science—from high school through graduate school—who both educated us and inspired us. It is one thing to convey knowledge. It is quite another thing to ignite a love of computer science and teaching that motivates one to write a textbook. We have both also enjoyed teaching bright and motivated students. They have also played an important motivating and refining role in this undertaking.

We also want to thank our respective spouses, Margaret and Kilian for their love and support during the lengthy and arduous process of writing a book of this size. It is hard to imagine finishing this book without them.
Part I

Introduction to Programming in C
Programming is an increasingly important skill, whether you aspire to a career in software development, or in other fields. In this text, you will learn how to program, starting from no prior knowledge. Even if you have some prior knowledge, you may wish to start at the beginning of the book, as the approach here may be quite different than what you have seen before.

Many new programmers (and some courses or texts) place undue focus on language syntax and language features—aiming to become an expert in whatever language they have heard is popular on the job market. While syntax is important—the computer cannot understand your program if you do not write it properly in a programming language—it is not the heart of programming. In fact, the key aspect of programming is *metacognition*—thinking about how you think. Specifically, programming is fundamentally about figuring out how to solve a class of problems, and writing down the *algorithm*—a set of steps to solve any problem in that class—in a clear and unambiguous manner. Programming languages (such as C, C++, Java, Scheme, or SML) figure into this equation primarily as a means to provide a clearly defined manner to write down the algorithm. Natural language, such as English, is too ambiguous and complicated for this purpose. A good programmer should be able to pick up a new language quite quickly. The key skills of programming are universal—learning a new language is largely just a matter of learning its syntax.

A natural consequence of being overly syntax-focused is that many novice programmers attempt to dive right into writing the code (in the programming language) as the first step. However, writing the code is actually a much later step in the process. A good programmer will plan first and write second, possibly breaking down a large programming task into several smaller tasks in the process. Even when cautioned to plan first and code second, many programming students ignore the advice—after all, why “waste” 30 minutes planning when you are time-crunched from all the work you have to do. This tradeoff, however, presents a false economy—30 minutes planning could save hours of trying to make the code work properly. Well planned code is not only more likely to be correct (or at least closer to correct), but is also easier to understand—and thus fix.

To try to better understand the importance of planning before you write, imagine an analogy to building a house or sky scraper. If you were tasked with building a sky scraper, would you
break ground and start building right away, figuring out how the building is designed as you go? Hopefully not. Instead, you (or an architect) would design blueprints for the building first. These blueprints would be iteratively refined until they meet everyone’s specifications—they must meet the requirements of the building’s owner, as well as be possible to build reasonably. Once the blueprints are completed, they must be approved by the local government. Actual construction only begins once the plans are fully completed. Programming should be done in a similar manner—come up with a complete plan (algorithm) first and build (implement in code) second.

We said that the heart of programming is to figure out how to solve a class of problems—not just one particular problem. The distinction here is best explained by an example. Consider the task of figuring out if a particular number (e.g., 7) is prime. With sufficient knowledge of math (i.e., the definition of a prime number and the rules of division), one can solve this problem—determining that 7 is in fact prime. However, a programming problem typically looks at a more general class of problems. We would typically not write a program to determine if 7 is prime, but rather a program which, given a number $N$, determines if $N$ is prime. Once we have an algorithm for this general class of problems, we can have the computer solve any particular instance of the problem for us.

When we examine a class of problems, we have parameters which tell us which particular problem in the class we are solving. In the previous example, the class of problems is parameterized by $N$—the number we want to test for primality. To develop an algorithm for this class of problems, we must account for all possible legal values of the parameters. As we will see later (in Chapter 3), programming languages let us restrict what type of information a parameter can represent, to limit the legal values to those which make sense in the context of the problem. For primality testing, we would want our parameter $N$ to be restricted such that it can only hold integer numbers. It would not make any sense to check if letters, words, or files are prime.

To write a program which takes any number $N$ and determines if $N$ is prime, we must first figure out the algorithm for this class of problems. As we said before, if we attack the problem by blindly writing code, we will end up with a mess—much like constructing a sky scraper with no plan. Coming up with the appropriate algorithm for a class of problems is a challenging task, and typically requires significant work and thought.

1.1 How to Write a Program

Figure 1.1 shows a high-level overview of the programming process. A programmer starts by devising the algorithm for the task she is trying to solve. We will split this planning phase into four steps in the process of writing a program, which we will discuss in more detail shortly. At the end of these four steps, the programmer should have a complete plan for the task at hand, and be convinced that the plan is a good one.

After devising a proper algorithm, she is ready for Step 5 of the programming process: translating her plan into code in the programming language she is using for her current project. Initially, translation to code will go slowly, as you will be unfamiliar with the syntax—likely, needing to look up the specific details often. However, even if slow, it should be fairly straightforward. You already devised the plan, so you should have done all the actual problem-solving tasks already. Your algorithm may have some complex steps, but that is fine. As we will see later, whenever your algorithm calls for a step that is too complicated to be simply translated into a few lines of code, you should turn that step into its own separate programming task, and repeat the programming process on it. We will discuss translation to code in much more detail in Chapter 4, as well as how to turn the code into something that the computer can run in Chapter ??.

Once the algorithm is implemented in code, the programmer must test her code, which is the 6th
Step of the programming process. By testing the program, the programmer tries to uncover errors in her algorithm or implementation. If the programmer finds errors in her program, she debugs the program (Step 7)—finding out the cause of the error, and fixing it. The programmer may need to return to the algorithm design steps (if the error lies in the algorithm) or to translation to code (if the error lies in the implementation) to correct the error. The programmer then repeats all of the later steps.

At some point, the programmer completes enough test cases with no errors to become convinced that her program is correct. Note that we said that the programmer becomes convinced that her program is correct. No amount of testing can guarantee that the program is correct. Instead, more testing increases the programmer’s confidence that the code is correct. When the programmer is convinced her code is correct, she has successfully completed the task at hand. We will discuss testing and debugging in much more detail in Chapter ??.

1.2 Algorithms

As we discussed earlier, an algorithm is a clear set of steps to solve any problem in a particular class. Typically, algorithms have at least one parameter, however, algorithms with no parameters exist—they are simply restricted to one specific problem, rather than a more general class. We can discuss and think about algorithms in the absence of any particular knowledge of computers—a good algorithm can not only be translated into code, but also could be executed by a person with no particular knowledge of the problem at hand.

Algorithms that computers work on deal with numbers—in fact Chapter 3 will discuss the concept of “Everything is a number,” which is a key principle in programming. Computers can only compute on numbers, however, Chapter 3 will also teach us how we can represent a variety of useful things (letters, words, images, videos, sound,...) as numbers so that computers can compute on them. As a simple example of an algorithm that works with numbers, we might consider the following algorithm (which takes one parameter, $N$, which is a non-negative integer):
Chapter 1: Introduction

Given a non-negative integer $N$:

Make a variable called $x$, set it equal to $(N+2)$

Count from 0 to $N$ (include both ends), and for each number (call it "i") that you count:

Write down the value of $(x * i)$

Update $x$ to be equal to $(x + i * N)$

When you finish counting, write down the value of $x$.

For any non-negative integer $N$ that I give you, you should be able to execute these steps. If you do these steps for $N = 2$, you should come up with the sequence of numbers $0 \ 4 \ 12 \ 10$. These steps are unambiguous as to what should happen. It is possible that you get the wrong answer if you misunderstand the directions, or make arithmetic mistakes, but otherwise, everyone who does them for a particular value of $N$ should get the same answer. We will also note that this algorithm can be converted into any programming language quite easily—all that is needed is to know the basic syntax of the particular language you want.

You may wonder why we would want an algorithm that generates this particular sequence of numbers. In this case, it is just a contrived algorithm to show as a simple introductory example. In reality, we are going to devise algorithms that solve some particular problem. However, devising the algorithm for a problem takes some significant work, and will be the focus of discussion of the rest of the chapter.

Even though computers can only work with numbers, we can envision algorithms that might be executed by humans which work on a variety of things. For example, we might write algorithms that operate on physical objects such as LEGO bricks or food. Even though such things would be difficult to implement on a computer (we would need the computer to control a robot to actually interact with the physical world), they are still instructive, as the fundamental algorithmic design principles are the same.

One exercise done at the start of some introductory programming courses is to have the students write down directions to make a peanut butter and jelly sandwich. The instructor then executes the algorithms, which are often imprecise and ambiguous. The instructor takes the most comical interpretation of the instructions to underscore that what the students wrote did not actually describe what they meant.

This exercise underscores an important point—you must specify exactly what you want the computer to do. The computer does not “know what you mean” when you write something vague, nor can it figure out an “etc”. Instead, you must be able to describe exactly what you want to do in a step-by-step fashion. Precisely describing the exact steps to perform a specific task is somewhat tricky, as we are used to people implicitly understanding details we omit. The computer will not do that for you (in any programming language).

Even though the “sandwich algorithm” exercise makes an important point about precisely describing the steps you want the computer to perform, it falls short in truly illustrating the hardest part of designing an algorithm. This algorithm has no parameters, so it just describes how to solve one particular problem (making a peanut butter and jelly sandwich). Real programming problems (typically) involve algorithms that take parameters. A more appropriate problem might be “Write an algorithm which takes a list of things you want in a sandwich, and describes how to make the sandwich.”

Such a problem is much more complex, but illustrates many concepts involved in devising a real algorithm. First, our algorithm cannot take a list of just anything to include in the sandwich—it really will only work with certain types of things, namely food. We would not expect our algorithm to be able to make us a “car, skyscraper, airplane” sandwich. These items are all the wrong type. We will learn more about types in programming in Chapter 3.
Our algorithm may also have to deal with error cases. Even if we specify the correct type of inputs, the particular values may be impossible to operate on correctly. For example, “chicken breast” is food, but if the chicken breast has not been cooked yet, we should not try to make a sandwich out of it. Another error case in our sandwich creation algorithm might be if we specify too much food to go inside the sandwich (how do you make a sandwich with an entire turkey, 40 pounds of carrots, and 3 gallons of ice cream?). Of course, if we were writing this sandwich algorithm for humans, we could ignore this craziness, because humans have “common sense”—however, computers do not.

Even if we ignore all of the error cases, our general algorithm is not as simple as just stacking up the ingredients on top of bread in the order they appear in the input. For example, we might have an input of “chicken, mustard, spinach, tomatoes.” Here, we probably want to spread the mustard on the bread first, then place the other ingredients on it (hopefully in an order that makes the most stable sandwich).

It would seem that writing a correct algorithm to make a sandwich from an arbitrary list of ingredients is quite a complex task. Even if we did not want to implement that algorithm in code, but rather have it be properly executed by a person with no common sense (or a professor with a comedic disregard for common sense), this task is quite challenging to do correctly. How could we go about this task and hope to get a good algorithm?

The wrong way to write an algorithm is to just throw some stuff on the page, and then try to straighten it out later. Imagine if we approached our sandwich example by writing down some steps, and having someone (with no common sense) try them out. After the kitchen catches on fire, we try to go in and figure out what went wrong. We then tweak the steps, and try again. This time, the kitchen explodes instead. We repeat this process until we finally get something that resembles a sandwich, and the house did not burn down.

The previous paragraph may sound silly, but is exactly how many novice (and intermediate) programmers approach programming tasks. They jump right into writing code (no time to plan! Busy schedule!), and it inevitably does not work. They then pour countless hours into trying to fix the code, even though they do not have a clear plan for what it is supposed to do. As they “fix” the code, it becomes a larger, more tangled mess. Eventually, the program sort-of-kind-of works, and they call it good enough.

Instead, you should devise an algorithm in a disciplined fashion. Figure 1.2 shows how you should approach designing your algorithm. We will spend the next few sections discussing each of these steps in detail. However, note that “translate to code” comes only after you have an algorithm that you have tested by hand—giving you some confidence that your plan is solid before you build on it.

If you plan well enough and translate it correctly, your code will just work the first time. If it
does not work the first time, you at least have a solid plan of what the code should be doing to guide your debugging.

### 1.3 Step 1: Work an Example Yourself

The first step in trying to design an algorithm is to work at least one instance of the problem—picking specific values for each parameter—yourself (by hand). Often this step will involve drawing a diagram of the problem at hand, in order to work it precisely. The more precisely you can perform this problem (including the more precisely you can draw a diagram of the situation if applicable), the easier the remainder of our steps will be. A good example of the sort of picture you might draw would be the diagrams drawn in many science classes (especially physics classes). Figure 1.2 shows multiple copies of the box for this step layered one on top of the other as you may need to perform this step multiple times to generalize the algorithm properly.

One of the examples of an algorithm that we mentioned early in this chapter was determining if a number is prime. If you were trying to write a function to determine if a number is prime, your first step would be to pick a number and figure out if it is prime. Just saying “ok, I know 7 is prime” is not of much use—you just used a fact you know, and did not actually work out the problem. For a problem such as this one, which has a “yes or no” answer, we probably want to work at least one example that comes up with a “yes” answer, and one that comes up with a “no” answer.

Another example would be if we wanted to write a program to compute $x$ raised to the $y$ power ($x^y$). To do Step 1, we would pick particular values for $x$ and $y$, and work them by hand. We might try $x = 3$ and $y = 4$, getting an answer of $3^4 = 81$.

If you get stuck at this step, it typically means one of two things. The first case is that the problem is ill specified—it is not clear what you are supposed to do. In such a situation, you must resolve how the problem should be solved before proceeding. In the case of a classroom setting, this resolution may require asking your professor or TA for more details. In an industrial setting, asking your technical lead or customer may be required. If you are solving a problem of your own creation, you may need to think harder about what the right answers should be and refine your definition of the problem.

The second case where Step 1 is difficult is when you lack domain knowledge—the knowledge of the particular field or discipline which the problem deals with. In our primality example, if you did not remember the definition of a prime number, that would be an example of lacking domain knowledge—the problem domain is mathematics, and you are lacking in math knowledge. No amount of programming expertise nor effort (“working harder”) will overcome this lack of domain knowledge. Instead, you must consult a source of domain expertise—a math textbook, website, or expert. Once you have the correct domain knowledge, you can proceed with solving your instance of the problem. Note that domain knowledge may come from domains other than math. It can come from any field, as programming is useful for processing any sort of information.

Sometimes, domain knowledge may come from particular fields of computer science or engineering. For example, if you intend to write a program which determines the meaning of English text, the relevant domain field is actually a sub-field of computer science, called Natural Language Processing. Here the domain knowledge would be the specific techniques developed to write programs which deal with natural language. A source of domain knowledge on English (an English professor or textbook) is unlikely to contain such information.
1.4 Step 2: Write Down What You Just Did

For this step, you must think about what you did to solve the problem, and write down the steps to solve that particular instance. Another way to think about this step, is to write down a clear set of instructions that anyone else could follow to reproduce your answer for the particular problem instance that you just solved. If you do multiple instances in Step 1, you will repeat Step 2 multiple times as well, once for each instance you did in Step 1. If an instruction is somewhat complex, that is all right, as long as the instruction has a clear meaning—later, we will turn these complex steps into their own programming problems, which will get solved separately.

The difficult part of Step 2 is thinking about exactly what you did to accomplish the problem. The difficulty here is that it is very easy to mentally gloss over small details, “easy” steps, or things that you do implicitly. This difficulty is best illustrated by the peanut butter and jelly exercise we mentioned earlier. Implicit assumptions about what to do, or relying on common sense lead to imprecise or omitted steps. The computer will not fill in any steps you omit, thus you must be careful to think through all the details.

Returning to our example of computing $x$ to the $y$, we might write down the following steps for $x = 3$ and $y = 4$.

Multiply 3 by 3
You get 9
Multiply 3 by 9
You get 27
Multiply 3 by 27
You get 81
81 is your answer.

The steps are very precise—and leave nothing to guess work. Anyone who can perform basic arithmetic can do follow these steps to get the right answer. Computers are very good at arithmetic, so none of these steps is even complex enough to require splitting into a sub-problem.

1.5 Step 3: Generalize Your Steps

Having solved one or more problems from the class we are interested in and written down the particular steps we executed to solve them, we are ready to try to generalize those steps into an algorithm. In our Step 2 steps, we solve particular instances, but now we need to find the pattern which allows us to solve the whole class. This generalization typically requires two activities. First, we must take particular values that we used and replace them with mathematical expressions of the parameters. Looking at our Step 2 steps for computing $3^4$, we would see that we are always multiplying 3 by something in each step. In the more general case, we will not always use 3—we are using 3 specifically because it is the value that we picked for $x$. We can generalize this slightly by replacing this occurrence of 3 with $x$:

Multiply $x$ by 3
You get 9
Multiply $x$ by 9
You get 27
Multiply $x$ by 27
You get 81
81 is your answer.

All of Programming, http://aop.cs.cornell.edu
Chapter 1: Introduction

The second common way to generalize steps is to find repetition—the same step repeated over and over. Often the number of times that the pattern repeats will depend on the parameters. We must generalize how many times to do the steps, as well as what the steps are. Sometimes, we may find steps which are almost repetitive, in which case we may need to adjust our steps to make them exactly repetitive. In our $3^4$ example, our multiplication steps are almost repetitive—both multiply $x$ by “something”, but that “something” changes (3 then 9 then 27). Examining the steps in more detail, we will see that the “something” we multiply is the answer from the previous step. We can then give it a name (and an initial value) to make all of these steps the same:

Start with $n = 3$
$n = \text{Multiply } x \text{ by } n$
$n = \text{Multiply } x \text{ by } n$
$n = \text{Multiply } x \text{ by } n$
$n$ is your answer.

Now, we have the same exact step repeated three times. We can now contemplate how many times this step repeats as a function of $x$ and/or $y$. We must be careful not to jump to the conclusion that it repeats $x$ times because $x = 3$—that is just a coincidence in this case. In this case, it repeats $y - 1$ times. The reason for this is that we need to multiply 4 3s together, and we already have one in $n$ at the start, so we need $y - 1$ more. This would lead to the following generalized steps:

Start with $n = 3$
Count up from 1 to $y - 1$ (inclusive), for each number you count,
$n = \text{Multiply } x \text{ by } n$
$n$ is your answer.

We need to make one more generalization of a specific value to a function of the parameters. We start with $n = 3$, however, we would not always want to start with 3. In the general case, we would want to start with $n = x$:

Start with $n = x$
Count up from 1 to $y - 1$ (inclusive), for each number you count,
$n = \text{Multiply } x \text{ by } n$
$n$ is your answer.

Sometimes you may find it difficult to see the pattern, making it hard to generalize the steps. When this happens, returning to Steps 1 and 2 may help. Doing more instances of the problem will provide more information for you to consider, possibly giving you insight into the patterns of your algorithm.

1.6 Step 4: Test Your Algorithm

After Step 3, we have an algorithm that we think is right. However, it is entirely possible that we have messed up along the way. The primary purpose of Step 4 is to ensure our steps are actually right before we proceed. To accomplish this, we test our algorithm with different values of the parameters than the ones we used to design our algorithm. We execute our algorithm by hand and compare the answer it obtains to the right answer. If they differ, then we know our algorithm is wrong. The more test cases (values of parameters) we use, the more confident we can become that
our algorithm is correct. Unfortunately, it is impossible to ensure that our algorithm is correct by
testing. The only way to be completely sure that your algorithm is correct is to formally prove its
correctness (using a mathematical proof), which is beyond the scope of this textbook.

One common type of mistake is mis-generalizing in Step 3. As we just discussed, one might
think that the steps repeated $x$ times because $x = 3$ and the steps repeated 3 times. If we had
written that down in Step 3, our algorithm would only work when $x = y - 1$, otherwise we would
count the wrong number of times, and get the wrong answer. If that were the case, we would
_hopefully_ detect the problem by testing our algorithm by hand in Step 4. When we detect such
a problem, we must go back and re-examine the generalizations we made in Step 3. Often, this
is best accomplished by returning to Steps 1 and 2 for whatever test case exposed the problem.
Redoing Steps 1 and 2 will give you a concrete set of steps to generalize differently. You can then
find where the generalization you came up with before is wrong, and revise it accordingly.

Another common type of mistake is that there are cases we did not consider in designing our
algorithm. In fact, in our $x^y$ example, we did not consider what happens when $y = 0$, and our
algorithm handles this case incorrectly. If you execute the algorithm by hand with $x = 2$, $y = 0$, you
should get $2^0 = 1$, however, you will get an answer of 2. Specifically, you will start with $n = x = 2$.
We would then try to count up from 1 to $0 - 1 = -1$, of which there are no numbers, so we would
be done counting right away. We would then give back $n$ (which is 2) as our answer.

To fix our algorithm, we would go back and revisit Steps 1 and 2 for the case that failed ($x = 2$,
$y = 0$). This case is a bit tricky since we just know that the answer is 1 without doing any work
($n^0 = 1$ for any $x$). The fact that the answer requires no work makes Step 2 a little different—we
just give an answer of 1. While this simplicity may seem nice, it actually makes it a little more
difficult to incorporate it into our generalized steps. We might be tempted to write generalized
steps like these:

If $y$ is 0 then
   1 is your answer
Otherwise:
   Start with $n = x$
   Count up from 1 to $y-1$ (inclusive), for each number you count,
      $n = \text{Multiply } x \text{ by } n$
      $n$ is your answer.

These steps check explicitly for the case that gave us a problem ($y = 0$), give the right answer
for that case, then perform the more general algorithm. For some problems, there may be corner
cases which require this sort of special attention. However, for this problem, we can do better.
Note that if you were unable to see the better solution, and were to take the above approach, it is
not wrong _per-se_, but it is not the best solution.

Instead, a better approach would be to realize that if we count no times, we need an answer
of 1, so we should start $n$ at 1 instead of at $x$. In doing so, we need to count 1 more time (to $y$
instead of to $y - 1$)—to multiply by $x$ one more time:

Start with $n = 1$
Count up from 1 to $y$ (inclusive), for each number you count,
   $n = \text{Multiply } x \text{ by } n$
   $n$ is your answer.

Whenever we detect problems with our algorithm in Step 4, we typically want to return to
Steps 1 and 2 to get more information to generalize from. Sometimes, we may see the problem
right away (e.g., if we made a trivial arithmetic mistake, or if executing the problematic test case by hand gives us insight into the correct generalization). If we see how to fix the problem, it is fine to fix it right away without redoing Steps 1 and 2, but if you are stuck, you should redo those steps until you find a solution. Whatever approach you take to fixing your algorithm, you should re-test it with all the test cases you have already used, as well as some new ones.

Determining good test cases is an important skill, which improves with practice. For testing in Step 4, you will want to test with cases which at least produce a few different answers (e.g., if your algorithm has a “yes” or “no” answer, you should test with parameters which produce both “yes” and “no”). You should also test any corner cases—cases where the behavior may be different from the more general cases. Whenever you have conditional decisions (including limits on where to count), you should test potential corner cases right around the boundaries of these conditions. For example, if your algorithm makes a decision based on whether or not \( x < 3 \), you might want to test with \( x = 2, x = 3, \) and \( x = 4 \). You can limit your “pencil and paper” testing somewhat, since you will do more testing on the actual code once you have written it.

1.7 Some Examples

Having learned the basic steps of designing an algorithm, it is useful to see several examples of them in action. We are going to work through four examples in a variety of forms. For two of the examples, we will work from a problem statement (that is a description of what the algorithm should do). For the other two examples, we will work from a set of examples that illustrate the pattern. Writing algorithms from either starting point is a useful skill for programmers, and the two skills are tightly interlinked—in both cases, we are trying to find the general solution/pattern, and write it down clearly.

1.7.1 A Numerical Sequence

For our first example, we will find the pattern from a set of given examples:

The numbers in the table below are the result of executing an algorithm which has one parameter, \( N \), which must be a non-negative integer, and produces sequences of integers as outputs. For values of \( N \) from 0 to 5, the algorithm produces the following sequences of numbers as outputs:

<table>
<thead>
<tr>
<th>( N )</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1 -1</td>
</tr>
<tr>
<td>2</td>
<td>4 2 0</td>
</tr>
<tr>
<td>3</td>
<td>9 7 5 3</td>
</tr>
<tr>
<td>4</td>
<td>16 14 12 10 8</td>
</tr>
<tr>
<td>5</td>
<td>25 23 21 19 17 15</td>
</tr>
</tbody>
</table>

We start with this Numerical Sequence example, since it is the simplest of the four, and the key skill—being able to analyze sequences of numbers to find patterns—is required for pretty much any other algorithm. In many algorithms you will design, you will find a sequence of numbers (after all: everything is a number), in which you need to find the pattern to describe the generalization.

Step 1: Work An Example Yourself. Unlike working a problem from a description, we have several examples already given to us—6 examples are already worked, so we have 6 “Step 1”s already done for us. However, it is still useful to walk through a couple in detail to have them in your head. Even just copying the numbers down directly helps bring them into your active memory.
Step 2: Write Down What You Just Did. In this particular case, there really isn’t anything more to what you did than just writing down the sequence of numbers. That is, for $N = 1$, $N = 2$, and $N = 3$, we might write down:

<table>
<thead>
<tr>
<th>N=1</th>
<th>N=2</th>
<th>N=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>I wrote down 1.</td>
<td>I wrote down 4.</td>
<td>I wrote down 9.</td>
</tr>
<tr>
<td>I wrote down -1.</td>
<td>I wrote down 2.</td>
<td>I wrote down 7.</td>
</tr>
<tr>
<td></td>
<td>I wrote down 0.</td>
<td>I wrote down 5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I wrote down 3.</td>
</tr>
</tbody>
</table>

Step 3: Generalize Your Steps. We clearly have repetition in these steps (we are, after all, writing down a sequence of numbers), but a lot of things depend on $N$. For each value of $N$, we are writing down a different number of numbers in the sequence. We might first figure out how many numbers we will write down for a given value of $N$. If the pattern is not immediately obvious, it may help to make a table like this one:

<table>
<thead>
<tr>
<th>N</th>
<th>How many numbers in the sequence?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Now we can see that each sequence has $N + 1$ numbers in it. We therefore want to repeat a step of the form “Write down (some number)” $N + 1$ times, where we need to figure out the formula for “(some number).” Since we want to repeatedly do a similar step, it is natural to “count them out”—that is, count from 0 to $N$ (inclusive; or 0 to $N + 1$ if we want to exclude the upper bound), name the number we are counting on, and then say what to do for each number we count. Let us count from 0 to $N$ (inclusive), and call each number that we count $i$.

Now, we need to figure out the formula for each number that we need to write in terms of $N$ and $i$. If we start by looking at $N = 4$ (for example; we could pick any of them):

16 14 12 10 8

We will see that the numbers decrease by 2 each time. We go from 16 to 14 (2 less), then to 12 (again, 2 less), and so on. For $N = 4$ in particular, we can come up with a formula of $16 - 2\times i$. If we go back to the entire problem, and look at each $N$ we were given, we might come up with the following table of formulas:

<table>
<thead>
<tr>
<th>N</th>
<th>Formula for the $i^{th}$ number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$1 - 2 \times i$</td>
</tr>
<tr>
<td>2</td>
<td>$4 - 2 \times i$</td>
</tr>
<tr>
<td>3</td>
<td>$9 - 2 \times i$</td>
</tr>
<tr>
<td>4</td>
<td>$16 - 2 \times i$</td>
</tr>
<tr>
<td>5</td>
<td>$25 - 2 \times i$</td>
</tr>
</tbody>
</table>

These formulas are pretty similar, except for two things. First, 0 is a bit of an “odd man out”—it only has one number (0), and thus we did not naturally come up with a $-2\times i$ on it, which
the other formulas have. However, since we start counting at \( i = 0 \), \( 2 \times 0 = 0 \), and anything minus 0 is itself, we can just add the missing \(-2 \times i\) to this formula without changing anything, giving us \( 0 - 2 \times i \) as the formula for \( N = 0 \).

Now, all of the formulas look the same, except that they start with a different number. That is, they are all \((\text{something}) - 2 \times i\), but that (something) differs. If we can figure out the pattern in the (something), we have a general formula, and just need to write down the steps. Scrutinizing the relationship between these numbers, we can see that the formula is \( N^2 - 2 \times i \).

Count from 0 to \( N \) (inclusive), call each number you count \( (i) \) and
Write down the number \((N \text{ squared } - 2 \times i)\)

If you cannot play these videos, your pdf reader does not support videos.

**Video 1.1: Testing our algorithm**

**Step 4: Test Your Algorithm.** Now that we have written this algorithm, we should test it. It is entirely possible that we made a mistake in generalizing the patterns. If we did, we would prefer to catch the problem now. If we write the code, then discover the problem, we have wasted work. We test the algorithm by picking a particular \( N \), and following the steps. If we used particular values of \( N \) to generalize from, we want to pick other values to test with. Video 1.1 demonstrates testing of these steps for \( N = 5 \).

After testing our algorithm by executing it step-by-step as in the video, we are more confident in its correctness. We know that our algorithm produces the right answer on at least one input (\( N = 5 \)). Of course, we cannot be completely sure our algorithm is right from testing (in general, no number of tests can assure correctness). However, the more we test our algorithm, the more confident we can become in it. Only once we are completely happy with our algorithm should we then proceed to translate it to code.

### 1.7.2 A Pattern of Squares

For our second example, we will look at a pattern of squares drawn on a grid. You may wonder why a programmer would be interested in drawing squares on a grid. Beyond this example serving us well for analyzing patterns in general, computer graphics ultimately boil down to drawing colored pixels on a 2D grid (the screen). In this particular example, we have an algorithm which is parameterized...
over one integer, $N$, and produces a pattern of red and blue squares on a grid that starts all white. The output of the algorithm for $N = 0$ to $N = 5$ is as follows:

![Grids showing the output of the algorithm for $N = 0$ to $N = 5$.](image)

To devise the algorithm, we should work through steps 1–4 (as we should with all problems). Video 1.2 walks through these steps to illustrate how we could come up with this algorithm.

We note that there are many correct algorithms for this problem. Even if we restrict ourselves to the ones we can come up with naturally (that is, as a result of working through steps 1–4, rather than trying something bizarre), there are still many choices which are equivalent and correct. Which algorithm you come up with would be determined by how you approach step 1.

The algorithm in the video works from left to right, filling in each column from bottom to top. If we had worked step 1 by working from the top down, filling in each row from left to right, we might have ended up with the following slightly different algorithm instead:

**Count down from $N$ to 0 (inclusive), call each number you count "y" and**

**Count from 0 to $y$ (inclusive), call each number you count "x" and**

- if $(x + y$ is multiple of 3)
  - then place a blue square at $(x,y)$
- otherwise place a red square at $(x,y)$

Of course, those are not the only two ways. You could have worked across the rows from the bottom up going right to left, and come up with a slightly different (but also equivalent) algorithm. Or even an entirely different approach, such as filling in the entire “triangle” with red squares, then going back to fill in the blue squares.

We emphasize this point, because it is important for you to understand that there is always more than one right answer to a programming problem. You might work a problem and come up with a correct solution, but find that it looks completely different from some other solution you
If you cannot play these videos, your pdf reader does not support videos.

Chapter 1: Introduction

1.7.3 Drawing a Rectangle

For our third example, we will work from the following problem statement:

Given $x$, $y$, width, and height, draw a blue filled-in $width \times height$ rectangle whose lower left corner is at $(x, y)$ on a 20x20 grid.

Like the last problem, this one involves drawing something on a 2D plane. As with the last one, this problem is one we might want to do in graphics—however, here, we are working the problem from a description of what we want to do, rather than from some examples. We might end up working a problem that is conceptually quite similar in a non-graphical context—if we represent data in a 2D grid (e.g., a table or matrix), we might want to set a rectangular range of data values to something in particular. Even though this second problem “looks” different, the algorithm would be quite similar. Video 1.3 walks through this problem.

1.7.4 Closest Point

For our fourth example, we will work from the following problem statement:

Given a set $S$ of $(x, y)$ points, and another point $P$, select the point from $S$ which is closest to $P$.
Chapter 1: Introduction

We might want to write a program to solve this problem for a variety of reasons. Some of
the most natural reasons would be that we actually want to apply this to physical locations. For
example, if we are writing a program that deals with maps and directions, we might want to locate
the gas station nearest to the user. The software could then find all of the gas stations in the area
\((S)\), query the GPS for the user’s location \((P)\), and then apply this algorithm to determine which
gas station is closest to the user.

However, this problem also has a variety of applications which are slightly more subtle. For
example, suppose we have some information describing the characteristics an item, and we want to
classify it based on the similarity of those characteristics to preexisting categories of items. If we
have two characteristics, then the first is the x-coordinate, and the second is the y-coordinate. If
we have more than two, we solve the problem in a higher-dimensional space (which is still the same
problem, we just compute distance differently). We then use the points describing the categories
that we desire as our \(S\), and the characteristics of the item we are classifying as our \(P\). The closest
item in \(S\) is the best match for \(P\). For example, suppose the key features in determining the breed
of a dog were its adult height (the x-coordinate) and its adult weight (the y-coordinate). If you
have a set \(S\) of representative dog breeds and you’d like to know the breed of your dog, \(P\), you
simply determine which member of \(S\) your dog is closest to.

Video 1.4 walks through the algorithmic design for this problem. We will revisit this problem
much later in Video ??, where we will turn this algorithm into C code.

All of Programming, http://aop.cs.cornell.edu
Chapter 1: Introduction

If you cannot play these videos, your pdf reader does not support videos.

Video 1.4: Devising an algorithm to find the closest point in a set $S$ to another point $P$.

1.8 Next Steps

At this point, you should have a basic grasp on the idea of developing simple algorithms. This skill is one that you will practice as you go through the rest of this book, as every programming problem’s key component is figuring out the correct algorithm. We cannot underscore enough the important of working through problems in a step-by-step fashion. Many novice programmers try to skip over the first several steps, and plunge right into writing code. The result is frequently a disaster, which they end up spending orders of magnitude more time trying to fix than they would have spent planning correctly in the first place.

The reasons that novice programmers give for skipping straight to step 5 vary, but one common one is “Step 3 (writing a generalized algorithm) seemed too hard.” This reason is quite possibly the worst reason to skip over Step 3—if making a correct plan is proving hard, how can you possibly hope to write correct code without the plan? Better is to repeat Steps 1 and 2 on more examples until you can find the pattern and write down the algorithm. Another common reason that novice programmers give for skipping the first steps is “to save time”—however, they often then report spending countless hours trying to debug the resulting code. It is well worth 10 or even 30 minutes of planning to avoid trying to debug a hopeless mess for multiple hours!

As you become more and more practiced at this process, you may find that steps 1–4 come naturally, and you can do them in your head without writing them down—much like may happen with mathematical skills. When these improvements in your programming skills happen, then there is nothing wrong with doing the easier steps in your head, as long as you are sure that you are
doing them correctly. However, whenever you are programming at the boundaries of your abilities, you will need to go through these steps—so it is quite important to remember how the full process works even as you become more skilled.

We will continue from here by first learning a bit about reading code in C, before we continue on to more about writing code. By reading code, we mean being able to understand exactly what a piece of code does, executing it step-by-step by hand. This skill is important for three reasons. First, it is very difficult to write when you cannot read. Reading the code will be a matter of drawing and updating diagrams which reflect the state of the program as the code executes. Writing code will be a matter of writing the syntax to effect the appropriate transformations—as spelled out in the algorithm—to the program’s state. Second, being able to read your code is crucial for being able to debug your code. Third, you may end up in a variety of settings (e.g., group class projects, coding teams in industry) where you must end up in a variety of settings (e.g., group class projects, coding teams in industry) where you must read and understand what other people’s code does so that you can work on it.

1.9 Practice Exercises

Selected questions have links to answers in the back of the book.

- Question 1.1: What is an algorithm?
- Question 1.2: What is a parameter?
- Question 1.3: How many tests are needed to ensure that an algorithm is correct?
- Question 1.4: The numbers in the table below are the result of executing an algorithm which has one parameter, N, which must be a non-negative integer, and produces sequences of integers as outputs. For values of N from 0 to 5, the algorithm produces the following sequences of numbers as outputs:

<table>
<thead>
<tr>
<th>N</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 2</td>
</tr>
<tr>
<td>1</td>
<td>3 5 7 9</td>
</tr>
<tr>
<td>2</td>
<td>6 8 10 12 14 16</td>
</tr>
<tr>
<td>3</td>
<td>9 11 13 15 17 19 21 23</td>
</tr>
<tr>
<td>4</td>
<td>12 14 16 18 20 22 24 26 28 30</td>
</tr>
<tr>
<td>5</td>
<td>15 17 19 21 23 25 27 29 31 33 35 37</td>
</tr>
</tbody>
</table>

Determine the algorithm that was used to generate the numbers in this table and

1. Write it down
2. Execute it for N=6, and write down your result.
3. Give your description of the algorithm to a friend who is not a programmer, and ask him/her to execute it for N=6. Compare your results to his/hers.
Question 1.5: The diagrams shown below are the result of executing an algorithm which has one parameter, \( N \), which must be a non-negative integer, and colors boxes on a 10x10 grid. For values of \( N \) from 0 to 5, the algorithm produces the following patterns:

Determine the algorithm that was used to draw these patterns and

1. Write it down
2. Execute it for \( N=6 \), and write down your result (possibly on graph paper).
3. Give your description of the algorithm to a friend who is not a programmer, and ask him/her to execute it for \( N=6 \). Compare your results to his/hers.
Chapter 1: Introduction

• Question 1.6: The numbers in the table below are the result of executing an algorithm which has one parameter, N, which must be a non-negative integer, and produces sequences of integers as outputs. For values of N from 0 to 5, the algorithm produces the following sequences of numbers as outputs:

<table>
<thead>
<tr>
<th>N</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
</tr>
<tr>
<td>2</td>
<td>0 2 2 3</td>
</tr>
<tr>
<td>3</td>
<td>0 2 4 3 4 5</td>
</tr>
<tr>
<td>4</td>
<td>0 2 4 6 4 5 6 7</td>
</tr>
<tr>
<td>5</td>
<td>0 2 4 6 8 5 6 7 8 9</td>
</tr>
</tbody>
</table>

Determine the algorithm that was used to generate the numbers in this table and

1. Write it down
2. Execute it for N=6, and write down your result.
3. Give your description of the algorithm to a friend who is not a programmer, and ask him/her to execute it for N=6. Compare your results to his/hers.

• Question 1.7: The diagrams shown below are the result of executing an algorithm which has one parameter, N, which must be a non-negative integer, and colors boxes on a 10x10 grid. For values of N from 0 to 5, the algorithm produces the following patterns:

Determine the algorithm that was used to draw these patterns and
Chapter 1: Introduction

1. Write it down
2. Execute it for N=6, and write down your result (possibly on graph paper).
3. Give your description of the algorithm to a friend who is not a programmer, and ask him/her to execute it for N=6. Compare your results to his/hers.

• Question 1.8: The numbers in the table below are the result of executing an algorithm which has one parameter, N, which must be a non-negative integer, and produces sequences of integers as outputs. For values of N from 0 to 5, the algorithm produces the following sequences of numbers as outputs:

<table>
<thead>
<tr>
<th>N</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1 0 3</td>
</tr>
<tr>
<td>1</td>
<td>-4 -3 0 5 12 21</td>
</tr>
<tr>
<td>2</td>
<td>-9 -8 -5 0 7 16 27 40 55</td>
</tr>
<tr>
<td>3</td>
<td>-16 -15 -12 -7 0 9 20 33 48 65 84 105</td>
</tr>
<tr>
<td>4</td>
<td>-25 -24 -21 -16 -9 0 11 24 39 56 75 96 119 144 171</td>
</tr>
</tbody>
</table>

Determine the algorithm that was used to generate the numbers in this table and

1. Write it down
2. Execute it for N=6, and write down your result.
3. Give your description of the algorithm to a friend who is not a programmer, and ask him/her to execute it for N=6. Compare your results to his/hers.
Before you can learn to write, you must learn to read. In the case of code, learning to read
means being able to take a piece of code, and execute it by hand in a step-by-step fashion. Correctly
eexecuting a piece of code by hand will allow you to determine the effects that the code will produce.
If you cannot understand what code does, you cannot possibly write it.

There are two important parts to executing code by hand. The first is understanding what
each statement does. The rest of the chapter will cover the basic statements of C, and we will
see more advanced topics later on. As we introduce these basic C constructs, we will explain
their syntax—the rules for how to write them according to the grammatical rules of C—and their
semantics—what they mean (i.e., what they make the program do). We suggest that you make
yourself a “quick reference” sheet, where you write down the syntactic rules and effects of each
construct as you learn it. You will want to refer to these later as you begin to write code (although,
after some practice, they will start to come naturally to you).

The second important part is keeping track of the state of the program in a correct fashion—that
is, what part of the code is currently being executed, as well as the values of the variables involved.
As we will see shortly, tracking the values of variables is not quite as simple as listing their names
and values (as we will sometimes make new variables, and destroy old ones). Accordingly, we will
keep a diagram in a particular way that allows us to track this state correctly and easily. We will
track our current location in the code with an arrow (→), which will typically rest between lines
of code. It will be after the last line we have executed, and before the next line we are going to
execute. Our basic process will be to evaluate the line of code after the execution arrow (according
to the rules we will learn here), updating our diagram appropriately. We will then advance the
arrow to the next line of code, and repeat the process, until the program exits.

As you work through this chapter, you will hopefully notice some similarities between our
executing code by hand, and the way in which we tested algorithms (Step 4 of various problems)
in the previous chapter. This similarity is not a coincidence, as code is just a formal way to
express an algorithm such that the computer can understand how to follow the steps. Whenever
you execute algorithms by hand—no matter what language they are in: English, C, C++, Java,
SML, Python,...—you will want to follow a very similar procedure so that you capture the effects

<table>
<thead>
<tr>
<th>2.1 Variables</th>
<th>2.2 Expressions</th>
<th>2.3 Functions</th>
<th>2.4 Conditional Statements</th>
<th>2.5 Shorthand</th>
<th>2.6 Loops</th>
<th>2.7 Higher-level Meaning</th>
<th>2.8 Practice Exercises</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 2

Reading Code

22
Chapter 2: Reading Code

precisely.

2.1 Variables

Programs track most of their state in variables—which you can think of as a box that stores a value. In order to use a variable, the programmer must declare it, specifying its type and name. The type specifies what kind of value can be held in a variable’s box (for example, whether it is a number, a letter, or text). We will learn about types in Chapter 3, but for now, we will use variables whose types are all int—meaning that the value in their box is a number.

2.1.1 Declaration

The name of a variable may be any valid identifier. An identifier is the formal programming term for a word that can be used to name something. In C, identifiers may be any length, and can contain letters, numbers, and underscores (_). They may only start with a letter or an underscore (not a number), and are case-sensitive (meaning that abc is different from Abc and ABC is different from both of them). The variable declaration ends with a semicolon—which is used to end many statements in C. A statement in a programming language is roughly analogous to a sentence in English—it is a complete line of code which can be executed for an effect. Figure 2.1 shows a variable declaration, and identifies each of the pieces.

When executing code by hand, the effect of a variable declaration is to create a new box, labeled with the name of the variable. In C, a newly declared variable is uninitialized, meaning that its value is undefined. When the computer actually executes the program, it has a finite (but quite large) number of “boxes” (memory locations), and the variable will be given one that is currently not in use. The value of the variable will be whatever value happened to be in the location previously, until a new value is assigned to the variable (which we will see shortly). Correspondingly, when we execute a variable declaration by hand, we will draw a box and place a ? in it for its value—indicating that it is unknown. If we ever use an unknown value as we execute our program, it indicates a problem with our program, since its behavior is undefined—its behavior will changed based on whatever the value actually is, which we cannot predict.

Video 2.1 shows the execution of code containing two variable declarations—x and y. At the start, the execution arrow is at the beginning of the code. The area on the right—which represents the state of the program—is empty. As the execution arrow advances across these statements, we execute their effects: drawing a box for each variable, with a ? in it, indicating that the variable is uninitialized.

2.1.2 Assignment

For variables to be useful, we must be able to change their values. To accomplish this, we use
**assignment statements**—statements which change the value contained in a box. An assignment statement starts with an *lvalue* on the *left*. An lvalue (pronounced “el-value”) must be something that “names a box”—indicating which box the assignment statement will change. The simplest lvalue is a variable, which names the variable’s own box. (Later we shall see how to name boxes in other ways, but for now, we will only consider variable names.) After the lvalue, comes a single equals sign (called the assignment operator), followed by an *rvalue* on the *right*, then a semicolon. The rvalue (pronounced “are-value”) must be an *expression* whose value shall be placed in the box.

An expression is a combination of values, and operations which evaluates to a value. For the moment, we will just consider numeric constants (such as 3), which evaluate simply to themselves (that is, 3 evaluates to the number 3). We will discuss more expressions shortly. Evaluating any assignment statement is a matter of figuring out what box the left side names, evaluating the right side to a value (e.g., a number), and then changing the value in the box named on the left side to the value from the right side.

Figure 2.2 show an example of an assignment statement, and identifies the individual pieces. This assignment statement assigns the value 3 to the variable `myVariable`. Its effect is to change the value in the box named `myVariable` to be 3.

The declaration and *initialization*—the first assignment—of a variable may be combined into a single statement, such as `int x = 3;` which has the same effect as the two individual statements `int x; x = 3;`. Video 2.2 shows the execution of a combination of variable declarations and assignment statements.

### 2.2 Expressions

As we mentioned previously, an expression is a combination of values and operations which evaluates to a value. We have already seen the simplest expressions—numerical constants, which evaluate to themselves. We can also use mathematical operators, such as `+`, `-`, `*`, and `/` to carry out arithmetic operations. For example, `7 + 3` evaluates to 10 and `4 * 6 + 9 * 3` evaluates to 51. These operators have the standard rules of *precedence*—multiplication and division occur before addition and subtraction—and *associativity*: `4 - 3 - 1` means `(4 - 3) - 1` not `4 - (3 - 1)`. Parenthesis may be used to enforce a specific order of operations—`4 * (6 + 9) * 3` evaluates to 180.

Another common operator which you may not be as familiar with is the *modulus* operator, `%`. The modulus operator evaluates to the remainder when dividing the first operand by the second. That is `a % b` (read “a modulus b”, or “a mod b” for short) is the remainder when `a` is divided by `b`. For example, `19 % 5 = 4` because `19/5 = 3` with a remainder of `4`, and `19 - 15 = 4`.

Variables may also appear in expressions. When a variable appears in an expression, it is evalu-
ated to a value by reading the current value out of its box. It is important to note that assignment statements involving variables on the right side are not algebraic equations to be solved—we cannot write \( x - y = z \times q \). Note that here, the left side of this statement does not “name a box”. If you want to solve an algebraic equation, you must do so in a step-by-step fashion.

We can, however, write perfectly meaningful assignment statements which are not valid in algebra. For example, a statement such as \( x = x + 1 \); is quite common in programming, but has no solution if you think of it as an algebraic equation. In programming, this statement means to take current value of \( x \), add 1 to it, and update \( x \)’s value to whatever that result is.

Video 2.3 shows the execution of some assignment statements with more complex expressions on their right-hand sides than in previous examples.

### 2.3 Functions

A function gives a name to a parameterized computation—it is the implementation in code of a specific algorithm. All of the code that you will read or write (in this book) will be inside of functions. There are two sides to using functions in your programming: declaring a function—which provides the definition for how a function behaves—and calling a function—which executes the definition of the function on specific values of the parameters.

Figure 2.3 shows a function declaration. The function’s name may be any valid identifier, just like a variable’s name. In this particular example, the function’s name is \( \text{myFunction} \). Immediately before the function’s name is its return type—the type of value that this function will compute. As mentioned earlier, we will learn more about types later. For now, we will just work with \( \text{ints} \), which are numbers. The fact that this function returns an \( \text{int} \) means that its “answer” is an \( \text{int} \). After the function’s name comes a set of parenthesis, with the parameter list inside. The parameter list looks like a comma separated list of variable declarations. Here, the function takes two parameters, \( x \), and \( y \), both of which are \( \text{ints} \). The similarity between parameters and variable declarations is not a coincidence—the parameters behave much like variables, but they are initialized by the function call (which we will discuss shortly).

The body of the function then comes between a set of curly braces, and is comprised of zero or more statements. The body of this function has two statements. The first statement in this function’s body is the now familiar declaration and initialization of a variable: \( z \) is declared as a variable of type \( \text{int} \), and initialized to the value of the expression \( x - 2 \times y \).

The second statement within the body of this function is a new type of statement which we have not seen before: a return statement. A return statement starts with the keyword return, which is then followed by an expression. The effect of this statement is to say what the “answer” is for the current function, leaving its computation and returning to the code which called it.

To understand this last concept completely, we must first see the other aspect of using a function—calling the function. A function is another kind of expression, whose value is whatever “answer” the called function comes up with when it is executed with the specified arguments—
values of its parameters. This “answer” is more formally called the function’s return value.

Evaluating a function call is more complex than evaluating the other kinds of expressions that we have seen so far—it may take many steps of executing the code in the function to determine its answer. In fact, code may call one function, which itself may call other functions before finally coming up with an answer. While this may seem daunting, we can do it properly by following a few rules for executing function calls by hand.

As a first step towards reading code with function calls, we must first group together the variables belonging to one particular function into a larger box, labeled with the function’s name, which is called a frame (or stack frame, since they are located on the call stack). Figure 2.4 shows an example of this organization.

Notice that in the example of Figure 2.4, one of the functions is named main. The function named main is special—execution of a program starts at the start of main. We start by drawing an empty frame for main, and putting the execution arrow right before the first line of code in main. We then execute statements of the code until main returns, which ends the program.

Calls to functions may appear in expressions, in which case we must evaluate the function to determine its return result. To do this evaluation, we take the following steps:

1. Draw a frame for the function being called. Place a box in that frame for each parameter that this function takes.

2. Initialize the parameters by evaluating the corresponding expressions in the function call, and copying the resulting values into the parameter’s box in the called function’s frame.

3. Mark the location of the function call, and note that location in the corner of the function’s frame.

4. Move the execution arrow immediately before the first line of code in the called function.

5. Evaluate the lines of code inside the called function.
6. When you reach a `return` statement, evaluate its argument to a value. Note down this return value.

7. Return the execution arrow back to where the function was called—you know this location because you noted it in the corner of the frame. You will return the arrow to the middle of the line of code (rather than the typical “between them”) because that line of code is part-way done.

8. Erase the frame for the called function.

9. Use the return value of the function as the value of the function call in the expression in which it appears.

A function call may also be used as a statement by itself, in which case, it is evaluated the same as above, except that its return value is not used for anything.

If you cannot play these videos, your pdf reader does not support videos.

Video 2.4: Execution of function calls

Video 2.4 demonstrates the execution of code with function calls.

### 2.3.1 Scope

So far, all of our code examples have had only one variable with a particular name. However, in real programs—which may be quite large and developed by multiple people—we may have many different variables with the same name. This possibility means that we need rules to determine which variable a particular name refers to. These rules are based on the notion of *scope*.

The scope of a variable is the region of code in which it is visible. Within a variable’s scope, its name may refer to it. Outside of a variable’s scope, nothing can refer to it directly. Most variables that you will use will be *local variables*—variables which are declared inside of a function—and
function parameters. In C, the scope of a local variable begins with its declaration and ends at the closing curly-brace (}) which closes the block of code—the code between matching open and close curly braces—that the variable was declared in. Function parameters have a scope of the entire function which they belong to.

Figure 2.5 shows a snippet of code (we have not learned the details of what most of this code does, but that is not important—we are just interested in the scope of the variables). The figure shows the same piece of code three times, with different scopes highlighted. The leftmost portion of the figures shows the scope of the parameters (x and y)—which is the entire function—in a blue box. The middle portion shows the scope of the variable n—which starts at its declaration and continues to the close curly brace which ends the function—in a red box. The right portion shows the scope of the variable q—which starts at its declaration and ends at the next curly brace—in a green box.

To determine which variable a name refers to, we must first determine which variable(s) with that name are in scope at the reference. If no variables of that name are in scope, then the reference is illegal. If exactly one variable is in scope, then the name refers to that variable. If multiple variables are in scope, we select the one whose declaration is in the innermost enclosing block. That is, if you went backwards out of blocks, through open curly braces, the variable which would go out of scope first is the one to use.

Figure 2.6 shows a code fragment with four different xs in it. (As the actual behavior of the code is irrelevant to this example, much of it is replaced with . . . .) The first x in the figure is declared outside of any of the functions—it is a global variable. The “box” for a global variable exists outside of any frames, and is created when the program starts. If the global variable is initialized in its declaration, the value is also placed in the box before the program starts. The areas where x references this variable are colored purple.

We note that there is a time and place to use global variables, but their use should be rare. When novice programmers learn about global variables, they often want to use them for all sorts of inappropriate purposes. Typically these uses reflect a lack of understanding of parameter passing or how functions return values. We recommend against using global variables for any problem in this book, and more generally unless it is truly the correct design approach.

The next x in our example is the parameter to the function f. The scope for this x begins at the
open curly brace (\{) of f’s body, and ends at the matching close curly brace (\}). The region of the
program where x references the parameter to f are shown in red. Observe that the red begins and
ends with the curly braces surrounding the body of f, but has a “hole” where there is a different
x in a smaller scope in the middle.

The “hole” in the red region corresponds to the portion of the code (shown in blue) where x references the local
variable declared inside of the while loop’s body. After this local variable x goes out of scope at the closing curly
brace of the block it was declared in, we return to the “red region” where the parameter of f is what we reference with
the name x.

Between the end of f and the declaration of a local variable named x inside of function g, the global variable
is what the name x references—shown in the figure by coloring this region of code purple. When there is a local
variable named x declared inside of g, then the name x references it (this area is shown in green) until it goes out
of scope, at which point the name x again references the global variable.

If all of that seems complicated, you will be comforted by the fact that thinking through such issues should not
come up in well-written code. Ideally, you should write your code such that you have at most one variable by any
particular name in scope at a time (related to this point: you should name your variables meaningfully—x is seldom
a good name for a variable, unless of course it represent the x coordinate of a point or something
similar). However, you should still know what the rule is, as it is common to many programming
languages. You may come across code which has multiple variables of the same name in scope at
some point, and need to understand how to read it.

### 2.3.2 Printing

Our example programs so far have computed results, but had no way to communicate them to the
user. Such programs would be useless in practice. Real programs have means to communicate with
their user, both to read input and to provide output. Many programs that you are accustomed to
have Graphical User Interfaces (GUIs), however, we will work primarily with programs that use a
command line interface. Writing GUIs is a more complex task, and requires a variety of additional
concepts.

Command line programs provide output to their user by **printing** it out on the terminal. In
C, printing output is accomplished by calling the `printf` function, which takes a string specifying
what to print. We will learn more about strings later (in Section ??), as they require knowledge
of pointers to understand. For now, you can think of them as being text—words, or sentences. In
much the same way that we can write down numerical literals (such as 3, or −42), we can write down
string literals by placing the string we want inside of quotation marks, e.g., "This is a string". If we wanted to print out the string, “Hello World”, we would type `printf("Hello World");`

The f in `printf` stands for “formatted,” meaning that `printf` does not just print literal strings,
but can take multiple arguments (of various types), format the output as a string, and print the
result. To format output in this way, the string argument of `printf` (which is called the “format
string”) includes special format specifiers, which start with a percent sign (%). For now, we will only concern ourselves with %d which specifies that an integer should be formatted as a decimal (base 10) number. For example, if we wrote the following code fragment:

```c
int x = 3;
int y = 4;
printf("x + y = %d", x + y);
```

it would print `x + y = 7`, because it would evaluate the expression `x + y` to get `3 + 4` which is `7`, and format the number `7` as a decimal number in place of the `%d` specifier. The rest of the string is printed literally.

Another type of special information we can include in the string is escape sequences. Escape sequences are two (or more) characters, the first of which is a backslash (\), which gives the remaining characters special meaning. The most common escape sequence you will encounter is \n, which means “newline”. If you want your print statement to print a newline character (which makes the next output begin at the start of the next line), then you do so with \n. If you want a literal backslash (that is, you actually want to print a backslash), \\ is the escape sequence for that purpose. We will note that you generally will want to print a newline in your output, not only so that it looks nice, but also because printf does not actually print the output to the screen until it encounters a newline character, or is otherwise forced to do so.

We will discuss the various format specifiers which printf accepts, as well as the escape sequences that C understands in Chapter 3.
Chapter 2: Reading Code

<table>
<thead>
<tr>
<th>expr1 == expr2</th>
<th>tests if expr1 is equal to expr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr1 != expr2</td>
<td>tests if expr1 is not equal to expr2</td>
</tr>
<tr>
<td>expr1 &lt; expr2</td>
<td>tests if expr1 is less than expr2</td>
</tr>
<tr>
<td>expr1 &lt;= expr2</td>
<td>tests if expr1 is less than or equal to expr2</td>
</tr>
<tr>
<td>expr1 &gt; expr2</td>
<td>tests if expr1 is greater than expr2</td>
</tr>
<tr>
<td>expr1 &gt;= expr2</td>
<td>tests if expr1 is greater than or equal to expr2</td>
</tr>
<tr>
<td>!expr</td>
<td>computes the logical NOT of expr</td>
</tr>
<tr>
<td>expr1 &amp;&amp; expr2</td>
<td>computes the logical AND of expr1 and expr2</td>
</tr>
<tr>
<td>expr1</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Logical Operators

Video 2.5 demonstrates the execution of code which prints output.

2.4 Conditional Statements

In addition to computing arithmetic combinations of their variables, programs often make decisions based on the values of their variables—executing different statements based on the value of expressions. In C, an if/else statement specifies that one block of code should be executed if a condition is true, and another block should be executed if that condition is false.

To writing meaningful if/else statements, we need to introduce operators which allow us to compare two expressions to produce true or false outcomes. In C, there are no distinct values for true or false, instead, false is 0, and anything which is non-zero is true. We will refer to true and false because they make more sense conceptually; the distinction should not make a practical difference in most cases.

Table 2.1 shows the C operators for conditional expressions. The first six (==, !=, <=, <, >, and >=) are relational operators—they compare two expressions for equality or inequality. For any of these operators, both operands (the expressions on the left and right) are evaluated to a value, then compared appropriately. The operator then produces a true or false value.

The last three operators in the table (!, &&, and ||) are boolean operators—they operate on true/false values. The first of these, ! performs the boolean NOT operation. It is a unary operator—meaning that is has one operand—which evaluates to true if its operand is false, and evaluates to false if its operand is true.

The && and || operators perform the logical AND and logical OR operations respectively. The logical AND of two values is true if and only if both values are true, otherwise it is false. The logical OR of two values is true if and only if either of the values are true, otherwise it is false.

Unlike previous operators that we have seen, && and || may know their answer from only one argument. In the case of &&, if either operand is false, then the result is false, regardless of the other value. Similarly for ||, if either operand is true, then the result is true regardless of the other value. C exploits this fact in the way that it evaluates && and || by making them short circuit—they may only evaluate one operand. Specifically, the first operand is always evaluated to a value; however, if the value of that operand determines the result of the entire && or ||—false for && or true for ||—then the second operand is not evaluated at all.

2.4.1 if/else
Now that we understand comparison operators, and can compare expressions, we can discuss the evaluation of if/else statements. The syntax for an if/else statement is shown in Figure 2.7. The keyword if is followed by an expression in parenthesis. This expression is evaluated to a value, to determine whether the “then” block or the “else” block is executed. The “then” block of code comes immediately after the expression. C does not have a then keyword (although some languages do), however, this block of code serves the same purpose regardless of the syntactic particulars of the language—it is executed if the conditional expression evaluates to true. After the “then” block, we have the keyword else, followed by the “else” block. This block of code is executed if the conditional expression evaluates to false.

When your execution arrow reaches an if statement, evaluate the conditional expression. Evaluating this expression proceeds just like evaluating any other expression. If the result is true, move the execution arrow immediately inside the “then” block and continue executing statements as usual. When your execution reaches the close curly brace that ends the “then” block, skip over the else block, placing your execution arrow immediately after the close curly brace of the “else” block, and continue executing statements from there.

If the result of the conditional expression is false, you should instead skip the “then” block and execute the “else” block. Move your execution arrow into the start of the “else” block, and continue executing statements from there. When your execution arrow reaches the close curly brace that ends the “else” block, simply move it past that curly brace (which has no effect—it just denotes the end of the block) and continue executing statements normally.

C permits if with no else, which is equivalent to an empty “else” block (as if the programmer had written else {}). If you execute an if with no else, then simply imagine the empty “else” block. If the conditional expression evaluates to true, you should execute the “then” block as previously described, however, there is no “else” block to skip. Instead, continue executing statements immediately after the end of the “then” block (skipping over the non-existent “else” block). If the conditional expression evaluates to false, then skip the “then” block, and execute whatever statements follow it (doing nothing for the “else” block).

if/else statements may be nested—one (or more) may occur in the “then” or “else” block of another if/else statement. When you encounter nested statements, the same rules apply. The inner statement is just one of the (possibly) many statements in the block, and is executed according to its rules—the condition is evaluated, whichever of the “then” or “else” blocks is appropriate is executed, and then execution continues after the end of the “else” block. When the execution arrow reaches the end of the outer “then” or “else” block, it behaves no differently than if there were no inner if statement. Video 2.6 demonstrates the execution of some if/else statements.

### 2.4.2 switch/case

Another way that programs can make decisions is to use switch/case. The syntax of switch/case is shown in Figure 2.8. Here, when the execution arrow reaches the switch statement, the selection expression—in parenthesis after the keyword switch—is evaluated to a value. This value is then used to determine which case to enter. The execution arrow then jumps to the corresponding case—the one whose label (the constant immediately after the keyword case) matches the selection expression’s value. If no label matches, then the execution arrow jumps to the default case if
If you cannot play these videos, your pdf reader does not support videos.

Video 2.6: Execution of if/else

there is one, and to the closing curly brace of the switch if not.

Once the execution arrow has jumped into a particular case, execution continues as normal until it encounters the keyword break. When the execution arrow reaches the break keyword, it jumps to the close curly brace which ends the switch statement. Note that reaching another case label does not end the current case. Unless the execution arrow encounters break, execution continues from one statement to the next. When the execution arrow passes from one case into the next like this, it is called “falling through” into the next case.

For example, if we were executing the code in Figure 2.8, and reached the switch statement with x having a value of 17 and y having a value of 16, then we would first evaluate the selection expression \((x - y)\), and get a value of 1. The execution arrow would then jump to case 1: and begin executing statements after it. We would execute \(y = 9;\). Then we would fall through the next case label—our execution arrow would move past it into the next case (the label itself has no effect). Then we would execute \(z = 42;\). Next, we would execute the break; statement, causing our execution arrow to jump to the close curly brace of the switch, after which we would continue executing whatever other statements are there.

Figure 2.8: Syntax of switch/case
Chapter 2: Reading Code

Video 2.7: Execution of `switch/case`.

Video 2.7 shows the execution of a `switch/case` statement under a few different conditions.

### 2.5 Shorthand

C (and many other programming languages) has shorthand—also called *syntactic sugar*—for a variety of common operations. These shorthands do not introduce any new behaviors. Instead, they just provide a shorter way to write common patterns of existing things we have seen. Table 2.2 shows the most common shorthand notations in C. These shorthand have exactly the same effect as their expanded meanings. Consequently, when you encounter a shorthand statement while executing code, you can execute it by considering what its fully written out form is, and performing the effects of that statement.

Another possible shorthand is to omit the curly braces around single statement blocks of code in certain cases. For example, if the “then” and/or “else” clause of an `if` statement is one single statement, the curly braces are not required. While you may encounter code written this way by other people, it is highly inadvisable to write code this way. Omitting the curly braces presents a danger if you modify the

<table>
<thead>
<tr>
<th>Shorthand</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x += y</code></td>
<td><code>x = x + y</code></td>
</tr>
<tr>
<td><code>x -= y</code></td>
<td><code>x = x - y</code></td>
</tr>
<tr>
<td><code>x *= y</code></td>
<td><code>x = x * y</code></td>
</tr>
<tr>
<td><code>x /= y</code></td>
<td><code>x = x / y</code></td>
</tr>
<tr>
<td><code>x++</code></td>
<td><code>x = x + 1</code></td>
</tr>
<tr>
<td><code>++x</code></td>
<td><code>x = x + 1</code></td>
</tr>
<tr>
<td><code>x--</code></td>
<td><code>x = x - 1</code></td>
</tr>
<tr>
<td><code>--x</code></td>
<td><code>x = x - 1</code></td>
</tr>
</tbody>
</table>

Table 2.2: Shorthands
code in the future. If you add another statement to the clause, but forget to add curly braces, then the statement will not actually be part of the clause, but rather the first statement after the if. Such errors have produced high-profile security vulnerabilities recently.

2.6 Loops

Programs often repeat the same block of code multiple times, using a loop. As you may recall from the examples in Section 1.7, algorithms often have repetitive behavior. Finding repetitive patterns is crucial to generalizing over inputs, as your program may need to perform similar work multiple times for different pieces of the input—and the number of repetitions will change with the characteristics of the inputs. There is another way to express repetition, called recursion, which you will learn about in Chapter ??.

2.6.1 while Loops

There are three kinds of loops in C. The first of these is the while loop. The syntax for a while loop is shown in Figure 2.9. The keyword while is followed by an expression in parenthesis. Much like an if statement, this expression is evaluated to determine whether or not to enter the block of code immediately following it, which is known as the body of the loop. If the conditional expression evaluates to true, the execution arrow moves inside the body of the loop and its statements are executed normally. The while loop differs from the if statement in what happens when the execution arrow reaches the closing curly brace. In the case of a while loop, it jumps up to the top of the loop, immediately before the while keyword. The conditional expression is then re-evaluated, and if it is still true, execution re-enters the loop body. If the conditional expression evaluates to false, then the execution arrow skips to immediately after the closing curly brace of the loop body, and proceeds from there.

Video 2.8 shows the execution of a while loop.

2.6.2 do-while Loops

Another type of loop in C is the do-while loop. Unlike a while loop, which checks its conditional expression at the top of the loop, the do-while loop checks its conditional expression at the bottom of the loop—after it has executed the body. While this distinction may seem contrived—either way the condition is checked between iterations—it is important at the start of the loop. A while loop may execute its body zero times, skipping the entire loop, if the condition is false initially. By contrast, a do-while loop is guaranteed to execute its body at least once because it executes the loop body before ever checking the condition.

Figure 2.10 shows the syntax of a do-while loop. The keyword do is followed by the loop body. After the loop body, the keyword while is followed by the conditional expression and a semicolon.
Chapter 2: Reading Code

Video 2.8: Execution of a while loop

Execution of a do-while loop proceeds by first entering the loop body and executing all of the statements contained in it. When the execution arrow reaches the while at the end of the loop body, its conditional expression is evaluated. If the expression evaluates to true, then the execution arrow jumps back to the start of the loop body. If the expression evaluates to false, then it moves past the end of the loop and execution continues with the next statement after the loop.

2.6.3 For loops

The third type of loop in C is a for loop. The for loop is syntactic sugar—it does not introduce any new behavior, but instead provides a more convenient syntax for a common programming idiom. In the case of for loops, the common idiom is counting from one number to another. Figure 2.11 shows the syntax of a for loop, and how it is de-sugared into a while loop—that is, how we could write the for loop in terms of the already familiar while loop. Knowing how the for loop de-sugars to a while loop tells us how to execute it. We can imagine the equivalent while loop, and follow the execution rules we have already learned for it.

The for keyword is followed by three pieces, separated by semicolons, inside of parenthesis. The first of these is the “initialization statement”. It happens once before the first time the loop’s condition is checked. In the de-sugaring, this statement appears right before the while loop. The second piece is not a statement (even though it is followed by a semicolon), but rather the conditional expression for the loop. In the de-sugaring, this expression is the conditional expression of the while loop. The third statement is the “increment statement”. In the de-sugaring, it appears immediately before the close curly brace of the loop body. After all of these is the loop body, which (except for the addition of the “increment statement” at the end) is the loop body of the while loop in the de-sugared version.
Chapter 2: Reading Code

Figure 2.11: Syntax of a for loop (left), and how you can understand it in terms an equivalent while loop (right)

If you examine Figure 2.11 carefully, you will notice that there is a set of curly braces around the entire piece of while-based code. These curly braces are there for a subtle, but important reason. The scope of any variables declared in the “initialization statement” of the for loop have a scope which is limited to the for loop. Recall that a variable typically has a scope which is limited to the curly braces which enclose its declaration. For a variable declared in the start of the for loop, the scope appears to be an exception to this rule, however, it is not if we think of it in terms of the de-sugaring shown above with the curly braces surrounding the declaration.

2.6.4 Nesting

Just as if/else statements may be nested, loops may also be nested. Similarly, loops follow exactly the same rules no matter how they are nested. In fact, if/else statements and loops may be nested within each other in any combinations. The rules are always the same regardless of any combinations or depths of nesting.

Video 2.9 shows the execution of some code with nested loops and if statements.

2.6.5 continue and break

Sometimes a programmer wants to leave the loop body early, rather than finishing all of the statements in side of it. There are two possible behaviors that a programmer might want when leaving the loop body early.

One behavior would be to exit the loop completely, making the execution arrow jump to immediately after the close curly brace which ends the loop (the same place that it goes when the loop’s condition evaluates to false). This behavior is obtained by using the break; statement—which we have already seen in the context of switch/case. Whenever the execution arrow encounters a break statement, it executes the statement by jumping out of the innermost enclosing loop (whether it is a while, do-while, or for loop), or switch statement. If the break statement is inside multiple of these which are nested together (e.g. a loop inside a case of a switch statement), then it exits only the most immediately enclosing one. If a break statement occurs and is not inside one of these loops or a switch statement, it is an error in the program.

The other possible behavior that the programmer might want to have is for the execution arrow to jump back to the top of the loop. This behavior is accomplished with the continue; statement. Executing the continue statement jumps to the top of the innermost enclosing loop (if it is not in a loop, it is an error). In the case of a for loop, the “increment statement” in the for loop is executed immediately before the jump. This fact complicates the de-sugaring of a for loop into a while loop slightly relative to the explanation given above. If the for loop contains any continue
Chapter 2: Reading Code

Video 2.9: Execution of a nested loops and ifs

statements, then the “increment statement” is written not only before the close curly brace of the loop, but also before any continue statements.

Video 2.10 shows how to transform a for loop with a continue statement inside of it into an equivalent while loop, then execute the resulting code. We note that in this example, simply using an if/else statement would be better—however, a good example of the use of continue is not easy to come by until we learn some more advanced concepts.

2.7 Higher-level Meaning

So far, we have discussed reading code in terms of step-by-step execution to determine its effects. While this skill is crucial for programmers, another useful skill is to be able to understand the meaning of a piece of code—what algorithm it implements and how it does so. This skill is useful to programmers, as they may need to understand—and possibly modify—code that they did not write.

The skill of reading code and ascertaining its higher level meaning is, in some ways more of a matter of reversing the process of writing code—you translate the code into the algorithmic steps it represents, then figure out what the purpose of that general algorithm is. Sometimes you accomplish this by working examples from the algorithm to see what it does. Sometimes the original programmer was helpful and wrote documentation explaining what it does and how it accomplishes that task.
Chapter 2: Reading Code

```c
void printRemainders (int lo, int hi, int n) {
    for (int i = lo; i < hi; i++) {
        if (i == 0) {
            printf("Cannot divide by 0.\n");
            continue;
        }
        printf("%d mod %d = %d\n", n, i, n % i);
    }
}

int main (void) {
    printRemainders(-2, 4, 5);
    return 0;
}
```

If you cannot play these videos, your pdf reader does not support videos.

Video 2.10: Execution of `continue` in a `for` loop

### 2.8 Practice Exercises

Selected questions have links to answers in the back of the book.
• Question 2.1: What does the following code print when it is executed?

```c
int f (int x, int y) {
    if (x < y) {
        return y - x;
    }
    return x + 5 - y;
}

int main (void) {
    int a = 3;
    int b = 4;
    int c = f (b, a);
    printf("c = %d\n", c);
    a = f(a, c);
    printf("a = %d\n", a);
    b = f(c, f(a, b));
    printf("b = %d\n", b);
    return 0;
}
```

• Question 2.2: What does the following code print when it is executed?

```c
main (void) {
    for (int x = 0; x < 3; x++) {
        for (int y = 0; y < 3; y++) {
            if (x-y % 2 ==0) {
                printf(" O ");
            } else if (x <= y) {
                printf(" X ");
            } else {
                printf(" ");
            }
        }
        printf("\n");
    }
    return EXIT_SUCCESS;
}
```
• Question 2.3: What does the following code print when it is executed?

```c
int f (int x, int y) {
    printf("In f(%d,%d)\n", x, y);
    if (x + 2 < y) {
        x += 3;
        return y * x;
    } else {
        return x + y + 2;
    }
}

int main(void) {
    int answer = 0;
    for (int i = 0; i < 4; i++) {
        answer += f (i, answer);
        printf("i = %d, answer = %d\n", i, answer);
    }
    return EXIT_SUCCESS;
}
```

• Question 2.4: Given the following code

```c
int g (int x, int y) {
    switch(x - y) {
    case 0:
        return x;
    case 4:
        y++;
        break;
    case 7:
        x--;
    case 9:
        return x*y;
    case 3:
        y = x + 9;
    default:
        return y - x;
    }
    return y;
}
```

What do each of the following expressions evaluate to?

1. \(g(14,7)\)
2. \(g(9,5)\)
3. \(g(3,0)\)
4. \(g(2,9)\)
5. \(g(5,5)\)
6. \(g(27,18)\)
• Question 2.5: Consider the following code, thinking about the differences between the functions f and g.

```c
void f(int x, int y) {
    while (x < y) {
        printf("%d", y-x);
        x++;
        y--;
    }
}
void g(int x, int y) {
    do {
        printf("%d", y-x);
        x++;
        y--;
    } while (x < y);
}
```

Come up with values of the parameters (x and y) where the two functions produce different output. What does each function produce for output on the parameter values you picked? Check your answer by swapping parameter values with a friend. Execute f(x,y) and g(x,y) by hand for your friend’s values of x and y, and see if you came up with the same answer. Have your friend do the same for your values of x and y.
• Question 2.6: What does the following code print when it is executed?

```c
int min(int a, int b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}

int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

int euclid(int a, int b) {
    printf("euclid(%d,%d)\n", a, b);
    int larger = max(a,b);
    int smaller = min(a,b);
    if (smaller == 0) {
        return larger;
    }
    return euclid(smaller, larger % smaller);
}

int main(void) {
    int x = euclid(9135, 426);
    printf("x = %d\n", x);
    return EXIT_SUCCESS;
}
```

(Hint: nothing unusual happens if a function calls itself. You just follow the same rules we have learned. We will talk a lot more about functions that call themselves in Chapter ??)

• Question 2.7: Trace the creation and destruction of stack frames for the following code. What stack frames exist after each line of code is executed?

```c
void aFinalFn() {
}

void sillyFunction() {
    aFinalFn();
}

void someFn() {
    sillyFunction();
    sillyFunction();
}

int main(void) {
    someFn();
    aFinalFn();
    return EXIT_SUCCESS;
}
```
Chapter 3

Types

3.1 Hardware Representations .................................................................................. 44
3.2 Basic Data Types ................................................................................................. 46
3.3 Printing Redux .................................................................................................... 51
3.4 Expressions Have Types ..................................................................................... 51
3.5 “Non-Numbers” .................................................................................................. 56
3.6 Complex, Custom Data Types ............................................................................. 58
3.7 Practice Exercises ............................................................................................... 63

Until this point, our programs have been declaring and manipulating integers only. We have declared variables such as \( x \) and \( y \) and given them values such as 3 or 4. What happens when we want to move past integers? What happens when we want to move past numbers altogether?

3.1 Hardware Representations

First and foremost, as far as the computer is concerned, there is no way to move “past numbers” because to the computer, \( \text{everything is a number} \). A computer stores \( \text{everything} \) as a series of 0’s and 1’s. Each 0 or 1 is called a \( \text{bit} \), and there are many ways to interpret these bits. This is where types come in. A \( \text{type} \) is a programming language construct that specifies both a size and an interpretation of a series of bits stored in a computer. For example, the type for working with integers is an \( \text{int} \), whose size is typically 32 bits and whose interpretation is an integer number directly represented in binary.

3.1.1 Binary Numbers

Before we delve into how to represent numbers in binary, let us briefly discuss the decimal system, which should be familiar to all of us. A decimal number is a number represented in base 10, in which there are 10 possible values for each digit (0–9). When these digits are concatenated to make strings of numbers, they are interpreted column by column. Beginning at the far right and moving to the left, we have the 1’s column, the 10’s column, the 100’s column, and so forth. The number 348, for example, has 8 ones, 4 tens, and 3 hundreds. The value of each column is formed by taking the number 10 and raising it to increasing exponents. The ones column is actually \( 10^0 = 1 \), the tens column is \( 10^1 = 10 \), the hundreds column is \( 10^2 = 100 \), and so forth. When we see a number in base 10, we automatically interpret it using the process shown in Figure 3.1(a), without giving it much thought.

A binary number is a number represented in base 2, in which there are only 2 possible values for each digit (0 and 1). The 0 and 1 correspond to low and high voltage values stored in your computer. Although it might be possible for a computer to store more than two voltage values and
therefore support a base larger than 2, it would be extremely difficult to support the 10 voltage values that would be required to support a base 10 number system in hardware. A familiarity with base 2 is helpful in understanding how your computer stores and interprets data.

Binary numbers are interpreted such that each bit (the name for a binary digit) holds the value $2^0$ raised to an increasing exponent, as shown in Figure 3.1(b). We begin with the rightmost bit (also called the least significant bit) which holds the value $2^0 = 1$, or the ones column. The next bit holds the value $2^1 = 2$, or the twos column. In base 10, each column is ten times larger than the one before it. In base 2, each column’s value grows by 2. The number 10_2 (the subscript indicates the base) has 1 two and no ones. It corresponds to the value 2 in base 10. Congratulations! You are now technically equipped to understand the age-old joke: “There are 10 types of people in the world. Those who understand binary and those who do not.”

### 3.1.2 Looking Under the Hood

When you are driving a car in traffic, it is probably not a good idea to think too much about what the engine is doing—in fact, you really do not need to know how it works in order to drive. This example illustrates an important concept in programming: abstraction—the separation of interface (what something does or how you use it) from implementation (how something works).

Abstraction often comes in multiple levels. Driving a car, the level of abstraction you care about is that the steering wheel turns the car, the gas pedal makes it go faster, and the brake makes it slow down. Your mechanic’s level of abstraction is how the pieces of the engine fit together, what level is appropriate for the brake fluid, and if your oil filter is screwed on tightly enough. The engineers who designed the car thought about the physics to make it all work efficiently. At each deeper level, you can think about details that were not important at higher levels, but are still crucial to making the system work. We could continue to lower and lower levels of abstraction until we start thinking about quantum interactions of atoms—fortunately you don’t need to worry about that to merge onto the interstate!

There are times, however, when it is a good idea to take a look “under the hood”—to go deeper than the abstraction levels that you typically care about. At the very least, you might want to know whether the car has a diesel engine before filling up the tank, or to be aware that your car...
has oil, and you should get it changed sometimes.

Similarly, you need not constantly consider the inner workings of your CPU in order to write good code. Thinking about variables as boxes that store values is a good level of abstraction. But, having some knowledge of what goes on under the hood can be important. When you first declare your variables and assign them a type, it is a good idea to pause and consider what this actually means at the hardware level.

As mentioned earlier, a type indicates both a size and an interpretation. Figure 3.2 shows you the now-familiar figure with code and its conceptual representation. For this chapter, we will add a third column, showing you the underlying representation at the hardware level. When you declare a variable \( x \) of type \texttt{int}, you should think about this conceptually as a box called \( x \) with a value 42 inside. But at a hardware level, the type \texttt{int} means that you have allocated 32 bits dedicated to this variable, and you have chosen for these bits to be interpreted as an integer number in order to yield the value 42.

**Hex.** As you may well imagine, reading and writing out a series of 32 of 1’s and 0’s is tedious at best and error-prone at worst. Because of this, many computer scientists choose to write out the values of numbers they are thinking about in binary using an encoding called hexadecimal, or hex for short. Hex is base 16, meaning that it represents a number with a 1’s column, a 16’s column, a 256’s column, and so on. As a hex digit can have 16 possible values (0–15), but our binary number system only has 10 possible symbols (0–9) we use the letters A-F to represent the values 10-15 in a single digit. The number eleven, for example, is represented by the single digit ‘B’ in hex. Numbers represented in binary can easily be converted to hex by simply grouping them into 4-digit clusters, each of which can be represented by a single hex digit. For example, the 4 rightmost bits in Figure 3.2 (colored blue) are 1010, which has the decimal value 10 and the hex value A. The next 4 bits in Figure 3.2 (colored green) are 0010, which has the decimal value 2 and the hex value 2. The remaining 24 bits in the number are all zeroes. Instead of writing out the entire 32 bit binary sequence, we can use 8 digits of hex \(0x0000002A\) or the shorthand \(0x2A\). (In both cases, the leading 0x (interchangeable with just x) indicates that the number is in hex.)

### 3.2 Basic Data Types

C supports a very small number of data types, each with a unique combination of size and interpretation. They are shown in Figure 3.3. As the caption of this figure notes, the sizes listed are common, and what we will use in general discussion in this book, but not guaranteed. In particular, it depends on the hardware, and the compiler—the program which turns your code into instructions that the computer can actually execute (more on this in Chapter ??).
Chapter 3: Types

<table>
<thead>
<tr>
<th>type</th>
<th>size (typical)</th>
<th>interpretation</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte (8 bits)</td>
<td>one ASCII character</td>
<td>'f'</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes (32 bits)</td>
<td>binary integer</td>
<td>42</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes (32 bits)</td>
<td>floating point number</td>
<td>3.141592</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes (64 bits)</td>
<td>floating point number</td>
<td>3.141592653589793</td>
</tr>
</tbody>
</table>

Figure 3.3: Basic data types supported in C. Note: sizes shown are typical, but can vary with compiler and hardware.

3.2.1 char

A char (pronounced either “car” or “char”) is the smallest data type—a mere 8 bits long—and is used to encode characters. With only 8 bits, there are only \(2^8 = 256\) possible values for a char (from 00000000 to 11111111). On most machines you will use, these 8 bits are interpreted via the American Standard Code for Information Interchange (or ASCII) character-encoding scheme, which maps 128 number sequences to letters, basic punctuation, and upper- and lower-case letters. A subset of this mapping is shown in Figure 3.4; please don’t try to memorize it. Another, much more expressive character-encoding scheme you may encounter (particularly when needing to encode non-English characters) is Unicode (which requires more than 1 byte).

If you look at the first line of code in Figure 3.5, you can see the char c declared and initialized to the value ‘A’. Notice that we wrote A in single quotation marks—these indicate a character literal. In the same way that we could write down a string literal in Section 2.3.2, we can also write down a character literal: the specific constant character value we want to use. Writing down this literal gives us the numerical value for A without us having to know that it is 65. If we did need to know, we could consult an ASCII table like the one in Figure 3.4. Being able to write ‘A’ instead of 65 is another example of abstraction—we do not need to know the ASCII encoding, we can just write down the character we want.

3.2.2 int

We have said that an int is a 32-bit value interpreted as an integer directly from its binary representation. As it turns out, this is only half of the story—the positive half of the story. If we dedicate all 32 bits to expressing positive numbers, we can express \(2^{32}\) values, from 0 up to...
### 3.2.3 float, double

The final two basic data types in C allow the programmer to express real numbers. Since there are an infinite number of real numbers, the computer cannot express them all (that would require an infinite number of bits!). Instead, for values that cannot be represented exactly, an approximation of the value is stored.

If you think about the fact that that computers can only store values as 0s and 1s, you may wonder how it is possible to store a real number, which has a fractional part. In much the same way that decimal representations of a number can have a fractional portion with places to the right of a decimal point (the tenth’s, hundredth’s, thousandth’s, etc. places), binary representations of numbers can have fractional portions after the binary point. The places to the right of the binary point are the half’s, quarter’s, eighth’s, etc. places.

One way we could (but often do not) choose to represent real numbers is fixed point. We could take 32 bits, and interpret them as having the binary point in the middle. That is, the most significant 16 bits would be the “integer” part, and the least 16 bits would be the “fractional” part. While this representation would be conceptually simple, it is also rather constrained—we could not represent very large numbers, nor could we represent very small numbers precisely.

Instead, the most common choice is similar to scientific notion. Recall that in decimal scientific
int main (void) {
    float p1 = 3.141592;
    double p2 = 3.141592653589793;
    ...
}

Sample Code
Conceptual Representation

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>3.141592</td>
</tr>
<tr>
<td>p2</td>
<td>3.141592653589793</td>
</tr>
</tbody>
</table>

Hardware Representation

-1^s x m x 2^e

<table>
<thead>
<tr>
<th>1 bit</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>exponent</td>
<td>mantissa</td>
</tr>
</tbody>
</table>

Figure 3.6: Floating Point Representation. A float has 32 bits and a double has 64 bits to express the 3 necessary fields to represent a floating point number.

notation, number 403 can be expressed as 4.03 × 10². Computers use floating point notation, the same notation but implicitly in base 2: \(m \times 2^e\). \(m\) is called the mantissa (though you may also hear it referred to as the significand). \(e\) is the exponent.

A float has 32 bits used to represent a floating point number. These 32 bits are divided into three fields. The lowest 23 bits encode the mantissa; the next 8 bits encode the exponent. The most significant bit is the sign bit, \(s\), which augments our formula as follows: \((-1)^s \times m \times 2^e\). (When \(s = 1\), the number is negative. When \(s = 0\), the number is positive.) A double has 64 bits and uses them by extending the mantissa to 52 bits and the exponent to 11 bits. Examples of both a float and a double are shown in Figure 3.6.

Standards. There would be many possible ways to divide a given number of bits into the mantissa and exponent fields. The arrangement here is part of the IEEE (Institute of Electrical and Electronics Engineers) Standard. Industry standards like these make it possible for engineers from a variety of companies to agree upon a single encoding by which floating point numbers can be represented and subsequently interpreted across all languages, platforms, and hardware products. Part of the IEEE Standard for floating point notation involves two adjustments to the bitwise representations of a float and a double. These adjustments (normalization and adding a bias) make the actual binary representation of these numbers less accessible to a first time observer. We encourage the interested reader to read the actual IEEE floating point Standard and allow the less curious reader simply to trust that there is a bitwise encoding for the numbers in Figure 3.6 which is just outside the scope of this textbook.

Precision. There are an infinite number of values between the numbers 0 and 1. It should be unsurprising, then, that when we use a finite number of bits to represent all possible floating point values, some precision will be lost. A float is said to represent single-precision floating point whereas a double is said to represent double-precision floating point. (Since a double has 64 bits, it can dedicate more bits to both the mantissa and exponent fields, allowing for more precision.) How does precision play out in practice? Figure 3.7 shows how unexpected (or at least un intuitive) things can happen due to imprecision. If you take the square root of 2.0 and store it in the float fRoot you get a particular value. If you store the number in a double, the value is the same number to the 8th decimal place; then the two values diverge. Interestingly, in neither case if you take the square root of 2.0 and square it, do you end up with exactly 2.0. Notice how the code in Figure 3.7 tests to see whether the root of 2.0 squared yields 2.0 and in neither case it does. As we will discuss in the next section, the default print setting for floats and doubles is to print up to 6 decimal places (see Figure 3.8). As a consequence, the user has no reason to think that these numbers are not exactly 2.0 and the fact that the neither test for equality to 2.0 passes is simply confusing.

It is important for programmers to understand precision when they choose types for their
Chapter 3: Types

Source Code

```c
float fRoot = sqrt(2.0);
float fSquared = fRoot * fRoot;
double dRoot = sqrt(2.0);
double dSquared = dRoot * dRoot;

if ((fSquared == 2.0) || (dSquared == 2.0)) {
    printf("This statement will never be printed!
")
} else {
    printf("Apparently sqrt(2.0)*sqrt(2.0) != 2.0. ???
")
}
printf("fSquared = %f, dSquared = %f
", fSquared, dSquared);
```

Conceptual Representation

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>fRoot</td>
<td>1.4142135386880710</td>
</tr>
<tr>
<td>fSquared</td>
<td>2.0000000000000004</td>
</tr>
<tr>
<td>dRoot</td>
<td>1.4142135827476095</td>
</tr>
<tr>
<td>dSquared</td>
<td>2.0000000000000004</td>
</tr>
</tbody>
</table>

Output

```
Apparently sqrt(2.0)*sqrt(2.0) != 2.0. ???
fSquared = 2.000000, dSquared = 2.000000
```

Figure 3.7: Precision. In practice, the imprecision associated with floating point arithmetic can produce surprising results. A `double` offers more precision than a `float`, but neither is immune to the problem of representing an infinite number of values with a finite number of bits.

---

Variables and when they perform tests on variables whose values are assumed to be known. Some programs will need more precision in order to run correctly. Some programs will have to allow for a small degree of imprecision in order to run correctly. Understanding exactly the level of precision required for your code is critical to writing correct code. For every project you begin (or join) it is definitely worth taking a minute to think about the code and how important precision might be in that particular domain. This is true particularly for programs that will ultimately be used to make life-and-death decisions for those who have no say over the precision decisions you are making for them.

It is also important to understand the cost. A `double` takes up twice as much space as a `float`. This may not matter for a single variable, but some programs declare thousands or even millions of variables at a time. If these variables do not require the precision of a `double`, choosing a `float` can make your code run faster and use less memory with no loss of correctness.
Chapter 3: Types

Source Code

char letter = 'A';
int negNumber = -1;
unsigned int age = 65;
float p1 = 3.141592;
double p2 = 3.14159265358979323;

printf("My name begins with %c\n", letter);
printf("Look, I'm negative! --> %d\n", negNumber);
printf("I am %d years old!\n", age);
printf(" in octal (base 8) = %o\n", age);
printf(" in hex (base 16) = %x\n", age);
printf("p1: decimal floating point = %f\n", p1);
printf(" scientific notation = %e\n", p1);
printf("p2: default decimal places = %f\n", p2);
printf(" 10 decimal places = %.10f\n", p2);

Output

My name begins with A
Look, I'm negative! --> -1
I am 65 years old!
in octal (base 8) = 101
in hex (base 16) = 41
p1: decimal floating point = 3.141592
 scientific notation = 3.141592e+00
p2: default decimal places = 3.141592
 10 decimal places = 3.1415920258

Figure 3.9: Printing in C. The above figure shows a few printing examples, incorporating a variety of format specifiers, decimal formats, and escape sequences.

3.3 Printing Redux

As we learned in Section 2.3.2, C supports printing formatted information via the function printf. Now that we have multiple types, we can explore the various format specifiers, which allow us to print variables of a variety of types. Figure 3.8 shows the most common specifiers. You should not try to memorize these (nor really anything in learning to program)—rather, you should know how/where to look up what you need. As you use the most common ones often, they will come to you naturally. You can find more format specifiers, as well as more information about these format specifiers in the man page for printf (see Section A.2 for more information about man pages).

Figure 3.9 shows some examples of these format specifiers being used. Here, the code (shown on the left) declares a few variables, and prints them out using the format specifiers described in Figure 3.8. Note that while we have already discussed hexadecimal (base 16), this example also makes reference to octal—which is base 8.

3.4 Expressions Have Types

In Section 2.2, we learned that expressions are evaluated to values—if you have $a+b*2$, the current value of $b$ is read out of its box, multiplied by 2, then the value of $a$ is read out of its box, and added to the product of $b*2$. The expression evaluates to the resulting sum.

Expressions also have types, which are determined by the types of the sub-expressions that make them up. The simplest expressions are constants, which have type int if they are integer constants ($e.g., 2$ or $46$), or type double if they are real constants ($e.g., 3.14$, or $-8.19$). The types of constants can be modified by applying a letter suffix if needed (U for unsigned, L for long, and f for float): $3.14f$ is a constant with type float, and $999999999999L$ is a constant with type long int. The next simplest type of expression is a variable, which has whatever type it was declared to have.

Most (but not all) expressions with binary operators—$e_1$ op $e_2$ ($e.g., a + b$ or $c * 4$)—have the same type as their operands. If $a$ and $b$ are doubles, then $a + b$ is a double as well. Likewise, if $c$ is an int, then $c * 4$ is also an int (note that 4 is an int).
Chapter 3: Types

The type of a function is its declared return type. That is, if you have

```c
int f (int x, int y) { ... }
```

then the expression `f(3, 4) + g(42.6, 'a')` has type `int`. We can see this from the fact that `f(3, 4)` has type `int` (`f` is declared to return an `int`), as does `g(42.6, 'a')`. As we just discussed, adding two `ints` results in an `int`.

### 3.4.1 Type Conversion

The next natural question is “what happens if you have a binary operator, and its operands have different types?” For example, if `a` is an `int` and `b` is a `double`, then what type is `a + b`? The answer to this question depends on the types involved.

Fundamentally, the first thing that must happen is that the two operands must be converted to the same type. Most operations can only be performed on operands of the same type. The processor has one instruction to add two 32-bit integers, a different instruction to add two 16-bit integers, a third one to add two 32-bit floats, a fourth to add two 64-bit doubles, and so on. The compiler must translate your code into one of these instructions, so, it must pick one of them and arrange to have the inputs in a proper format in order to be able to perform the math.

When the two operands have different types, the compiler attempts to add a *type conversion* (sometimes called a *type promotion*) to make the types the same. If no type conversion is possible, the compiler will issue an error message and refuse to compile the code. When the compiler inserts a type conversion, it typically must add instructions to the program which cause the processor to explicitly change the bit representation from the size and representation used by the original type to the size and representation used by the new type. The compiler chooses which operand to convert based on what gives the “best” answer. In our `int + double` example, the compiler will convert the `int` to a `double` to avoid losing the fractional part of the number.

There are four common ways that the bit representations must be changed to convert from one type to another during a type promotion. When converting from a smaller signed integer type to a longer signed integer, the number must be *sign extended*—the sign bit (most significant bit) must be copied an appropriate number of times to fill in the additional bits. When converting from a smaller unsigned integer type to a longer unsigned integer type, the number must be *zero extended*—the additional bits are filled in with all zeros. The third common way that the bit representation can be changed during an automatic conversion happens when a longer integer type is converted to a shorter integer type. Here, the bit pattern is *truncated* to fit—the most significant bits are thrown away, and only the least significant bits are retained.

The fourth way that the bit representation may need to be changed is to fully calculate what the representation of the value is in the new type. For example, when converting from an integer type to a real type, the compiler must insert an instruction which requests that the CPU compute the floating point (binary scientific notation) representation of that integer.

There are other cases where a type conversion does not need to alter the bit pattern, instead just changing how it is interpreted. For example, converting from a *signed* `int` to an *unsigned* `int` leaves the bit pattern unchanged. However, if the value was originally negative, it will now be interpreted as a large positive number. Consider the following code:

```c
unsigned int bigNum = 100;
int littleNum = -100;
```

---

1There are some operations which explicitly work with different types, but the types they work with are still limited in certain ways.
Chapter 3: Types

3.1 if (bigNum > littleNum)
   printf("Obviously, 100 is bigger than -100!\n");
else
   printf("Something unexpected has happened!\n");

When this code runs, it prints “Something unexpected has happened!” The bit pattern of littleNum (which has a leading 1 because it is negative) is preserved; the value is changed to a number larger than 100 (because under an unsigned interpretation, this leading 1 indicates a very large number). We will note that the compiler produces a warning (an indication that you probably did something bad—which means you should go fix your code!) for this behavior, as comparing signed integers to unsigned integers is typically a bad idea for exactly this reason.

When you declare a variable and assign it a particular type, you specify how you would like the data associated with that variable—the bit pattern “in the box”—to be interpreted for the entirety of its life span. There are some occasions, however, when you or the compiler may have to temporarily treat the variable as though it were of another type. When a programmer does this, it is called casting and when a compiler does it, it is called type conversion or type promotion. It is extremely important to understand when to do the former and when the compiler is doing the latter because it can often be the cause of confusion and consequently errors. We will note that while understanding when to cast is important, understanding that you should generally not cast is even more important—sprinkling casts into your code to make errors go away indicates that you are not actually fixing your code, but rather hiding the problems with it.

3.4.2 Casting

Sometimes, the programmer wants to explicitly request a conversion from one type to another—either because the compiler has no reason to insert it automatically (the types are already the same, but a different type of operation is desired), or because the compiler does not consider the conversion “safe” enough to do automatically. This explicitly requested conversion is called casting and is written in the code by placing the desired type in parenthesis before the expression whose value should be converted. For example, (double)x evaluates x (by reading the value in its box), then converts it to a double.

To see why we would want this capability, let us begin with a seemingly benign example. We want to write a program that calculates how many hours you would work per day if you stretched the 40 hour work week across 7 days instead of 5. A naïve implementation of the code (shown in Video 3.1) might begin with two ints, nHours and nDays. Here, int is a perfectly reasonable type as we are working only in integer numbers of hours (40) and days (5). This code then divides the number of hours by the number of days and stores the result in the float avgWorkDay. If you execute this code carefully by hand, you will find that when it prints the answer out, it will print 5.0. Somehow our work week just got shortened to 35 hours!

In this case, the problem lies in the fact that we divided two ints, and integer division always produces an integer result—in this case 5. When the compiler looks at this expression, there are only integers involved, so it sees no need to convert either operand to any other type. It therefore generates instructions that request the CPU to perform integer division, producing an integer result.

However, when the compiler examines the assignment, it sees that the type on the left (the type of the box it will store the value in) is float, while the type of the expression on the right (the type of the value that the division results in) is int. It then inserts the type conversion instruction at the point of the assignment: converting the integer division result to a floating point number as it puts it in the box. Video 3.1 illustrates this execution.

All of Programming, http://aop.cs.cornell.edu
Chapter 3: Types

If you cannot play these videos, your pdf reader does not support videos.

Video 3.1: Execution of our naïve code for computing the hours in a 7 day work week.

Here, what we really wanted to do was to convert both operands to a real type (float or double) before the division occurs, then perform the division on real numbers. We can achieve this goal by introducing an explicit cast—requesting a type conversion. We could explicitly cast both operands, however, casting either one is sufficient to achieve our goal. Once one operand is converted to a real type, the compiler is forced to automatically convert the other. We prefer writing \( a / (\text{double})b \) over \( (\text{double})a/ b \) even though they are the same, as the former does not require the reader of the code to remember the relative precedence (“order of operations”) between a cast and the mathematical operator. However, we note that casting has very high operator precedence—it happens quite early in the order of operations.

Video 3.2 illustrates the execution of the modified code with the cast added. Observe how the cast does not affect the boxes, only the intermediate numbers that are being worked with in the computation.

### 3.4.3 Overflow & Underflow

Figure 3.3 showed us some of the basic types supported in C and their sizes. The fact that each type has a set size creates a limit on the smallest and largest possible number that can be stored in a variable of a particular type. For example, a \texttt{short} is typically 16 bits, meaning it can express exactly \(2^{16}\) possible values. If these values are split between positive and negative numbers, then the largest possible number that can be stored in a short is \(0111111111111111\), or 32767.

What happens if you try to add 1 to this number? Adding 1 yields an unsurprising \(1000000000000000\). The bit pattern is expected. But the \textit{interpretation} of a signed short with this bit pattern is \(-32768\), which could be surprising (the very popular xkcd comic, illustrates this principle nicely:

All of Programming, http://aop.cs.cornell.edu
Chapter 3: Types

Video 3.2: Execution of our fixed code for computing the hours in a 7 day work week.

http://xkcd.com/571/). If the short were unsigned, the same bit pattern 1000000000000000
would be interpreted as an unsurprising 32768.

This odd behavior is an example of overflow: an operation results in a number that is too
large to be represented by the result type of the operation. The opposite effect is called underflow
in which an operation results in a number that is too small to be represented by the result type of
the operation. Overflow is a natural consequence of the size limitations of types.

Note that overflow (and underflow) are actions that occur during a specific operation. It is
correct to say “Performing a 16-bit signed addition of 32767 + 1 results in overflow.” It is not
correct to say “-32768 overflowed.” The number -32768 by itself is perfectly fine. The problem of
overflow (or underflow) happens when you get -32768 as your answer for 32767 + 1. The operation
does not have to be a “math” operation to exhibit overflow. Assignment of a larger type to a
smaller type can result in overflow as well. Consider the following code:

```c
short int s;
int x = 99999;
s = x;
printf("%d\n", s);
```

In this code, the assignment of x (which is a 32-bit int) to s (which is a 16-bit short int)
overflows—the truncation performed in the type conversion discards non-zero bits. This code will
print out ~31073, which would be quite unexpected to a person who does not understand overflow.

Whether overflow is a problem for the correctness of your program is context-specific. Clocks,
for example, experience overflow twice a day without problems. (That 12:59 is followed by 1:00 is
the intended behavior). As a programmer, realize that your choice of type determines the upper
and lower limits of each variable, and you are responsible for knowing the possibility and impact
3.5 “Non-Numbers”

It is worth restating: everything is a number. This rule is fundamental to understanding how computers work and is one of the most important concepts in programming. For every variable you create in any programming language, the value of that variable—the data that you place “in the box” of every conceptual diagram you draw—is stored in your computer as a series of zeros and ones. This fact is easy to accept for a positive integer, whose base 10 representation is simply converted to base 2 and then stored in a series of bits. Understanding how negative numbers and floating point numbers are also represented as a series of zeros and ones may be a little less straightforward, but is still appeals to our general intuitions about numbers.

Extending this rule to things that do not seem like numbers—words, colors, pictures, songs, movies—may seem like a much harder conceptual leap. However, with our newfound understanding that computers can only operate on numbers, we must realize that all of these things must be numbers too—after all, our computers operate on them regularly.

Finding a way to encode these “non-number” data types is a simply a matter of coming up with a new convention for encoding the information as bits, and interpreting the bits to mean the original information. These new conventions are not longer included as basic data types of the C programming language (though some of them are basic data types in languages other than C). Instead, new types are formed by combining the basic types to achieve the programmer’s goals. These more complex types may be widely accepted programming conventions (like the representation of strings), or may be something done by one single programmer specific to their programming task.

3.5.1 Strings

A string is a sequence of characters that ends with a special character called the null terminator, which can be written with the character literal ‘\0’ (pronounced “backslash zero”) that signals the end of the string. A string is referred to by the location of the first character in memory and each 8-bit character is read until the ‘\0’ is detected. A simple drawing of this concept is shown in Figure 3.10.

Strings are not a basic data type in C, meaning you cannot simply declare and use them as you would an int or a double. To give you a tiny glimpse into the complexity of the matter, consider how large a string should be. Is there pre-defined number of bits that should correspond to a string data type? Since each string has a unique number of characters, this does not seem like a choice that can be made up front. In fact, the size of a string will need to be dynamically determined on
Chapter 3: Types

a per-string basis. To truly understand how to create and use strings, an understanding of pointers (the topic of Chapter ??) is required. This is one reason why Figure 3.10 is deliberately lacking in details—because we haven’t yet explained the concepts necessary to show you how to declare and instantiate them. We will delay further discussions of strings to Section ??.

3.5.2 Images

Your computer frequently displays images—whether its the windows and icons on your screen, or the lolcats you view in your web-browser. These may seem like they are not numbers, however, they are actually just many numbers put together. The first step to representing an image as a number is to represent a color as a number.

While there are many ways to represent a color as a number, the most common is RGB encoding, which encodes each color by specifying how much red, green, and blue they contain. Typically, this encoding is done with each component being represented on a scale from 0 to 255. The RGB values for the color red are: R=255, G=0, B=0. Orange is R=255, G=127, B=0. If you search the Internet, you will find many online tools that will let you select a color, and then tell you its corresponding RGB encoding.

Once we can encode a single color numerically, an image is encoded as a 2D grid of colors. Each “dot” in this grid is called a pixel. As with strings, understanding how to store a 2D sequence requires an understanding of pointers, which will come later. However, for now, it suffices to understand that an image can be encoded as many numbers organized in a logical format.

You may have noticed that computers typically have a variety of image formats, such as JPG, BMP, PNG, TIFF, and many others. Each of these encodes the image numerically, however, the specific details differ between the formats. Some image formats compress the image data—performing math on the colors (after all, the colors are just numbers!) to encode the image data in fewer bits, reducing the size of the data that must be stored on disk and/or transferred across the Internet.

3.5.3 Sound

Sound is another common aspect of computer use that seems like it is not a number. However, sound is a naturally a waveform, which can easily be represented as a sequence of numbers. The most direct numeric representation of a sound wave is to record the “height” of the wave at periodic intervals, forming a sequence of numbers. The frequency of these intervals is called the sampling rate (higher sampling rates result in better quality of the sound), and is typically 22 kHz or 44kHz—that is 22,000 or 44,000 samples per second. Stereo sound simply has 2 sequences of numbers—one for the left channel and one for the right channel. As with images, there are many typical formats (e.g., WAV, AIFF, AAC, etc.), some of which are compressed (again, the sound is just numbers, so we can do math on it).

3.5.4 Videos

Videos (including those found in this book) again seem to defy the “everything is a number” rule—however, by now, you should see the path to numeric encoding. A video is a sequence of images (called “frames”) and the corresponding sound. We have already seen how to encode images and sound as numbers. The simplest approach would be to encode the video as the sequence of images plus the sound. While this approach gives us a bunch of numbers, it would be huge—one minute of a 512 pixel x 256 pixel video at 32 frames per second with a stereo sound track at 44 kHz would require about 725 Megabytes (almost 1 GB). Correspondingly, all common movie formats (e.g.,
3.6 Complex, Custom Data Types

You may be starting to notice that the definitions of many data types are essentially a set of agreed upon conventions. One of the great things about rich programming languages like C is that they give a programmer the power to create new data types and associated conventions. Some conventions, like the IEEE floating point standard, are agreed upon across multiple programming languages, compilers, machine languages, and the architecture of the processors they run on. This requires the coordination of hundreds of companies and tens of thousands of engineers. Other conventions can be more local, existing only in a particular code base, or a collection of files that all use a common library. This may require the coordination of multiple people (who are usually working together already) or may only affect a single person who simply wishes to produce clean, modifiable, and debuggable programs.

Suppose you are designing a program that regularly draws and computes various properties of rectangles. It would be very convenient to have a data type that captures the basic properties of a rectangle. In C, this is accomplished via the keyword \texttt{struct}.

3.6.1 \texttt{struct}

A \texttt{struct} allows a programmer to bundle multiple variables into a single entity. For example, if we wish to define a rectangle via its 4 coordinates on an X-Y plane\(^2\) as shown in Figure 3.11(left), these four coordinates can be bundled into a single, conglomerate data structure, whose internal structure will look like the code in Figure 3.11(center). Structs are represented conceptually with a single box in which all the component fields reside, each with their own box. Figure 3.11(right) shows a variable called myRect with its 4 fields.

Syntactically, there are multiple ways to declare, define, and use structs. Figure 3.12 shows 4 different syntactic options that all create the same conceptual struct. Regardless of which syntactic option you choose, the drawing of your conceptual representation will be the same. It is not important for you to be “fluent” in all four options. You may choose a single approach and stick with it. However, it is important for you to know about all four options because others contributing

\(^2\)We will work with rectangles that are vertical or horizontal, rather than at an angle for simplicity.
## Types

### Options 1-4

<table>
<thead>
<tr>
<th>Options</th>
<th>Creating new tags (1-3) and types (2-4)</th>
<th>Instantiating the variable myRect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Define a tag <strong>(rect_t)</strong> only. Tag can only be used with the word <code>struct</code> as a prefix.</td>
<td><code>struct rect_t { int left; int bottom; int right; int top; };</code></td>
<td><code>int main() { struct rect_t myRect; myRect.left = 1; ... }</code></td>
</tr>
<tr>
<td>2. Define a tag <strong>(_rect_t)</strong> and then define its type alias <strong>(rect_t)</strong>. Struct declaration and typedef can occur in either order. Tag can be used on its own with struct prefix.</td>
<td><code>struct _rect_t { int left; int bottom; int right; int top; }; typedef struct _rect_t rect_t;</code></td>
<td><code>int main() { rect_t myRect; myRect.left = 1; ... }</code></td>
</tr>
<tr>
<td>3. Abbreviation from 2. Declaration &amp; definition occur in the same statement.</td>
<td><code>typedef struct _rect_t rect_t;</code></td>
<td><code>int main() { rect_t myRect; myRect.left = 1; ... }</code></td>
</tr>
<tr>
<td>4. Type definition with no tag declaration. Downside: struct cannot refer to itself (see linked lists in Chapter 21).</td>
<td><code>typedef struct { int left; int bottom; int right; int top; } rect_t;</code></td>
<td><code>int main() { rect_t myRect; myRect.left = 1; ... }</code></td>
</tr>
</tbody>
</table>

Figure 3.12: Various syntactic options for creating struct tags, types, and instantiating variables.

---

to the same code base as you may have a different style, and internet searches will also result in many versions of the effectively the same code. You need to be aware of these differences so that you can correctly understand and extend code whose syntax differs from your preferred style.

Struct declarations do not go inside functions; they live in the global scope of the program, next to function declarations and definitions. All of them use the keyword `struct`. Option 1 in Figure 3.12 begins with the keyword `struct`, followed by the `tag` of our choosing. In this case, we use the tag `rect_t`. Ending the tag in “_t” is a convention that makes it easier to recognize the name as identifying a type throughout your code. A name such as `rect` would be acceptable, just a little less reader-friendly. Everything inside the braces belongs to the definition of the newly defined struct named `rect_t`. The semi-colon indicates the completion of the definition.

The far right column of Figure 3.12 shows how to instantiate a variable for each syntactic option. For Option 1, the type of the variable is `struct rect_t`, and the name of the variable is `myRect`. Once you declare the variable, you can access the fields of the struct using a dot (period): `myRect.top` gives you access to the field `top` of type `int` inside the `myRect` variable. Note: when you instantiate a variable of type `struct rect_t`, you choose a top level name (`myRect`) only. The names of the fields are determined in the definition of the structure and cannot be customized on a per-instance basis.

Video 3.3 shows an example of a declaration of a `struct` for a rectangle, as well as its initialization, and use. Note that the assignment statements follow the same basic rules we have seen so far: you find the box named by the left side, evaluate the right side to a value, and store that value into the box named by the left side. The dot operator just gives you a different way to name a box—you can name a “box inside a box”.

A key part of good programming is using good abstractions. Structs are another form of abstraction. Once we have a rectangle struct, other pieces of code can operate on rectangles without looking at the implementation. We could write many functions to manipulate rectangles, and those
Chapter 3: Types

If you cannot play these videos, your pdf reader does not support videos.

Video 3.3: Declaration and use of a struct for a rectangle.

functions could be the only pieces of code that know the internal details of rectangles—the rest of the code could just call those functions.

However, part of using good abstractions is using them correctly. In the case of structs, remember that their primary purpose is to group together data that belongs logically together. In this example, we use a struct for a rectangle—something that logically makes sense as a combination of other pieces of data. In Figure 3.11 we illustrate the connection between the conceptual idea (the visualization on the left), and the declaration in the middle. We can think about operations on rectangles and understand what they are conceptually, without looking at the implementation details.

While it may seem silly to say: do not just group data together into structs without a logical purpose. Sometimes novice programmers are tempted to just put a bunch of things in a struct because they get used in the same parts of a program (to pass one parameter around instead of a few). However, if you cannot articulate why those data make sense together, they do not belong in a struct together.

3.6.2 typedef

Many consider Option 1 in Figure 3.12 to be somewhat unwieldy, because the type of the variable includes the word struct in it. For example, suppose you wanted a function called shrinkRect that takes a rectangle as its input and returns a smaller rectangle as its output. Using the syntax of Option 1, the function would have the signature struct rect_t shrinkRect(struct rect_t
typedef unsigned int rgb_t;

rgb_t getRedForPixel(int x, int y) {...}
rgb_t getGreenForPixel(int x, int y) {...}
rgb_t getBlueForPixel(int x, int y) {...}

int main(void) {
    rgb_t red, green, blue;
    red = getRedValue();
    green = getGreenValue();
    ...
}

// nothing else changes!

typedef unsigned int rgb_t;

Figure 3.13: Use of Typedef. Left: code that defines and uses a new data type, rgb_t to store color values. Right: a change to the definition of the type requires no subsequent code changes.

typedef unsigned int rgb_t;

// nothing else changes!

char change definition of rgb_t

typedef unsigned int rgb_t;

// nothing else changes!

char change definition of rgb_t

Figure 3.13: Use of Typedef. Left: code that defines and uses a new data type, rgb_t to store color values. Right: a change to the definition of the type requires no subsequent code changes.

shrinkThisRectangle). Depending on how often you need to write out the type of the structure, this syntax can become cumbersome and make your code appear cluttered.

The solution to needing to type out “struct rect_t” every time you want to declare, pass, or use your new struct is to create a new data type that is explicitly of type struct. We do this using the keyword typedef. The exact syntax is shown in Option 2 of Figure 3.12. The first lines declare the _rect_t struct in the same way as before. However, after this struct definition, the last line (typedef struct _rect_t rect_t;) is the declaration of the type “rect_t” which is defined as having the type “struct _rect_t”. Note: when providing a typedef (as in Option 2) many programmers use the underscore naming convention for the tag (_rect_t) to differentiate between the tag and the new type, rect_t. This common convention makes code easier to read and encourages the use of the type over the tag. Options 3 and 4 also “typedef” a new type, however, they both combine the typedef into a single statement with the structure declaration.

Although typedefs can simplify the use of structs, that is far from their only use. Any time that you are writing code in a specific context, typedefs can help you make your code more readable, by naming a type according to its meaning and use. For example, suppose you are writing a program that deals with colors.

In the context of programming color characteristics, you might want to define a new data type for the colors in an RGB value. For example, you could create a new data type called rgb_t (which represents one of the red, green, or blue components of the color), that is of type unsigned int (because we know the values should be positive integers) and then declare variables red, green, and blue of type rgb_t. An example of this is shown on the left side of Figure 3.13. Typedefs provide a helpful abstraction for programmers. Instead of having to write “unsigned int” throughout her code, or frankly even think about the range of acceptable values in RGB representations, the programmer simply uses the custom type rgb_t and gives it no further thought.

Typedefs have another nice property of limiting the definition of a particular type to a single place in the code base. Suppose a programmer wished to conserve the space dedicated to variables and therefore wished to use an unsigned char instead of an unsigned int (after all, the values from 0 to 255 all fit within the 8-bits of an unsigned char). Without a typedef, this change would require a tedious and error-prone search of many (but by no means all—it may be used for variables unrelated to colors) instances of unsigned int throughout the code, changing these
types to unsigned char. With a typedef, the programmer simply changes the single line of code in which rgb_t was defined (see the right side of Figure 3.13). No other code changes are required.

Heads up about typedef. The use of typedefs is somewhat controversial in some programming circles. In the context of structs, there are those who believe that it is important not to abstract the struct away from a type. They believe that programmers should always know when a particular variable is a struct and when it is not. Similarly, they believe that programmers should always be aware of the actual types of the data they use lest they fall prey to typing errors that could have been otherwise avoided. Use typedefs when the abstraction simplifies rather than obfuscates your code.

3.6.3 Enumerated Types

The last form of custom type that a programmer can create is called an enumerated type. Enumerated types are named constants that can increase the readability and the correctness of your code. They are most useful when you have a type of data with a set of values that you would like to label by their conceptual name (rather than using a raw number) and either the particular numerical values do not matter (as long as they are distinct), or they occur in naturally in a sequential progression. For example, until 2011 the United States’ Homeland Security maintained a color-coded terrorism threat advisory scale that it used to maintain heightened or more relaxed security in various locations including major airports. There were five threat levels from green to red in ascending order of severity.

These five threat levels could be recorded in an enumerated type which we can create ourselves as shown in Figure 3.14. We begin with the keyword enum, followed by the name of the new enumerated type, in this case threat_level_t. The various threat levels are placed in curly braces, as shown. Each level is assigned a constant value, staring with 0.\(^3\) The enumerated names are constant—they are not assignable variables. Their values cannot change throughout the program. The convention for indicating that a name denotes a constant is to write the name in all uppercase. However, variables of the enumerated type can be created, and assigned to normally.

Because enumerated types have integer values, they can be used in constructs such as simple value comparisons, switch statements, and for loops. Figure 3.14 shows an example of the first two. Video 3.4 illustrates the execution of this code.

\(^3\)You can override this default by explicitly setting the value of the any particular constant: {LOW = 1, GUARDED...
Chapter 3: Types

Another example of enumerated types would be if we wanted to make a program which regularly refers to a small set of fruits: grapes, apples, oranges, bananas, and pears. Suppose we want to represent each of these as a number (because we regularly use constructs like switch statements on the fruits themselves), but we do not really care which number each is represented as. We can make a enumerated type, `enum fruit_t { GRAPE, APPLE,...};` and then use these constants throughout our code.

3.7 Practice Exercises

Selected questions have links to answers in the back of the book.

- Question 3.1: Why is “Everything is a Number” an important rule of programming?
- Question 3.2: What is a “char”?
- Question 3.3: What are “float”s and “double”s? How are they similar? How are they different
- Question 3.4: What two things do a variable’s type tell the compiler about that variable?
• Question 3.5: In the table below there are four columns: one for binary, one for hexadecimal (hex), one for decimal (base 10) if you interpret the number as 8-bit signed two’s complement binary, and one for decimal if you interpret the number as 8-bit unsigned two’s complement binary. In each row, you are given one of the numbers, and should fill in the other three columns by performing the appropriate conversions:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hex</th>
<th>Decimal (Unsigned)</th>
<th>(Signed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0x3C</td>
<td>200</td>
<td>-42</td>
</tr>
<tr>
<td>0110101</td>
<td>0x7F</td>
<td>100</td>
<td>87</td>
</tr>
</tbody>
</table>

• Question 3.6: What is “type promotion”? What is “casting”? How are they similar? How are they different?

• Question 3.7: Assume that we have executed int a = 4; and int b = 5;. What are the type and value of each of the following expressions:
  1. a/b
  2. (double)(a/b)
  3. a/(double)b
  4. (double)a/b
  5. a-b/2
  6. a-b/2.0

• Question 3.8: What happens if integer arithmetic results in a number too large to represent?

• Question 3.9: How are colors represented as numbers?

• Question 3.10: What is a string? How does it related to the “everything is a number” principle?

• Question 3.11: What does typedef do?

• Question 3.12: Suppose you are writing software in which you need unique sequence numbers, and decide that unsigned long is sufficiently large as a type to work with them. How could you give this type a name (e.g., seq_t) so that you can use that name throughout your program? Write the C statement which would accomplish this goal.
Writing code is (or at least, should be), 90% planning. Investing an extra 10 minutes in carefully planning out a piece of code can save hours of debugging a snarled mess later on. Many novice programmers mistakenly jump right into writing code without a plan, only to end up pouring hours into what should be a relatively short task.

Planning first is not only the best approach for novices, but also for skilled programmers. However, if you see a highly experienced programmer in action, you may not see her planning when working on a relatively easy problem. Not seeing her planning does not mean that she is not doing it, but just that she is capable of doing all of the planning in her head. As you advance in programming skill, this will eventually happen for you as well—there will be certain problems that you can just solve in your head and write down the solution. Of course, having practiced the skills required to solve harder problems will be key, as your skills will be put to better use if you work on problems at the difficult end of your capabilities.

As we discussed in Chapter 1, planning for programming primarily consists of developing the algorithm to solve the relevant problem. Once the algorithm is devised (and tested), translating it to code becomes relatively straightforward. Once you have implemented your program in code, you will need to test—and likely debug—that implementation. Having a clear plan of what the program should do at each step makes the debugging process significantly easier.

Even though Chapter 1 explained Steps 1–4, and worked some examples, we are going to revisit them now. One reason for revisiting these steps is that they are crucial to programming, and it is likely that they have somewhat faded from your mind, as we last saw them 3 chapters ago. However, we are also going to revisit them now as you have been introduced to the material in Chapter 2 and Chapter 3 which you did not know in Chapter 1. Accordingly, we can now talk about types, and representing everything as numbers. After we revisit Steps 1–4, we will continue on to Step 5, translating our algorithms to code. At the end of this chapter, Video 4.1 will work an entire problem from Step 1 to Step 5.
Chapter 4: Writing Code

4.1 Step 1: Work an example yourself

The first step to devising an algorithm is to work an instance of the problem yourself. As we discussed earlier, if you cannot do the problem, you cannot hope to write an algorithm to do it—that is like trying to explain to someone how to do something which you yourself do not understand how to do. However, you have to not only be able to do the problem, but also do it methodically enough that you can analyze what you did and generalize it.

Often, a key part of working the problem yourself is drawing a picture of the problem and its solution. Drawing a clear and precise picture allows you to visualize the state of the problem as you manipulate it. Having a clear idea of the state of the problem, and how you are manipulating it will help you with the next step, in which you write down precisely what you did on this instance of the problem.

We will use the following problem as an example to work from for the rest of this chapter:

Given two rectangles, compute the rectangle which represents their intersection. You may assume the rectangles are vertical or horizontal.

The first thing we should do here is to work at least one instance of this problem (we may want to work more). In order to do this, we need a bit of domain knowledge—what a rectangle is (a shape with 4 sides, such that adjacent sides are at right angles), and what their intersection is (the area that is within both of them).

What instance we pick is really up to us. For some problems, some instances will be more insightful than others, and some will expose corner cases—inputs where our algorithm needs to behave specially. The most important rule in picking a particular instance of the problem is to pick one that you can work completely and precisely by hand.

Figure 4.1 shows the results of Step 1 for the rectangle intersection problem. We picked an instance of the problem—here the yellow-shaded rectangle from (−4, 1) to (8, 6) intersecting with the blue-shaded rectangle from (1, −1) to (4, 7). The resulting intersection is the green-shaded rectangle from (1, 1) to (4, 6).

You should note a few things about this example. First, while the yellow/blue/green coloring is not truly a part of the problem, there is nothing wrong with adding extra information to your diagram to help you understand what is going on. Second, note that the diagram is done precisely—we drew a Cartesian coordinate grid, and placed the rectangles at their correct coordinates. This precision not only ensured that any information we obtained from analyzing our diagram was correct and not a result of sloppy drawing (though whether some relationship is generally true, or a consequence of the specific case we chose is not guaranteed by a careful drawing). You typically do not need to draw things with draftsman-level precision, but the more precise you can be, the better.

In this case, we can tell the answer just by looking at the picture and seeing where the green region is. However, to write a program to do this, we need to figure out the math behind it—we need to be able to work the problem in some way other than just looking at the diagram and seeing the answer. Sometimes trying work things mathematically is hard when you can just see the answer. Learning to put the obvious aside and think about what is going on is a key programming skill to learn, but takes some time.
Chapter 4: Writing Code

If you struggle with it, it may be useful to work another instance of the problem, but eliminate extra information that lets you jump straight to the answer without understanding. Figure 4.2 shows a different instance of the rectangle problem with the Cartesian grid removed (note that it was still drawn such that the rectangles are the right size and in the correct relative positions). We can still precisely work the problem from this diagram, but it is a little harder to just look at the Cartesian grid and see the answer. Take a second to work out the answer before you continue.

Note that there is nothing wrong with working a few instances of the problem, taking different approaches as you do it, and including/excluding various extra information as you do so. In general, it is better to spend extra time in an earlier step of programming than getting stuck in a later step (if you do get stuck, you might want to go back to an earlier step and redo it with another instance of the problem). For Step 1, doing a few different instances of the problem is preferable to moving into Step 2 and only being able to come up with “I just did it—it was obvious.”

4.2 Step 2: Write down what you just did

Now, you are ready to think about what it was exactly that you just did, and write down a step-by-step approach to solve one specific instance of the problem. Using our example from Figure 4.2, this would basically be a set of steps anyone could follow to find the intersection of the rectangle from $(-2,1)$ to $(6,3)$ with the rectangle from $(-1,-1)$ to $(4,4)$. Note that you are not trying to generalize to any rectangles here, just writing down what you did for this particular pair of rectangles.

There are actually two important pieces to think about here. The first is how you represented the problem with numbers. Remember the key rule of programming: Everything is a number. Since a rectangle is not a number (in the way, for example, that the price of bread is), we will have to find a way to properly represent a rectangle using a number (or several). If you go back and read the descriptions of the instances of the problems we worked, you will find that we already have been representing each rectangle with 4 numbers—two for the bottom left corner, and two for the top right corner. Now that we have assured ourselves that rectangles are numbers, we know that we can happily compute on them—we also have an idea of what information we should think of a rectangle as having (each of which is just a number): a bottom, a left, a top, and a right. This analysis leads us to make the same definition of a rectangle as in Section 3.6.1, but we underscore here as it is an important part of the programming process.

The second thing we need to think about is what exactly it was that we did to come up with our answer, and write it down. These steps can be anywhere in the spectrum of pseudocode—notation that looks like programming, but does not obey any formal rules of syntax—to pure natural language (e.g., English) that you are comfortable with. The important thing here is not any particular notation, but to have a clear idea of what you did in a step-by-step fashion before you try to generalize the steps.

For example, we might write the following:

I found the intersection of
Chapter 4: Writing Code

left:  -2
bottom: 1
right:  6
top:    3

and
left:  -1
bottom:  1
right:  4
top:    4

by making a rectangle with
left:  -1
bottom: 1
right:  4
top:    3

In this case, we do not have many steps, but it is still crucial for us to write them down.

4.3 Step 3: Generalize your steps

Now that we know what we did for this particular instance, we need to generalize to all instances of the problem. We will note that this step is often the most difficult (you have to think about why you did what you did, recognize patterns, and figure out how to deal with any possible inputs) and the most mistake prone (which is why we test the algorithm in Step 4 before we proceed).

4.3.1 Generalizing Values

One aspect of generalizing your algorithm is to scrutinize each value you used, and contemplate what it is in the general case. Is it a constant that does not change depending on the inputs? Does it depend on one (or more) of the parameters? If it does depend on some of the parameters, what is the relationship between them?

Going back to the rectangle example on which we did Step 2, we came up with \(-1\) for the left value of the answer rectangle. We can quickly rule out the idea that this is a constant—surely not all rectangles have \(-1\) as the left side of their intersection (counterexamples would be easy to come by if we needed to convince ourselves).

Now we are left figuring out how \(-1\) relates to the input parameters. It could be that the left value of the answer rectangle matches one of the values of the input rectangles—both the left and the bottom of the second rectangle are \(-1\). It could be the case that it has some mathematical relationship to another value—maybe the left of the first rectangle divided by 2, or plus 1, or maybe the negative of the bottom of the first rectangle. Any of these would yield \(-1\), and work in this case, but we need to think about why the answer is \(-1\) to figure out the correct generalization.

Sometimes this analysis is quite difficult. Whenever you get stuck on generalization, it can help to repeat Steps 1 and 2, to give us more information to work with and more insight. For example, looking back at the other example we worked first in Step 1, we can rule out some of the ideas we pondered in the prior paragraph. From these two examples, we might draw the conclusion that the left value of the intersection is the left value of the second rectangle. We might proceed similarly.
to generate the following generalized algorithm (as with Step 2, notational specifics do not matter as long as you are precise enough that each step has a clear meaning):

To find the intersection of two rectangles, \( r_1 \) and \( r_2 \):

Your answer is a rectangle with

- left: \( r_2 \)'s left
- bottom: \( r_1 \)'s bottom
- right: \( r_2 \)'s right
- top: \( r_1 \)'s top

While these generalized steps accurately describe the two examples we did, they are in fact not a correct generalization. Figure 4.3 shows a pair of rectangles where our algorithm gives the wrong answer—shown a red dashed rectangle. If we make an incorrect generalization such as this, we should catch it in Step 4 (or if not, then when we test the code at the end of Step 5). In such a case, we must return to Step 3 before proceeding, and fix our algorithm.

When you detect a mis-generalization of your algorithm, you have the advantage that you have already worked at least one example which highlights a case you need to analyze carefully. In this case, we can see that we want \( r_1 \)'s right (not \( r_2 \)'s right) for the right side of the answer, and \( r_2 \)'s bottom (not \( r_1 \)'s bottom) for the bottom side of the answer. Note that \( r_1 \)'s right and \( r_2 \)'s bottom did not work for the earlier cases, so we cannot simply change our algorithm to use those in all cases. Instead, we must think carefully about when we need which one and why.

Careful scrutiny will lead us to conclude that we need the minimum of \( r_1 \)'s right and \( r_2 \)'s right, and the maximum of \( r_1 \)'s bottom and \( r_2 \)'s bottom. We may also realize that we should do something similar for the left and top (if not, we should find that out when repeating Step 4). We could then come up with the following correctly generalized steps:

To find the intersection of two rectangles, \( r_1 \) and \( r_2 \):

Make a rectangle (called \( \text{ans} \)) with

- left: maximum of \( r_1 \)'s left and \( r_2 \)'s left
- bottom: maximum or \( r_1 \)'s bottom and \( r_2 \)'s bottom
- right: minimum of \( r_1 \)'s right and \( r_2 \)'s right
- top: minimum of \( r_1 \)'s top and \( r_2 \)'s top

That rectangle called \( \text{ans} \) is your answer.

We will note that in the case of rectangles that do not intersect, this algorithm will produce an illogical rectangle as the answer (its top will be less than its bottom and/or its left will be greater than its right). For the purpose of this problem, we will say that giving such an invalid rectangle in these cases is the intended behavior of the algorithm—in part because we have not learned how to represent “no such thing” easily.
4.3.2 Generalizing Repetitions

Another important part of generalizing an algorithm is to look for repetitions of the same (or similar) steps. When similar steps repeat, you will want to generalize your algorithm in terms of how many times the steps repeat (or until what condition is met). To examine this aspect of generalizing, we will deviate from our rectangle example (which does not have this type of repetition), and consider a slightly different problem for a moment:

Given an integer \(N (>0)\), print a right triangle of *s, with height \(N\) and base \(N\). For example, if \(N = 4\), you would print

*  
**  
***  
****

We might work an example with \(N=5\), and end up with the following result from Step 2:

Print 1 star  
Print a newline  
Print 2 stars  
Print a newline  
Print 3 stars  
Print a newline  
Print 4 stars  
Print a newline  
Print 5 stars  
Print a newline

Here, we are doing almost the same thing (Print \(i\) stars; Print a newline) 5 times. Once we observe the repetition, we can take one step towards generalizing the algorithm by re-writing the algorithm like this:

Count (call it \(i\)) from 1 to 5 (inclusive)  
    Print \(i\) stars  
    Print a newline

Notice that the way we have re-written the algorithm here gives us two new constants to scrutinize: the 1 and the 5 in the range that we count from/to. Careful consideration of these would show that 1 is truly a constant (we always start counting at 1 for this algorithm), but 5 should be generalized to \(N\):

Count (call it \(i\)) from 1 to \(N\) (inclusive)  
    Print \(i\) stars  
    Print a newline

This algorithm is correct for the triangle-of-stars problem.

Sometimes it takes a little more work to make the steps of your algorithm match up so that you can describe them in terms of repetition. For example, consider the following problem:

Given a list of numbers, find their sum.
We might work this problem on the list of numbers 3, 5, 42, 11, and end up with the following result from Step 2:

Add 3 + 5 (= 8)
Add 8 + 42 (= 50)
Add 50 + 11 (= 61)
Your answer is 61

Scrubining each of these constants might lead us to the following more general steps:

Add (the 1st number) + (the 2nd number)
Add (the previous total) + (the 3rd number)
Add (the previous total) + (the 4th number)
Your answer is (the previous total)

Here, we almost, but not quite, have a nice repetitive pattern. We can, however, make the steps match up:

previous_total = 0
previous_total = Add previous_total + (the 1st number)
previous_total = Add previous_total + (the 2nd number)
previous_total = Add previous_total + (the 3rd number)
previous_total = Add previous_total + (the 4th number)
Your answer is previous_total

Note that mathematically speaking, what we did was exploit the fact that 0 is the additive identity—0 + N = N for any number N. We will also note that starting with the identity element as our answer before doing math to the items in a list is typically a good idea, since the list may be empty. Often, the correct answer when performing math on an empty list is the identity element of the operation you are performing. That is, the sum of an empty list of numbers is 0, the product of an empty list of numbers is 1 (the multiplicative identity). Now that we have re-arranged our steps, we can generalize nicely:

previous_total = 0
Count (call it i) from 1 to how many numbers you have

previous_total = Add previous_total + (the ith number)
Your answer is previous_total

In this example, we also did something that will make Step 5 (translating to code) a bit easier—naming values that we want to manipulate. In particular, we gave a name to the running total we compute, which means that not only is it clear exactly what we are referencing when we say previous_total, but also that when we reach Step 5, this will translate directly into a variable.

4.3.3 Generalizing Conditional Behavior

Sometimes when we are generalizing, we will have steps which appear sometimes, but not others. Such a situation may be a matter in which we perform a step for some parameter values, but not for others; or in which we have steps that are almost repetitive, but some actions which appear in some repetitions but not in others. In either case, we need to figure out under what conditions we should do those steps.
Chapter 4: Writing Code

It may take some work and thinking to determine the patterns for what conditions we need to perform those steps under, and what conditions we do not. As with many things in generalizing, if it is not immediately apparent, it can be quite useful to work more examples—giving you more information to generalize from. You might also find it informative to make a table of the circumstances (parameter values, information specific to each repetition, etc.) and whether or not the steps are done under those circumstances.

Once you have figured out the pattern, you can express the step in the algorithm more generally by describing the condition that should be determined, and what to do if that condition is true, and what to do if it is false. Doing so makes your algorithm a little bit more general, and may help you express a large sequence of steps as repetition, since they will now be more uniform.

4.3.4 Generalization: An Iterative Process

Generalization is an iterative process—you take what you have, generalize (or rewrite it) a bit, and then try to generalize that result more. Sometimes one step of generalization opens up new avenues of generalization that were not visible before. We have already seen how recognizing repetitive patterns can lead to the opportunity to generalize in terms of how many times you do the repeated steps. You may also end up exposing the repetitive pattern of some steps only once you have figured out what the generalization of the values in those steps is.

4.4 Step 4: Test Your Algorithm

Once you have generalized your Algorithm, it is time to test it out. To test it out, you should work it on different instances of the problem than the one(s) you used to come up with it. The goal here is to find out if you mis-generalized before you proceed. We have already seen one instance of mis-generalization in our rectangle problem, in which our algorithm was too specific to the examples from which we had built it (always using r1’s bottom, r2’s left, etc....). Testing on these same examples would not have revealed any problems.

In doing this testing, you want to strike a balance—enough testing to give you confidence that your algorithm is correct before you proceed, but not an excessive amount of testing. Note that in this testing, you perform your steps by hand, so it may be somewhat slow for a long or complex algorithm. You can do more extensive testing after you translate your algorithm to code. The tradeoff there is that the computer will execute your test cases (which is fast), but if your algorithm is not correct, you have spent time implementing the wrong algorithm.

Here are some guidelines to help you devise a good set of test cases to use in Step 4:

- Try test cases that are qualitatively different from what you used to design your algorithm. In the case of our rectangle example, the two examples we used to build the algorithm were both fairly similar, but the third example (which we used to show the flaw) was noticeably different—the rectangles overlapped in a very different way.

- Try to find corner cases—inputs where your algorithm behaves differently. If your algorithm takes a list of things as an input, try it with an empty list. If you count from 1 to N (inclusive), try N=0 (you will count no times) and N=1 (you will count only one time).

- Try to obtain statement coverage—that is, between all of your test cases, each line in the algorithm should be executed at least once. We will discuss various forms of test case coverage later in Chapter ??.
• Examine your algorithm and see if there are any apparent oddities in its behavior (it always answers “true”, it never answers “0” even though that seems like a plausible answer,…), and think about whether or not you can get a test case where the right answer is something that your algorithm cannot give as its answer.

4.5 Step 5: Translation to Code

Now that you are confident in your algorithm, it is time to translate it into code. This task is something that you can do with pencil and paper (e.g., as you often will need to do in a programming class on exams), but most of the time, you will want to actually type your code into an editor so that you can compile and run your program. Here, we will primarily focus on the mechanics of the translation from algorithmic steps to code. We strongly recommend that you acquaint yourself with a programmer’s editor (Emacs or Vim) and use it whenever you program. We cover Emacs in Appendix ??, if you need an introduction.

We should start Step 5 by writing down the declaration of the function that we are writing, with its body (the code inside of it) replaced by the generalized algorithm from Step 3, written as comments. Comments are lines in a program which have a syntactic indication that they are for humans only (to make notes on how things work, and help people read and understand your code), and not an actual part of the behavior of the program. When you execute code by hand, you should simply skip over comments, as they have no effect. In C, there are two forms of comments: // comments to the end of the line, and /*...*/ makes everything between the slash-star and the star-slash into a comment.

One thing we may need to do in writing down the function declaration is to figure out its parameter types and return type. These may be given to us—in a class programming problem, you may be told as part of the assignment description and in a professional setting, it may be part of an interface agreed upon between members of the project team—however, if you do not know, you need to figure this out before proceeding.

Returning to our rectangle intersection example, we know that the function we are writing takes two rectangles, and returns a rectangle. Earlier, we decided that a rectangle could be represented as four numbers—suggesting a struct. However, as you learned earlier, there are a variety of different types of numbers—should these numbers be ints? floats? doubles? Or some other type of number?

The answer to the question about which type of number we need is “It depends”. You may be surprised to learn that “It depends” is often a perfectly valid answer to many questions related to programming, however, if you give this answer, you should describe what it depends on, and what the answer is under various circumstances.

For our rectangle example, the type that we need depends on what we are doing with the rectangles. One of the real number types (float or double) would make sense if we are writing a math-related program where our rectangles can have fractional coordinates. Choosing between float and double is a matter of what precision and what range we need on our rectangles. If we are doing computer graphics, and working in the coordinates of the screen (which come in discrete pixels), then int makes the most sense, as you cannot have fractional pieces. For this example, we will assume that we want to use floats.

With this decision made, we would start our translation to code by declaring the function and writing the algorithm in comments. We then go through and translate each of the steps into code, line by line. If you have written good (i.e., clear and precise) steps in Step 3, this translation should be fairly straightforward—most steps you will want to implement naturally translate into
Chapter 4: Writing Code

the syntax we have already learned:

**Repetition** Whenever you have discovered repetition while generalizing your algorithm, it translates into a loop. Typically, if your repetition involves counting, you will use a *for* loop. Otherwise, if you are sure you always want to do the body at least once, a *do-while* is the most appropriate type. In other cases (which typically align with steps like “as long as (something)...” *while* loops are generally your best bet. If your algorithm calls for you to “stop repeating things” or “stop counting” you will want to translate that idea into a *break* statement. Meanwhile, if your algorithm calls for you to skip the rest of the steps in the current repetition, and go back the start of the loop, that translates into a *continue* statement.

**Decision Making** Whenever your algorithm calls for you to make a decision, that will translate into either *if/else* or *switch/case*. You will typically only want *switch/case* when you are making a decision based on many possible numerical values of one expression. Otherwise, you will want *if/else*.

**Math** Generally, when your algorithm calls for mathematical computations, these translate directly into expressions in your program which compute that math.

**Names** When your algorithm names a value and manipulates it, that translates into a variable in your program. You need to think about what type the variable has, and declare it before you use it. Be sure to initialize your variable by assigning to it before you use it—which your algorithm should do anyways (if not, what value did you use when testing it in Step 4?).

**Altering Values** Whenever your algorithm manipulates the values that it works with, these translate into assignment statements—you are changing the value of the corresponding variable.

**The answer is...** When your algorithm knows the answer and has no more work to do, you should write a *return* statement, which returns the answer that you have computed.

**Complicated Steps** Whenever you have a complex line in your algorithm—something that you cannot translate directly into a few lines of code—you should call another function to perform the work of that step. In some cases, this function will already exist—either because you (or some member of your programming team) has already written it, or because it exists in the standard C library (or another library you are using). In this case, you can call the existing function (possibly reading its documentation to find its exact arguments), and move on to translating the next line of your algorithm.

In other cases, there will not already be a function to do what you need. In these cases, you should decide what arguments the function takes, what its exact behavior is, and what you want to call it. Write this information down (either on paper, or in comments elsewhere in your source code), but do not worry about defining the function yet. Instead, just call the function you *will write in the future* and move on to translating the next line of your algorithm. When you finish writing the code for this algorithm, you will go implement the function you just called—this is a programming problem all of its own, so you will go through all of the Steps for it.

Abstracting code out into a separate function has another advantage—you can reuse that function to solve other problems later. As you write other code, you may find that you need to perform the same tasks that you already did in earlier programming problems. If you pulled the code for these tasks into their own functions, you can simply call those functions.
Copy/pasting code is generally a terrible idea—whenever you find yourself inclined to do so, you should instead find a logical way to abstract it out into a function and call that function from the places where you need that functionality.

With a clearly defined algorithm, the translation to code should proceed in a fairly straightforward manner. Initially, you may need to look up the syntax of various statements (you did make that quick reference sheet we recommended in Chapter 2, right?), but you should quickly become familiar with them. If you find yourself struggling with this translation, it likely either means that your description of your algorithm is too vague (in which case, you need to go back to it, think about what precisely you meant, and refine it), or that the pieces of your algorithm are complex and you are getting hung up on them, rather than calling a function (as described above) to do that piece, which you will write afterwards.

The process of taking large, complex pieces, and separating them out into their own function—known as top-down design—is crucial as you write larger and larger programs. Initially, we will write individual functions to serve a small, simple purpose—we might write one or two additional functions to implement a complex step. However, as your programming skill expands, you will write larger, more complex programs. Here, you may end up writing dozens of functions—solving progressively smaller problems until you reach a piece small enough that you do not need to break it down any further. While it may seem advantageous to just write everything in one giant function, such an approach not only makes the programming more difficult, but also tends to result in a complex mess that is difficult to test and debug. Whenever you have a chance to pull a well-defined logical piece of your program out into its own function, you should consider this an opportunity, not a burden. We will talk much more about writing larger programs in Chapter ?? after you master the basic programming concepts and are ready to write significant pieces of code.

4.5.1 Composability

When you are translating your code from your algorithmic description to C (or whatever other language you want), you can translate an instruction into code in the same way, no matter what other steps it is near, or what conditions or repetitions it is inside of. That is, you do not have to do anything special to write a loop inside of another loop, nor to write a conditional statement inside of a loop—you can just put the pieces together and they work as expected.

The ability to put things together and have them work as expected is called composability and is important to building not only programs, but other complex systems. If you put a for loop inside of an if statement, you do not need to worry about any special rules or odd behaviors: you only need to know how a for loop and an if statement work, and you can reason about the behavior of their combination.

In general, modern programming languages are designed so that features and language constructs can be composed, and work as expected. C (and later C++) follow this principle pretty well, so you can compose pretty much anything you learn from this book with pretty much anything else (with one notable exception that we will discuss in Section ??).

4.5.2 Finishing Our Earlier Examples

Now that we have seen how to translate our generalized steps into code, we can finish our earlier examples. We will start with the simpler one, the triangle of stars. We see here the code, as the translation of our generalized steps from earlier (which are placed in the code as comments—each line corresponds to the directions from the algorithm on the line right before it):
Chapter 4: Writing Code

```c
// prints i stars
void printIStars(int i) {
  // Count (call it j) from 1 to i (inclusive)
  for (int j = 1; j <= i; j++) {
    // Print a star
    printf("*");
  }
}

// prints a triangle of n stars
void printStarTriangle(int n) {
  // Count (call it i) from 1 to n (inclusive)
  for (int i = 1; i <= n; i++) {
    // Print i stars
    printIStars (i);
    // Print a newline
    printf("\n");
  }
}
```

Note how we abstracted out `printIStars`. The resulting function is small enough and simple enough that we would have been justified in writing it inline if we saw exactly how to do it right away. However, there is nothing wrong with pulling it out into its own function and solving it separately. In fact, if it is not immediately obvious what to write to translate it, abstracting it out is exactly what you should do.

The other example we saw earlier was our rectangle intersection problem. We can translate these steps into code as well, using the same approach. Again, we include the generalized steps as comments, and abstract out the `maximum` and `minimum` functions—here doing so allows us to avoid writing the code for this functionality twice, as we need each in two places. Here is the resulting code:
Chapter 4: Writing Code

// a rectangle with left, bottom, top, and right
struct _rect {
  float left;
  float bottom;
  float top;
  float right;
};
typedef struct _rect rect;

float minimum (float f1, float f2) {
  // compare f1 to f2
  if (f1 < f2) {
    // if f1 is smaller than f2, then f1 is your answer
    return f1;
  }
  else {
    // otherwise, f2 is your answer
    return f2;
  }
}

float maximum (float f1, float f2) {
  // compare f1 to f2
  if (f1 > f2) {
    // if f1 is larger than f2, then f1 is your answer
    return f1;
  }
  else {
    // otherwise, f2 is your answer
    return f2;
  }
}

// To find the intersection of two rectangles, r1 and r2:
rect intersection(rect r1, rect r2) {
  // Make a rectangle (called ans) with
  rect ans;
  //   left: maximum of r1's left and r2's left
  ans.left = max (r1.left, r2.left);
  //   bottom: maximum of r1's bottom and r2's bottom
  ans.bottom = max (r1.bottom, r2.bottom);
  //   right: minimum of r1's right and r2's right
  ans.right = minimum (r1.right, r2.right);
  //   top: minimum of r1's top and r2's top
  ans.top = minimum (r1.top, r2.top);
  // The rectangle called ans is your answer
  return ans;
}
If you cannot play these videos, your pdf reader does not support videos.

Video 4.1: Writing the \texttt{isPrime} function.

4.6 A Complete Example

Video 4.1 walks through a complete example—writing the \texttt{isPrime} function, which takes one integer \(N\) and determines if \(N\) is prime (in which case, it returns 1) or not (in which case it returns 0).

4.7 Next Steps: Compiling, Running, Testing and Debugging

Steps 6 (testing your program) and 7 (debugging it) require you to be able to run your code. The computer’s processor does not actually understand the source code directly, so the source code must be first be translated to the numerically encoded (\textit{everything} is a number) instructions that the processor can understand. In the next chapter, we will discuss this process, as well as some of the details of what you need to put in your source file to make a complete program. We will then discuss testing and debugging in Chapter ??.

4.8 Practice Exercises

Selected questions have links to answers in the back of the book.
• Question 4.1: Write a function \texttt{myAbs} which takes an integer \( n \), and returns an integer which is the absolute value of \( n \).

• Question 4.2: Write a function \texttt{avg3} which takes three integers \( (a, b, \text{ and } c) \) and returns the floating point number which is their average. Be careful—\( you \) may need to think back to what you learned in \textbf{Section 3.4.2} to be sure you get the right answer (in particular, if \( a=1, b=1, c=2 \), you should get 1.333, not 1.0).

• Question 4.3: Write a function \texttt{myRound} which takes a double \( d \), and returns an integer. This function should round \( d \) to the nearest integer, and return the rounded result. (\textbf{Hint:} to get the fractional portion of \( d \) (the part after the decimal), think about how you can get the integral portion (the part before the decimal) using what you learned in \textbf{Chapter 3}—then think about what mathematical operation you can use to compute the fractional portion from the information you have).

• Question 4.4: Write a function \texttt{factorial} which takes an integer \( n \), and returns an int which is the factorial of \( n \) (\( n! \) in math notation).

• Question 4.5: Write a function \texttt{isPow2} which takes an integer \( n \), and returns an int which is 1 ("true") if \( n \) is a power of 2, and 0 ("false") if it is not. Note that 1 is power of 2 (it is \( 2^0 \)), and 0 is not a power of 2. Note: some approaches to this problem involve computing \( 2^i \). In C, if you write \( 2^i \) it will NOT compute \( 2^i \)—instead, it will compute the bitwise exclusive-or (XOR) of 2 and \( i \). If you want to compute \( 2^i \) easily, you can write \( 1<<i \) (where \( << \) is the binary left shift operator—so it takes the number "1" and puts "\( i \)" 0s after it in the binary representation.

• Question 4.6: Write a function \texttt{printBinary} which takes an integer \( n \), and returns \texttt{void}. This function should print the binary representation of the number \( n \). You may assume that the number \( n \) has at most 20 bits.

• Question 4.7: Write a function \texttt{printFactors} which takes an integer \( n \), and returns \texttt{void}. This function should print the prime factorization of \( n \)—that is, it should print a multiplication expression composed of prime numbers which, if evaluated would result in \( n \). For example, given the number 132, your function should print \( 2 \times 2 \times 3 \times 11 \). If your function is given a number that is 1 or less, your function should print nothing. \textbf{Hint:} you should end up with at least 2 steps that are too complex to translate directly into a single line—these should result in additional functions you write.

• Question 4.8: Write a function \texttt{isPerfect} which takes an integer \( n \), and determines if \( n \) is \texttt{perfect} (hint: the definition of "perfect number" is domain knowledge in math—if you do not know it, look it up before you attempt step 1). If \( n \) is perfect, your function should return 1, otherwise it should return 0.

• Question 4.9: Write a function \texttt{power} which takes two unsigned integers, \( x \) and \( y \), and returns an unsigned integer. This function should compute and return \( x^y \) (that is, \( x \) to the \( y \) power).
• Question 4.10: For any of the previous questions, take your code, and trade it with a friend. First, you should examine the syntax of each other’s code—does it follow the rules that we specified in Chapter 2? If not, explain to each other what you think is not correct. Next, have her execute your code by hand (for some reasonable parameter values), while you execute her by hand. Determine if each other’s code is correct. If not, help each other fix and understand what might be wrong, and figure out how to fix it.
Appendices
Programming is all about making something. Whenever you make something, you do well to invest time and effort into learning the tools that help you with that task. Over the years, programmers have developed a wide variety of tools that support development efforts in various ways. If you want to become a serious programmer, mastering these tools is crucial.

New programmers often wonder why they should invest the effort into learning these sorts of tools. It is possible to program with more-familiar seeming environments, which require less up-front effort to learn the basics. For example, you could write and compile your programs in a graphical environment called an “IDE” (which stands for integrated development environment), which will have a more familiar “buttons and menus” interface. Choosing tools like these which are designed for the ease of novices represent a short-term benefit and long-term loss. If you are studying programming casually (e.g., just taking one or two required courses), then the time investment to learn these tools is not likely to be worthwhile. However, if your goal is to become a professional programmer, you will want to become well-versed in the tools of your trade.

Figure 4 shows the long-term tradeoff of using a tool designed for novices versus using a tool designed for experts. The x-axis of this graph represents time spent learning and using the tool. The y-axis represents proficiency (what you can do) with the tool. The red line shows the progression of proficiency with a tool designed for a novice. At time 0—when you first start using the tool—you have a basic proficiency with it. This basic proficiency stems from the fact that the tool is setup to be easy for novices—it is “user friendly.” As you spend more time with the tool, you learn more features and tricks, but your proficiency quickly plateaus as you reach the limits of your tool.

The blue line shows the progression with a tool designed for expert use. At time 0, the tool is difficult to use. It does not fit with the paradigms you are used to. As you spend time and effort learning the tool, your proficiency increases. At some point, your rate of learning increases too as you become familiar with the terminology and paradigms of the tool—you know what to look for, what to ask about, and where to look when you do not know something. As you continue to learn, your proficiency progresses past the plateau you could achieve in the tool designed for novices. You may eventually plateau, but when you do, that plateau will be much higher with the tool designed for experts. For some tools, you may never plateau—one author has used emacs for 15 years and still learns new things regularly.

This tradeoff is not unique to programming. Professional photographers, for example, use equipment that gives them full control over their art. They make decisions about shutter speed, aperture, and light sensitivity every few minutes. This level of control is likely overkill for taking casual photos. And a novice user might find that their first dozen photos taken on professional equipment are blurry, over-exposed, or that they missed the shot entirely while fiddling with camera settings. At the same time, few professional photographers would give up their professional equipment and replace it with a “point-and-click” camera for any serious artistic undertaking.

Another reason to invest time and effort into learning programming tools (if you want to be a professional programmer) is the perception associated with your tool choices. Using the tools of an expert programmer (especially if you use them well) sets up the perception that you are an expert programmer. Several students have reported that when interviewing for jobs, the fact they used the tools described in this appendix was important in interviews. Think of the photography analogy: would you hire a photographer who only knows how to use cheap disposable cameras?
Appendix A

UNIX Basics

A.1 In the Beginning Was the Command Line ............................................. 84
A.2 Getting Help: man and help .............................................................. 85
A.3 Directories ......................................................................................... 87
A.4 Displaying Files ................................................................................ 90
A.5 Moving, Copying, and Deleting ......................................................... 91
A.6 Pattern Expansion: Globbing and Braces ........................................... 91
A.7 Redirection and Pipes ......................................................................... 92
A.8 Searching ......................................................................................... 94
A.9 Command Shells ............................................................................... 95
A.10 Scripting ......................................................................................... 96
A.11 Environment Variables ..................................................................... 98
A.12 Remote Login: ssh ........................................................................... 99

Unix is a multi-tasking, multi-user operating system, which is well-suited to programming and programming-related tasks (running servers, etc.). Technically speaking, UNIX refers to a specific operating system developed at Bell Labs in the 1970s, however, today it is more commonly used (slightly imprecisely) to mean “any UNIX-like” operating system, such as Linux, Free BSD, Solaris, AIX, and even Mac OSX.¹ Here, we will use the more general term, and note that you are most likely to use Linux or Mac OSX.

Unix is a great example of the tools for experts versus tools for novices tradeoffs discussed in the introduction to these appendices. If you are reading this section, odds are good that you fall into the relatively large set of people who are “master novices” when it comes to using a computer—that is, you have mastered all of the skills of a novice system. You can use a graphical interface to open files, send email, browse the web, and play music. Maybe you can even fix a few things when something goes wrong. However, you would be hard pressed to make your computer perform moderately sophisticated tasks in an automated fashion.

As a simple example, suppose you had 50 files in a directory (aka “folder”) and wanted to rename them all by replacing _ with - in their names (but otherwise leaving the names unchanged). As a “master novice” you could perform this task in the graphical interface by hand—clicking each file, clicking rename, and typing in the new name. However, such an approach would be incredibly tedious and time consuming. An expert user would use the command line (which we will introduce shortly) to rename all 50 files in a single command, taking only a few seconds of work.

¹Many people use Mac OSX without really thinking of it as UNIX-like, however, that it is a UNIX variant can easily be revealed if you open the Terminal application
Appendix A: UNIX Basics

A.1 In the Beginning Was the Command Line

While Unix has a graphical interface (GUI), its users often make use of the command line.\(^2\) In its simplest usage, the command line has you type the name of the program you want to run, whereas a GUI-based operating system might have you double-click on an icon of the program you want to run. The command line interface can be intimidating or frustrating at first, but an expert user will often prefer the command line to a GUI. Beyond being the natural environment to program in, it allows for us to perform more sophisticated tasks, especially automating those which might otherwise be repetitive.

To reach a command line prompt, you will need to use a terminal emulator (commonly referred to as just a “terminal”), which is a program that emulates a text-mode terminal. If you are running a UNIX based system (Linux or Mac OSX), a terminal is available natively. In Linux, if you are using the graphical environment, you can run `xterm`, or you can switch to an actual text-mode terminal by pressing Ctrl-Alt-F1 (to switch back to the graphical interface, you can press Ctrl-Alt-F7). If you are running Mac OSX, you can run the Terminal application (typically found under Applications → Utilities).

If you are running Windows, there are some command line options (typically called `cmd` or `command`, depending the version of Windows), however, these tend to be quite simplistic by UNIX standards. You could install a tool called Cygwin, which provides the basics of a UNIX environment if you wanted. However, if you have access to a UNIX server (e.g., if you are taking a class and your teacher has set one up for your to work on), it is typically easier to just log into the server remotely and work there. This is explained in more detail in Section A.12.

Once you have started your terminal, it should display a command prompt (or just “prompt” for short). Figure A.1 shows a picture of a typical command prompt. The prompt not only lets you know that the shell is ready for you to give it a command, but also provides some information. In this case, it gives the current username (drew, displayed before the @) and the hostname of the system you are on (in this case, the system is named fenrir, displayed after the @). It then has a : and the current directory. In this case, the current directory is ~, which is UNIX shorthand for “your home directory” (which we will elaborate on momentarily). After that, the $ is the typical symbol for the end of the prompt for a typical user, indicating that a command can be entered. The grey box is the cursor, which indicates where you are typing input. The cursor blinks, which is not shown in the figure.

The prompt displays this information since it is typically useful to know immediately without having to run a command to find out. While it may seem trivial to remember who you are, or what computer you are on, it is quite common to work across multiple computers.\(^3\) For example, a developer may have one terminal open on their local computer, one logged into a server shared by their development team, and a third logged into a system for experimentation and testing. Likewise, one may have multiple usernames on the same system for different purposes.\(^4\) Exactly what information the prompt displays is configurable, which we will discuss briefly later.

\(^2\)The title of this section was borrowed from an essay written by Neal Stephenson in 1999 with the same title. It’s a little dusty these days, but still a very good read if you’re curious about what an operating system is and the history of how we’ve ended up with the OS options we have.

\(^3\)Type the command “hostname” at the prompt to discover the full name of the machine you are logged into.

\(^4\)Type the command “whoami” at the prompt to discover the username currently logged in.
Appendix A: UNIX Basics

A.2 Getting Help: man and help

The first commands we will learn are those which provide built in help. The first, and most versatile
of these is the man command (which is short for “manual”). This command displays the manual
page (“man page” for short) for whatever you request—commands, library functions, and a variety
of other important topics. For example, if you type man -S3 printf, then your computer will
display the man page for the printf function from the C library.

Before we discuss the details of the man command, we will take a brief aside to discuss command
line arguments. Like many UNIX commands, man takes arguments on the command line to specify
exactly what it should do. In the case of man, these arguments (typically) specify which page (or
pages) you want it to display for you. In general, command line arguments are separated from the
command name (and each other) by white space (one or more spaces or tabs). In the example
above, we gave the man command two arguments: -S3 and printf.

Of these two arguments, the first is an “option.” Options are arguments which differ from
“normal” arguments in that they start with a - and change the behavior of the command, rather
than specifying the typical details of the program (such as which page to display or what file to
act on). In the particular example above, the -S3 argument tells man to look in section 3 of the
manual, which is dedicated to the C library.

Before we delve into the options and the details of the various sections of the manual, we will look
at what the manual displays in a bit more detail. Figure A.2 shows the output of man -S3 printf.
This page actually has information not only for printf, but also for a variety of related functions,
which are all listed at the top of the page. The SYNOPSIS section lists the #include file to use,
as well as the functions’ prototypes. At the bottom of the screen is the start of the DESCRIPTION
section which describes the behavior of the function in detail. This description runs off the bottom
of the screen, but you can scroll up and down with the arrow keys. You can also use d and u to
scroll a page at a time. You can also quit by pressing q. These are the most important and useful
keys to know, but there are a variety of other ones you can use, which you can find out about by
pressing h (for help).

If you were to continue scrolling down through the man page for printf, you would find out
everything you could ever want to know about it (including all the various options and features for
the format string, what values it returns under various conditions, etc.). We are not interested in
the details of printf for this discussion, only that the man page provides them.

The manual includes pages on topics other than just the C library, such as commands. For
example, in Section A.3, we will introduce the ls command. If you wanted to know more details
of this command, you could do man ls to read about it. The manual page describes what command
line arguments ls expects, as well as the details of the various options it accepts.

Now You Try: Man Pages

Read the man page for ls. Find out what options you can give the ls command
to (a) have it list in “long format” (with more details) and (b) use unit suffixes
for Megabytes, Gigabytes, etc...when it lists the sizes.
Appendix A: UNIX Basics

Unlike `printf`, we did not specify a section of the manual for `ls`. In fact, not specifying the section explicitly is the common case—`man` will look through the sections sequentially trying to find the page we requested. If there is nothing with the same name in an earlier section of the manual, then you do not need to specify the section. In the case of `ls`, the page we are looking for is in Section 1—which has information about executable programs and shell commands. In fact, when you run `man ls`, you can see that it found the page in section 1 by looking in the top left corner, where you will see `LS(1)`. The `(1)` denotes section 1 of the manual.

If we just type `man printf` we get the man page for the `printf` command from section 1 ("`printf(1)`"). This page corresponds to the executable command `printf` which lets you print things at your shell. For example, you could type `printf "Hello %d\n" 42` at your shell and it would print out `Hello 42`. While this may not seem useful Section A.10 introduces “shell scripts” which can automate various tasks. When writing a script, it might be useful to print information out such as this. Since `man` finds this page first, if we want the C library function `printf` (for example, if we are programming and need to look up a format specifier that we do not remember), we need to explicitly ask for section 3 with the `-S3` option, as section 3 has C library reference.

So far, we have seen two section of the manual: 1 which is for executable programs and shell commands, and 3 which is for C library function reference. How would we find these out if we did not have this book handy? Also, how do we find out about the other sections of the manual? The `man` command, like most other commands has its own manual page too, so we could just read that. In fact, if we type `man man`, the computer will display the manual page for the `man` command.

Scrolling down a screen or so into the `DESCRIPTION` section shows the following table of sections:
Appendix A: UNIX Basics

The table below shows the section numbers of the manual followed by the types of pages they contain.

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions eg /etc/passwd
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

Scrolling down further in the manual will show various examples of how to use man, as well as the various options it accepts.

New users of the man system often face the conundrum that reading a man page is great for the details of something if you know what you need, but how do you find the right page if you do not know what you are looking for? There are two main ways to find this sort of information. The first is to use the -k option, which asks man to do a keyword search. For example, suppose you wanted to find a C function to compare two strings. Running the command man -k compare lists about 56 commands and C library functions that have the word “compare” in their description. You can then look through this list, find things that look relevant, and read their respective pages to find the details.

The other way to find things is to look in the SEE ALSO section at the end of another page if you know something related but not quite right. This section, which you can find at the end of each man page, lists the other pages which the author thought were relevant to someone reading the page she wrote.

Now You Try: Searching The Man Pages

Use man -k to find a command which will omit repeated lines from its input.

A.3 Directories

The discussion of the prompt introduced three important concepts: directories, the current directory, and the user’s home directory. Directories are an organizational unit on the filesystem, which contain files and/or other directories. You may be familiar with the concept under the name “folder”, which is the graphical metaphor for the directory. The actual technical term, which is the correct way to refer to the organizational unit on the filesystem is “directory”. Folder is really only appropriate when referring to the iconography used in many graphical interfaces.

To understand the importance of the “current directory,” we must first understand the concept of path names—how we specify a particular file or directory. In UNIX, the filesystem is organized in a hierarchical structure, starting from the root, which is called / . Inside the root directory, there are other directories and files. The directories may themselves contain more directories and files, and so on. Each file (or directory—directories are actually a special type of file) can be named with a path. A path is how to locate the file in the system. An absolute path name specifies all of the directories that must be traversed, starting at the root. Components of a path name are
separated by a / . For example, /home/drew/myfile.txt is an absolute pathname, which specifies the myfile.txt inside of the drew directory, which is itself inside of the home directory, inside the root directory of the file system.

The “current directory” (also called the “current working directory” or “working directory”) of a program is the directory which a relative path name starts from. A relative path name is a path name which does not begin with / (path names which begin with / are absolute path names). Effectively, a relative path name is turned into an absolute path name by prepending the path to the current directory to the front of it. That is, if the current working directory is /home/drew then the relative path name textbook/chapter4.tex refers to /home/drew/textbook/chapter4.tex.

All programs have a current directory, including the command shell. When you first start your command shell, its current directory is your home directory. On a UNIX system, each user has a home directory, which is where they store their files. Typically the name of user’s home directory matches their user name. On Linux systems, they are typically found in /home (so a user named “drew” would have a home directory of /home/drew). Mac OSX typically places the home directories in /Users (so “drew” would have /Users/drew). The home directory is important enough that it has its own abbreviation, ~. Using ~ by itself refers to your own home directory. Using ~ immediately followed by a user name refers to the home directory of that user (e.g., ~fred would refer to fred’s home directory).

<table>
<thead>
<tr>
<th>Now You Try: Current Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use the pwd command to find out what the current working directory of your command shell is.</td>
</tr>
</tbody>
</table>

There are a handful of useful directory-related commands that you should know. The first is cd, which stands for “change directory.” This command changes the current directory to a different directory that you specify as its command line argument (recall from earlier that command line arguments are written on the command line after the command name and are separated from it by white space). For example, cd / would change the current directory to / (the root of the filesystem). Note that without the space (cd/) the command shell interprets it as a command named “cd/” with no arguments, and gives an error message that it cannot find the command.

The argument to cd can be the pathname (relative or absolute—as a general rule, you can use either) for any directory that you have permission to access. We will discuss permissions in more detail shortly, but for now, it will suffice to say that if you do not have permission to access the directory that you request, cd will give you an error message and not change the directory.

Another useful command is ls which lists the contents of a directory—what files and directories are inside of it. With no arguments, ls lists the contents of the current directory. If specify one or more path names as arguments, ls will list information about them. For path names that specify directories, ls will display the contents of the directories. For path names that specify regular files, ls will list information about the files named.

Figure A.3 shows an example of using the cd and ls commands. The first command in the example is cd examples, which changes the current directory to the relative path examples. Since the current directory is /home/drew, this makes an absolute path of /home/drew/examples (which is called ~/examples for short). On the second line, you can see that the prompt now shows the current directory as ~/examples. The second command is ls, which lists the contents of the examples directory (since there are no arguments, ls lists the current directory’s contents). In this example, the current directory has 2 directories (dir1 and dir2) and 2 regular files (myfile.c and myfile.txt) in it. The default on most systems is for ls to color code its output: directories are
shown in dark blue, while regular files are shown in plain white. There are other file types, which are also shown in different colors.

The \texttt{ls} command (like \texttt{man}, and many other UNIX commands) also can take special arguments called “options”. For example, for \texttt{ls} the `-1` option requests that \texttt{ls} print extra information about each file that it lists. The `-a` option requests that \texttt{ls} list all files. By contrast, its default behavior is to skip over files whose names begin with . (\textit{i.e.}, a dot). While this behavior may seem odd, it arises from the UNIX convention that files are named with a . if and only if you typically do not want to see. One common use of these “dot files” is for configuration files (or directories). For example, a command shell (which parses and executes the commands you type at the prompt) maintains a configuration file in each user’s home directory. For the command shell \texttt{bash} (see Section A.9.), this file is called \texttt{.bashrc}. For the command shell \texttt{tsch} (see Section A.9.), this file is called \texttt{.cshrc}.

The other common files whose names start with . are the special directory names . and .. In any directory, . refers to that directory itself (so \texttt{cd .} would do nothing—it would change to the directory you are already in). This name can be useful when you need to explicitly specify something in the current directory (.\texttt{/myCommand}). The name .. refers to the \textit{parent} directory of the current directory—that is, the directory that this directory is inside of. Using \texttt{cd ..} takes you “one level up” in the directory hierarchy. The exception to this is the .. in the root directory, which refers back to the root directory itself, since you cannot go “up” any higher.

The \texttt{ls} command has many other options, as do many UNIX commands. Over time, you will become familiar with the options that you use frequently. However, you may wonder how you find out about other options that you do not know about. Like most UNIX commands, \texttt{ls} has a man page (as we discussed in Section A.2) which describes how to use the command, as well as the various options it takes. You can read this manual page by typing \texttt{man ls} at the command prompt.

Two other useful directory-related commands are \texttt{mkdir} and \texttt{rmdir}. The \texttt{mkdir} command takes one argument and creates a directory by the specified name. The \texttt{rmdir} command takes one argument and removes (deletes) the specified directory. To delete a directory using \texttt{rmdir}, the directory must be empty (it must contain no files or directories, except for . and .. which cannot be deleted).
Appendix A: UNIX Basics

Now You Try: Directory Commands

If you are not already in your home directory, cd to it.

- Make a directory called example
- List the contents of your current directory (you should see the example directory you just made)
- Use cd to change directories into the example directory
- Use ls to look at the contents of your new current directory.
- Use cd .. to go back up one level
- Remove the example directory that you created.

A.4 Displaying Files

Now that we have the basics of directories, we will learn some useful commands to manipulate regular files. We will start with commands to display the contents of files: cat, more, less, head, and tail.

The first of these, cat, reads one or more files, concatenates them together (which is where it gets its name), and prints them out. As you may have guessed by now, cat determines which file(s) to read and print based on its command line arguments. It will print out each file you name, in the order that you name them.

If you do not give cat any command line arguments, then it will read standard input and print it out. Typically, standard input is the input of the terminal that you run a program from—meaning it is usually what you type. If you just run cat with no arguments, this means it will print back what you type in. While that may sound somewhat useless, it can become more useful when either standard input or standard output (where it prints: typically the terminal’s screen) are redirected or piped somewhere else. We will discuss redirection and pipes in Section A.7.

While you can use cat to display the contents of a file, you typically want a bit more functionality than just printing the file out. The more command displays one screenfull and then waits until you press a key before displaying the next screenfull. It gets its name from the fact that it prompts ---More-- to indicate that you should press a key to see more text. The less command supercedes more and provides more functionality: you can scroll up and down with the arrow keys, and search for text. Many systems actually run less whenever you ask for more.

There are also commands to show just the start (head) or just the end (tail) of a file. Each of these commands can take an argument of how many lines to display from the requested file. Of course, for full details on any of these commands, see their man pages.

Note that these commands just let you view the contents of files. We will discuss editing files in Appendix ??

Now You Try: Looking at Files

UNIX has a system dictionary, in /usr/share/dict/words (which contains one word per line). Use the head command to print the first 20 lines of this file. Use the tail command to print the last 25 lines of this file.
A.5 Moving, Copying, and Deleting

Another task you may wish to perform is to move (mv), copy (cp), or delete (rm—stands for “remove”) files. The first two take a source and a destination, in that order. That is where to move (or copy) the file from, followed by where to move (or copy) it to. If you give either of these commands more than 2 arguments, they assume that the first N-1 are sources, and the last is the destination, which must be a directory. In this case, each of the sources is moved (or copied) into that directory, keeping its original filename.

The rm command takes any number of arguments, and deletes each file that you specify. If you want to delete a directory, you can use the rmdir command instead. If you use rmdir, the directory must be empty—it must contain no files or subdirectories (other than . and ..). You can also use rm to recursively delete all files and directories contained within a directory by giving it the -r option. Use rm with care: once you delete something, it is gone.5

Now You Try: Basic File Movements

- Copy the system dictionary to your home directory.
- Rename (move) the copy you created to have the name mydictionary (note: you don’t actually need two separate steps: you can specify this name when you copy).
- Use ls to look at the contents of your home directory
- Delete mydictionary

A.6 Pattern Expansion: Globbing and Braces

You may (frequently) find yourself wishing to manipulate many files at once that conform to some pattern—for example, removing all files whose name ends with ~ (editors typically make backup files while you edit by appending ~ to the name). You may have many of these files, and typing in all of their names would be tedious.

Because these names follow a pattern, you can use globbing—patterns which expand to multiple arguments based on the file names in the current directory—to describe them succinctly. In this particular case, you could do rm *~ (note there is no space between the * and the ~; doing rm * ~ would expand the * to all files in the directory, and then ~ would be a separate argument after all of them). Here, * is a pattern which means “match anything”. The entire pattern *~ matches any file name (in the current directory) whose name ends with ~. The shell expands the glob before passing the command line arguments to rm—that is, it will replace *~ with the appropriately matching names, and rm will see all of those names as its command line arguments.

There are some other UNIX globbing patterns besides just *. One of them is ? which matches any one character. By contrast, * matches any number (including 0) of characters. You can also specify a certain set of characters to match with [...], or to exclude with [!...]. For example, if you were to use the pattern file0[123].txt it would match file01.txt, file02.txt, and

5Many machines that are maintained by an IT staff do have periodic backups. In the event that you have accidentally/tragically deleted something that you desperately need again, it is worth contacting your IT department to see whether there might be an available backup. They might, for example, be able to provide you with a snapshot of the deleted files as they looked at midnight the day before.
file03.txt. If you did file0[!123].txt, then it would not match those names, but would match names like file09.txt, file0x.txt, or file0..txt (and many others).

Sometimes, you may wish to use one of these special characters literally—that is, you might want to use * to mean just the character *. In this case, you can escape the character to remove its special meaning. For example, rm \* will remove exactly the file named *, whereas rm * will remove all files in the current directory.

Another form of pattern expansion that UNIX supports is brace expansion. Brace expansion takes a list of comma-separated choices in curly braces, such as {a,b,c} and replaces the surround argument with one version for each item in the list, using that item in place of the list. For example rm file{1,a,X}.txt would expand to rm file1.txt filea.txt fileX.txt. This particular example could be accomplished with globbing as well (using rm file[1aX].txt), however, there are uses for brace expansion which globbing is ill-suited for.

One major difference between globbing and brace expansion is that globbing operates on the file names in the local directory. Suppose you wanted to copy some specific files from a remote computer. As we will discuss in Section A.12, the scp program lets you securely copy files from one computer to another. You could do scp user@computer::~/file{1,2,3}.txt ./ to copy three files (file1.txt, file2.txt, and file3.txt). Globbing is not appropriate here, since you don’t want to expand based on the names of local files.

Brace expansion is also useful when the choices are longer than one character each. For example, rm dir1/dir2/{abc,xyz}.txt. Brace expansion can also be used multiple times in one argument, in which case you get all possible pairings of the expansions. For example {a,b,c}{1,2,3} expands to 9 arguments (a1, a2, a3, b1, b2, b3, c1, c2, c3).

**Now You Try: Expansions**

- Use brace expansion and the echo command (which prints is arguments) to print all 9 combinations of chicken, turkey, and beef with cheddar, swiss, and blue.
- List all of the files in /bin whose names start with s.

### A.7 Redirection and Pipes

When you run a program under UNIX, it has access to three “files” by default: stdin, stdout, and stderr. In the typical scheme of things, all three of these are connected to the terminal in which the program is running. stdin can be read for input from the user typing at the terminal, and stdout and stderr can be printed to to write output to the terminal, with the former nominally being for error-related printing, and the later for everything else.

However, where these files read and write can be redirected on the command line where you run the program. Redirecting the input or output of a program means that instead of the file reading from/writing to the terminal’s keyboard/screen, it will read/write the file you request instead. Redirection is accomplished with the < (for input) and/or > (for output) operators. For example, ./myProgram < file1.txt > output.txt runs the program myProgram with its input redirected from file1.txt and its output redirected to output.txt.

You can also redirect stderr by using 2>. The reason for the 2 is that stderr is file descriptor number 2. In yet another example of how everything is a number (the key lesson of Chapter 3), programs communicate with the operating system kernel about files in terms of file descriptors—numeric handles representing open files. When a program opens a file, the OS kernel returns a file
Appendix A: UNIX Basics

descriptor which the program uses for all future requests about that file until it closes it. Note that while in Chapter ?? we discuss IO in terms of FILE*s, these actually are structures which wrap the file descriptor in more state for the C library. Standard input, output, and error are just file descriptors (0, 1, and 2 respectively) that are open before the program starts.

You can, in fact, redirect other file descriptors other than the standard three. For example, if you wrote ".cmd 3< f1 4> f2", it would open the file f1 for reading as file descriptor 3 and f2 for writing as file descriptor 4 before starting the program. You can also use the <> operator (possibly with a number before it) to redirect a file descriptor for both reading and writing. The advanced behaviors described in this paragraph are relatively uncommon as few programs expect such file descriptors to be open when the program starts.

Two more commonly used features of redirection are >>, which redirects the output to a file, but appends to the original contents rather than erasing it, and 2>&1 which redirects one file descriptor (in this case 2—stderr) to refer to exactly the same file as another (in this case 1—stdout).

UNIX also supports a special form of input redirection called a “here document.” A here document lets you write a literal multi-line input for the program, and redirect its input to be what you wrote. Redirecting input with a here document involves the << operator, followed by the “here tag”—the word that you will use to indicate where the here document ends. While this tag can be anything (that does not appear on a line by itself in the input), it is traditionally EOF (which stands for “end of file”). For example:

```
 1 drew@fenrir:˜$ cat << EOF
 2 > This is a here document
 3 > which will all serve as the input for the cat program.
 4 > Until it ends with the here tag on a line by itself
 5 > (which is right below this)
 6 > EOF
```

The above would run the cat command with its input redirected to be the multiple lines of text between the two EOF markers. Note: the “>” characters above are not entered by the user. They appear at the beginning of each line as a sub-prompt to complete the here document. In some settings, this prompt might be a “?”. When run with no arguments, cat reads standard input (in this case, the text of the here document) and prints it out. Here documents can be quite useful when writing scripts, which are basically programs in the shell. We will discuss them in more detail in Section A.10

Another way that the inputs/outputs of programs can be manipulated is with pipes. A pipe connects the output of one program to the input of another program. Using a pipe from the command shell is a matter of placing the | (read “pipe”) between two commands. The output of the first command becomes the input of the second command. For example, diff x.c y.c | less runs the command diff x.c y.c, which prints the differences between the two files x.c and y.c, however, since the output is piped to less, it will serve as less’s input. With no arguments, less reads stdin and lets you scroll around in it. This entire command line lets you scroll through the differences between the files, which may be quite useful if there are a large set of differences.

It would be possible to achieve a similar effect with redirection and two commands: diff x.c y.c > temp then less temp, however there are subtle, yet important, differences. With the redirection approach, the diff command is run completely, writing to a file on disk, then the less command is run using that file as input. With the pipe approach, the two programs are run at the same time, with the output from diff being passed directly to less through the OS kernel’s memory. This distinction may make a significant difference in speed and disk-space used if the output of the first command is quite large. The pipe approach is also more convenient to type.
You can build command pipelines with more than two commands—connecting the output of the first to the input of the second, the output of the second to the input of the third, and so on. In fact, command pipelines with three or four commands are quite common amongst experienced UNIX users. Part of the UNIX philosophy is to make commands which perform one task well, and connect them together as needed.

Note that the command shell processes redirections and pipes before the requested program actually starts. They are not included in the command line arguments of the program.

### Now You Try: Pipes and Redirection

- Use echo and redirection to create a file called myName.txt with your name in it.
- Use head to print the first 5000 words of the system dictionary, and then pipe the output to tail so that you only see the last 300 of those 5000.
- Perform the previous example, but pipe that output to less, so that you can scroll through the results.

### A.8 Searching

One common and important task when using a computer is searching for things. For example, suppose you have many C source files, and you want to search through them to see where you called `myFunction`. You could use the `grep` command, which searches one or more files (or standard input if you do not specify any file names) for a particular pattern. The simplest of patterns is a literal string: `myFunction` matches exactly itself. Therefore, you could do `grep myFunction *.c`, and it would search in all files ending with `.c` in the current directory (recall that the shell expands the * glob), and print out each matching line as well as the file in which it occurs.

The previous example is quite useful, but is just a taste of the power of `grep`. The patterns that `grep` can search for are not limited to just exactly matching one string, but rather support more general patterns. Grep, and a variety of other tools that use similar patterns, describe them as “regular expressions” (“regexps” for short), which is mostly true—technically speaking, grep’s patterns support features which go beyond the capabilities of true regular expressions. As one contrived example, suppose you wanted a list of all words in the English language with any 4 characters, then `w`, then any 3 characters. You could use `grep` to search the system dictionary (`/usr/share/dict/words`) for a regexp that matches exactly this criteria:

```
grep '^\{4\}w\{3\}$' /usr/share/dict/words
```

This pattern may seem complex, but is really a few simple pieces strung together. The `s around the outside of the pattern tell the command shell that we do not want it to interpret special characters in that argument, but rather pass it as-is to grep. The `^` at the start of the pattern matches the start of the line. The `.` matches any character, and is followed by `\{4\}` which specifies 4 repetitions of the prior pattern (we could have instead written `\.{4}` if we wanted). The `w` matches exactly the letter `w`. The `\{3\}` matches any three characters, in the same way as the `\{4\}` matched any 4 characters. Finally the `$` at the end matches the end of the line. Without the `^` and `$` we could match the rest of the pattern anywhere in a line (which we might want sometimes).

Our goal here is not to discuss all the intricacies of `grep`, nor the possibilities for its patterns, but rather to introduce you to the tool, and let you know that you can search for rather complex...
Appendix A: UNIX Basics

patterns if you need to. We will note that regexps and the shell use special characters (*, {, etc) for different purposes. Often you will want to enclose your pattern in ' to prevent the shell from expanding globs and braces, and applying other special meanings to characters in your patterns.

Another type of searching that you might want to do is to find files that meet a specific criteria. One might be inclined to approach this by using ls and piping the output to grep. Such an approach is possible (and looking in the man page for ls shows that the -R option makes it recursively look through subdirectories). This approach could work, as long as you only want the criteria to include the name of the file you are looking for, though even then, it is not the best way.

A better way is to use the find command, which takes the criteria to look for, and the path to look in. The criteria can be the name of the file, or other things like “find files newer than some specific file.” The criteria to look for are specified as options to find—for example -name pattern specifies to find files whose name matches pattern. The -name pattern is one of the most commonly used ones, and the pattern can include shell glob patterns. However, these must be escaped with a \ to prevent the shell from expanding them before passing the argument to find. Again, we are not going to go into the details of find here, but want you to know that it exists, and you can read all about it in its manpage if you need to.

Now You Try: Searching

- Use grep to find all the words in the system dictionary that have “sho” anywhere in them.
- Use grep to find all the words in the system dictionary that have “sho” at the start.
- Use grep to find all the words in the system dictionary that have an “s”, followed by 0 or more characters, followed by an “h”, 0 or more characters, then an “o” (Note: the regexp for this pattern is s.*h.*o).
- Use the find command to list all files in /usr with “net” in their names somewhere.

A.9 Command Shells

We’ve been a little vague about the command line. The truth of the matter is that when you type command at a terminal prompt, there is a program that parses, interprets, and executes these commands for you. This program is called a command shell. At a minimum, a UNIX command shell supports all UNIX commands (such as cd, ls, rm, etc.). However, most UNIX command shells provide more sophisticated features, effectively forming a programming language of their own. This programming language allows an experienced user to write “shell scripts” which contain algorithms implemented in shell commands to automate tasks (which in some cases may be quite complex).

There are a variety of command shells. One of the most popular is bash. Another, slightly older but still rather prevalent is tcsh (pronounced “tee-see-shell”). We will briefly introduce both to you. Command shell preferences (much like text editor preferences) can be a heated topic. A quick internet search will supply you with hours of arguments about which is better. We recommend being pragmatic in your choice. If those around you (co-workers, friends, TAs, instructors) are all gravitating towards a particular shell, this is the one you should use. It increases the amount of help you can get from and give to others, and it decreases the number of problems which may arise due to differences in the shells.

All of Programming, http://aop.cs.cornell.edu 95
Both Linux and Mac OSX should run bash by default, unless you have changed your default shell. If they run some other shell, you can just type bash at the prompt to run a bash shell (it too is a program, just like any other). If you are running Windows, bash is not built in.

As a final note, bash and tcsh are only two of many command shells, most ending in sh. To name just a few: sh, csh, zsh, and dash. Become familiar with one; dabble with the rest on a need to know basis only.

A.10 Scripting

UNIX command shells are not simply an interface to run programs, they are a kind of programming language themselves. Programs written in command shells are called scripts. These scripts contain programs built from shell commands, often involving running other programs. The shell scripting language has most of the programming constructs you would expect from learning to program in C—variables, conditional statements, loops, and functions. As with many things in this appendix, our goal is not to provide a comprehensive guide to the topic, but to introduce you to the idea so that you can seek out more information when the tool is useful to you. This section will specifically discuss bash scripts, but the code examples will be given in both bash (on the left) and tcsh (on the right) in order to give you some familiarity with the latter and to show you how various scripting languages differ. Note that we do not expect (or even suggest) you to learn both of these. Instead, we present both for the eventuality where you search for how to perform some task and find results in a shell that is not the one you use—you will have at least seen that there are different shells, and that they generally provide similar functionality, even if with slightly different syntax.

As with most programming languages, shell scripts have variables. Unlike C, bash scripts are untyped. You do not declare the types of your variables—nor even declare the variables before you use them. To assign to a variable, you simply write variable=value. Unlike C, bash does not require a semicolon to end a statement. Instead a statement may be terminated by either a newline or a semicolon.

Using a variable in bash requires putting a dollar sign ($) before the variable’s name. For example, we could do the following:

<table>
<thead>
<tr>
<th>bash</th>
<th>tcsh</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 variable=&quot;hello world&quot;</td>
<td>1 set variable=&quot;hello world&quot;</td>
</tr>
<tr>
<td>2 echo $variable</td>
<td>2 echo $variable</td>
</tr>
</tbody>
</table>

This very simple script assigns the string "hello world" to the variable variable, and then runs the command echo $variable, which the shell expands to echo hello world before running the command (Recall that echo is a program which simply prints out its command line arguments). So the scripts behavior is to print hello world. You can try this at your command shell, or you can write these commands into a file, save it and run it.

If you save a script into a file, you need to make the file executable in order to be able to run it. UNIX tracks permissions for files, and by default they are not executable (though when you compile programs with a compiler like gcc, it adds execution permissions at the end of linking the binary). We will not go into the full details of permissions here, but just mention that you can run chmod u+x filename to add execute permissions for the user who owns the file (typically, the owner is you if you just created it).

If you create bash scripts, it is convention (though not required) to name them with .sh at the end of the name. Following this conventions makes it easy for people (including yourself) to realize that the file is an executable shell script, and can not only be run, but also read by a human (as compared to a compiled binary program, which is not human readable).
Appendix A: UNIX Basics

Additionally when you save a script in a file, you should start it with a line indicating what program should interpret the script. Such a line starts with `#!` and then has the full path of the program capable of running the script. Note that the `#` is read “hash” or “pound” and the `!` is read “bang”, so the `#!` combination is either read “pound-bang” or “hash-bang”, with the later sometimes shortened to “shebang”.

For a bash script, this line would read `#!/bin/bash`. This line lets the kernel know that the script should be interpreted by bash. You can write scripts for other shells (which have different syntaxes), or other scripting languages, such as perl. Note that `#` is “comment to end of line” in bash (and most other scripting languages), so the line will have no effect in the script itself. If no such line is present, then it will be run by the default shell. In our example, the complete script would look like this:

```
#!/bin/bash
variable="hello world"
echo $variable
```

As with much programming, the most usefulness comes when we can have the computer repeat tasks for us. bash has loops to repeat tasks with variations. The most common loop in bash is the `for` loop, although there is also a `while` loop. bash’s `for` loop behaves slightly differently from C’s. In bash and tcsh, the syntaxes are:

```
for variable in alist
do
  commands
done
```

Here, `variable` can be whatever variable name you want. The loop will iterate once per item in the `alist` (in order), with the current item being assigned to the variable before executing the commands that form the loop body. For example,

```
for i in oneFish twoFish redFish
done
```

will print

```
Current fish is oneFish
Current fish is twoFish
Current fish is redFish
```

Note that the list of things can be the value of a variable, shell glob (e.g., `*.c`), or the output of command—using back-tick expansion. When you write a command inside of back-ticks (`, the character that shares a key with the tilde, on the far left of the numbers row on an American keyboard), bash runs that command, and replaces the back-tick expression with the output of that command.

The following example uses back-tick expansion to run the command `find . -name ".c"` (finding all `.c` files in the current directory and its sub-directories). The output of this `find` command becomes the list of things that the `for` loop iterates over:

```
for i in *.c
done
```

will print

```
All of Programming, http://aop.cs.cornell.edu
```
To return to our motivating example at the start of this appendix—renaming all files in the current directory to replace _ with -. If we take a second to introduce the tr command, we now have the skills to do this task with a quick for loop. In its simplest usage, the tr command takes two arguments—the first is a list of characters to replace, and the second is the list of characters to replace them with. It reads standard input, and for each character that it reads, it either prints its replacement (if that character appears in the first list of characters, it prints the corresponding character from the second list), otherwise it prints the character unmodified. With this command, we can use a for loop to iterate over all the files in the current directory, use the mv command to rename them, and use back-tick expansion and tr to compute the new name.

While this loop may seem a bit unfamiliar to you now, as you gain experience with shell scripting, such a command will come naturally to you whenever you need to perform a repetitive task. If you want to count over numbers (as you would with a for loop in C), you can use the seq command and back-tick expansion to generate the list of numbers that you want to iterate through.

We could write an entire book on shell scripting, but that is not the purpose of this text. Instead, we will suggest that those interested in reading more about shell scripting consult the wealth of existing resources available on the Internet. One such resource is the Advanced Bash Scripting Guide available on the Linux Documentation Project web-site.

### A.11 Environment Variables

Some variables have special meaning to the shell or certain programs. For example, the PATH variable specifies where the shell should look for programs to execute. When you type a program name without any directory (_e.g., ssh has no directory in its name, as opposed to ./myProgram which names a particular directory), the shell searches through the components of the PATH in order, looking for a matching program name. If it finds one, it runs it. Otherwise, it reports an error. You can see what your current PATH is by echo $PATH (since PATH is a variable, and echo prints its command line arguments). An example PATH is

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Notice that the value of the PATH is a colon delimited list of directory names.

Another variable which controls the shell’s behavior is IFS—the internal field separator. This variable controls how the shell divides input up into fields. Consider the following loop:

```
for i in `cat someFile`
done
```

When bash goes to execute the loop, it has to split up the output of cat someFile into fields (what to set i to for each iteration of the loop). The current value of IFS controls how this splitting
is done. The default value of IFS causes the input to be split into fields at any whitespace. However, you might want to split the fields differently: at only newline (IFS="\n"), at commas (IFS=','), or at some other separator you desire.

There are a variety of other environment variables, and we will of course not go into them all here. However, we will mention two useful things to understand about environment variables. First, most variables are local to the shell you run them in. By default, the variables will not be passed down to programs that you run from within the shell. If you want a variable to appear in the environment of commands you run, you should export it. Typically this is done when the variable is assigned (export myVar="hi"), but can be done later (myVar="hi" ... export myVar).

Second, you can read (and manipulate) environment variables from programs that you write. One way to do so is with the `getenv` function (from `stdlib.h`), which takes the name of an environment variable, and gives you its value. You can also declare `main` to take a third argument `char ** envp`, which is a pointer to an array of strings containing the environment variables (in the form "variable=value").

### A.12 Remote Login: ssh

To log into a remote UNIX system, you will need to use an `ssh` program. `ssh` stands for “secure shell” and provides an encrypted connection to a terminal on a remote computer. When you “ssh into” another computer, you run an ssh client on your computer, which connects to an ssh server on the other computer. The client and server setup an encrypted session, and then you login with your username and password. These credentials are sent over the encrypted connection, so they are protected from attackers who might try to eavesdrop on the connection. Once you are authenticated, you can type commands in your local terminal, and the ssh program will encrypt them, and send them to the server. The server will execute the commands, encrypt the output, and send it back, where your client will decrypt it and display it.

If you are using a UNIX based system, sshing to a remote computer is just a matter of type `ssh username@servername` at the command prompt in your terminal (if your username is the same on both systems, you can omit the `username@` part). The ssh program will then ask you for your password (unless you have other authentication methods setup). After successfully authenticating, you will be provided with a command prompt on the remote system, and can execute commands on it as you desire.

If you are using a Windows machine, the easiest way to ssh is to find or download a program that is an `ssh` client. One example that is both open-source and commonly available is called PuTTY. Another is called `SSH Secure Shell`. There are more options than these. Often Windows machines maintained by major universities will have at least one of these installed, possibly residing in a directory called Utilities or Internet. Simply start the program and enter the appropriate information about the username and server you are trying to connect to.

A companion of `ssh` is `scp` which allows you to copy files securely from one computer to another over the same protocol as `ssh`. Use of `scp` is much the same as use of `cp` except that the source or destination file (or both) can be on another computer. You specify copying to/from another computer by prefixing the file name with `user@server:`. For example, for user `smith123` to copy a local file called `myFile` into a directory called `myDir` on a remote server called `myserver.edu`, she would type `scp myFile smith123@myserver.edu:myDir/`. There are many other features available for `ssh`. Consult `man ssh` and `man scp` for more details.
Index

abstraction, 45, 59, 74
algorithm, 2
arguments, 25
assignment statements, 24

binary point, 48
bit, 44
block, 28
body, 35
break, 37
call stack, 26
calling, 25
case, see switch-case
casting, 53
character literal, 47
characters, 47
command shell, 95
comments, 73
compiler, 46
composability, 75
conditional expressions, 31
continue, 37
corner cases, 66, 72
do-while loop, 35
doi-else, see if-else
enumerated type, 62
escape sequences, 30
expression, 24
fixed point, 48
for loop, 36
frame, 26
function, 25
global variable, 28
GUI, 29
hex, 46
hexadecimal, 46
identifier, 23
if-else, 31–32
initialization, 24
label, 32
least significant bit, 45
local variables, 27
loop, 35
lvalue, 24
main (function), 26
modulus, 24
null terminator, 56
octal, 51
parameter list, 25
parameters, 3
pixel, 57
printing, 29
rect_t, 59
recursion, 35
return statement, 25
return type(of a function), 25
return value, 26
RGB encoding, 57
rvalue, 24
scope, 27
selection expression, 32
semantics, 22
sign extended, 52
stack frame, 26
state, 22
statement coverage, 72
struct, 58
switch-case, 32–34
syntactic sugar, 34
syntax, 22
Index

tag, 59
testing, 9
top-down design, 75
type, 23, 44, 46
type conversion, 52, 53
type promotion, 52, 53
typedef, 61
value, 24
variable
  declaration, 23
  in expressions, 24
  initialization, 24
  uninitialized, 23
variables, 23

while loop, 35

zero extended, 52