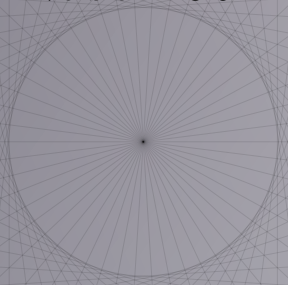
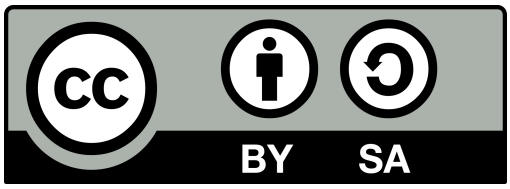
Computer Science I

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

2018/08/09 16:16:18
Version 1.3.6



Copyleft (Copyright)



The entirety of this book is free and is released under a **Creative Commons Attribution-ShareAlike 4.0 International License** (see http://creativecommons.org/licenses/by-sa/4.0/ for details).

Draft Notice

This book is a draft that has been released for evaluation and comment. Some of the later chapters are included as placeholders and indicators for the intended scope of the final draft, but are intentionally left blank. The author encourages people to send feedback including suggestions, corrections, and reviews to inform and influence the final draft. Thank you in advance to anyone helping out or sending constructive criticisms.

Preface

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

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

I've been teaching Computer Science since 2008 and was a Teaching Assistant long before that. Before that I was a student. During that entire time I've been continually disappointed in the value (note, not quality) of textbooks, particularly Computer Science textbooks and especially introductory textbooks. Of primary concern are the costs, which have far outstripped inflation over the last decade [30] while not providing any real additional value. New editions with trivial changes are released on a regular basis in an attempt to nullify the used book market. Publishers engage in questionable business practices and unfortunately many institutions are complicit in this process.

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

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

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

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

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

The exercises in this book are a variety of exercises I've used in my courses over the years. They have been made as generic as possible so that they could be assigned using any language. While some have emphasized the use of "real-world" exercises (whatever that means), my exercises have focused more on solving problems of a mathematical

nature (most of my students have been Engineering students). Some of them are more easily understood if students have had Calculus but it is not absolutely necessary.

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

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

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

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

Acknowledgements

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

This book is dedicated to my family.

Contents

Copyleft (Copyright) i

Draft Notice iii

Preface v

Acknowledgements ix

1. Introduction 1
- 1.1. Problem Solving . . . 2
- 1.2. Computing Basics . . . 4
- 1.3. Basic Program Structure . . . 5
- 1.4. Syntax Rules & Pseudocode . . . 12
- 1.5. Documentation, Comments, and Coding Style . . . 14

2. Basics 17
- 2.1. Control Flow . . . 17
  - 2.1.1. Flowcharts . . . 17
- 2.2. Variables . . . 18
  - 2.2.1. Naming Rules & Conventions . . . 19
  - 2.2.2. Types . . . 22
  - 2.2.3. Declaring Variables: Dynamic vs. Static Typing . . . 31
  - 2.2.4. Scoping . . . 32
- 2.3. Operators . . . 33
  - 2.3.1. Assignment Operators . . . 33
  - 2.3.2. Numerical Operators . . . 35
  - 2.3.3. String Concatenation . . . 37
  - 2.3.4. Order of Precedence . . . 38
  - 2.3.5. Common Numerical Errors . . . 38
  - 2.3.6. Other Operators . . . 40
- 2.4. Basic Input/Output . . . 41
  - 2.4.1. Standard Input & Output . . . 42
  - 2.4.2. Graphical User Interfaces . . . 42
  - 2.4.3. Output Using `printf()` -style Formatting . . . 43
  - 2.4.4. Command Line Input . . . 44

2.5. Debugging . . . 46
2.5.1. Types of Errors . . . 46
2.5.2. Strategies . . . 49
2.6. Examples . . . 50
2.6.1. Temperature Conversion . . . 50
2.6.2. Quadratic Roots . . . 51
2.7. Exercises . . . 51

3. Conditionals **65**
3.1. Logical Operators . . . 65
3.1.1. Comparison Operators . . . 66
3.1.2. Negation . . . 68
3.1.3. Logical And . . . 69
3.1.4. Logical Or . . . 70
3.1.5. Compound Statements . . . 71
3.1.6. Short Circuiting . . . 74
3.2. The If Statement . . . 75
3.3. The If-Else Statement . . . 76
3.4. The If-Else-If Statement . . . 78
3.5. Ternary If-Else Operator . . . 82
3.6. Examples . . . 82
3.6.1. Meal Discount . . . 82
3.6.2. Look Before You Leap . . . 83
3.6.3. Comparing Elements . . . 84
3.6.4. Life & Taxes . . . 85
3.7. Exercises . . . 87

4. Loops **95**
4.1. While Loops . . . 97
4.1.1. Example . . . 98
4.2. For Loops . . . 99
4.2.1. Example . . . 100
4.3. Do-While Loops . . . 100
4.4. Foreach Loops . . . 102
4.5. Other Issues . . . 103
4.5.1. Nested Loops . . . 103
4.5.2. Infinite Loops . . . 103
4.5.3. Common Errors . . . 105
4.5.4. Equivalency of Loops . . . 106
4.6. Problem Solving With Loops . . . 106
4.7. Examples . . . 107
4.7.1. For vs While Loop . . . 107
4.7.2. Primality Testing . . . 108
4.7.3. Paying the Piper . . . 109

4.8. Exercises . . . 111

5. Functions **133**
5.1. Defining & Using Functions . . . 134
5.1.1. Function Signatures . . . 134
5.1.2. Calling Functions . . . 136
5.1.3. Organizing . . . 137
5.2. How Functions Work . . . 137
5.2.1. Call By Value . . . 140
5.2.2. Call By Reference . . . 140
5.3. Other Issues . . . 142
5.3.1. Functions as Entities . . . 142
5.3.2. Function Overloading . . . 144
5.3.3. Variable Argument Functions . . . 145
5.3.4. Optional Parameters & Default Values . . . 145
5.4. Exercises . . . 146

6. Error Handling **151**
6.1. Error Handling . . . 153
6.2. Error Handling Strategies . . . 153
6.2.1. Defensive Programming . . . 153
6.2.2. Exceptions . . . 155
6.3. Exercises . . . 157

7. Arrays, Collections & Dynamic Memory **159**
7.1. Basic Usage . . . 160
7.2. Static & Dynamic Memory . . . 162
7.2.1. Dynamic Memory . . . 164
7.2.2. Shallow vs. Deep Copies . . . 166
7.3. Multidimensional Arrays . . . 166
7.4. Other Collections . . . 167
7.5. Exercises . . . 168

8. Strings **177**
8.1. Basic Operations . . . 177
8.2. Comparisons . . . 178
8.3. Tokenizing . . . 179
8.4. Exercises . . . 179

9. File Input/Output **183**
9.1. Processing Files . . . 183
9.1.1. Paths . . . 184
9.1.2. Error Handling . . . 185
9.1.3. Buffered and Unbuffered . . . 187

9.1.4. Binary vs Text Files . . . 187
9.2. Exercises . . . 188

10. Encapsulation & Objects **197**
10.1. Objects . . . 198
10.1.1. Defining . . . 198
10.1.2. Creating . . . 199
10.1.3. Using Objects . . . 200
10.2. Design Principles & Best Practices . . . 200
10.3. Exercises . . . 201

11. Recursion **203**
11.1. Writing Recursive Functions . . . 204
11.1.1. Tail Recursion . . . 205
11.2. Avoiding Recursion . . . 206
11.2.1. Memoization . . . 208
11.3. Exercises . . . 209

12. Searching & Sorting **211**
12.1. Searching . . . 211
12.1.1. Linear Search . . . 212
12.1.2. Binary Search . . . 213
12.1.3. Analysis . . . 215
12.2. Sorting . . . 220
12.2.1. Selection Sort . . . 221
12.2.2. Insertion Sort . . . 224
12.2.3. Quick Sort . . . 227
12.2.4. Merge Sort . . . 232
12.2.5. Other Sorts . . . 237
12.2.6. Comparison & Summary . . . 237
12.3. Searching & Sorting In Practice . . . 238
12.3.1. Using Libraries and Comparators . . . 238
12.3.2. Preventing Arithmetic Errors . . . 239
12.3.3. Avoiding the Difference Trick . . . 241
12.3.4. Importance of a Total Order . . . 242
12.3.5. Artificial Ordering . . . 242
12.3.6. Sorting Stability . . . 243
12.4. Exercises . . . 244

13. Graphical User Interfaces & Event Driven Programming **247**

14. Introduction to Databases & Database Connectivity **249**

I. The C Programming Language 251

15. Basics 253
15.1. Getting Started: Hello World . . . 253
15.2. Basic Elements . . . 254
15.2.1. Basic Syntax Rules . . . 255
15.2.2. Preprocessor Directives . . . 255
15.2.3. Comments . . . 258
15.2.4. The `main()` Function . . . 259
15.3. Variables . . . 260
15.3.1. Declaration & Assignment . . . 260
15.4. Operators . . . 262
15.5. Basic I/O . . . 263
15.6. Examples . . . 265
15.6.1. Converting Units . . . 265
15.6.2. Computing Quadratic Roots . . . 267

16. Conditionals 271
16.1. Logical Operators . . . 271
16.1.1. Order of Precedence . . . 273
16.1.2. Comparing Strings and Characters . . . 273
16.2. If, If-Else, If-Else-If Statements . . . 274
16.3. Examples . . . 276
16.3.1. Computing a Logarithm . . . 276
16.3.2. Life & Taxes . . . 277
16.3.3. Quadratic Roots Revisited . . . 279

17. Loops 283
17.1. While Loops . . . 283
17.2. For Loops . . . 285
17.3. Do-While Loops . . . 285
17.4. Other Issues . . . 286
17.5. Examples . . . 287
17.5.1. Normalizing a Number . . . 287
17.5.2. Summation . . . 287
17.5.3. Nested Loops . . . 288
17.5.4. Paying the Piper . . . 288

18. Functions 291
18.1. Defining & Using Functions . . . 291
18.1.1. Declaration: Prototypes . . . 291
18.1.2. Void Functions . . . 293
18.1.3. Organizing Functions . . . 294
18.1.4. Calling Functions . . . 294

18.2. Pointers . . . 295
18.2.1. Passing By Reference . . . 297
18.2.2. Function Pointers . . . 300
18.3. Examples . . . 301
18.3.1. Generalized Rounding . . . 301
18.3.2. Quadratic Roots . . . 303

19. Error Handling **305**
19.1. Language Supported Error Codes . . . 305
19.1.1. POSIX Error Codes . . . 306
19.2. Error Handling By Design . . . 308
19.3. Enumerated Types . . . 309
19.4. Using Enumerated Types for Error Codes . . . 310

20. Arrays **313**
20.1. Basic Usage . . . 313
20.2. Dynamic Memory . . . 316
20.3. Using Arrays with Functions . . . 318
20.4. Multidimensional Arrays . . . 319
20.4.1. Contiguous 2-D Arrays . . . 322
20.5. Dynamic Data Structures . . . 324

21. Strings **325**
21.1. Character Arrays . . . 325
21.2. String Library . . . 327
21.3. Arrays of Strings . . . 330
21.4. Comparisons . . . 331
21.5. Conversions . . . 332
21.6. Tokenizing . . . 333

22. File I/O **335**
22.1. Opening Files . . . 335
22.2. Reading & Writing . . . 336
22.2.1. Plaintext Files . . . 336
22.2.2. Binary Files . . . 338
22.3. Closing Files . . . 339

23. Structures **341**
23.1. Defining Structures . . . 341
23.1.1. Alternative Declarations . . . 342
23.1.2. Nested Structures . . . 343
23.2. Usage . . . 344
23.2.1. Declaration & Initialization . . . 344
23.2.2. Selection Operators . . . 346

23.3. Arrays of Structures . . . 348
23.4. Using Structures With Functions . . . 351
23.4.1. Factory Functions . . . 353
23.4.2. To String Functions . . . 354
23.4.3. Passing Arrays of Structures . . . 355

24. Recursion **357**

25. Searching & Sorting **361**
25.1. Comparator Functions . . . 361
25.2. Function Pointers . . . 365
25.3. Searching & Sorting . . . 372
25.3.1. Searching . . . 372
25.3.2. Sorting . . . 374
25.3.3. Examples . . . 374
25.4. Other Considerations . . . 377
25.4.1. Sorting Pointers to Elements . . . 377

II. The Java Programming Language 381

26. Basics **383**
26.1. Getting Started: Hello World . . . 384
26.2. Basic Elements . . . 385
26.2.1. Basic Syntax Rules . . . 385
26.2.2. Program Structure . . . 386
26.2.3. The `main()` Method . . . 389
26.2.4. Comments . . . 389
26.3. Variables . . . 390
26.3.1. Declaration & Assignment . . . 391
26.4. Operators . . . 393
26.5. Basic I/O . . . 395
26.6. Examples . . . 396
26.6.1. Converting Units . . . 396
26.6.2. Computing Quadratic Roots . . . 400

27. Conditionals **403**
27.1. Logical Operators . . . 403
27.1.1. Order of Precedence . . . 405
27.1.2. Comparing Strings and Characters . . . 406
27.2. If, If-Else, If-Else-If Statements . . . 407
27.3. Examples . . . 408
27.3.1. Computing a Logarithm . . . 408
27.3.2. Life & Taxes . . . 410

27.3.3. Quadratic Roots Revisited . . . 411

28. Loops **415**
28.1. While Loops . . . 415
28.2. For Loops . . . 417
28.3. Do-While Loops . . . 417
28.4. Enhanced For Loops . . . 418
28.5. Examples . . . 419
28.5.1. Normalizing a Number . . . 419
28.5.2. Summation . . . 419
28.5.3. Nested Loops . . . 420
28.5.4. Paying the Piper . . . 421

29. Methods **423**
29.1. Defining Methods . . . 424
29.1.1. Void Methods . . . 426
29.1.2. Using Methods . . . 426
29.1.3. Passing By Reference . . . 427
29.2. Examples . . . 428
29.2.1. Generalized Rounding . . . 428

30. Error Handling & Exceptions **431**
30.1. Exceptions . . . 431
30.1.1. Catching Exceptions . . . 431
30.1.2. Throwing Exceptions . . . 433
30.1.3. Creating Custom Exceptions . . . 433
30.1.4. Checked Exceptions . . . 434
30.2. Enumerated Types . . . 436
30.2.1. More Tricks . . . 437

31. Arrays **439**
31.1. Basic Usage . . . 439
31.2. Dynamic Memory . . . 442
31.3. Using Arrays with Methods . . . 442
31.4. Multidimensional Arrays . . . 443
31.5. Dynamic Data Structures . . . 444

32. Strings **449**
32.1. Basics . . . 449
32.2. String Methods . . . 450
32.3. Arrays of Strings . . . 452
32.4. Comparisons . . . 453
32.5. Tokenizing . . . 455

33. File I/O **457**
33.1. File Input . . . 457
33.2. File Output . . . 459

34. Objects **461**
34.1. Data Visibility . . . 462
34.2. Methods . . . 463
34.2.1. Accessor & Mutator Methods . . . 464
34.3. Constructors . . . 467
34.4. Usage . . . 469
34.5. Common Methods . . . 469
34.6. Composition . . . 472
34.7. Example . . . 474

35. Recursion **479**

36. Searching & Sorting **483**
36.1. Comparators . . . 483
36.2. Searching & Sorting . . . 487
36.2.1. Searching . . . 487
36.2.2. Sorting . . . 488
36.3. Other Considerations . . . 488
36.3.1. Sorted Collections . . . 488
36.3.2. Handling `null` values . . . 490
36.3.3. Importance of `equals()` and `hashCode()` Methods . . . 491
36.3.4. Java 8: Lambda Expressions . . . 493

III. The PHP Programming Language **495**

37. Basics **497**
37.1. Getting Started: Hello World . . . 498
37.2. Basic Elements . . . 499
37.2.1. Basic Syntax Rules . . . 499
37.2.2. PHP Tags . . . 500
37.2.3. Libraries . . . 500
37.2.4. Comments . . . 500
37.2.5. Entry Point & Command Line Arguments . . . 502
37.3. Variables . . . 502
37.3.1. Using Variables . . . 503
37.4. Operators . . . 504
37.4.1. Type Juggling . . . 505
37.4.2. String Concatenation . . . 508
37.5. Basic I/O . . . 508

37.6. Examples . . . 509
37.6.1. Converting Units . . . 509
37.6.2. Computing Quadratic Roots . . . 512

38. Conditionals **515**
38.1. Logical Operators . . . 515
38.1.1. Order of Precedence . . . 517
38.2. If, If-Else, If-Else-If Statements . . . 517
38.3. Examples . . . 519
38.3.1. Computing a Logarithm . . . 519
38.3.2. Life & Taxes . . . 520
38.3.3. Quadratic Roots Revisited . . . 522

39. Loops **527**
39.1. While Loops . . . 527
39.2. For Loops . . . 528
39.3. Do-While Loops . . . 529
39.4. Foreach Loops . . . 529
39.5. Examples . . . 530
39.5.1. Normalizing a Number . . . 530
39.5.2. Summation . . . 530
39.5.3. Nested Loops . . . 531
39.5.4. Paying the Piper . . . 531

40. Functions **535**
40.1. Defining & Using Functions . . . 535
40.1.1. Declaring Functions . . . 535
40.1.2. Organizing Functions . . . 537
40.1.3. Calling Functions . . . 537
40.1.4. Passing By Reference . . . 538
40.1.5. Optional & Default Parameters . . . 539
40.1.6. Function Pointers . . . 540
40.2. Examples . . . 540
40.2.1. Generalized Rounding . . . 540
40.2.2. Quadratic Roots . . . 541

41. Error Handling & Exceptions **543**
41.1. Throwing Exceptions . . . 543
41.2. Catching Exceptions . . . 543
41.3. Creating Custom Exceptions . . . 544

42. Arrays **547**
42.1. Creating Arrays . . . 547

42.2. Indexing . . . 547
42.2.1. Strings as Indices . . . 549
42.2.2. Non-Contiguous Indices . . . 549
42.2.3. Key-Value Initialization . . . 549
42.3. Useful Functions . . . 550
42.4. Iteration . . . 551
42.5. Adding Elements . . . 552
42.6. Removing Elements . . . 553
42.7. Using Arrays in Functions . . . 554
42.8. Multidimensional Arrays . . . 555

43. Strings **557**
43.1. Basics . . . 557
43.2. String Functions . . . 558
43.3. Arrays of Strings . . . 560
43.4. Comparisons . . . 560
43.5. Tokenizing . . . 561

44. File I/O **563**
44.1. Opening Files . . . 563
44.2. Reading & Writing . . . 564
44.2.1. Using URLs . . . 565
44.2.2. Closing Files . . . 565

45. Objects **567**
45.1. Data Visibility . . . 568
45.2. Methods . . . 568
45.2.1. Accessor & Mutator Methods . . . 570
45.3. Constructors . . . 571
45.4. Usage . . . 572
45.5. Common Methods . . . 573
45.6. Composition . . . 573
45.7. Example . . . 574

46. Recursion **577**

47. Searching & Sorting **581**
47.1. Comparator Functions . . . 581
47.1.1. Searching . . . 584
47.1.2. Sorting . . . 584

Glossary **587**

Acronyms **599**

Index **610**

References **613**

List of Algorithms

1.1. An example of pseudocode: finding a minimum value 13

2.1. Assignment Operator Demonstration 34
2.2. Addition and Subtraction Demonstration 35
2.3. Multiplication and Division Demonstration 36
2.4. Temperature Conversion Program 50
2.5. Quadratic Roots Program 51

3.1. An if-statement 76
3.2. An if-else Statement 79
3.3. Example If-Else-If Statement 79
3.4. General If-Else-If Statement 81
3.5. If-Else-If Statement With a Bug 81
3.6. A simple receipt program 83
3.7. Preventing Division By Zero Using an If Statement 83
3.8. Comparing Students by Name 84
3.9. Computing Tax Liability with If-Else-If 86
3.10. Computing Tax Credit with If-Else-If 86

4.1. Counter-Controlled While Loop 97
4.2. Normalizing a Number With a While Loop 99
4.3. A General For Loop 99

4.4. Counter-Controlled For Loop 100
4.5. Summation of Numbers in a For Loop 100
4.6. Counter-Controlled Do-While Loop 101
4.7. Flag-Controlled Do-While Loop 102
4.8. Example Foreach Loop 102
4.9. Foreach Loop Computing Grades 103
4.10. Nested For Loops 103
4.11. Infinite Loop 104
4.12. Computing the Geometric Series Using a For Loop 107
4.13. Computing the Geometric Series Using a While Loop 108
4.14. Determining if a Number is Prime or Composite 109
4.15. Counting the number of primes. 109
4.16. Computing a loan amortization schedule 111
4.17. Scaling a Value 123

5.1. A function in pseudocode 135
5.2. Using a function 137

11.1. Recursive CountDown(n) Function 203
11.2. Recursive Fibonacci(n) Function 204
11.3. Recursive Fibonacci(n) Function With Memoization 208

12.1. Linear Search 212
12.2. Recursive Binary Search Algorithm, BinarySearch(A, l, r, e_k) 214
12.3. Iterative Binary Search Algorithm, BinarySearch(A, e_k) 215
12.4. Selection Sort 222
12.5. Insertion Sort 225
12.6. QuickSort 229

12.7. In-Place PARTITION 229

12.8. MERGESORT 233

12.9. MERGE 234

List of Code Samples

1.1. A simple program in C . . . 9
1.2. A simple program in C, compiled to assembly . . . 10
1.3. A simple program in C, resulting machine code formatted in hexadecimal (partial) . . . 11

2.1. Example of variable scoping in C . . . 32
2.2. Compound Assignment Operators in C . . . 41
2.3. `printf()` examples in C . . . 45
2.4. Output Result . . . 45

4.1. Zune Bug . . . 105

15.1. Hello World Program in C . . . 254
15.2. Fahrenheit-to-Celsius Conversion Program in C . . . 268
15.3. Quadratic Roots Program in C . . . 269

16.1. Examples of Conditional Statements in C . . . 275
16.2. Logarithm Calculator Program in C . . . 280
16.3. Tax Program in C . . . 281
16.4. Quadratic Roots Program in C With Error Checking . . . 282

17.1. While Loop in C . . . 283
17.2. Flag-controlled While Loop in C . . . 284
17.3. For Loop in C . . . 285
17.4. Do-While Loop in C . . . 286
17.5. Normalizing a Number with a While Loop in C . . . 287
17.6. Summation of Numbers using a For Loop in C . . . 287
17.7. Nested For Loops in C . . . 288
17.8. Loan Amortization Program in C . . . 290

19.1. Using the `errno.h` library . . . 307

23.1. A `Student` structure declaration . . . 344

25.1. C Function Pointer Syntax Examples . . . 371
25.2. C Search Examples . . . 375
25.3. C Sort Examples . . . 376
25.4. C Comparator Function for Strings . . . 377

25.5. Sorting Structures via Pointers 378
25.6. Handling Null Values 379

26.1. Hello World Program in Java 384
26.2. Basic Input/Output in Java 396
26.3. Fahrenheit-to-Celsius Conversion Program in Java 399
26.4. Quadratic Roots Program in Java 402

27.1. Examples of Conditional Statements in Java 407
27.2. Logarithm Calculator Program in Java 412
27.3. Tax Program in Java 413
27.4. Quadratic Roots Program in Java With Error Checking 414

28.1. While Loop in Java 415
28.2. Flag-controlled While Loop in Java 416
28.3. For Loop in Java 417
28.4. Do-While Loop in Java 418
28.5. Enhanced For Loops in Java Example 1 418
28.6. Enhanced For Loops in Java Example 2 419
28.7. Normalizing a Number with a While Loop in Java 419
28.8. Summation of Numbers using a For Loop in Java 420
28.9. Nested For Loops in Java 420
28.10Loan Amortization Program in Java 422

34.1. The completed Java `Student` class. 477

36.1. Java Search Examples 489
36.2. Using Java Collection's Sort Method 490
36.3. Handling Null Values in Java Comparators 491

37.1. Hello World Program in PHP 498
37.2. Hello World Program in PHP with HTML 498
37.3. Type Juggling in PHP 506
37.4. Fahrenheit-to-Celsius Conversion Program in PHP 511
37.5. Quadratic Roots Program in PHP 513

38.1. Examples of Conditional Statements in PHP 518
38.2. Logarithm Calculator Program in C 523
38.3. Tax Program in PHP 524
38.4. Quadratic Roots Program in PHP With Error Checking 525

39.1. While Loop in PHP 527
39.2. Flag-controlled While Loop in PHP 528
39.3. For Loop in PHP 529
39.4. Do-While Loop in PHP 529
39.5. Normalizing a Number with a While Loop in PHP 530

39.6. Summation of Numbers using a For Loop in PHP 531
39.7. Nested For Loops in PHP 531
39.8. Loan Amortization Program in PHP 533

44.1. Processing a file line-by-line in PHP 564

45.1. The completed PHP `Student` class. 576

47.1. Using PHP's `usort()` Function 585

List of Figures

1.1. Depiction of Computer Memory 6
1.2. A Compiling Process 8

2.1. Types of Flowchart Nodes 18
2.2. Example of a flowchart for a simple ATM process 19
2.3. Elements of a `printf()` statement in C 44
2.4. Intersection of two circles. 63

3.1. Control flow diagrams for sequential control flow and an if-statement. . . 77
3.2. An if-else Flow Chart 78
3.3. Control Flow for an If-Else-If Statement 80
3.4. Quadrants of the Cartesian Plane 87
3.5. Three types of triangles 90
3.6. Intersection of Two Rectangles 91
3.7. Examples of Floor Tiling 93

4.1. A Typical Loop Flow Chart 96
4.2. A Do-While Loop Flow Chart. The continuation condition is checked *after* the loop body. 101
4.3. Plot of $f(x) = \frac{\sin x}{x}$ 116
4.4. A rectangle for the interval $[-5, 5]$. 117
4.5. Follow the bouncing ball 118
4.6. Sampling points in a circle 119
4.7. Regular polygons 120
4.8. A polygon and its centroid. Whoo! 130

5.1. A function declaration (prototype) in the C programming language with the return type, identifier, and parameter list labeled. 135
5.2. Program Stack 139
5.3. Demonstration of Pass By Value 141
5.4. Demonstration of Pass By Reference 143

7.1. Example of an Array 160
7.2. Example returning a static array 163
7.3. Pitfalls of Returning Static Arrays 174
7.4. Depiction of Application Memory. 175
7.5. Shallow vs. Deep Copies 176

9.1. Linux Tree Directory Structure . . . 186
9.2. An example polygon for $n = 5$ . . . 188
9.3. A Word Search . . . 189
9.4. A solved Sudoku puzzle . . . 191
9.5. A DNA Sequence . . . 194
9.6. Codon Table for RNA to Protein Translation . . . 195

11.1. Recursive Fibonacci Computation Tree . . . 207

12.1. Array of Integers . . . 212
12.2. A Sorted Array . . . 213
12.3. Binary Search Example . . . 216
12.4. Example of the benefit of ordered (indexed) elements in Windows 7 . . . 220
12.5. Selection Sort Example . . . 223
12.6. Insertion Sort Example . . . 226
12.7. Partitioning Example 1 . . . 230
12.8. Partitioning Example 2 . . . 230
12.9. Partitioning Example 3 . . . 231
12.10Merge Sort Example . . . 235
12.11Merge Example . . . 236
12.12Generalized Sorting with a Comparator . . . 240

18.1. Pointer Operations . . . 298

20.1. Dynamically Allocating Multidimensional Arrays . . . 321
20.2. Contiguous Two Dimensional Array . . . 324

21.1. Example of a character array (string) in C. . . . 325

23.1. Contiguous Structure Array . . . 350
23.2. Array of Structure Pointers . . . 350
23.3. Hybrid Array of Structures . . . 352

1. Introduction

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

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

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

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

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

1.1. Problem Solving

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

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

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

Ultimately, what is desired in a solution? What are the objectives that need to be achieved? What would an ideal solution look like or what would it do? Who would use the solution and how would they use it? By answering these questions, we get an idea of *where we want to be*. Once we know where we are and where we want to be, the problem solving process can begin: how do we get from point A to point B?

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

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

1. Design
2. Implementation

3. Testing
4. Refinement

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

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

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

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

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

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

A bottom-up strategy attempts to first completely define the smallest components or

entities that make up a system *first*. Once these have been defined and implemented, they are combined and interactions between them are defined to produce a more complex system.

1.2. Computing Basics

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

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

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

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

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

Computers "speak" in binary code. Binary is nothing more than a structured collection of 0s and 1s. A single 0 or 1 is referred to as a bit. Bits can be collected to form larger chunks of information: 8 bits form a byte, 1024 bytes is referred to as a kilobyte, etc. Table 1.1 contains a several more binary units. Each unit is in terms of a power of 2 instead of a power of 10. As humans, we are more familiar with decimal–base-10 numbers and so units are usually expressed as powers of 10, kilo- refers to 10^3, mega- is 10^6, etc. However, since binary is base-2 (0 or 1), units are associated with the closest power of 2. Computers are binary machines because it is the most practical to implement in electronic devices. 0s and 1s can be easily represented by low/high voltage; low/high frequency; on-off; etc. It is much easier to design and implement systems that switch between only two states.

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

Table 1.1.: Various units of digital information with respect to bytes. Memory is usually measured using powers of two.

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

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

Main memory, sometimes referred to as Random Access Memory (RAM) consists of a collection of *addresses* along with *contents*. An address usually refers to a single byte of memory (called *byte-addressing*). The content, that is the byte of data that is stored at an address, can be anything. It can represent a number, a letter, etc. To the computer it is all just a bunch of 0s and 1s. For convenience, memory addresses are represented using hexadecimal, which is a base-16 counting system using the symbols $0, 1, \ldots, 9, a, b, c, d, e, f$. Numbers are prefixed with a `0x` to indicate they represent hexadecimal numbers. Figure 1.1 depicts memory and its address/contents.

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

1.3. Basic Program Structure

Programs start out as *source code*, a collection of instructions usually written in a high-level programming language. A source file containing source code is nothing more than

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

Figure 1.1.: Depiction of Computer Memory. Each address refers to a byte, but different types of data (integers, floating-point numbers, characters) may require different amounts of memory. Memory addresses and some data is represented in *hexadecimal*.

a plain text file that can be edited by any text editor. However, many developers and programmers utilize modern Integrated Development Environment (IDE) that provide a text editor with *code highlighting*: various elements are displayed in different colors to make the code more readable and elements can be easily identified. Mistakes such as unclosed comments or curly brackets can be readily apparent with such editors. IDEs can also provide automated compile/build features and other tools that make the development process easier and faster.

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

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

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

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

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

In contrast, some languages are *interpreted*, not compiled. The source code is contained in a file usually referred to as a *script*. Rather than being run directly by an operating system, the operating system loads and execute another program called an *interpreter*. The interpreter then loads the script, parses, and execute its instructions. Interpreted

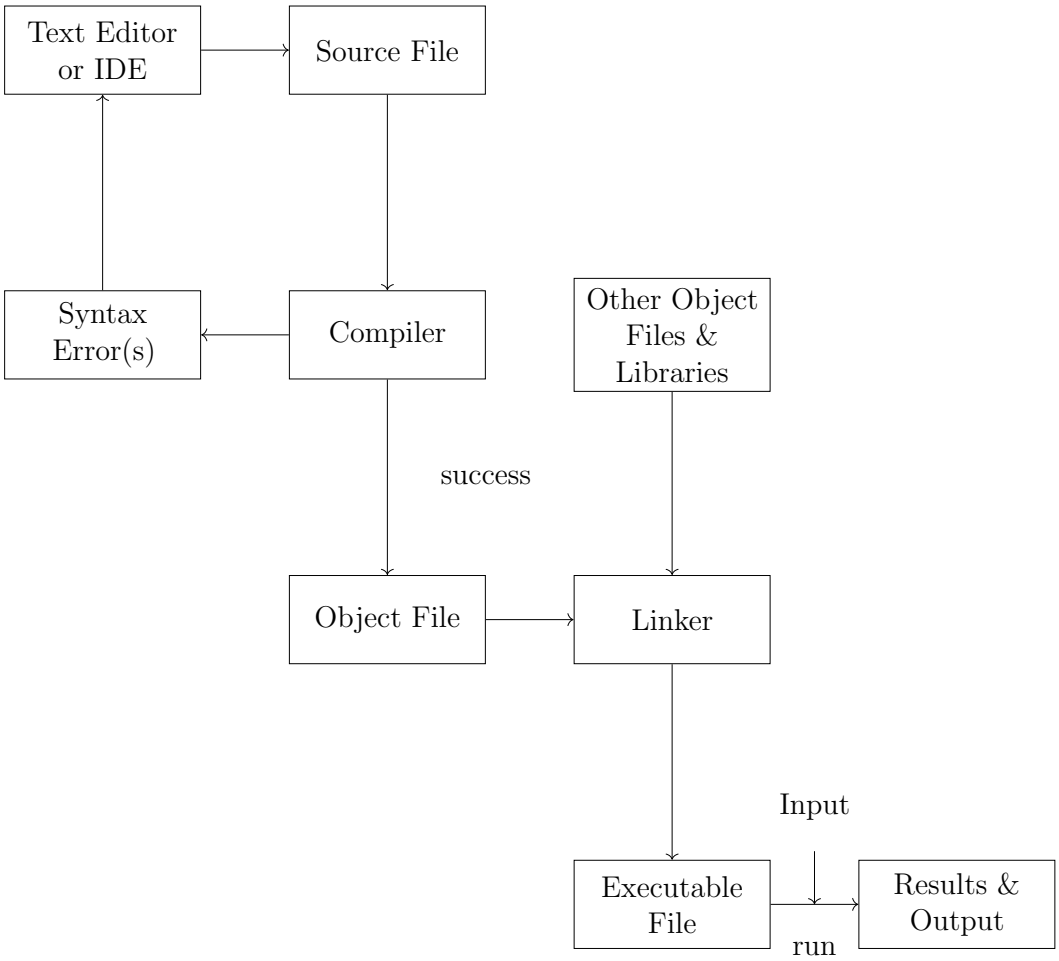


Figure 1.2.: A Compiling Process

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

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

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

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

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

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

  return 0;
}
```

Code Sample 1.1.: A simple program in C

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

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

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

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

.subsections_via_symbols
```

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

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

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

languages may still have a predefined main function, but in general, a script starts executing starting with the first instruction in the script file. Adhering to the syntax rules is still important, but since interpreted languages are not compiled, syntax errors become runtime errors. A program may run fine until its first syntax error at which point it fails.

There are other ways of compiling and running programs. Java for example represents a compromise between compiled and interpreted languages. Java source code is compiled into Java bytecode which is not actually machine code that the operating system and hardware can run directly. Instead, it is compiled code for a Java Virtual Machine (JVM). This allows a developer to write highly portable code, compile it once and it is runnable on any JVM on any system (write-once, compile-once, run-anywhere).

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

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

1.4. Syntax Rules & Pseudocode

Programming languages are a lot like human languages in that they have *syntax* rules. These rules dictate the appropriate arrangements of words, punctuation, and other symbols that form valid statements in the language. For example, in many programming languages, commands or statements are terminated by semicolons (just as most sentences are ended with a period). This is an example of "punctuation" in a programming language. In English paragraphs are separated by lines, in programming languages *blocks* of code are separated by curly brackets. Variables are comparable to nouns and operations and functions are comparable to verbs. Complex documents often have footnotes that provide additional explanations; code has *comments* that provide documentation and explanation for important elements. English is read top-to-bottom, left-to-right. Programming

languages are similar: individual executable commands are written one per line. When a program executes, each command executes one after the other, top-to-bottom. This is known as sequential control flow.

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

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

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

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

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

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

1.5. Documentation, Comments, and Coding Style

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

It would be difficult to read an essay that contained random line breaks, paragraphs were not indented, it contained different spacing or different fonts, etc. Likewise, code should be legible. Well written code is consistent and makes good use of whitespace and indentation. Code within the same code block should be indented at the same level. Nested blocks should be further indented just like the outline of an essay or table of contents.

Code should also be well documented. Each line or segment of code should be clear enough that it tells the user *what* the code does and *how* it does it. This is referred to as "self-documenting" code. A person familiar with the particular language should be able to read your code and immediately understand what it does. In addition, well-written code should contain sufficient and clear *comments*. A comment in a program is intended for a human user to read. A comment is ultimately ignored by the compiler/interpreter and has no effect on the actual program. Good comments tell the user *why* the code was written or why it was written the way it was. Comments provide a high-level description of what a block of code, function, or program does. If the particular method or algorithm is of interest, it should also be documented.

There are typically two ways to write comments. Single line comments usually begin with two forward slashes, `//`.[1] Everything after the slashes until the next line is ignored. Multiline comments begin with a `/*` and end with a `*/`; everything between them is ignored even if it spans multiple lines. This syntax is shared among many languages including C, Java, PHP and others. Some examples:

```
1  double x = sqrt(y); //this is a single line comment
2
3  /*
4    This is a multiline comment
5    each line is ignored, but allows
6    for better formatting
7  */
8
9  /**
10  * This is a doc-style comment, usually placed in
```

[1] You can remember the difference between a forward slash `/` and a backslash `\` by thinking of a person facing right and either leaning backwards (backslash) or forwards (forward slash).

```
11   * front of major portions of code such as a function
12   * to provide documentation
13   * It begins with a forward-slash-star-star
14   */
```

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

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

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

2. Basics

2.1. Control Flow

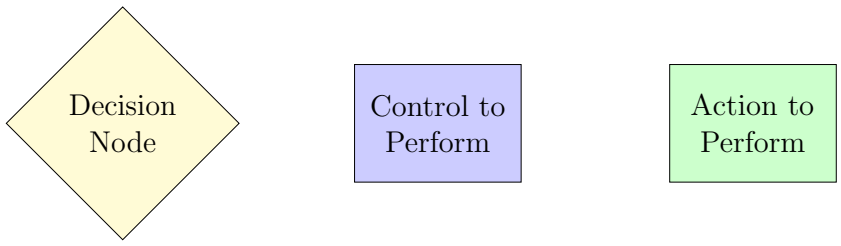
The flow of control (or simply control flow) is how a program processes its instructions. Typically, programs operate in a linear or *sequential* flow of control. Executable statements or instructions in a program are performed one after another. In source code, the order that instructions are written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

This type of control flow is usually associated with a procedural programming paradigm (which is closely related to imperative or structured programming paradigms). Though this text will mostly focus on languages that are procedural (or that have strong procedural aspects), it is important to understand that there are other programming language paradigms. Functional programming languages such as Scheme and Haskell achieve computation through the evaluation of mathematical functions with as little or no ("pure" functional) state at all. Declarative languages such as those used in database languages like SQL or in spreadsheets like Excel specify computation by expressing the logic of computation rather than explicitly specifying control flow. For a more formal introduction to programming language paradigms, a good resource is *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages* by Tate [36].

2.1.1. Flowcharts

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

Some boxes represent *decisions* to be made which may have one or more alternate routes (more than one directed edge going out of the box) depending on the the result of the



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

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

decision. Decision boxes are usually depicted with a diamond shaped box.

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

2.2. Variables

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

$$x = 3y + 5$$

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

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

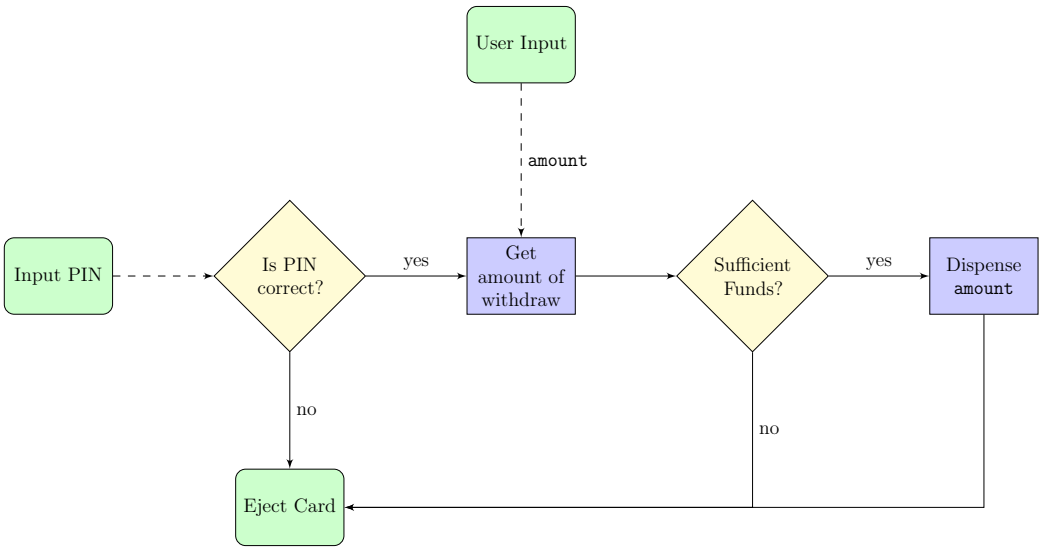


Figure 2.2.: Example of a flowchart for a simple ATM process

2.2.1. Naming Rules & Conventions

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

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

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

however, using a mixture of lowercase and uppercase letters to refer to the same variable is discouraged: it is difficult to read, inconsistent, and just plain ugly.

Beyond the naming rules that languages may enforce, most languages have established naming conventions; a set of guidelines and best-practices for choosing identifier names for variables (as well as functions, methods, and class names). Conventions may be widely adopted on a per-language basis or may be established within a certain library, framework or by an organization that may have official *style guides*. Naming conventions are intended to give source code consistency which ultimately improves readability and makes it easier to understand. Following a consistent convention can also greatly reduce the chance for errors and mistakes. Good naming conventions also has an aesthetic appeal; code should be beautiful.

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

`average_score`, `number_of_students`, `miles_per_hour`

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

`averageScore`, `numberOfStudents`, `milesPerHour`

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

`AverageScore`, `NumberOfStudents`, `MilesPerHour`

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

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

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

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

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

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

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

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

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

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

the variable `apr` may be preferred over the longer `annualPercentageRate`.

- Avoid pluralizations, use singular forms – English is not a very consistent language when it comes to rules like pluralizations. For most cases you simply add "s"; for others you add "es" or change the "y" to "i" and add "es". Some words are the same form for singular and plural such as "glasses."[2] Other words have completely different forms ("focus" becomes "foci"). Still yet there are instances in which *multiple* words are acceptable: the plural of "person" can be "persons" *or* "people". Avoiding plural forms keeps things simple and consistent: you don't need to be a grammarian in order easily read code. One potential exception to this is when using a collection such as an array to hold more than one element or the variable represents a quantity that is pluralized (as with `numberOfStudents` above).

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

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

2.2.2. Types

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

The type of a variable is important because it affects how the raw binary data stored at a memory location is interpreted. Moreover, some types take a different amount of memory to store. For example, an integer type could take 32 bits while a floating point type could take 64 bits. Programming languages may support different types and may do so in different ways. In the next few sections we'll describe some common types that are supported by many languages.

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

Numeric Types

At their most basic, computers are number crunching machines. Thus, the most basic type of variable that can be used in a computer program is a *numeric type*. There are several numeric types that are supported by various programming languages. The most simple is an *integer* type which can represent whole numbers 0, 1, 2, etc. and their negations, $-1, -2, \ldots$. *Floating point* numeric types represent decimal numbers such as $0.5, 3.14, 4.0$, etc. However, neither integer nor floating point numbers can represent *every* possible number since they use a finite number of bits to represent the number. We will examine this in detail below. For now, let's understand how a computer represents both integers and floating point numbers in memory.

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

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

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

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

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

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

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

as follows.

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

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

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

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

Note that the upper bound follows from the fact that

$$\underbrace{\texttt{0b}\,\texttt{11}\ldots\texttt{011}}_{n \text{ bits}} = \sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1$$

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

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

Table 2.1.: Counting in Binary

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

Table 2.2.: Ranges for various signed integer types

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

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

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

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

For example,

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

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

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

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

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

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

$$0.110011 \times 2^3$$

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

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

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

Most modern programming languages implement floating point numbers according to the Institute of Electrical and Electronics Engineers (IEEE) 754 Standard [20] (also called the International Electrotechnical Commission (IEC) 60559 [19]). When represented in binary, a fixed number of bits must be used to represent the sign, mantissa and exponent. The standard defines several precisions that each use a fixed number of bits with a resulting number of significant digits (base-10) of precision. Table 2.3 contains a summary of a few of the most commonly implemented precisions.

Just as with integers, the finite precision of floating point numbers results in several limitations. First, irrational numbers such as $\pi = 3.14159\ldots$ can only be approximated out to a certain number of digits. For example, with single precision $\pi \approx 3.1415927$ which is accurate only to the 6th decimal place and with double precision, $\pi \approx 3.1415926535897931$ approximate to only 15 decimal places.[3] In fact, *regardless* of how many bits we allow in our representation, an irrational number like π (that never repeats and never terminates) will only ever be an approximation. Real numbers like π require an infinite precision, but computers are only finite machines.

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

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

which when represented in scientific notation in decimal is

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

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

Programming languages usually only support the common single and double precisions defined by the IEEE 754 standard as those are commonly supported by hardware.

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

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

Characters & Strings

Another type of data is textual data which can either be single characters or a sequence of characters which are called strings. Strings are sometimes used for human readable data such as messages or output, but may also model general data. For example, DNA is usually encoded using strings consisting of the characters C, G, A, T (corresponding to the nucleases cytosine, guanine, adenine, and thymine). Numerical characters and punctuation can also be used in strings in which case they do not represent numbers, but instead may represent textual versions of numerical data.

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

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

There are several other nice design features built into the ASCII table. For example, to convert between uppercase and lowercase versions, you only need to "flip" the second bit (0 for uppercase, 1 for lowercase). There are also several special characters that

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

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

need to be *escaped* to be defined. For example, though your keyboard has a tab and an enter key, if you wanted to code those characters, you would need to specify them in some way other than using those keys (since typing those keys will affect what you are typing rather than specifying a character). The standard way to escape characters is to use a backslash along with another, single character. The three most common are the (horizontal) tab, `\t`, the endline character, `\n`, and the null terminating character, `\0`. The tab and endline character are used to specify their whitespace characters respectively. The null character is used in some languages to denote the *end* of a string and is not printable.

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

Even 256 possible characters are not enough to represent the wide array of international characters when you consider languages like Chinese Japanese Korean (CJK). Unicode was developed to solve this problem by establishing a standard encoding that supports 1,112,064 possible characters, though only a fraction of these are actually currently assigned.[4] Unicode is backward compatible, so it works with plain ASCII characters. In fact, the most common encoding for Unicode, UTF-8 uses a *variable* number of bytes to encode characters. 1-byte encodings correspond to plain ASCII, there are also 2, 3, and 4-byte encodings.

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

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

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

string types more fully in Chapter 8.

Boolean Types

A Boolean is another type of variable that is used to hold a truth value, either *true* or *false*, of a logical statement. Some programming languages explicitly support a built-in Boolean type while others implicitly support them. For languages that have explicit Boolean types, typically the keywords `true` and `false` are used, but logical expressions such as $x \leq 10$ can also be evaluated and assigned to Boolean variables.

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

Object & Reference Types

Not everything is a number or string. Often, we wish to model real-world entities such as people, locations, accounts, or even interactions such as exchanges or transactions. Most programming languages allow you to create *user-defined types* by using objects or structures. Objects and structures allow you to group multiple pieces of data together into one logical entity; this is known as encapsulation. For example, a Student object may consist of a first-name, last-name, GPA, year, major, etc. Grouping these separate pieces of data together allows us to define a more complex type. We explore these concepts in more depth in Chapter 10.

In contrast to the built-in numeric, character/string, and Boolean types (also called primitive data types) user-defined types do not necessarily take a fixed amount of memory to represent. Since they are user-defined, it is up to the programmer to specify how they get created and how they are represented in memory. A variable that refers to an object or structure is usually a reference or pointer: a reference to where the object is stored in memory on a computer. Many programming languages use the keyword `null` (or sometimes `NULL` or a some variation) to indicate an invalid reference. The `null` keyword is often used to refer to uninitialized or "missing" data.

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

2.2.3. Declaring Variables: Dynamic vs. Static Typing

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

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

```
int x;
```

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

In other languages, typically *interpreted* languages, you do *not* have to declare a variable before using it. Such languages are generally referred to as dynamically typed languages. Instead of declaring a variable to have a particular type, the type of a variable is determined by the type of value that is assigned to it. If you assign an integer to a variable it *becomes* an integer. If you assign a string to it, it *becomes* a string type. Moreover, a variable's type can *change* during the execution of a program. If you reassign a value to a variable, it dynamically changes its type to match the type of the value assigned.

In PHP for example, a line like

```
$x = 10;
```

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

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

```c
{
  int a;
  {
    //this is a new code block inside the outer block
    int b;
    //at this point in the code, both a and b are in-scope
  }
  //at this point, only a is in-scope, b is out-of-scope
}
```

Code Sample 2.1.: Example of variable scoping in C

2.2.4. Scoping

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

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

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

On the same token, however, global variables make the state and execution of a program less predictable: if any piece of code can access a global variable, then potentially any piece of code could *change* that variable. Imagine some questionable code changing the value of our global π variable to 3. For this reason, using global variables is generally considered bad practice.[5] Even if no code performs such an egregious operation, the fact that anything *can* change the value means that when testing, you must test for

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

the potential that anything *will* change the value, greatly increasing the complexity of software testing. To capture the advantages of a global variable while avoiding the disadvantages, it is common to only allow global *constants*; variables whose values cannot be changed once set.

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

2.3. Operators

Now that we have variables, we need a way to work with variables. That is, given two variables we may wish to add them together. Or we may wish to take two strings and combine them to form a new string. In programming languages this is accomplished through operators which operate on one or more operands. An operator takes the values of its operands and combines them in some way to produce a new value. If an operator is applied to variable(s), then the values used in the operation are the values stored in the variable at the time that the operator is evaluated.

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

2.3.1. Assignment Operators

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

$$a \leftarrow 10$$

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

```
a = 10;
```

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

```
10 = a;
```

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

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

$$a \leftarrow 10$$

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

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

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

$$b \leftarrow a$$

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

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

Algorithm 2.1: Assignment Operator Demonstration

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

2.3.2. Numerical Operators

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

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

$$a \leftarrow -10$$

or

$$a \leftarrow -b$$

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

Addition & Subtraction

You can also add (sum) two numbers using the + (plus) operator and subtract using the − (minus) operator in a straightforward way. Note that most languages can distinguish the minus operator and the negation operator by how you use it just like a mathematical expression. If applied to one operand, it is interpreted as a negation operator. If applied to two operands, it represents subtraction. Some examples can be found in Algorithm 2.2.

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

Algorithm 2.2: Addition and Subtraction Demonstration

Multiplication & Division

You can also multiply and divide literals and variables. In mathematical expressions multiplication is represented as $a \times b$ or $a \cdot b$ or simply just ab and division is represented

as $a \div b$ or a/b or $\frac{a}{b}$. In our pseudocode, we'll generally use $a \cdot b$ and $\frac{a}{b}$, but in programming languages it is difficult to type these symbols. Usually programming languages use `*` for multiplication and `/` for division. Similar examples are provided in Algorithm 2.3.

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

Algorithm 2.3: Multiplication and Division Demonstration

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

Since both operands are integers, the result must be an integer. Normally, $3/2 = 1.5$, but since the result must be an integer, the fractional part gets truncated (cut-off) and only the integral part is kept for the final result. This can lead to weird results such as $1/3 = 0$ and $99/100 = 0$. The result is *not* rounded down or up; instead the fractional part is completely thrown out. Care must be taken when dividing integer variables in a statically typed language. Type casting can be used to force variables to change their type for the purposes of certain operations so that the full answer is preserved. For example, in C we can write

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

Integer Division

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

$$a = qb + r$$

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

$$a \bmod b$$

For example,

$$13 \bmod 5 = 3$$

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

$$10 \bmod 5 = 0$$

Many programming languages support this operation using the percent sign. For example,

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

2.3.3. String Concatenation

Strings can also be combined to form new strings. In fact, strings can often be combined with non-string variables to form new strings. You would typically do this in order to convert a numerical value to a string representation so that it can be output to the user or to a file for longterm storage. The operation of combining strings is referred to as string concatenation. Some languages support this through the same plus operator that is used with addition. For example,

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

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

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

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

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

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

2.3.4. Order of Precedence

In mathematics, when you write an expression such as:

$$a + b \cdot c$$

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

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

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

Numerical operators are similar when used in most programming languages. The same order of precedence is used and parentheses can be used to change the order of evaluation.

2.3.5. Common Numerical Errors

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

- Division by zero: $\frac{a}{b}$ where $b = 0$. This is an undefined operation in mathematics and also in programming languages. Depending on the language, any number of things may happen. It may be a fatal error or *exception*; the program may continue executing but give "garbage" results from then on; the result may be a special value such as null, "NaN" (not-a-number) or "INF" (a special representation of infinity). It is best to avoid such an operation entirely using conditionals statements and defensive programming (see Chapter 3).
- Other potentially invalid operations involve common mathematical functions. For example, $\sqrt{-1}$ would be a complex result, i which some languages do support.

However, many do not. Similarly, the natural logarithm of zero, $\ln(0)$ and negative values, $\ln(-1)$ is undefined. In either case you could expect a result like "NaN" or "INF."

- Still other operations seem like they should be valid, but because of how numbers are represented in binary, the results are invalid. Recall that for a 32-bit signed, two's complement number, the maximum representable value is 2,147,483,647. Suppose this maximum value is stored in a variable, b. Now suppose we attempt to add one more,

  $$c \leftarrow b + 1$$

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

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

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

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

  in binary. However, the 32nd bit is the sign bit, so this is a negative number. In particular, if this is a two's complement integer, it has the decimal value $-2,147,483,648$ which is obviously wrong. Another example would be if we have a "large number, say 2 billion and attempt to double it (multiply by 2). We would expect 4 billion as a result, but again overflow occurs and the result (using 32-bit signed two's complement integers) is $-294,967,296$.

- A similar phenomenon can happen with floating point numbers. If an operation (say multiplying two "small" numbers together) results in a number that is smaller than the smallest floating point number that can be represented, the result is said to have resulted in underflow. The result can essentially be zero, or an error can be raised to indicate that underflow has occurred. The consequences of underflow can be very complex.

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

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

2.3.6. Other Operators

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

Increment Operators

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

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

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

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

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

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

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

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

the variable `i` again now has the value 11, but the value of `x` is 16 since `++i` takes on the new, incremented value of 11. Appropriately using each can lead to some very concise code, but it is important to remember the difference.

Compound Assignment Operators

If we want to increment or decrement a variable by an amount other than 1 we can do so using *compound assignment* operators that combine an arithmetic operator and an assignment operator into one. For example, `a += 10` would add 10 to the variable `a`.

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

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

Code Sample 2.2.: Compound Assignment Operators in C

The same could be achieved by coding `a = a + 10`, but the former is a bit shorter as we don't have to repeat the variable.

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

2.4. Basic Input/Output

Not all variables can be coded using literals. Sometimes a program needs to read in values as input from a user who can give different values on different runs of a program. Likewise, a computer program often needs to produce output to the user to be of any use.

The most basic types of programs are interactive programs that interact with a human user. Generally, the program may interactive the user to enter some input value(s) or make some choices. It may then compute some values and respond to the user with some output. In the following sections we'll overview the various types of input and output (I/O for short) that are available.

2.4.1. Standard Input & Output

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

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

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

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

2.4.2. Graphical User Interfaces

A much more user-oriented way of reading input and displaying output is to use a Graphical User Interface (GUI). GUIs can be implemented as traditional "thick-client" applications (programs that are installed locally on your machine) or as "thin-client" applications such as a web application. They typically support general "widgets" such as input boxes, buttons, sliders, etc. that allow a user to interact with the program in a more visual way. They also allow the programmer to do better input validation. Widgets could be design so that *only* good input is allowed by creating *modal* restrictions: the user is only allowed to select one of several "radio" buttons for example. GUIs also support visual feedback cues to the user: popups, color coding, and other elements can be used to give feedback on errors and indicate invalid selections.

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

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

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

2.4.3. Output Using `printf()`-style Formatting

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

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

The function works by providing it a number of *arguments*. The first argument is always a string that specifies the formatting of the result using several *placeholders* (flags that begin with a percent sign) which will be replaced with values stored in variables but in a formatted manner. Subsequent arguments to the function are the list of variables to be printed; each argument is delimited by a comma. Figure 2.3 gives an example of of a `printf()` statement with two placeholders. The placeholders are ultimately replaced with the values stored in the provided variables a, b. If a, b held the values 10 and 2.718281, the code would end up printing

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

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

- `%d` formats an integer variable or literal

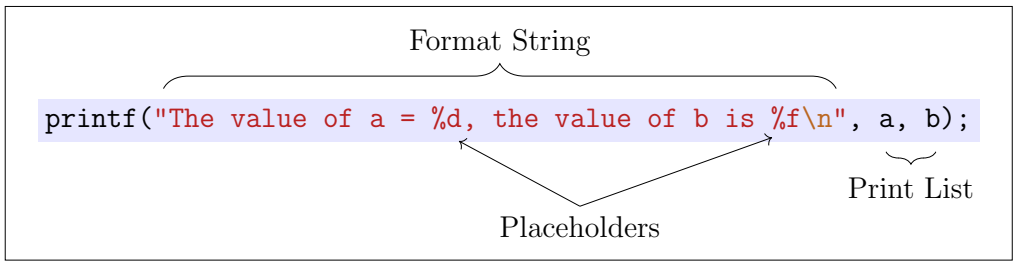


Figure 2.3.: Elements of a `printf()` statement in C

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

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

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

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

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

2.4.4. Command Line Input

Not all programs are interactive. In fact, the vast majority of software is developed to interact with other software and does not expect that a user is sitting at the console

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

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

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

Code Sample 2.3.: `printf()` examples in C

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

Code Sample 2.4.: Result of computation in Code Sample 2.3. Spaces are highlighted with a ␣ for clarity.

constantly providing it with input. Most languages and operating systems support non-interactive input from the Command Line Interface (CLI). This is input that is provided at the command line when the program is executed. Input provided from the command line are usually referred to as *command line arguments*. For example, if we invoke a program named `myProgram` from the command line prompt using something like the following:

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

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

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

2.5. Debugging

Making mistakes in programming is inevitable. Even the most expert of software developers make mistakes.[6] Errors in computer programs are usually referred to as bugs. The term was popularized by Grace Hopper in 1947 while working on a Mark II Computer at a US Navy research lab. Literally, a moth stuck in the computer was impeding its operation. Removing the moth or "debugging" the computer fixed it. In this section we will identify general types of errors and outline ways to address them.

2.5.1. Types of Errors

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

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

Syntax Errors

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

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

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

Runtime Errors

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

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

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

Logic Errors

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

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

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

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

2.5.2. Strategies

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

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

In general, debugging strategies attempt to isolate a problem to the smallest possible code segment. Thus, it is best practice to design your code using good procedural abstraction and place your code into functions and methods (see Chapter 5). It is also good practice to create test cases and test suites as you develop these small pieces of code.

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

In the end, good debugging skills, just like good coding skills, come from experience. A seasoned expert may be able to look at an error message and immediately diagnose the problem. Or, a bug can escape the detection of hundreds of the best developers and software tools and end up costing millions of dollars and thousands of man-hours.

2.6. Examples

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

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

2.6.1. Temperature Conversion

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

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

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

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

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

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

Algorithm 2.4: Temperature Conversion Program

2.6.2. Quadratic Roots

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

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

using the quadratic formula,

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

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

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

Algorithm 2.5: Quadratic Roots Program

2.7. Exercises

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

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

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

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

where

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

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

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

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

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

$$APY = e^i - 1$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

using the well-known quadratic formula:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$4\pi r^2$$

and the volume of a sphere with radius r is

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

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

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

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

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

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

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

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

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

```
Day Converter
=============
```

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

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

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

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

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

Your output should looks something like the following.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Where

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

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

simple formula:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

kilograms of Strontium-90 left.

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

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

Exercise 2.24. In sports, the *magic number* is a number that indicates the combination of the number of games that a leader in a division must win and/or the 2nd place team must lose for the leader to clinch the division. The magic number can be computed using the following formula:

$$G + 1 - W_A - L_B$$

where G is the number of games played in the season, W_A is the number of wins the leader currently has and L_B is the number of losses the 2nd place team currently has.

Write a program that prompts the user to enter:

- G, the total number of games in the season
- W_A the number of wins of the leading team
- L_A the number of losses of the leading team
- W_B the number of wins of the second place team
- L_B the number of losses of the second place

Your program will then output the current win percentages of both teams, the magic number of the leading team as well as the percentage of the remaining games that must go in team A's favor to satisfy the magic number (for this, we will assume A and B do not play each other).

For example, if a user enters the values 162, 96, 58, 93, 62, the output should look something like the following.

```
Team Wins Loss Perc   Magic Number
A    96   58   62.34% 5
B    93   62   60.00%
With 15 total games remaining, 33.33% must go in Team A's favor
```

Exercise 2.25. The *Doppler Effect* is a change in the observed spectral frequency of light when objects in space are moving toward or away from us. If the spectral frequency of the object is known, then the relative velocity can be estimated based on either the blue-shift (for objects traveling toward the observer) or the red-shift (for objects traveling away from the observer) of the observed frequency.

The blue-shift equation to determine velocity is given by

$$v_b = c\left(\frac{\lambda}{\lambda_b} - 1\right)$$

The red-shift equation to determine velocity is given by

$$v_a = c\left(1 - \frac{\lambda}{\lambda_r}\right)$$

where

- c is the speed of light (299,792.458 km/s)
- λ is the actual spectral line of the object (ex: hydrogen is 434nm)
- λ_r is the observed (red-shifted) spectral line and λ_b is the observed (blue-shifted) spectral line

Write a program that prompts the user to enter which spectral shift they want to compute (1 for blue-shift, 2 for red-shift). The program should then prompt for the spectral line of an object and the observed (shifted) frequency and output the velocity of the distant object. For example, if a user enters the values 1 (blue-shifted), 434 (nm) and 487 (nm), the output should look something like the following.

```
Spectral Line: 434nm
Observed Line: 487nm
Relative Velocity: 32626.28 km/s
```

Exercise 2.26. Radiometric dating is a technique by which the age of rocks, minerals, and other material can be estimated by measuring the proportion of radioactive isotopes it still has to its decay products. It can be computed with the following formula:

$$D = D_0 + N(e^{\lambda t} - 1)$$

where

- t is age of the sample,
- D is number of atoms of the daughter isotope in the sample,
- D_0 is number of atoms of the daughter isotope in the original composition,
- N is number of atoms of the parent isotope in the sample
- λ is the decay constant of the parent isotope,

  $$\lambda = \frac{\ln 2}{t_{1/2}}$$

  where $t_{1/2}$ is the half-life of the parent isotope (in years).

Write a program that prompts the user to enter D, D_0, N, and $t_{1/2}$ and computes the approximate age of the material, t.

For example, if the user were to enter 150, 50, 300, 28.8 (Strontium-90's half-life) then the program should output something like the following.

```
The sample appears to be 11.953080 years old.
```

Exercise 2.27. Suppose you have two circles each centered at $(0, 0)$ and $(d, 0)$ with radii of R, r respectively. These circles may intersect at two points, forming an asymmetric "lens" as in Figure 2.4.

The area of this lens can be computed using the following formula:

$$A = r^2 \cos^{-1}\left(\frac{d^2 + r^2 - R^2}{2dr}\right) + R^2 \cos^{-1}\left(\frac{d^2 + R^2 - r^2}{2dR}\right) - \frac{1}{2}\sqrt{(-d + r + R)(d + r - R)(d - r + R)(d + r + R)}$$

Write a program that prompts the user for the two radii and the x-offset d and computes the area of this lens. Your program should handle the special cases where the two circles do not intersect and when they intersect at a single point (the area is zero).

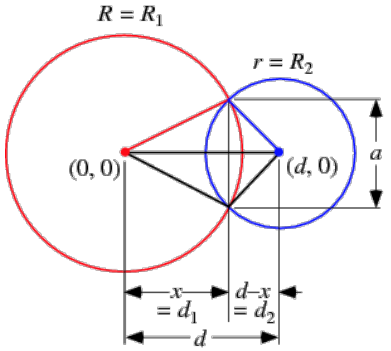


Figure 2.4.: Intersection of two circles.

3. Conditionals

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

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

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

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

3.1. Logical Operators

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

3.1.1. Comparison Operators

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

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

$$a \leq b$$

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

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

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

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

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

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

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

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

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

or they can be between a variable and a constant

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

or

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

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

Table 3.1.: Comparison Operators

Comparisons can also be used with more complex expressions such as

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

which could commonly be expressed in code as

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

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

Pitfalls

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

$$0 \leq x \leq 10$$

and in code, we may try to do something like

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

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

Some languages would treat this as a syntax error and not allow such an expression to be compiled since you cannot compare a Boolean value to a numerical value. However, other languages *may* allow this, typically representing *true* with some nonzero value such as 1 and *false* with 0. In either case, the expression would evaluate to *true* since both

$0 \leq 10$ and $1 \leq 10$. However, this is clearly wrong: if x had a value of 20 for example, the first expression would evaluate to *false*, making the entire expression *true*, but $20 \not\leq 10$. The solution is to use *logical operators* to express the same logic using *two* comparison operators (see Section 3.1.3).

Another common pitfall when programming is to mistake the assignment operator (typically only one equals sign, `=`) and the equality operator (typically two equal signs, `==`). As before, some languages will not allow it. The expression `a = 10` would *not* have a truth value associated with it. Attempts to use the expression in a logical statement would be a syntax error. Other languages may permit the expression and would give it a truth value equal to the value of the variable. For example, `a = 10` would take on the value 10 and be treated as *true* (nonzero value) while `a = 0` would take on the value 0 and be treated as *false* (zero). In either case, we probably do not get the result that we want. Take care that you use proper equality comparison operators.

Other Considerations

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

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

3.1.2. Negation

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

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

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

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

denote the negation operator[1], examples:

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

We will adopt this notation in our pseudocode, however most programming languages use the exclamation mark, `!` for the negation operator, similar to its usage in the inequality comparison operator, `!=`. The negation operator applies to the variable or statement immediately following it, thus

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

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

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

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

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

are equivalent, but the second expression is preferred.

3.1.3. Logical And

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

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

[1] This notation was first used by Heyting, 1930 [16]; prior to that the tilde symbol was used ($\sim p$ for example) by Peano [33] and Whitehead & Russell [37]. However, the tilde operator has been adopted to mean *bit-wise* negation in programming languages.

[2] In logic, the "wedge" symbol, $p \wedge q$ is used to denote the logical AND. It was first used again by Heyting, 1930 [16] but should not be confused for the keyboard caret, ^, symbol. Many programming languages do use the caret as an operator, but it is usually the *exclusive-or* operator which is *true* if and only if exactly one of its operands is *true*.

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

Table 3.3.: Logical AND Operator

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

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

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

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

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

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

3.1.4. Logical Or

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

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

As with the logical AND, the logical OR is used to combine logical statements to make

[3] In logic, the "vee" symbol, $p \vee q$ is used to denote the logical OR. It was first used by Russell, 1906 [35].

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

Table 3.4.: Logical Or Operator

more complex statements. For example,

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

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

We can also write statements with multiple Or operators,

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

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

3.1.5. Compound Statements

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

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

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

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

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

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

Tautologies and Contradictions

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

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

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

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

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

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

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

Similarly, the statement

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

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

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

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

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

De Morgan's Laws

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

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

Order	Operator
1	$\neg$
2	AND
3	OR

Table 3.5.: Logical Operator Order of Precedence

are equivalent to each other;

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

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

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

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

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

which is more concise and arguably more readable.

Order of Precedence

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

For example, the statement

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

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

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

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

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

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

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

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

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

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

and

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

3.1.6. Short Circuiting

Consider the following statement:

$$a \text{ And } b$$

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

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

Historically, avoiding even a few operations in old computers meant a difference on the order of milliseconds or even seconds. Thus, it made sense to avoid unnecessary operations. This is now known as short circuiting and to this day is still supported in most programming languages.[4] Though the differences are less stark in terms of CPU resources, most developers and programmers have come to *expect* this behavior and write statements under the assumption that short-circuiting will occur.

[4] Historically, the short-circuited version of the And operator was known as *McCarthy's sequential conjunction operation* which was formally defined by John McCarthy (1962) as "if p then q, else false", eliminating the evaluation of q if p is *false* [23].

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

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

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

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

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

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

but subtraction and division are not,

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

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

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

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

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

$$a \text{ Or } b$$

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

3.2. The If Statement

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

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

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

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

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

Algorithm 3.1: An if-statement

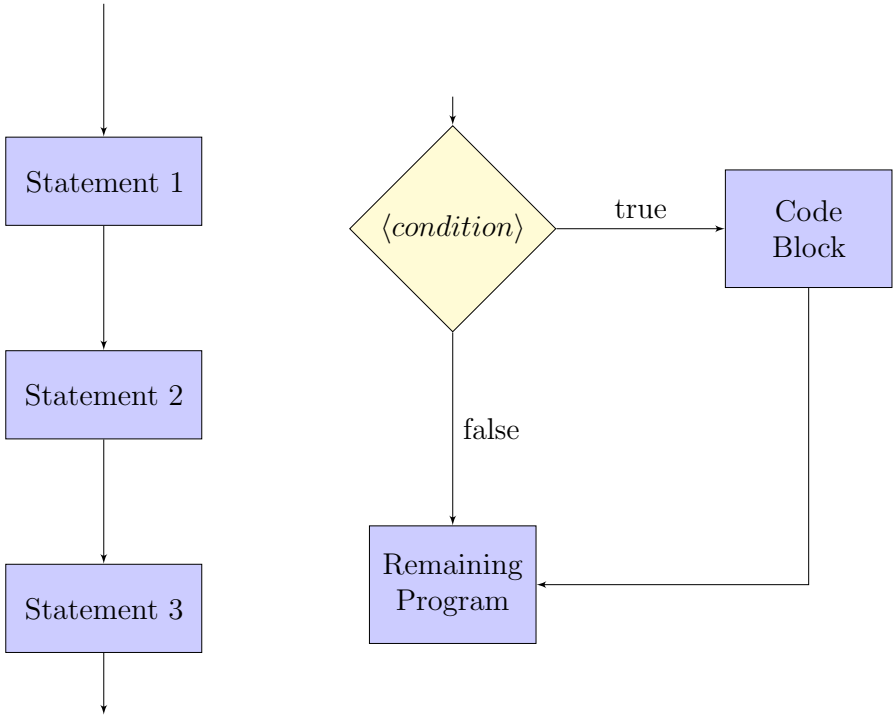
As in the flow chart, if the ⟨*condition*⟩ evaluates to *true*, then the code block bound to the statement executes in its entirety. Otherwise, if the condition evaluates to *false*, the code block bound to the statement is skipped in its entirety.

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

3.3. The If-Else Statement

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

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



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

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

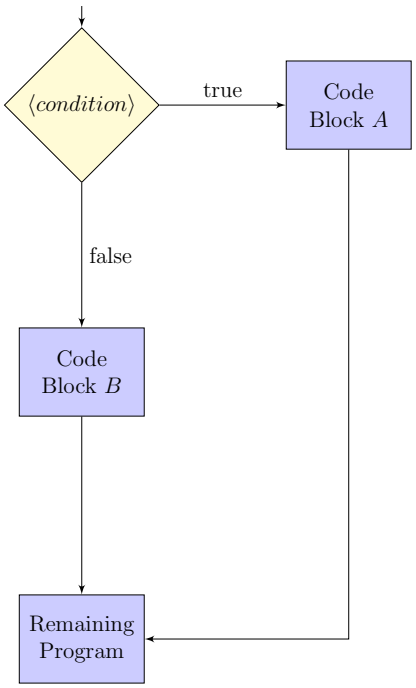


Figure 3.2.: An if-else Flow Chart

Just as with an if-statement, the keyword "if" is used. In fact, the if-statement is simply just an if-else statement with the else block omitted (equivalently, we could have defined an *empty* else block, but since it would have no effect, a simple if-statement with no else block is preferred). It is common to most programming languages to use the "else" keyword to denote the else block of code. Since there is only one ⟨*condition*⟩ to evaluate and it can only be *true* or *false*, it is not necessary to specify the conditions under which the else block executes. It is assumed that if the ⟨*condition*⟩ evaluates to *false*, the else block executes.

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

3.4. The If-Else-If Statement

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

```
1 IF (⟨condition⟩) THEN
2 |   Code Block A
3 ELSE
4 |   Code Block B
5 END
```

Algorithm 3.2: An if-else Statement

To illustrate, consider the case in which we have exactly three mutually exclusive possibilities. At a particular university, there are three possible semesters depending on the month. January through May is the Spring semester, June/July is the Summer semester, and August through December is the Fall semester. These possibilities are mutually exclusive because it cannot be *both* Spring and Summer at the same time for example. Suppose we have the current month stored in a variable named *month*. Algorithm 3.3 expresses the logic for determining which semester it is using an if-else-if statement.

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

Algorithm 3.3: Example If-Else-If Statement

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

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

In our example, we only identified three possibilities. You can generalize an if-else-if statement to specify as many conditions as you like. This generalization is depicted in

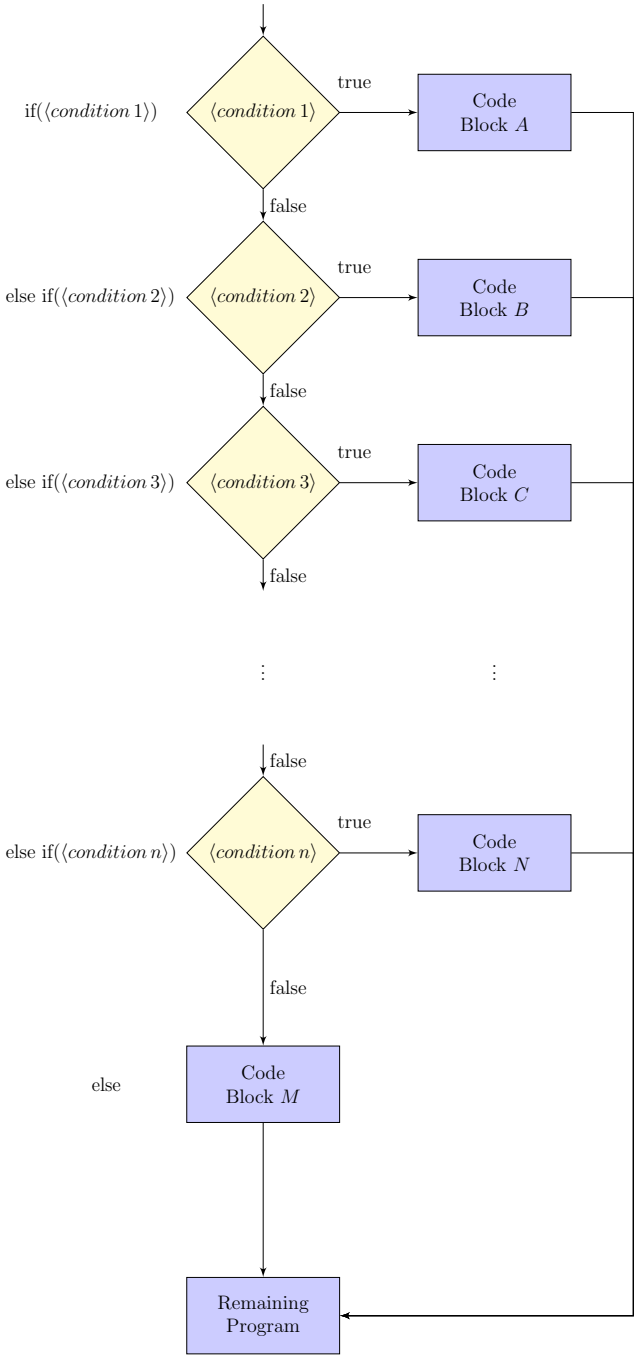


Figure 3.3.: Control Flow for an If-Else-If Statement. Each condition is evaluated in sequence. The first condition that evaluates to *true* results in the corresponding code block being executed. After executing, the program continues. Thus, each code block is *mutually exclusive*: at most *one* of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

Algorithm 3.4 and visualized in Figure 3.3. Similar to the if-statement, the else-statement and subsequent code block is optional. If omitted, then it may be possible that *none* of the code blocks is executed.

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

Algorithm 3.4: General If-Else-If Statement

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

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

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

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

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

Alternatively, we could have followed the example in Algorithm 3.3 and completely

specified both lower bounds and upper bounds in our condition. For example, the condition for "intrusive" could have been

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

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

3.5. Ternary If-Else Operator

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

```
E ? X : Y
```

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

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

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

If $a < b$ is true, then `min` will take on the value a. Otherwise it will take on the value b (in which case $a \geq b$ and so b is minimal). Most programming languages support this special syntax as it provides a nice convenience (yet another example of syntactic sugar).

3.6. Examples

3.6.1. Meal Discount

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

To do this, we first prompt the user to enter a subtotal. We can then ask the user if there is a discount to be computed. If the user answers yes, then we again prompt them for an amount (to allow different types of discounts). Otherwise, the discount will be

zero. We can then proceed to calculate each of the amounts above. To do this we'll need an if-statement. We could also use a conditional statement to check to see if the input makes sense: we wouldn't want a discount amount that is greater than 100%. The full algorithm is presented in Algorithm 3.6.

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

Algorithm 3.6: A simple receipt program

3.6.2. Look Before You Leap

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

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

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

Algorithm 3.7: Preventing Division By Zero Using an If Statement

This approach to programming is known as defensive programming. We are essentially checking the conditions for an invalid operation *before* performing that operation. In the example above, we simply chose not to perform the operation. Alternatively, we could use an if-else statement to perform alternate operations or *handle* the situation differently. Defensive programming is akin to "looking before leaping": before taking a potentially dangerous step, you look to see if you are at the edge of a cliff, and if so you don't take that dangerous step.

3.6.3. Comparing Elements

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

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

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

Algorithm 3.8: Comparing Students by Name

3.6.4. Life & Taxes

Another example in which there are several cases that have to be considered is computing an income tax liability using marginal tax brackets. Table 3.6 contains the 2014 US Federal tax margins and marginal rates for a married couple filing jointly based on the Adjusted Gross Income (income after deductions).

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

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

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

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

As an example: suppose that a couple has $35,000 AGI (placing them in the second tax bracket) and has two children. Their tax liability is

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

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

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

contain our initial tax liability.

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

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

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

```
1  IF income ≥ 110,000 THEN
2  |  credit ← 0
3  ELSE IF numberOfChildren ≤ 3 THEN
4  |  credit ← numberOfChildren ∗ 1,000
5  ELSE
6  |  credit ← 3000
7  END
   //Now adjust the tax, taking care that its a nonrefundable credit
8  IF credit > tax THEN
9  |  tax ← 0
10 ELSE
11 |  tax ← (tax − credit)
12 END
```

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

3.7. Exercises

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

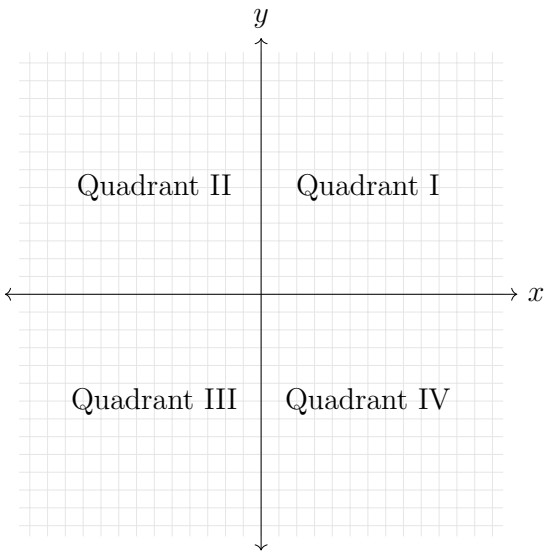


Figure 3.4.: Quadrants of the Cartesian Plane

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

Exercise 3.3. Price Per Mile. Write a program to determine which type of gas is the better deal with respect to price-per-mile driven. For example, suppose unleaded costs $2.50 per gallon and your vehicle is able to get an average of 30 miles per gallon. The true cost of unleaded is thus 8.33 cents per mile. Now suppose that the ethanol fuel costs only $2.25 per gallon but only yields 25 miles per gallon, thus 9 cents per mile, a worse deal.

Write a program that prompts the user to enter the price (per gallon) of one type of gas as well as the miles-per-gallon. Then prompt for the same two values for a second type of gas and compute the true cost of each type. Print a message indicating which type of gas is the better deal. For example, if the user enters the values above the output would look something like:

```
Gas A: $0.0833 per mile
Gas B: $0.0900 per mile
Gas A is the better deal.
```

Exercise 3.4. Various substances have different boiling points. A selection of substances and their boiling points can be found in Table 3.7. Write a program that prompts the user for the observed boiling point of a substance in degrees Celsius and identifies the substance if the observed boiling point is within 5% of the expected boiling point. If the data input is more than 5% higher or lower than any of the boiling points in the table, it should output `Unknown substance`.

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

Table 3.7.: Expected Boiling Points

Exercise 3.5. Electrical resistance in various metals can be measured using nano-ohm metres ($n\Omega \cdot m$). Table 3.8 gives the resistivity of several metals.

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

Table 3.8.: Resistivity of several metals

Write a program that prompts the user for an observed resistivity of an unknown material (as nano-ohm metres) and identifies the substance if the observed resistivity is within $\pm 3\%$ of the known resistivity of any of the materials in Table 3.8. If the input value lies outside the $\pm 3\%$ range, output `Unknown substance`.

Exercise 3.6. The visible light spectrum is measured in nanometer (nm) frequencies. Ranges roughly correspond to visible colors as depicted in Table 3.9.

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

Table 3.9.: Visible Light Spectrum Ranges

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

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

(i) Hardness must be greater than 50

(ii) Carbon content must be less than 0.7

(iii) Tensile strength must be greater than 5600

A grade of 5 thru 10 is is assigned to the steel according to the conditions in Table 3.10. Write a program that will read in the hardness, carbon content, and tensile strength as

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

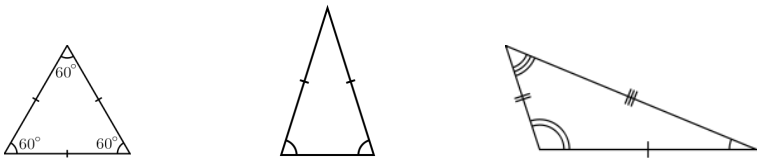
Table 3.10.: Grades of Steel

inputs and output the corresponding grade of the steel.

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

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

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



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

Figure 3.5.: Three types of triangles

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

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

where m is the person's mass (in lbs) and h is the person's height (in whole inches). Write a program that reads in a person's mass and height as input and outputs a characterization of the person's health with respect to the categories in Table 3.11.

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

Table 3.11.: BMI Categories

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

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

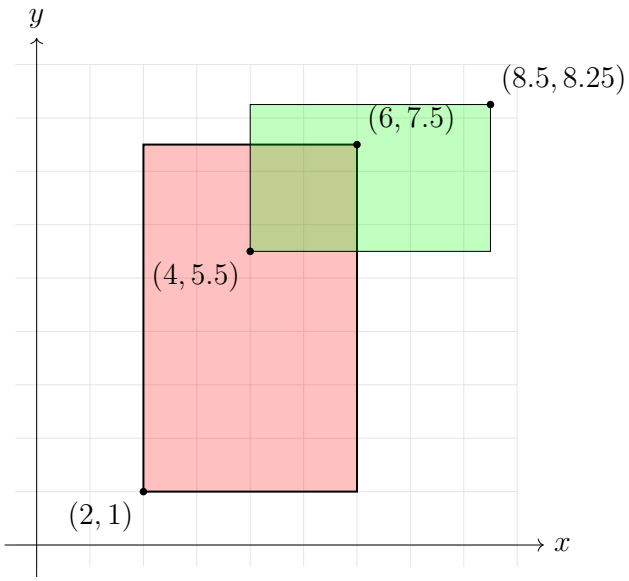


Figure 3.6.: Intersection of Two Rectangles

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

Your program should read in $x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4$ from the user and perform the computation above. As an example, the values $2, 1, 6, 7.5, 4, 5.5, 8.5, 8.25$ would correspond to the two rectangles in Figure 3.6.

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

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

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

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

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

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

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

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

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

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

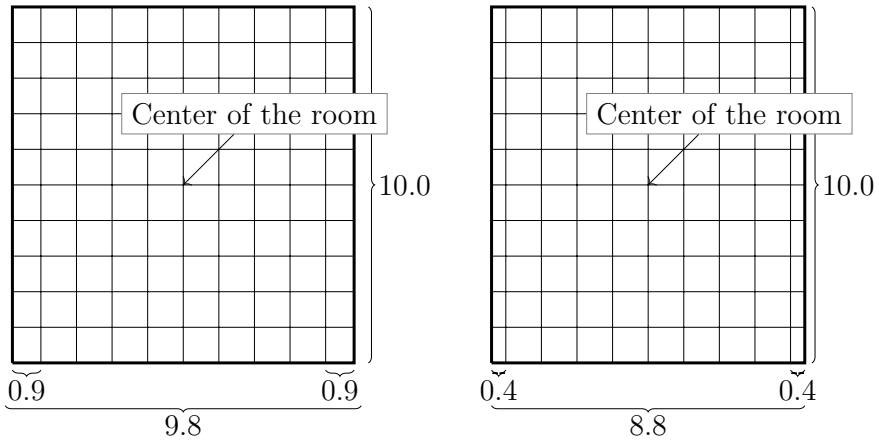
Of course, if the user is under their average daily use, a different message should be presented. You are allowed/encouraged to compute any other stats for the user that you feel would be useful.

Exercise 3.12. Write a program to help a floor tile company determine how many tiles they need to send to a work site to tile a floor in a room. For simplicity, assume that all rooms are perfectly rectangular with no obstructions; we will also omit any additional measurements related to grouting.

Further, we will assume that all tile is laid in a grid pattern centered at the center of the room. That is, four tiles will meet at their corners at the center of the room with tiles laid out to the edge of the room. Thus, it may be the case that the final row and/or column at the edge may need to be cut. Also note that if the cut is short enough, the remaining tile can be used on the other end of the room (same goes for the corners).

The program will take the following input:

- w - the width of the room
- l - the length of the room



(a) Example 1 (b) Example 2

Figure 3.7.: Examples of Floor Tiling

- t - width/length of the tile (all tiles are perfectly square)

If we can use whole tiles to perfectly fit the room, then we do so. For example, on the input $(10, 10, 1)$, we could perfectly tile a 10×10 room with 100 1×1 tiles. If the tiles don't perfectly fit, then we have to consider the possibility of waste and/or reuse. Consider the examples in Figure 3.7.

The first example is from the input $(9.8, 100, 1)$. In this case, we lay the tiles from the center of the room (8 full tile lengths) but are left with 0.9 on either side. If we cut a tile to fit the left side, we are left with only .1 tile which is too short for the right side. Therefore, we are forced to waste the 0.1 length and cut a full tile for the right side. In all, 100 tiles are required.

The second example is from the input $(8.8, 100, 1)$. In this case, we again lay tiles from the center of the room (8 full tile lengths) and are left with 0.4 lengths on either side. Here, we *can* reuse the cut tile: cut a tile on one side 0.4 with 0.6 remaining, and cut 0.4 on the other side of the tile (with the center 0.2 length of the tile being waste). Thus, both sides can be tiled with a single tile, meaning only 90 full tiles are needed to tile this room.

You may further assume that tiles used on the length-side end of the room *cannot* be used to tile the width-side of the room (and vice versa). Your program will compute and output the number of tiles required.

4. Loops

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

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

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

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

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

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

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

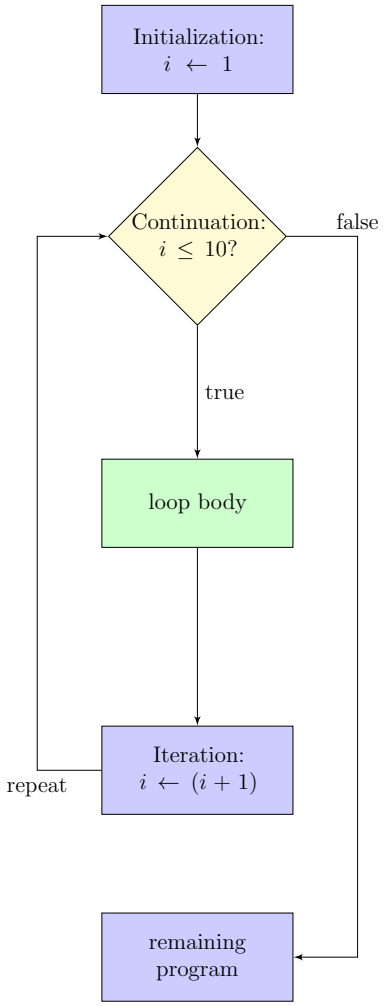


Figure 4.1.: A Typical Loop Flow Chart

continues.

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

As a simple example, consider the following outline.

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

The code outline above specifies that some action is to be performed once for each value: $i = 1, i = 2, \ldots, i = 10$, after which the loop terminates. Overall, the loop executes a total of 10 times. Prior to each of the 10 executions, the value of i is checked; as it is less than or equal to 10, the action is performed. At the end of each of the 10 iterations, the variable i is incremented by 1 and the termination condition is checked again, repeating the process. There are several different types of loops that vary in syntax and style but they all have the same three basic components.

4.1. While Loops

A *while loop* is a type of loop that places the three components in their logical order. The initialization statement is written before the loop code. Typically the keyword WHILE is used to specify the continuation/termination condition. Finally, the iteration statement is usually performed at the end of the loop *inside* the code block associated with the loop. A small, counter-controlled while loop is presented in Algorithm 4.1 which illustrates the previous example of iterating a variable i from 1 to 10.

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

Algorithm 4.1: Counter-Controlled While Loop

Prior to the WHILE statement, the variable i is initialized to 1. This action is only

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

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

4.1.1. Example

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

As an example, consider the problem of *normalizing* a number as is typically done in scientific notation. Given a number x (for simplicity, we'll consider $x \geq 1$), we divide it by 10 until its value is in the interval $[1, 10)$, keeping track of how many times we've divided by 10. For example, if we have the number $x = 32,145.234$, we would divide by 10 four times, resulting in 3.2145234 so that we could express it as

$$3.2145234 \times 10^4$$

A simple realization of this process is presented in Algorithm 4.2. Rather than some fixed number of iterations, the number of times the loop executes depends on how large x is. For the example mentioned, it executes 4 times; for an input of $x = 10,000,000$ it would execute 7 times. A while loop allows us to specify the repetition process without

having to know up front how many times it will execute.

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

Algorithm 4.2: Normalizing a Number With a While Loop

4.2. For Loops

A *for loop* is similar to a while loop but allows you to specify the three components on the same line. In many cases, this results in a loop that is more readable; if the code block in a while loop is long it may be difficult to see the initialization, continuation, and iteration statements clearly. For loops are typically used to iterate over elements stored in a collection such as an array (see Chapter 7). Usually the keyword FOR is used to identify all three components. A general example is given in Algorithm 4.3.

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

Algorithm 4.3: A General For Loop

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

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

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

Algorithm 4.4: Counter-Controlled For Loop

4.2.1. Example

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

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

Algorithm 4.5: Summation of Numbers in a For Loop

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

4.3. Do-While Loops

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

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

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

Algorithm 4.6: Counter-Controlled Do-While Loop

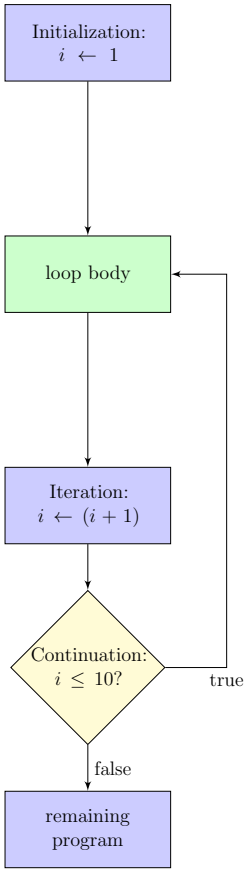


Figure 4.2.: A Do-While Loop Flow Chart. The continuation condition is checked *after* the loop body.

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

From this perspective, a do-while loop can also be seen as a do-until loop: perform a task *until* some condition is no longer satisfied. The subtle wording difference implies

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

Algorithm 4.7: Flag-Controlled Do-While Loop

that we'll perform the action before checking to see if it should be performed again.

4.4. Foreach Loops

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

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

Algorithm 4.8: Example Foreach Loop

How the elements are stored in the collection and how they are iterated over is not our (primary) concern. We simply want to apply the same block of code to each element, the foreach loop handles the details on how each element is iterated over. The syntax also provides a way to refer to each element (the a variable in the algorithm). On each iteration of the loop, the foreach loop updates the reference a to the *next* element in the array. The loop terminates after it has iterated through each and every one of the elements. In this way, a foreach loop simplifies the syntax: we don't have to specify any of the three components ourselves. As a more concrete example, consider iterating over each student in a course roster. For each student, we wish to compute their grade and then email them the results. The foreach loop allows us to do this without worrying about the iteration details (see Algorithm 4.9).

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

Algorithm 4.9: Foreach Loop Computing Grades

4.5. Other Issues

4.5.1. Nested Loops

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

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

Algorithm 4.10: Nested For Loops

The outer for loop executes a total of 20 times while the inner for loop executes 10 times. Since the inner for loop is *nested* inside the outer loop, the entire inner loop executes all 10 iterations *for each* of the 20 iterations of the outer loop. Thus, in total the inner most output operation executes $10 \times 20 = 200$ times. Specifically, it outputs $(1,1),(1,2),\ldots,(1,10),(2,1),(2,2),\ldots,(2,10),(3,1),\ldots,(20,10)$. Nested loops are commonly used when iterating over elements in two-dimensional arrays such as tabular data or matrices. Nested loops can also be used to process all *pairs* in a collection of elements.

4.5.2. Infinite Loops

Sometimes a simple mistake in the design of a loop can make it execute forever. For example, if we accidentally iterate a variable in the wrong direction or write the *opposite*

termination/continuation condition. Such a loop is referred to as an *infinite loop*. As an example, suppose we forgot the increment operation from a previous example.

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

Algorithm 4.11: Infinite Loop

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

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

The Zune Bug

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

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

The failure was that this code was never tested on the "corner cases" that it was designed for. No one thought to test the driver to see if it worked on the last day of a leap year.

```
while(days > 365) {
  if(IsLeapYear(year)) {
    if(days > 366) {
      days -= 366;
      year += 1;
   }
  } else {
    days -= 365;
    year += 1;
  }
}
```

Code Sample 4.1.: Zune Bug

The code worked the vast majority of the time, but this illustrates the need for rigorous testing.

4.5.3. Common Errors

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

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

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

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

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

prone.

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

4.5.4. Equivalency of Loops

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

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

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

4.6. Problem Solving With Loops

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

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

4.7. Examples

4.7.1. For vs While Loop

Let's consider how to write a loop to compute the classic geometric series,

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

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

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

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

```
   INPUT  : x, n ≥ 0
   OUTPUT : An approximated value of 1/(1−x) using a geometric series
1  sum ← 0
2  FOR (k = 0; k < n; k ← (k + 1)) DO
3  |  sum ← (sum + x^k)
4  END
5  output sum
```

Algorithm 4.12: Computing the Geometric Series Using a For Loop

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

This approach will be more straightforward with a while loop since the continuation condition will be more along the lines of "while the estimation is not yet good enough, continue the summation." This approach will also be easier if we keep track of both a *current* and a *previous* value of the summation, then computing and checking the difference will be easier.

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

Algorithm 4.13: Computing the Geometric Series Using a While Loop

On lines 1–2 we initialize our values to ensure that the while loop will execute at least once. In the continuation condition, we use the absolute value of the difference as the series can *oscillate* between negative and positive values.

4.7.2. Primality Testing

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

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

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

Now consider this more general problem: given an integer $m > 1$, determine *how many*

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

Algorithm 4.14: Determining if a Number is Prime or Composite

prime numbers $\leq m$ there are. A key observation is that we've *already solved* part of the problem: determining if a given number is prime in the previous exercise. To solve this more general problem, we could *reuse* or adapt our previous solution. In particular, we could surround the previous solution in an *outer* loop and iterate over integers from 2 up to m. The inner loop would then determine if the integer is prime and instead of outputting a result, could increment a counter of the number of primes it has found so far. This solution is presented in Algorithm 4.15.

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

Algorithm 4.15: Counting the number of primes.

4.7.3. Paying the Piper

Banks issue loans to customers as one lump sum called a *principle* P that the borrower must pay back over a number of *terms*. Usually payments are made on a monthly basis.

Further, banks charge an amount of *interest* on a loan measured as an Annual Percentage Rate (APR). Given these conditions, the borrower makes monthly payments determined by the following formula.

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

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

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

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

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

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

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

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

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

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

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

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

If we were to actually implement this we'd need to be more careful. This outlines the basic process, but keep in mind that US dollars are only accurate to cents. A monthly payment

can't be \$43.871 cents. We'll need to take care to round properly. This introduces another issue: by rounding the final month's payment may not match the expected monthly payment (we may over or under pay in the final month). An actual implementation may need to handle the final month's payment separately with different logic and operations than are outside the loop.

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

Algorithm 4.16: Computing a loan amortization schedule

4.8. Exercises

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

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

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

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

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

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

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

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

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

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

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

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

Write a program that takes the following input:

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

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

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

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

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

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

Exercise 4.5. Write a program that computes various statistics on a collection of numbers that can be read in from the command line, as command line arguments, or via other means. In particular, given n numbers,

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

your program should compute the following statistics:

- The minimum number
- The maximum number
- The mean,

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- The variance,

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

- And the standard deviation,

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

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

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

your output should look something like:

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

Exercise 4.6. The ancient Greek mathematician Euclid developed a method for finding the greatest common divisor of two positive integers, a and b. His method is as follows:

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

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

Exercise 4.7. Write a program to estimate the value of $e \approx 2.718281\ldots$ using the series:

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

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

Exercise 4.8. The value of π can be expressed by the following infinite series:

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

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

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

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

Exercise 4.9. The sine function can be approximated using the following Taylor series.

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

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

Exercise 4.10. One way to compute π is to use Machin's formula:

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}$$

To compute the arctan function, you could use the following series:

$$\arctan x = \frac{x}{1} - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots = \sum_{i=0}^{\infty} \frac{(-1)^i}{2k+1} x^{2i+1}$$

Write a program to estimate π using these formulas but allowing the user to specify how many terms to use in the series to compute it. Compare the estimate with the built-in definition of π in your language.

Exercise 4.11. The *arithmetic-geometric* mean of two numbers x, y, denoted $M(x, y)$ (or $\text{agm}(x, y)$) can be computed iteratively as follows. Initially, $a_1 = \frac{1}{2}(x + y)$ and $g_1 = \sqrt{xy}$ (i.e. the normal arithmetic and geometric means). Then, compute

$$\begin{aligned} a_{n+1} &= \tfrac{1}{2}(a_n + g_n) \\ g_{n+1} &= \sqrt{a_n g_n} \end{aligned}$$

The two sequences will converge to the same number which is the arithmetic-geometric mean of x, y. Obviously we cannot compute an infinite sequence, so we compute until $|a_n - g_n| < \epsilon$ for some small number ϵ.

Exercise 4.12. The integral of a function is a measure of the area under its curve. One numerical method for computing the integral of a function $f(x)$ on an interval $[a, b]$ is the *rectangle* rule. Specifically, an interval $[a, b]$ is split up into n equal subintervals of size $h = \frac{b-a}{n}$. Then the integral is approximated by computing:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(a + ih) \cdot h$$

Write a program to approximate an integral using the rectangle method. For this particular exercise you will integrate the function

$$f(x) = \frac{\sin x}{x}$$

For reference, the function is depicted in Figure 4.3. Write a program that will read the end points a, b and the number of subintervals n and computes the integral of f using the rectangle method. It should then output the approximation.

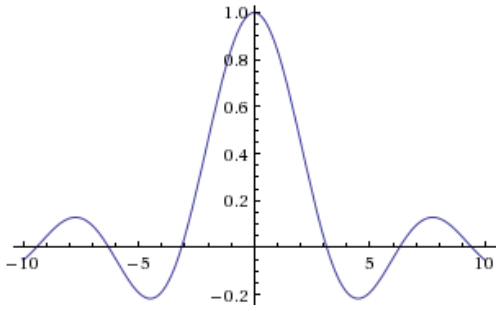


Figure 4.3.: Plot of $f(x) = \frac{\sin x}{x}$

Exercise 4.13. Another way to compute an integral is to a technique called *Monte Carlo Integration*, a randomized numerical integration method.

Given the interval $[a, b]$, we enclose the function in a region of interest with a rectangle of a known area A_r. We then randomly select n points within the rectangle and count the number of random points that are within the function's curve. If m of the n points are within the curve, we can estimate the integral to be

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{m}{n} A_r$$

Consider again the function $f(x) = \frac{\sin(x)}{x}$. Note that the global maximum and minimum of this function are 1 and ≈ -0.2172 respectively. Therefore, we can also restrict the rectangle along the y-axis from $-.25$ to 1. That is, the lower left of the rectangle will be $(a, -.25)$ and the upper right will be $(b, 1)$ for a known area of

$$A_r = |a - b| \times 1.25$$

Figure 4.4 illustrates the rectangle for the interval $[-5, 5]$.

Write a program that will takes as input interval values a, b, and an integer n and perform a Monte Carlo estimate of the integral of the function above. Realize that this is just an approximation and it is randomized so your answers may not match exactly and may be different on various executions of your program. Take care that you handle points within the curve but under the x-axis correctly.

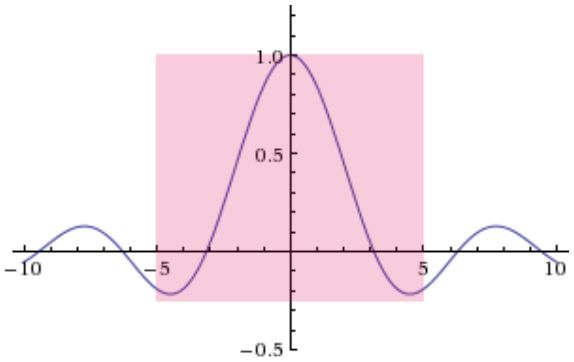


Figure 4.4.: A rectangle for the interval $[-5, 5]$.

Exercise 4.14. Consider a ball trapped in a 2-D box. Suppose that it has an initial position (x, y) within the box (the box's dimensions are specified by its lower left (x_ℓ, y_ℓ) and an upper right (x_r, y_r) points) along with an initial angle of travel θ in the range $[0, 2\pi)$. As the ball travels in this direction it will eventually collide with one of the sides of the box and bounce off. For this model, we will assume no loss of velocity (it keeps going) and its angle of reflection is perfect.

Write a program that takes as input, $x, y, \theta, x_\ell, y_\ell, x_r, y_r$, and an integer n and computes the first $n-1$ Euclidean points on the box's perimeter that the ball bounces off of in its travel (include the initial point in your printout for a total of n points). You may assume that the input will always be "good" (the ball will always begin somewhere inside the box and the lower left and upper right points will not be reversed).

As an example, consider the inputs:

$$x = 1, y = 1, \theta = .392699, x_\ell = 0, y_\ell = 0, x_r = 4, y_r = 3, n = 20$$

Starting at $(1, 1)$, the ball travels up and to the right bouncing off the right wall. Figure 4.5 illustrates this and the subsequent bounces back and forth.

Your output should simply be the points and should look something like the following.

```
(1.000000, 1.000000)
(4.000000, 2.242640)
(2.171572, 3.000000)
(0.000000, 2.100506)
(4.000000, 0.443652)
(2.928929, 0.000000)
(0.000000, 1.213202)
(4.000000, 2.870056)
(3.686287, 3.000000)
(0.000000, 1.473090)
```

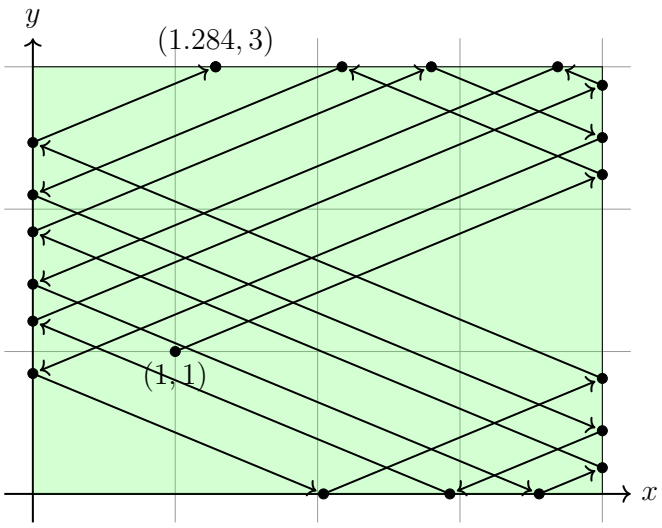


Figure 4.5.: Follow the bouncing ball

```
(3.556355, 0.000000)
(4.000000, 0.183764)
(0.000000, 1.840617)
(2.798998, 3.000000)
(4.000000, 2.502529)
(0.000000, 0.845675)
(2.041640, 0.000000)
(4.000000, 0.811179)
(0.000000, 2.468033)
(1.284282, 3.000000)
```

Exercise 4.15. An integer $n \geq 2$ is *prime* if its only divisors are 1 and itself, n. For example, $2, 3, 5, 7, 11, \ldots$ are primes. Write a program that outputs all prime numbers 2 up to m where m is read as input.

Exercise 4.16. An integer $n \geq 2$ is *prime* if the only integers that evenly divide it are 1 and n itself, otherwise it is *composite*. The *prime factorization* of an integer is a list of its prime divisors along with their multiplicities. For example, the prime decomposition of $188,760$ is:

$$188,760 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 5 \cdot 11 \cdot 11 \cdot 13$$

Write a program that takes an integer n as input and outputs the prime factorization of n. If n is invalid, an appropriate error message should be displayed instead. Your output should look something like the following.

```
1001 = 7 * 11 * 13
```

Exercise 4.17. One way of estimating π is to randomly sample points within a 2×2 square centered at the origin. If the distance between the randomly chosen point (x, y) and the origin is less than or equal to 1, then the point lies inside the unit circle centered at the origin and we count it. If the point lies outside the circle then we can ignore it. If

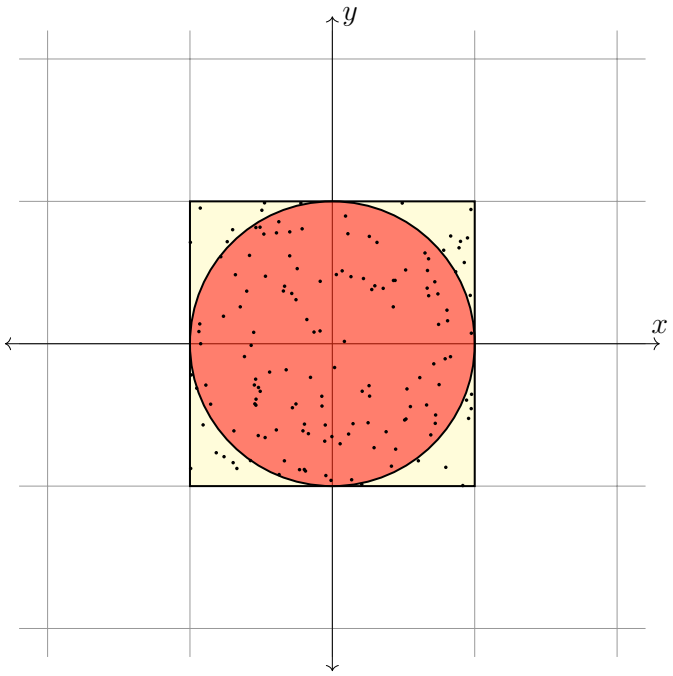


Figure 4.6.: Sampling points in a circle

we sample n points and m of them lie within the circle, then π can be estimated as

$$\pi \approx \frac{4m}{n}$$

Given a point (x, y), its distance from the origin is simply

$$\sqrt{x^2 + y^2}$$

This idea is illustrated in Figure 4.6. Example code is given to randomly generate numbers within a bound. Write a program that takes an integer n as input and randomly samples n points within the 2×2 square and outputs an approximation of π.

Of course, you'll need a way to generate random numbers within the range $[-1, 1]$. Since you are using some randomization, the result is just an *approximation* and may not match exactly or even be the same between two different runs of your program.

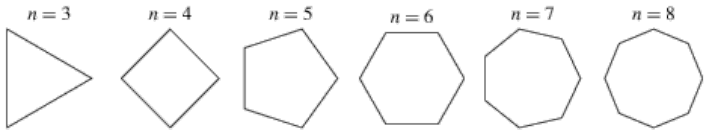


Figure 4.7.: Regular polygons

Exercise 4.18. A regular polygon is a polygon that is equiangular. That is, it has n sides and n points whose angle from the center are all equal in measure. Examples for $n = 3$ through $n = 8$ can be found in Figure 4.7.

Write a program that takes n and a radius r as inputs and computes the points of a regular n-sided polygon centered at the origin $(0, 0)$. Each point should be distance r from the origin and the first point should lie on the positive x-axis. Each subsequent point should be at an angle θ equal to $\frac{2\pi}{n}$ from the previous point. Recall that given the polar coordinates θ, r we can convert to cartesian coordinates (x, y) using the following.

$$\begin{aligned} x &= r \cdot \cos\theta \\ y &= r \cdot \sin\theta \end{aligned}$$

Your program should be robust enough to check for invalid inputs. If invalid, an error message should be printed and the program should exit.

For example, running your program with $n = 5, r = 6$ should produce the points of a pentagon with "radius" 6. The output should look something like:

```
Regular 5-sided polygon with radius 6.0:
(6.0000, 0.0000)
(1.8541, 5.7063)
(-4.8541, 3.5267)
(-4.8541, -3.5267)
(1.8541, -5.7063)
```

Exercise 4.19. Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be two points in the cartesian plane which define a *line* segment. Suppose we travel along this line starting at p_1 taking n steps that are an equal distance apart until we reach p_2. We wish to know which points correspond to each of these steps and which step along this path is *closest* to anther point $p_3 = (x_3, y_3)$. Recall that the distance between two points can be computed using the Euclidean distance formula:

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

Write a program that takes three points and an integer n as inputs and outputs a sequence of points along the line defined by p_1, p_2 that are distance $\frac{\delta}{n}$ apart from each other. It

should also indicate which of these computed points is the closest to the third point. For example, the execution of your program with inputs $0, 2, -5.5, 7.75, -2, 3, 10$ should produce output that looks something like:

```
(0.00, 2.00) to (-5.50, 7.75) distance: 7.9569
(0.00, 2.00)
(-0.55, 2.58)
(-1.10, 3.15)
(-1.65, 3.72)  <-- Closest point to (-2, 3)
(-2.20, 4.30)
(-2.75, 4.88)
(-3.30, 5.45)
(-3.85, 6.02)
(-4.40, 6.60)
(-4.95, 7.17)
(-5.50, 7.75)
```

Exercise 4.20. The natural log of a number x is usually computed using some numerical approximation method. One such method is to use the following Taylor series.

$$\ln x = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \frac{(x-1)^4}{4} + \cdots$$

However, this only works for $|x-1| \leq 1$ (except for $x = 0$) and diverges otherwise. For x such that $|x| > 1$, we can use the series

$$\ln \frac{y}{y-1} = \frac{1}{y} + \frac{1}{2y^2} + \frac{1}{3y^3} + \cdots$$

where $y = \frac{x}{x-1}$. Of course such an infinite computation cannot be performed by a computer. Instead, we approximate $\ln x$ by computing the series out to a finite number of terms, n. Your program should print an error message and exit for $x \leq 0$; otherwise it should use the first series for $0 < x \leq 1$ and the second for $x > 1$.

Another series that has better convergence properties and works for any range of x is as follows

$$\ln x = \ln \frac{1+y}{1-y} = 2y \left(1 + \frac{1}{3}y^2 + \frac{1}{5}y^4 + \frac{1}{7}y^6 \cdots\right)$$

where $y = \frac{(x-1)}{(x+1)}$.

You will write a program that approximates $\ln x$ using these two methods computed to n terms. You will also compute the error of each method by comparing the approximated value to the standard math library's `log` function.

Your program should accept x and n as inputs. It should be robust enough to reject any invalid inputs ($\ln x$ is not defined for $x = 0$ you may also print an error for any negative value; n must be at least one). It will then compute an approximation using both methods and print the relative error of each method.

For example, the execution of your program with inputs $3.1415, 6$ should produce output that looks something like:

```
Taylor Series: ln(3.1415) ~= 1.11976
  Error: 0.02494
Other Series:  ln(3.1415) ~= 1.14466
  Error: 0.00004
```

Exercise 4.21. There are many different numerical methods to compute the square root of a number. In this exercise, you will implement several of these methods.

(a) The Exponential Identity Method involves the following identity:

$$\sqrt{x} = e^{\frac{1}{2}\ln(x)}$$

Which assumes the use of built-in (or math-library) functions for e and the natural log, ln.

(b) The Babylonian Method involves iteratively computing the following recurrence:

$$a_i = \frac{1}{2}\left(a_{i-1} + \frac{x}{a_{i-1}}\right)$$

where $a_1 = 1.0$. Computation is repeated until $|a_i - a_{i-1}| \leq \delta$ where δ is some small constant value.

(c) A method developed for one of the first electronic computers (EDSAC [29]) involves the following iteration. Let $a_0 = x$, $c_0 = x - 1$. Then compute

$$\begin{aligned} a_{i+1} &= a_i - \tfrac{a_i c_i}{2} \\ c_{i+1} &= \tfrac{c_i^2(c_i - 3)}{4} \end{aligned}$$

The iteration is performed for as many iterations as specified (n), or until the change in a is negligible. The resulting value for a is used as an approximation for $\sqrt{x} \approx a$. However, this method only works for values of x such that $0 < x < 3$. We can easily overcome this by *scaling* x by some power of 4 so that the scaled value of x satisfies $\frac{1}{2} \leq x < 2$. After applying the method we can then scale back up by the appropriate value of 2 (since $\sqrt{4} = 2$). Algorithm 4.17 describes how to scale x.

Write a program to compute the square root of an input number using these methods and compare your results.

```
1 power ← 0
2 WHILE x < 1/2 DO
      //Scale up
3     x ← (x · 4)
4     power ← (power − 1)
5 END
6 WHILE x ≥ 2 DO
      //Scale down
7     x ← x/4
8     power ← (power + 1)
9 END
```

Algorithm 4.17: Scaling a value x so that it satisfies $\frac{1}{2} \leq x < 2$. After execution, *power* indicates what power of 2 the value x was scaled by.

Exercise 4.22. There are many different numerical methods to compute the natural logarithm of a number, $\ln x$. In this exercise, you will implement several of these methods.

(a) A formula version approximates the natural logarithm as:

$$\ln(x) \approx \frac{\pi}{2M(1, 4/s)} - m \ln(2)$$

Where $M(a, b)$ is the *arithmetic-geometric* mean and $s = x2^m$. In this formula, m is a parameter (a larger m provides more precision).

(b) The standard Taylor Series for the natural logarithm is:

$$\ln(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n}(x-1)^n$$

As we cannot compute an infinite series, we will simply compute the series to the first m terms. Also note that this series is not convergent for values $x > 1$

(c) Borchardt's algorithm is an iterative method that works as follows. Let

$$a_0 = \frac{1+x}{2} \qquad b_0 = \sqrt{x}$$

Then repeat:

$$\begin{aligned} a_{k+1} &= \tfrac{a_k + b_k}{2} \\ b_{k+1} &= \sqrt{a_{k+1} b_k} \end{aligned}$$

until the absolute difference between a_k, b_k is small; that is $|a_k - b_k| < \epsilon$. Then the logarithm is approximated as

$$\ln(x) \approx 2\frac{x-1}{a_k + b_k}$$

(d) Newton's method works if x is sufficiently close to 1. It works by setting $y_0 = 1$ and then computing

$$y_{n+1} = y_n + 2\frac{x - e^{y_n}}{x + e^{y_n}}$$

The iteration is performed m times.

To ensure that some of the methods above work, you may need to *scale* the number x to be as close as possible to 1. One way to do this is to divide or multiply by e until x is close to 1. Suppose we divided by e k times; that is $x = z \cdot e^k$ where z is close to 1. Then

$$\ln(x) = \ln(z \cdot e^k) = \ln(z) + \ln(e^k) = \ln(z) + k$$

Thus, we can apply the methods above to Newton's method to z and add k to the result to get $\ln(x)$. A similar trick can be used to ensure that the Taylor Series method is convergent.

Exercise 4.23. Write a program that takes a positive integer n as a command line argument and outputs a list of all *pairs* of integers x, y such that $1 \leq x, y \leq n$ whose sum, $x + y$ is prime. For example, if $n = 5$, the output should look something like the following.

```
1 + 2 = 3 is prime
1 + 4 = 5 is prime
2 + 3 = 5 is prime
2 + 5 = 7 is prime
3 + 4 = 7 is prime
```

Exercise 4.24. Consider the following variation on the classical "FizzBuzz" challenge. Write a program that will print out numbers 1 through n where n is provided as a command line argument. However, if the number is a perfect square (that is, the square of some integer; for example $1 = 1^2, 4 = 2^2, 9 = 3^2$, etc.) print "Go Huskers" instead. If the number is a prime $(2, 3, 5, 7,$ etc.) print "Go Cubs" instead.

Exercise 4.25. Implement a program to solve the classic "Rainfall Problem" (which has been used in CS education studies to assess programming knowledge and skills). Write an interactive program that repeatedly prompts the user to enter an integer (which

represents the amount of rainfall on a particular day) until it reads the integer 99999. After 99999 is entered, it should print out the *correct* average. That is, it should not count the final 99999. Negative values should also be ignored. For example, if the user entered the sequence `4 0 -1 10 99999` the output should look something like the following.

```
Average rainfall: 4.66
```

Exercise 4.26. Write a program that takes an integer n and a subsequent list of integers as command line arguments and determines which number(s) between 1 and n are missing from the list. For example, if the following numbers are given to the program: `10 5 2 3 9 2 8 8` your output should look something like:

```
Missing numbers 1 thru 10:
1, 4, 6, 7, 10
```

Exercise 4.27. Write a program that takes a list of pairs of numbers representing latitudes/longitudes (on the scale $[-180, 180]$ (negative values correspond to the southern and western hemispheres). Then, starting with the first pair, calculate the intermediate air distances between each location as well as a final total distance.

To compute air distance from location A to a location B, use the Spherical Law of Cosines:

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

where

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

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

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

For example, if the command line arguments were

```
40.8206 -96.756 41.8806 -87.6742 41.9483 -87.6556 28.0222 -81.7329
```

your output should look something like:

```
(40.8206, -96.7560) to (41.8806, -87.6742): 766.8053km
(41.8806, -87.6742) to (41.9483, -87.6556): 7.6836km
(41.9483, -87.6556) to (28.0222, -81.7329): 1638.7151km
Total Distance: 2413.2040
```

Exercise 4.28. A DNA sequence is made up of a sequence of four nucleotide bases, A, C, G, T (adenine, cytosine, guanine, thymine). One particularly interesting statistic of a DNA sequence is finding a *CG island*: a subsequence that contains the highest frequency of guanine and cytosine.

For simplicity, we will be interested in subsequences of a particular length, n that will be provided as part of the input.

Write a program that takes, as command line arguments, an integer n and a DNA sequence. The program should then find all subsequences of the given DNA string of length n with the maximal frequency of C and G in it. For example, if the DNA sequence is

```
ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTGCTGCTCTCCGGGGCCACGGC
```

and the "window" size that we're interested in is $n = 5$ then you would scan the sequence and find every subsequence with the maximum number of C or G bases. Your output should include *all* CG Islands (by indices) in the sequence similar to the following.

```
n = 5
highest frequency: 5 / 5 = 100.00%
CG Islands:
15 thru 20: CCCCC
16 thru 21: CCCCG
17 thru 22: CCCGG
18 thru 23: CCGGC
19 thru 24: CGGCC
42 thru 47: CCGGG
43 thru 48: CGGGG
44 thru 49: GGGGC
45 thru 50: GGGCC
```

Exercise 4.29. Write a program that will assist people in saving for retirement using a tax-deferred 401k program.

Your program will read the following inputs as command line arguments.

- An initial starting balance
- A monthly contribution amount (we'll assume its the same over the life of the savings plan)
- An (average) annual rate of return (on the scale $[0, 1]$)
- An (average) annual rate of inflation (on the scale $[0, 1]$)
- A number of years until retirement

Your program will then compute a monthly savings table detailing the (inflation-adjusted) interest earned each month, contribution, and new balance. The inflation-adjusted rate of return can be computed with the following formula.

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

To get the monthly rate, simply divide by 12. Each month, interest is applied to the balance at this rate (prior to the monthly deposit) and the monthly contribution is added. Thus, the earnings compound month to month.

Be sure that your program handles bad inputs as well as it can. It should also round to the nearest cent for every figure. Finally, as of 2014, annual 401k contributions cannot exceed $17,500. If the user's proposed savings schedule violates this limit, display an error message instead of the savings table.

For inputs `10000 500 0.09 0.012 10` your output should look something like the following:

```
 Month    Interest     Balance
     1 $     64.23 $  10564.23
     2 $     67.85 $  11132.08
     3 $     71.50 $  11703.58
     4 $     75.17 $  12278.75
     5 $     78.87 $  12857.62
     6 $     82.58 $  13440.20
     7 $     86.33 $  14026.53
     8 $     90.09 $  14616.62
     9 $     93.88 $  15210.50
```

```
...
  116 $    678.19 $ 106767.24
  117 $    685.76 $ 107953.00
  118 $    693.37 $ 109146.37
  119 $    701.04 $ 110347.41
  120 $    708.75 $ 111556.16
Total Interest Earned: $  41556.16
Total Nest Egg: $ 111556.16
```

Exercise 4.30. An *affine cipher* is an encryption scheme that encrypts messages using the following function:

$$e_k(x) = (ax + b) \bmod n$$

Where n is some integer and $0 \leq a, b, x \leq n - 1$. That is, we fix n, which will be used to encode an alphabet as in Table 4.1.

x	character
0	(space)
1	A
2	B
3	C
⋮	⋮
25	Y
26	Z
27	.
28	!

Table 4.1.: Character Mapping for $n = 29$

Then we choose integers a, b to define the encryption function. Suppose $a = 10, b = 13$, then

$$e_k(x) = (10x + 13) \bmod 29$$

So to encrypt "HELLO!" we would encode it as $8, 5, 12, 12, 15, 27$, then encrypt them,

$$e_k(8) = (10 \cdot 8 + 13) \bmod 29 = 6$$

$$e_k(5) = (10 \cdot 5 + 13) \bmod 29 = 5$$

$$e_k(12) = (10 \cdot 12 + 13) \bmod 29 = 17$$

$$e_k(12) = (10 \cdot 12 + 13) \bmod 29 = 17$$

$$e_k(15) = (10 \cdot 15 + 13) \bmod 29 = 18$$

$$e_k(28) = (10 \cdot 28 + 13) \bmod 29 = 3$$

Which, when mapped back to characters using our encoding is "FEQQRC."

To decrypt a message we need to invert the decryption function, that is,

$$d_k(y) = \left(a^{-1} \cdot (y - b)\right) \bmod n$$

where a^{-1} is the inverse of a modulo n. The inverse of an integer a is the value such that

$$(a \cdot a^{-1}) \bmod n = 1$$

so for $a = 10, n = 29$, the inverse, $10^{-1} \bmod 29 = 3$ since $3 \cdot 10 \bmod 29 = 1$. Given a and n, how can we find an inverse, a^{-1}? Obviously it cannot be zero, nor can it be 1 (1 is its own inverse). There is a simple algorithm (the Extended Euclidean Algorithm) that can solve this problem, but $n = 29$ is small enough that a brute-force strategy of testing all possibilities will suffice.

Write a program that takes a, b and an encrypted message as command line arguments and decrypts the message. Your program should print the decrypted message and other cipher information to the standard output. For example:

```
a    = 10
b    = 13
a^-1 = 3
Encrypted Message: FEQQRC
Decrypted Message: HELLO!
```

Exercise 4.31. A *centroid* (or barycenter) of a plane figure is the arithmetic mean of all the points in the shape. The centroid of a non-self-intersecting closed polygon defined by n vertices $(x_0, y_0), (x_1, y_1), \ldots, (x_{n_1}, y_{n-1})$ is the point (C_x, C_y) which can be computed using:

$$C_{\mathrm{x}} = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i\ y_{i+1} - x_{i+1}\ y_i)$$

$$C_{\mathrm{y}} = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i\ y_{i+1} - x_{i+1}\ y_i)$$

with A being the polygon's signed area:

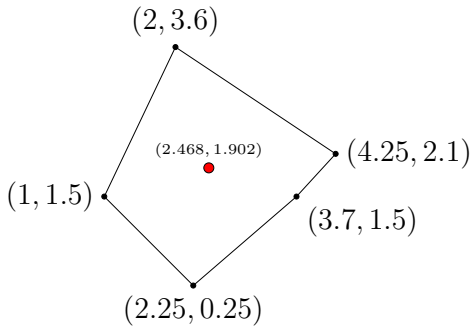


Figure 4.8.: A polygon and its centroid. Whoo!

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i\ y_{i+1} - x_{i+1}\ y_i)$$

In these formulas, the vertices are assumed to be numbered in order of their occurrence along the polygon's perimeter. Furthermore, the vertex $(x_n, y_n) = (x_0, y_0)$.

Write a program that takes n pairs of x-y coordinates as command line arguments that represent a polygon and computes its centroid. For example, the an input such as

```
1 1.5 2 3.6 4.25 2.1 3.7 1.5 2.25 0.25
```

would correspond to the polygon in Figure 4.8 with a centroid of $(2.46, 1.90)$. The output should look something like the following.

```
Centroid: (2.468405, 1.902874)
```

Exercise 4.32. Suppose we are given a set of n points $(x_0, y_0), (x_1, y_1), \ldots, (x_{n_1}, y_{n-1})$ in the plane. Write a program that outputs instructions for someone to "walk" to each point in the order that they are given starting at the origin by only going in the cardinal directions (not "as the bird flies"). For example, if the following input is given,

```
3 5 2 7 9 5
```

Then the output should look like the following.

```
Go north 5.00 units
Go east 3.00 units
Go north 2.00 units
```

```
Go west 1.00 units
Go south 2.00 units
Go east 7.00 units
```

Exercise 4.33. A *histogram* is a graphical representation of the distribution of numerical data. Typically, a histogram is represented as a (vertical) bar graph. However, we'll limit our attention to graphing a horizontal ASCII histogram.

In particular, write a program that takes a sequence of numerical values representing grades (on a 0–100 scale; reject any invalid values). The program will then display a horizontal graph of the distribution of these values over 6 fixed ranges that looks something like the following.

```
  0 -  49  (2, 11.11%) ******
 50 -  59  (1,  5.56%) ***
 60 -  69  (1,  5.56%) ***
 70 -  79  (5, 27.78%) **************
 80 -  89  (4, 22.22%) ************
 90 - 100  (5, 27.78%) **************
```

In the diagram, each star, `*` represents (at least) 2%

Exercise 4.34. Shannon entropy is a measure of *information* in a piece of data; specifically it is the expected value (average) of the information the data contains. A higher entropy value corresponds to more random data while 0 indicates highly structured data. Entropy is calculated using the following formula.

$$\mathcal{H}(X) = -\sum_{i=1}^{n} p(x_i) \cdot \log_2 p(x_i)$$

where $x_1, x_2, \ldots, x_n$ are the distinct symbols in the data sequence X and $p(x_i)$ is the probability of the x_i-th symbol. If the length of X is n, then this is simply

$$p(x_i) = \frac{c(x_i)}{n}$$

where $c(x_i)$ is a *count* of the number of times x_i appears in X. For example, the string "Hello World" has 8 distinct symbols with the probability of "l" being .2727, "o" being .1818 and the remaining being .0909. Thus the entropy would be

$$\mathcal{H}(\texttt{'Hello World'}) = 2.845351$$

Write a program that takes a string as command line input (which may contain spaces) and computes its entropy.

5. Functions

In mathematics, a function is a mapping from a set of *inputs* to a set of *outputs* such that each input is mapped to exactly one output. For example, the function

$$f(x) = x^2$$

maps numeric values to their squares. The input is a variable x. When we assign an actual value to x and evaluate the function, then the function has a value, its output. For example, setting $x = 2$ as input, the output would be $2^2 = 4$. Mathematical functions can have multiple inputs,

$$f(x, y) = x^2 + y^2 \quad f(x, y, z) = 2x + 3y - 4z$$

However, a function will only ever have a single output value.

In programming languages, a function (sometimes called subroutine or procedure) can take multiple inputs and produce one output value. We've already seen some examples of these functions. For example, most languages provide a math library that you can use to evaluate the square root or sine of a value x. We've also seen some functions with multiple input values such as the "power" function that allows you to compute $f(x, y) = x^y$. Finally, the main entry point to many programs is defined by a main function.

More formally, a function is a sequence of instructions (code) that is packaged into a *unit* that can be reused. A function performs a specific task: given a number of inputs, it executes some sequence of operations (executes some code) and "returns" (outputs) a result. The output can be captured into a variable or used in an expression by whatever code *invoked* or "called" the function.

Defining and using functions in programming has numerous advantages. The most obvious advantage is that it allows you a way to *organize* code. By separating a program it into distinct units of code it is more organized and it is clearer what each piece or segment of code does. This also facilitates top-down design: one way to approach a problem is to split it up into a series of subproblems until each subproblem is either trivial to deal with, or an existing, "off-the-shelf" solution already exists for it. Functions may serve as the logical unit for each subproblem.

Another advantage is that by organizing code into functions, those functions can be *reused* in multiple places either in your program/project or even in other programs/projects.

A prime example of this are the standard libraries available in most programming languages that provide functions to perform standard input/output or mathematical functions. These standard libraries provide functions that are used by thousands of different programs across multiple different platforms.

Functions also form an *isolated* unit of code. This allows for better and easier *testing*. By isolating pieces of code, we can rigorously test those pieces of code by themselves without worrying about the larger program or contexts.

Functions facilitates procedural abstraction. Placing code into functions allows you to abstract the details of how the function computes its answer. As an example: consider a standard math library's square root function: it may use some interpolation method, a Taylor series, or some other method entirely to compute the square root of a given number. However, by putting this functionality into a function, we, as programmers, do not need to concern ourselves about these details. Instead, we simply *use* the function, allowing us to focus on the larger issues at hand in our program.

5.1. Defining & Using Functions

Like variables, many programming languages may require that you declare a function before you can use it. A function declaration may simply include a description of the function's input/output and name. A function declaration may require you to define the function's body at the same time or separately. Functions can also have scope: some areas of the code may be able to "see" the function or know about it and be able to invoke the function while other areas of the code may *not* be able to see the function and therefore may not be able to invoke it.

Some interpreted programming languages use function hoisting which allows you to use/invoke functions *before* you declare them. This works because the interpreter does an initial scan of the code and identifies all function declarations. Only after it has "hoisted" all functions into scope does it start to execute the program. Thus, a function declaration can appear *after* it has been used and it will still work.

5.1.1. Function Signatures

A function is usually defined by its signature: every function can be identified by its name (also called an *identifier*), its list of *input parameters*, and its *output*. A function signature allows the programming language to uniquely identify each function so that

when you invoke a function there is no ambiguity in which function you are calling.

```
1 FUNCTION sum(a, b)
2 |   x ← a + b
3 |   return x
4 END
```

Algorithm 5.1: A function in pseudocode. In this case, the name (identifier) of the function is *sum* and it has two parameters, a and b. Its body is contained in lines 2–3. Its return value is indicated by the return statement on line 3.

A function declaration in pseudocode is presented in Algorithm 5.1. In the pseudocode, explicit variable types are omitted, and thus the return type is inferred from the return statement. In Figure 5.1 we have provided an example of a function declaration in the C programming language with each element labeled.

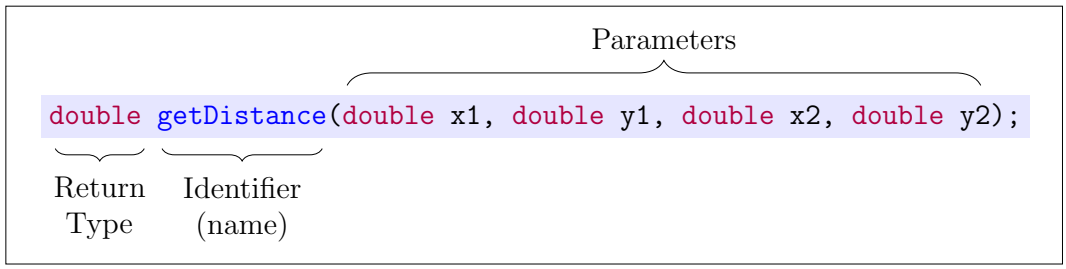


Figure 5.1.: A function declaration (prototype) in the C programming language with the return type, identifier, and parameter list labeled.

Some languages only allow you to use one identifier for one function (like variables) while other languages allow you to define multiple functions with the same identifier as long as the parameter list is different (see Section 5.3.2 below). In general, like variables, function names are case sensitive. Also similar to variables, modern lower camel casing is used with function names.

When defining the parameters to a function (its input), you usually provide a comma delimited list of variable names. In the case of statically typed languages, the types of the variable parameters are also specified. The order is important as when you invoke the function, the number of inputs must match the number of parameters in the function declaration. The variable types may also need to match. In some dynamically typed languages, you may be able to call functions with different types or you may be able to omit some of the parameters (see Section 5.3.4 below).

Similarly, the return type of the function may need to be specified in statically typed languages while with dynamic languages, functions may conditionally return different types. We generally refer to the "return value" or "return type" because when a function

is done executing, it "returns" the control flow back to the line of code that invoked it, returning its computed value.

You can also define functions that may not have any inputs or may not have any output. Some languages use the keyword `void` to indicate no return value and such functions are known as "void functions." When a function doesn't have any input values, its parameter list is usually empty.

The function signature may be accompanied by the function *body* which contains the actual code that specifies what the function does. Typically the function body is demarcated with a code block using opening and closing curly brackets, `{ ... }`. Within the function you can generally write any valid code including declaring variables. When you declare a variable inside a function, however, it is *local* to that function. That is, the variable's scope is only defined within the function. A local variable cannot be accessed outside the function, indeed the local variable does not usually survive when the function ends its execution and returns control back to line of code that called it. Function parameters are essentially locally scoped variables as well and can usually be treated as such.

5.1.2. Calling Functions

When a function has been defined and is in scope, you can *invoke* or "call" the function by coding the function name and providing the input parameters which can be either variables or literals. When provided as inputs, parameters are referred to as *arguments* to the function. The arguments are typically provided as a comma delimited list and placed inside parentheses.

Invoking a function changes the usual flow of control. When invoked, control flow is transferred over to the function. When the function finishes executing the code in its body, control flow returns to the point in the code that invoked it. It is common for a program to be written so that a function calls another function and that function calls another. This can form a deep chain of function calls in which the flow of control is transferred multiple times. Upon the completion of each function, control is returned back to the function that called it, referred to as the *calling function*.

If a function returns a value it can either be captured in a variable using an assignment

operator or by using it in an expression.

```
1 a ← 10
2 b ← 20
3 c ← sum(a, b)
```

Algorithm 5.2: Using a function. We invoke a function by indicating its name (identifier) and passing it arguments.

5.1.3. Organizing

Functions provide code organization, but functions themselves should also be organized. We've seen this with standard libraries. Functions that provide basic input/output are all grouped together into one library. Functions that involve math functions are grouped together into a separate math library.

Some languages allow you to define and "import" individual libraries which organize similar functions together. Some languages do this by collecting functions into "utility" classes or *modules*. Only when you import these modules do the functions come into scope and can be used in your code. If you do not import these modules, then the functions are out of scope and cannot be used.

In some languages once a function is imported it is part of the *global scope* and can be "seen" by any part of the code. This can cause *conflicts*: if you import modules from two different libraries each with different functions that have the same name or signature, then the two function definitions may be in conflict or it may make your code ambiguous as to which function you intend to invoke. This is sometimes referred to as "polluting the namespace." There are several techniques that can avoid this situation. Some languages allow you to place functions into a *namespace* to keep functions with the same name in different "spaces." Other languages allow you to place functions into different classes and then invoke them by explicitly specifying *which* class's function you want to call. Yet other languages don't have great support for function organization and it is the library designer's responsibility to avoid naming conflicts, typically by adding a library-specific prefix to every function.

5.2. How Functions Work

To understand how functions work in practice, it is necessary to understand how a program operates at a lower level. In particular, each program has a program stack (also called a call stack). A stack is a data structure that holds elements in a Last-In First-Out

(LIFO) manner. Elements are added to the "top" of the stack in an operation called *push* and elements can be removed from the top of the stack in an operation called *pop*. In general, elements cannot be inserted or removed from the middle or "bottom" of the stack.

In the context of a program, a call stack is used to keep track of the flow of control. Depending on the operating system, compiler and architecture, the details of how elements are stored in the program stack may vary. In general when a program begins, the operating system loads it into memory at the bottom of the call stack. Global variables (static data) are stored on top of the main program. Each time a function is called, a new *stack frame* is created and pushed on top of the stack. This frame contains enough space to hold values for the arguments passed to the function, local variables declared and used by the function, as well as a space for a return value and a return *address*. The return address is a memory location that the program should return to after the execution of the function. That way, when the function finishes its execution, the stack frame can be removed (popped) and the lower stack frame of the calling function is preserved. This is a very efficient way to keep track of the flow of control in a program. As each function calls another function, each stack frame is preserved by pushing a new one on top of the program stack. Each time a function terminates execution and returns, the removal of the stack frame means that all local variables go *out of scope*. Thus, variables that are local to a function are not accessible outside the function.

To illustrate, consider the following snippet of C code. The `main()` function invokes the `average()` function which in turn invokes the `sum()` function. Each invocation creates a new stack frame on top of the last in the program stack which is depicted in Figure 5.2.

```
1  double sum(double a, double b) {
2    double x = a + b;
3    return x;
4  }
5
6  double average(double a, double b) {
7    double y = sum(a, b) / 2.0;
8    return y;
9  }
10
11 int main(int argc, char **argv) {
12   double n = 10.0;
13   double m = 16.0;
14   double ave = average(n, m);
15   printf("average = %f\n", ave);
16   return 0;
17 }
```

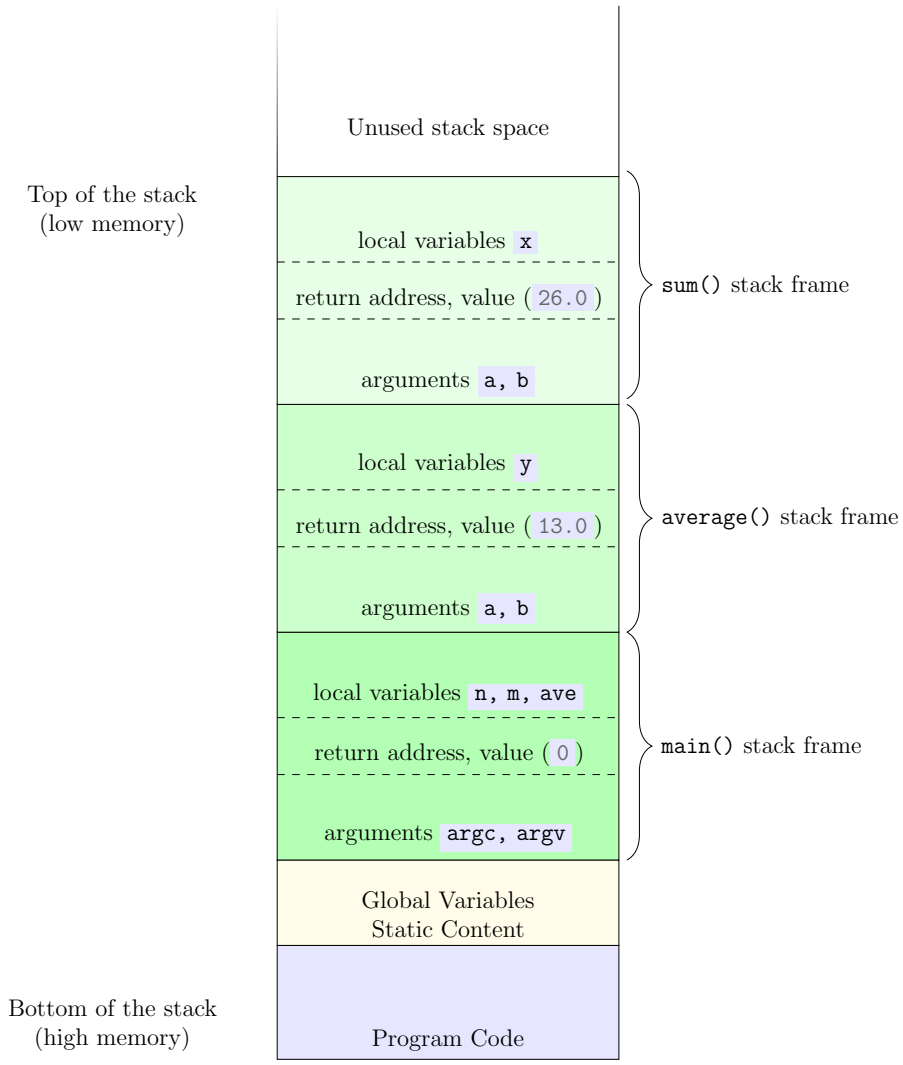


Figure 5.2.: Program Stack. At the bottom we have the program's code, followed by static content such as global variables. Each function call has its own stack frame along with its own arguments and local variables. In particular, the variable arguments `a` and `b` in two different stack frames are completely different variables. Upon returning from the `sum()` function call, the top-most stack frame would be popped and removed, returning to the code for the `average()` function via the return address. The stack is depicted bottom-up with high memory at the bottom and low memory at the top, but this may differ depending on the architecture.

5.2.1. Call By Value

When a function is invoked, arguments are passed to it. When you invoke a function you can pass it variables as arguments. However, the variables themselves are not passed to the function, but instead the values *stored* in the variables at the time that you call the function are passed to the function. This mechanism is known as call by value and the variable values are *passed by value* to the function.

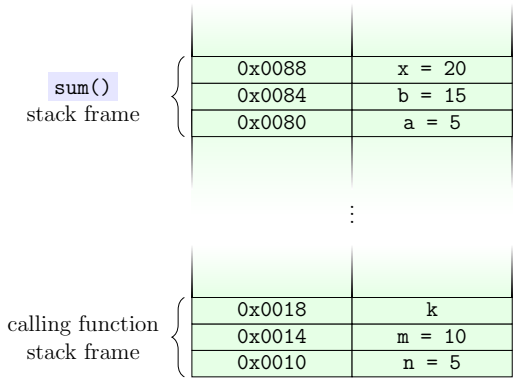
Recall that the arguments passed to a function are placed in a new stack frame for that function. Thus, in reality *copies* of the values of the variables are passed to the function. Any changes to the parameters inside the function have *no effect* on the original variables that were "passed" to the function when it was invoked.

To illustrate, consider the following C code. We have a function `sum` that takes two integer parameters `a` and `b` which are passed by value. Inside `sum`, we create another variable `x` which is the sum of the two passed variables. We then *change* the value of the first variable, `a` to `10`. Elsewhere in the code we call `sum` on two variables, `n`, `m` with values 5 and 15 respectively. The invocation of the function `sum` means that the two values, 5 and 15, stored in the variables are copied into a new stack frame. Thus, changing the value to the first parameter *changes the copy* and has no effect on the variable `n`. At the end of this code snippet `n` retains its original value of 5. The program stack frames are depicted in Figure 5.3.

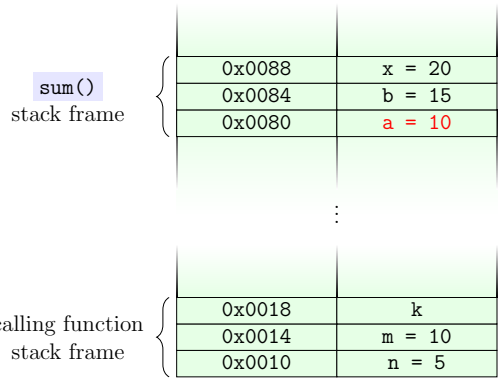
```
1  int sum(int a, int b) {
2    int x = a + b;
3    a = 10;
4    return x;
5  }
6
7  ...
8
9  int n = 5;
10 int m = 15;
11 int k = sum(n, m);
```

5.2.2. Call By Reference

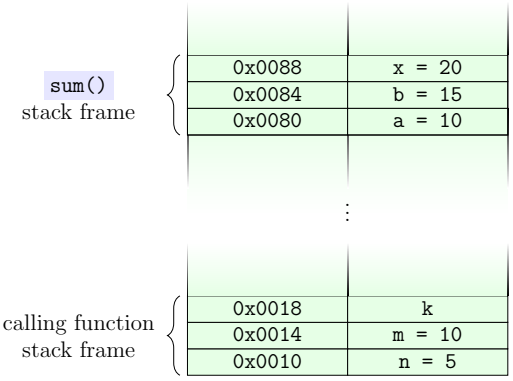
Some languages allow you to pass a parameter to a function by providing its memory address instead of the value stored in it. Since the memory address is being provided to the function, the function is able to access the original variable and manipulate the



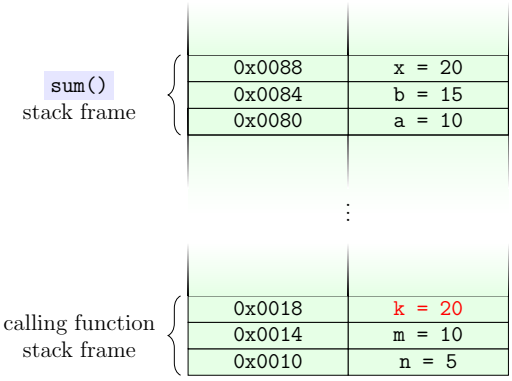
(a) Upon invocation of the `sum()` function, a new stack frame is created which holds the parameters and local variable. The parameter variables `a` and `b` are *distinct* from the original argument variables `n` and `m`.



(b) The change to the variable `a` in the `sum()` function changes the parameter variable, but the original variable `n` is unaffected.



(c) When the `sum()` function finishes execution, its stack frame is removed and the variables `a`, `b`, and `x` are no longer valid. The return value 20 is stored in another return value location.



(d) The returned value is stored in the variable `k` and the variable `n` retains its original value.

Figure 5.3.: Demonstration of Pass By Value. Passing variables by value means that *copies* of the values stored in the variables are provided to the function. Changes to parameter variables do not affect the original variables.

contents stored at that memory address. In particular, the function is now able to make changes to the original variable. This mechanism is known as call by reference and the variables are *passed by reference*.

To illustrate, consider the following C code. Here, the variable `a` is passed by reference (`b` is still passed by value, the `*a` and `&n` in the following code are dereferencing and referencing operators respectively. For details, see Section 18.2). Below when we invoke the `sum()` function, we pass not the value stored in `n`, but the memory address of the variable `n`. Thus, when we change the value of the variable `a` in the function, we are actually changing the value of `n` (since we have access to its memory location). At the conclusion of this snippet of code, the value stored in `n` has been changed to 10. The program stack frames are depicted in Figure 5.4.

```
1  int sum(int *a, int b) {
2    int x = *a + b;
3    *a = 10;
4    return x;
5  }
6
7  ...
8
9  int n = 5;
10 int m = 15;
11 int k = sum(&n, m);
```

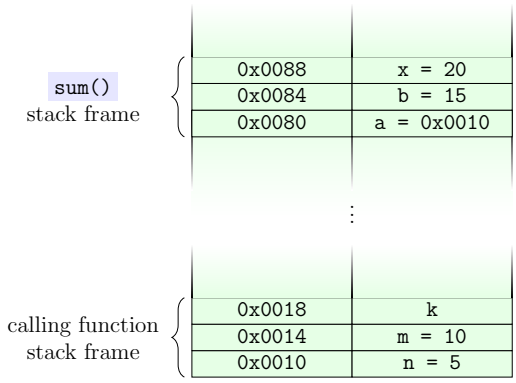
Whether or not a variable is passed by value or by reference depends on the language, type of variable, and the syntax used.

5.3. Other Issues

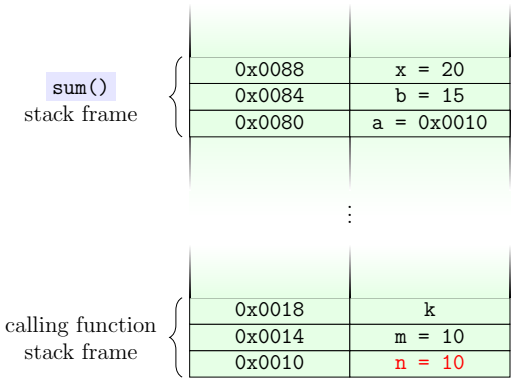
5.3.1. Functions as Entities

In programming languages, any entity that can be stored in a variable or passed as an argument to a function or returned as a value from a function is referred to as a "first-class citizen." Numerical values for example are usually first-class citizens as they can be stored in variables and passed to functions.

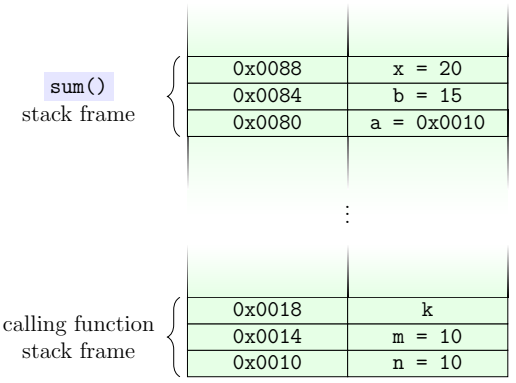
Functional Programming is a programming language paradigm in which functions themselves are first-class citizens. That is, functions can be assigned to variables, functions



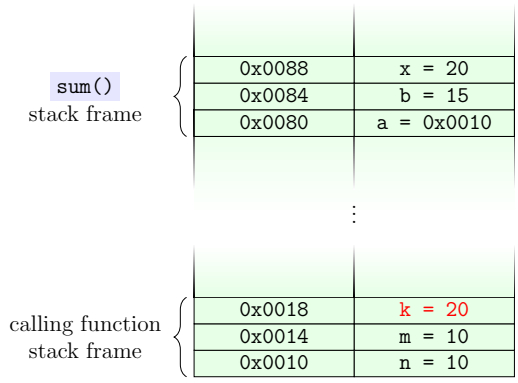
(a) Upon invocation of the `sum()` function, a new stack frame is created which holds the parameters and local variable. The parameter variable `a` holds the memory location of the original variable `n`.



(b) The change to the variable `a` in the `sum()` function actually changes what the variable `a` *references*. That is, the original variable `n`.



(c) When the `sum()` function finishes execution, its stack frame is removed and the variables `a`, `b`, and `x` are no longer valid. The return value 20 is stored in another return value location.



(d) The returned value is stored in the variable `k` and the value in the variable `n` has now changed.

Figure 5.4.: Demonstration of Pass By Reference. Passing variables by reference means that the *memory address* of the variables are provided to the function. The function is able to make changes to the original variable because it knows where it is stored.

can be passed to other functions as arguments, and functions can even return functions as a result. This is done as a matter of course in functional programming languages such as Haskell and Clojure, but many programming languages contain some functional aspects.

For example, some languages support the same concept by using function *pointers* which are essentially references to where the function is stored in memory. As a memory location is essentially a number, it can be passed around in functions and be stored in a variable. Purists would argue that this is not sufficient to call a function a "first-class citizen" in such a language. They may argue that a language must be able to create new functions at runtime for it to be considered a language in which functions are "true" first-class citizens.

In any case, there are several advantages to being able to pass functions around as arguments or store them in variables. Passing a function to another function as an argument gives you the ability to provide a callback. A callback is simply a function that gets passed to another function as an argument. The idea is that the function that receives the callback will execute or "call back" the passed function at some point.

Using callbacks enables us to program a "generic" function that provides some generalized functionality. Then more specific behavior can be be implemented in the callback function. For example, we could create a generic *sort* function that sorts elements in a collection. We could make the sort function generic so that it could sort any type of data: numbers, strings, objects, etc. A callback would provide more specific behavior on how to *order* individual elements in the sorted array.

As another example, consider GUI Programming in which we want to design a user interface. In particular, we may be able to create a button element in our interface. We need to be able to specify what happens when the user clicks the button. This could be achieved by passing in a function as a callback to "register" it with the click "event."

A related issue is anonymous functions (also known as lambda expressions). Typically, we simply want to create a function so that we can pass it as a callback to another function. We may have no intention of actually calling this function directly as it may not be of much use other than passing it as a callback. Some languages allow you to define a function "inline" without an identifier so that it can be passed to another function. Since the function has no name and cannot be invoked by other sections of the code (other than the function we passed it to), it is known as an anonymous function.

5.3.2. Function Overloading

Some languages do not allow you to define more than one function with the same name in the same scope. This is to prevent ambiguity in the code. When you write code to invoke a function and there are several functions with that name, which one are you

actually calling?

Some languages *do* allow you to define multiple functions with the same name as long as they differ in either the number (also called *arity*) or type of parameters. For example, you could define two absolute value, $|x|$ functions with the same name, but one of them takes a floating point number while the other takes an integer as its parameter. This is known as function overloading because you are "overloading" the code by defining multiple functions with the same name.

The ambiguity problem is solved by requiring that each function with the same name differs in their parameters. If you invoke the absolute value function and pass it a floating point number, clearly you meant to call the first version. If you passed it an integer, it is clear that you intended to invoke the second version. Depending on the type and number of arguments you pass to a function, the compiler or interpreter is able to determine *which* version you intend to call and is able to make the right function call. This process is known as static dispatch.

In a language without function overloading, we would be forced to use different names for functions that perform the same operation but on different types.

5.3.3. Variable Argument Functions

Many languages allow you to define special functions that take a *variable number* of parameters. Often they are referred to as "vararg" (short for variable argument) functions. The syntax for doing so varies as does how you can write a function to operate on a variable number of arguments (usually through some array or collection data structure).

The standard `printf` (print formatted) function found in many languages is a good example of a vararg function. The `printf` function allows you to use one function to print any number of variables or values. Without a vararg function, you would have to implement a `printf` version for no arguments, 1 argument, 2 arguments, etc. Even then, you would only be able to support up to n arguments for as many functions as you defined. By using a vararg function, we can write a single function that operates on all of these possibilities. It is import to note, a vararg function is *not* an example of function overloading. There is still only *one* function defined, it just takes a variable number of arguments.

5.3.4. Optional Parameters & Default Values

Suppose that you define a function which has, say, three parameters. Now suppose you invoke the function but only provide it 2 of the 3 arguments that it expects. Some languages would not allow this and it would be considered a syntax or runtime error. Yet

other languages may have very complex rules about what happens when an argument is omitted. Some languages allow you to omit some arguments when calling functions as a feature of the language. That is, the parameters to a function are *optional*.

When a language allows parameters to be optional, it usually also allows you to define *default values* to the parameters if the calling function does not provide them. If a user calls the function without specifying a parameter, it takes on the default value. Alternatively, the default could be a non-value like "null" or "undefined." Inside the function you could implement logic that determined whether or not a parameter was passed to the function and alter the behavior of the function accordingly.

5.4. Exercises

Exercise 5.1. Recall that the *greatest common divisor* (gcd) of two positive integers, a and b is the largest positive integer that divides both a and b. Adapt the solution from Exercise 4.6 into a function. If the language you use supports it, return the gcd via a pass by reference variable.

Exercise 5.2. Write a function that *scales* an input x to to its scientific notation scale so that $1 \leq x < 10$. If you language supports pass by reference, the amount that x is shifted should be stored in a pass-by-reference parameter. For example, a call to this function with $x = 314.15$ should return 3.1415 and the amount it is scaled by is $n = -2$.

Exercise 5.3. Write a function that returns the most significant digit of a floating point number. The function should only return an integer in the range 1–9 (it should return zero only if $x = 0$).

Exercise 5.4. Write a function that, given an integer x, sums the values of its digits. That is, for $x = 29423$ the sum $2 + 9 + 4 + 2 + 3 = 20$.

Exercise 5.5. Write a function to convert radians to degrees using the formula,

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

Write another function to covert degrees to radians.

Exercise 5.6. Write functions to compute the diameter, circumference and area of a circle given its radius. If your language supports pass by reference, compute all three of these with one function.

Exercise 5.7. The *arithmetic-geometric* mean of two numbers x, y, denoted $M(x, y)$ (or $\text{agm}(x, y)$) can be computed iteratively as follows. Initially, $a_1 = \frac{1}{2}(x + y)$ and $g_1 = \sqrt{xy}$ (i.e. the normal arithmetic and geometric means). Then, compute

$$\begin{aligned} a_{n+1} &= \tfrac{1}{2}(a_n + g_n) \\ g_{n+1} &= \sqrt{a_n g_n} \end{aligned}$$

The two sequences will converge to the same number which is the arithmetic-geometric mean of x, y. Obviously we cannot compute an infinite sequence, so we compute until $|a_n - g_n| < \epsilon$ for some small number ϵ.

Exercise 5.8. Write a function to compute the annual percentage yield (APY) given an annual percentage rate (APR) using the formula

$$APY = e^{APR} - 1$$

Exercise 5.9. Write a function that will compute the air distance between two locations given their latitudes and longitudes. Use the formula as in Exercise 2.14.

Exercise 5.10. Write a function to convert a color represented in the RGB (red-green-blue) color model (used in digital monitors) to a CMYK (cyan-magenta-yellow-key) used in printing. RGB values are integers in the range $[0, 255]$ while CMYK are fractional numbers in the range $[0, 1]$. To convert to CMYK, you first need to scale each integer value to the range $[0, 1]$ by simply computing

$$r' = \frac{r}{255}, \quad g' = \frac{g}{255}, \quad b' = \frac{b}{255}$$

and then using the following formulas:

$$\begin{aligned} K &= 1 - \max\{r', g', b'\} \\ C &= (1 - r' - k)/(1 - k) \\ M &= (1 - g' - k)/(1 - k) \\ Y &= (1 - b' - k)/(1 - k) \end{aligned}$$

Exercise 5.11. Write a function to convert from CMYK to RGB using the following formulas.

$$\begin{aligned} r &= 255 \cdot (1 - C) \cdot (1 - K) \\ g &= 255 \cdot (1 - M) \cdot (1 - K) \\ b &= 255 \cdot (1 - Y) \cdot (1 - K) \end{aligned}$$

Exercise 5.12. Write some functions to convert an RGB color to a gray scale, "removing" the color values. An RGB color value is grayscale if all three components have the same value. To transform a color value to grayscale, there there are several possible techniques. The average method simply sets all three values to the average:

$$\frac{r + g + b}{3}$$

The lightness method averages the most prominent and least prominent colors:

$$\frac{\max\{r, g, b\} + \min\{r, g, b\}}{2}$$

The luminosity technique uses a weighted average to account for a human perceptual preference toward green:

$$0.21r + 0.72g + 0.07b$$

Exercise 5.13. Adapt the methods to compute a square root in Exercise 4.21 into functions.

Exercise 5.14. Adapt the methods to compute the natural logarithm in Exercise 4.22 into functions.

Exercise 5.15. Weight (mass in the presence of gravity) can be measured in several scales: kilogram force (kgf), pounds (lbs), ounces (oz), or Newtons (N). To convert between these scales, you can use the following facts:

- 1 kgf is equal to 2.20462 pounds
- There are 16 ounces in a pound
- 1 kgf is equal to 9.80665 Newtons

Write a collection of functions to convert between these scales.

Exercise 5.16. Length can be measured by several different units. We will concern ourselves with the following scales: kilometer, mile, nautical mile, and furlong. A measure in each one of these scales can be converted to another using the following facts.

- One mile is equivalent to 1.609347219 kilometers
- One nautical mile is equivalent to 1.15078 miles
- A furlong is $\frac{1}{8}$-th of a mile

Write a collection of functions to convert between these scales.

Exercise 5.17. Temperature can be measured in several scales: Celsius, Kelvin, Fahrenheit, and Newton. To convert between these scales, you can use the following conversion table.

From/To	Celsius	Kelvin	Fahrenheit	Newton
Celsius	–	$c + 273.15$	$c\frac{9}{5} + 32$	$c\frac{33}{100}$
Kelvin	$k - 273.15$	–	$\frac{9}{5}k - 459.67$	$.33k - 90.1395$
Fahrenheit	$(f - 32)\frac{5}{9}$	$\frac{5}{9}f + 255.372$	–	$\frac{11}{60}f - \frac{88}{15}$
Newton	$n\frac{100}{33}$	$\frac{100}{33}n + 273.15$	$\frac{60}{11}n + 32$	–

Table 5.1.: Conversion Chart

Write a collection of functions to convert between these scales.

Exercise 5.18. Energy can be measured in several different scales: calories (c), joules (J), ergs (erg) and foot-pound force (ft-lbf) among others. To convert between these scales, you can use the following facts:

- 1 erg equals $1.0 \times 10^{-7} J$
- 1 ft-lbs equals 1.3558 joules
- 1 calorie is equal to 4.184 joules

Write a collection of functions to convert between these scales.

Exercise 5.19. Pressure is a measure of force applied to the surface of an object per unit area. There are several units that can be used to measure pressure:

- Pascal (Pa) which is one Newton per square meter
- Pound-force Per Square Inch (psi)
- Atmosphere (atm) or standard atmospheric pressure
- The torr, an absolute scale for pressure

To convert between these units, you can use the following formulas.

- 1 psi is equal to 6,894.75729 Pascals, 1 psi is equal to 0.06804596 atmospheres
- 1 atmosphere is equal to 101,325 Pascals
- 1 torr is equal to $\frac{1}{760}$ atmosphere and $\frac{101{,}325}{760}$ Pascals

Write a collection of functions to convert between these scales.

6. Error Handling

Writing perfect code is difficult. The more complex a system or code base, the more likely it is to have bugs. That is, flaws or mistakes in a program that result in incorrect behavior or unintended consequences. The term "bug" has been used in engineering for quite a while. The term was popularized in the context of computer systems by Grace Hopper who, when working on the Naval Mark II computer in 1946, tracked a malfunction to a literal bug, a moth, trapped in a relay [2].

Some of the biggest modern engineering failures can be tracked to simple software bugs. For example, on September 26th, 1983 a newly installed Soviet early warning system gave indication that nuclear missiles had been launched on the Soviet Union by the United States. Stanislav Petrov, a lieutenant colonel in the Soviet Air Defense Forces and duty officer at the time, did not trust the new system and did not report the incident to superiors who may have ordered a counter strike. Petrov was correct as the false alarm was caused by sunlight reflections off of high altitude clouds as well as other bugs in the newly deployed system [28].

In September 1999 the Mars Climate Orbiter, a project intended to study the Martian climate and atmosphere was lost after it entered into the upper atmosphere of Mars and disintegrated. The error was due to a subsystem that measured the craft's momentum in a non-standard pound force per second unit when all other systems expected the standard newton second unit [1]. The loss of the project was calculated at over $125 million.

There are numerous other examples, some that have caused inconvenience to users (such as the Zune bug mentioned in Section 4.5.2) to bugs in medical devices that have cost dozens of lives to those resulting in the loss of millions of dollars [6].

In some sense, Software Engineering and programming is unique. If you build a bridge and forget one bolt its likely not going to cause the bridge to collapse. If you draw up plans for a development and the land survey is a few inches off overall, its not a catastrophic problem. However, if you forget one character or are off by one number in a program, it can cause a complete system failure.

There are a variety of reasons for why bugs make it into systems. Bugs could be the result of a fundamental misunderstanding of the problem or requirements. Poor management and the pressure of time constraints to deliver a project may make developers more careless. A lack of proper testing may mean many more bugs survive the development

process than otherwise should have. Even expert programmers can overlook a simple mistake when writing thousands of lines of code.

Given the potential for error, it is important to have good software development methodologies that emphasize testing a system at all levels. Working in teams where regular code reviews are held so that colleagues can examine, critique, and catch potential bugs are essential for writing robust code.

Modern coding tools and techniques can also help to improve the robustness of code. For example, debuggers are tools that help a developer debug (that is, find and fix the cause of an error) a program. Debuggers allow you to simulate the execution of a program statement-by-statement and view the current state of the program such as variable values. You can "step through" the execution line by line to find where an error occurs in order to localize and identify a bug.

Other tools allow you to perform static analysis on source code to search for *potential* problems. That is, problems that are not syntax errors and are not necessarily bugs that are causing problems, but instead are anti-patterns or code smells. Anti-patterns are essentially common bad-habits that can be found in code. They are an *attempted* solution to a commonly encountered problem but which don't actually solve the problem or introduces new problems. Code smells are "symptoms" in a source code that indicate a possible deeper design or implementation flaw. Failure to adhere to good programming principles such as properly initializing variables or failure to check for null values are examples of smells. Static analysis tools automatically examine the code base for potential issues like these. For example, a lint (or linter) is a tool that can examine source code suspicious or non-portable code or code that does not comply with generally accepted standards or ways of doing things.

Even if code contains no bugs, it is still susceptible to errors. For example, a program could connect to a remote database to query and process data. However, if the network connection is temporarily unavailable, the program will not be able to execute properly. Because of the potential of such errors, it is important to write code that is not only bug free but is also robust and resilient. We must anticipate possible error conditions and write code to detect, prevent, or recover from such errors. Generally, this is referred to as *error handling*.

Much of what we now consider Software Engineering was pioneered by people like Margaret Hamilton who was the lead Apollo flight software designer at NASA. During the Apollo 11 Moon landing (1969), an error in one system caused the lander's computer to become overworked with data. However, because the system was designed with a robust architecture, it could detect and handle such situations by prioritizing more important tasks (those related to landing) over lower priority tasks. The resilience that was built into the system is credited with its success [11].

6.1. Error Handling

In general, errors are potential conditions or situations that can reasonably be anticipated by a developer. For example, if we write code to open and process a file, there are several things that could go wrong. The file may not exist, or we may not have permission on the system to read it, or the formatting in the file may be corrupted or not as expected. Still yet, everything could be fine with the file, but it may contain erroneous or invalid values.

If an error can be anticipated, we may be able to write code that detects the particular error condition and *handles* it by executing code that may be able to recover from the error condition. In the case of a missing file for example, we may be able to prompt the user for an alternate file.

We may be able to detect but not necessarily recover from certain errors. For example, if the file has been corrupted in example above, there may not be a way to properly "fix" it. If it contains invalid data, we may not even want the program to fix it as it may indicate a bug or other issue that needs to be addressed. Still yet, there may be some error conditions that we cannot recover from at all as they are completely unexpected. In such instances, we may want the error to result in the termination of the program in which case the error is considered *fatal*.

6.2. Error Handling Strategies

There are several general strategies for performing error handling. We'll look at two general methods here: defensive programming and exceptions.

6.2.1. Defensive Programming

Defensive programming is a "look before you leap" strategy. Suppose we have a potentially "dangerous" section of code; that is, a line or block of code whose execution could encounter or result in an error condition. Before we execute the code, we perform a check to see if the error condition is present (usually using a conditional statement). If the error condition does not hold, then we proceed with the code as normal. However, if the error condition does hold, instead of executing the code, we execute alternative code that *handles* the error.

Suppose we are about to divide by a number. To prevent a division by zero error, we can check if our denominator is zero or not. If it is, then we raise or handle the error instead of performing the division. What should be done in such a case? We could, as an alternative, use a predefined value as a result instead. Or we could notify the user and

ask for an alternative. Or we could log the error and proceed as normal. Or we could decide that the error is so egregious that it should be fatal and terminate the execution of the program.

Which is the right way to handle this error? It really depends on your design requirements really. This raises the question, though: "who" is responsible for making these decisions? Suppose we're designing a function for a library that is not just for our project but others as well (as is the case with the standard library functions). Further, the function we're designing could have multiple different error conditions that it checks for. In this scenario there are two entities that could handle the errors: the function itself and the code that invokes the function.

Suppose that we decide to handle the errors *inside* the function. As designers of the function, we've made the decision to handle the errors *for* the user (the code that invokes our function). Regardless of how we decide to handle the errors, this design decision has essentially taken any decision making ability away from users. This is not very flexible for someone using our code. If they have different design considerations or requirements, they may need or want to handle the errors in a different way than we did.

Now suppose that we decide *not* to handle the errors inside our function. Defensive programming may still be used to prevent the execution of code that results in an error. However, we now need a way to *communicate* the error condition to the calling function so that it can know what type of error happened and handle it appropriately.

Error Codes

One common pattern to communicate errors to a calling function is to use the return type as an *error code*. Usually this is an integer type. By convention 0 is used to indicate "no error" and various other non-zero values are used to indicate various types of errors. Depending on the system and standard used, error codes may have a predefined value or may be specific to an application or library.

One problem with using the return type to indicate errors is that functions are no longer able to use the return type to return an actual computed value. If a language supports pass by reference, then this is not generally a problem. However, even with such languages there are situations where the return type *must* be used to return a value. In such cases, the function can still communicate a general error message by returning some flag value such as null.

Alternatively, a language may support error codes by using a shared global variable that can be set by a function to indicate an error. The calling function can then examine the variable to see if an error occurred during the invocation of the function.

Limitations

Defensive programming has its limitations. Let's return to the example of processing a file. To check for all four of the error conditions we identified, we would need a series of checks similar to the following.

```
1  IF file does not exists THEN
2  |  return an error code
3  END
4  IF we do not have permissions THEN
5  |  return an error code
6  END
7  IF the file is corrupted THEN
8  |  return an error code
9  END
10 IF the file contains invalid values THEN
11 |  return an error code
12 END
13 process file data
```

A problem arises when an error condition is checked and does not hold. Then, later in the execution, circumstances change and the error condition does hold. However, since it was already checked for, the program remains under the assumption that the error condition does not hold. For example, suppose that another process or program deletes the file that we wish to process after its existence has been checked but before we start processing it. Because of the sequential nature of our program, this type of error checking is susceptible to these issues.

6.2.2. Exceptions

An exception is an event or occurrence of an anomalous, erroneous or "exceptional" condition that requires special handling. Exceptions interrupt the normal flow of control in a program by handing the flow of control over to *exception handlers*.

Languages usually support exception handling using a try-catch control structure such as the following.

```
try {
  //potentially dangerous code here
```

```
} catch(Exception e) {
  //exception handling code here
}
```

The `try` is used to encapsulate potentially dangerous code, or simply code that would fail if an error condition occurs. If an error occurs at some point within the `try` block, control flow is immediately transferred to the `catch` block. The `catch` block is where you specify *how* to handle the exception. If the code in the `try` block does not result in an exception, then control flow will skip over the `catch` statement and resume normally after.

It is important to understand how exceptions interrupt the normal control flow. For example, consider the following pseudocode.

```
try {
  statement1;
  statement2;
  statement3;
} catch(Exception e) {
  //exception handling code here
}
```

Suppose `statement1` executes with no error but that when `statement2` executes, it results an exception. Control flow is then transferred to the `catch` block, skipping `statement3` entirely. In general, there may not be a mechanism for your `catch` block to recover and execute `statement3`. Therefore, it may be necessary to make your `try-catch` blocks fine-grained, perhaps having only a single statement within the `try` statement.

Some languages only support a generic `Exception` and the type of error may need to be communicated through other means such as a string error message. Still other languages may support many different types of exceptions and you may be able to provide *multiple* `catch` statements to handle each one differently. In such languages, the order in which you place your `catch` statements may be important. Similar to an if-else-if statement, the first one that matches the exception will be the one that executes. Thus, it is best practice to order your `catch` blocks from the most specific to the most general.

Some languages also support a third `finally` control statement as in the following example.

```
try {
  //potentially dangerous code here
} catch(Exception e) {
  //exception handling code here
} finally {
  //unconditionally executed code here
}
```

The `try-catch` block operates as previously described. However, the `finally` block will execute *regardless* of whether or not an exception was raised. If no exception was raised, then the `try` block will fully execute and the `finally` block will execute immediately after. If an exception was raised, control flow will be transferred to the `catch` block. After the `catch` block has executed, the `finally` block will execute.

`finally` blocks are generally used to handle resources that need to be "cleaned up" whether or not an exception occurs. For example, opening a connection to a database to retrieve and process data. Whether or not an exception occurs during this process the connection will need to be properly closed as it represents a substantial amount of resources (a network connection, memory and processing time on both the server and client machines, etc.). Failure to properly close the connection may result in wasted resources. By placing the clean up code inside a `finally` statement, we can be assured that it will execute regardless of an error or exception.

In addition to handling exceptions, a language may allow you to "throw" your own exceptions by using the keyword `throw`. In this way you can also practice defensive programming. You could write a conditional statement to check for an error condition and then `throw` an appropriate exception.

6.3. Exercises

Exercise 6.1. Rewrite the function to compute the GCD in Exercise 5.1 to handle invalid inputs.

Exercise 6.2. Rewrite the function to compute statistics of a circle in Exercise 5.6 to handle invalid input (negative radius).

Exercise 6.3. Rewrite the function to compute the annual percentage yield in Exercise 5.8 to handle invalid input.

Exercise 6.4. Rewrite the function to compute air distance in Exercise 5.9 to handle invalid input (latitude/longitude values outside the range $[-180, 180]$).

Exercise 6.5. Rewrite the function to convert from RGB to CMYK in Exercise 5.10 to handle invalid inputs (values outside the range $[0, 255]$).

Exercise 6.6. Rewrite the function to convert from CMYK to RGB in Exercise 5.11 to handle invalid inputs.

Exercise 6.7. Rewrite the square root functions from Exercise 5.13 to handle invalid inputs.

Exercise 6.8. Rewrite the natural logarithm functions from Exercise 5.14 to handle invalid inputs.

Exercise 6.9. Rewrite the weight conversion functions from Exercise 5.15 to handle invalid inputs.

Exercise 6.10. Rewrite the length conversion functions from Exercise 5.16 to handle invalid inputs.

Exercise 6.11. Rewrite the temperature conversion functions from Exercise 5.17 to handle invalid inputs.

Exercise 6.12. Rewrite the energy conversion functions from Exercise 5.18 to handle invalid inputs.

Exercise 6.13. Rewrite the pressure conversion functions from Exercise 5.19 to handle invalid inputs.

7. Arrays, Collections & Dynamic Memory

Rarely do we ever deal with a single piece of data in a program. Instead, most data is made up of a collection of similar elements. A program to compute grades would be designed to operate on an entire roster of students. Scientific data represents a collection of many different samples or measurements. A bank maintains and processes many different accounts, etc.

Arrays are a way to collect similar pieces of data together in an *ordered collection*. The pieces of data stored in an array are referred to as *elements* and are stored in an ordered manner. That is, there is a "first" element, "second" element, etc. and a "last" element. Though the elements are ordered, they are not necessarily *sorted* in any particular manner. Instead, the order is usually determined by the order in which you place elements into the array.

Some languages only allow you to store the same types of elements in an array. For example, an integer array would only be able to store integers, an array of floating-point numbers would only be able to store floating-point numbers. Other languages allow you to create arrays that can hold *mixed elements*. A mixed array would be able to hold elements of any type, so it could hold integers, floating-point numbers, strings, objects, or even other arrays.

Some languages treat arrays as full-fledged object types that not only hold elements, but have methods that can be called to manipulate or transform the contents of the array. Other languages treat arrays as a primitive type, simply using arrays as storage data structures.

Languages can vary greatly in how they implement and represent arrays of elements, but many have the same basic usage patterns allowing you to create arrays and manipulate their contents.

index	0	1	2	3	4	5	6	7	8	9
contents	2	3	5	7	11	13	17	19	23	29

Figure 7.1.: An integer array of size 10. Using zero-indexing, the first element is at index 0, the last at index 9.

7.1. Basic Usage

Creating Arrays

Though there can be great variation in how a language uses arrays, there are some commonalities. Languages may allow you to create *static* arrays or *dynamically allocated arrays* (see Section 7.2 below for a detailed discussion). Static arrays are generally created using the program stack space while dynamically allocated arrays are stored in the *heap*. In either case you generally declare an array by specifying its size. In statically typed languages, you must also declare the array's type (integer, floating-point, etc.).

Indexing Arrays

Once an array has been created you can use it by assigning values to it or by retrieving values from it. Because there is more than one element, you must specify *which* element you are assigning or retrieving. This is generally done through *indexing*. An *index* is an integer that specifies an element in the array. The index is used in conjunction with (usually) square brackets and the array's identifier. For example, if the array's identifier is `arr` and the index is an integer value stored in the variable `i`, we would refer to the i-th element using the syntax `arr[i]`. An example is presented in Figure 7.1.

For most programming languages, indices start at zero. This is known as zero-indexing.[1] Thus, the first element is at `arr[0]`, the second at `arr[1]` etc. When an array is stored in memory, each element is usually stored one after the other in one contiguous space in memory. Further, each element is of a specific type which is represented using a fixed number of bytes in memory. Thus the index actually acts as an *offset* in memory from the beginning of the array. For example, if we have an array of integers which each take 4 bytes each, then the 5th element would be stored at index 4, which is an an offset equal to $4 \times 4 = 16$ bytes away from the beginning of the array. The first element, being at index 0 is $4 \times 0 = 0$ bytes from the beginning of the array (that is, the first element is at the beginning of the array).

Once an element has been indexed, it can essentially be treated as a regular variable,

[1] Some languages do use 1-indexing but there are very strong arguments in favor of zero-indexing [13].

assigning and retrieving values as you would regular variables. Care must be taken so that you do not make a reference to an element that does not exist. For example, using a negative index or an index $i \geq n$ in an array of n elements. Depending on the language, indexing an array element that is out-of-bounds may result in undefined behavior, an exception being thrown, or a corruption of memory.

Iteration

Since we can simply index elements in an array, it is natural to iterate over elements in an array using a for loop. We can create a for loop using an index variable i which starts at 0, and increments by one on each iteration, accessing the i-th element using the syntax described above, `arr[i]`. One issue is that such a loop would need to know how large the array is in order to define a terminating condition.

```
1 n ← size of array arr
2 FOR (i ← 0; i < (n − 1); i ← (i + 1)) DO
3 |   process element arr[i]
4 END
```

Some languages build the size of the array into a property that can be accessed. Java, for example, has a `arr.length` property. Other languages provide functions that you can use to determine their size. Still other languages (such as C), place the burden of "bookkeeping" the size of an array on you, the programmer. Whenever you pass an array to a function you need to also pass a size parameter that informs the function of how many elements are in the array. Yet other functions may also require you to pass the size of each element in the array.

Some languages also support a basic *foreach* loop (cf. Section 4.4). A foreach loop is syntactic sugar that allows you to iterate over the elements in an array (usually in order) without the need for boilerplate code that creates and increments an index variable.

```
1 FOREACH element a in arr DO
2 |   process element a
3 END
```

The actual syntax may vary if a language supports such a loop.

Using Arrays in Functions

Most programming languages allow you to use arrays as both function parameters and as return types. You can pass arrays to functions and functions can be defined that return arrays. Typically, when arrays are passed to functions, they are passed by reference so as to avoid making an entirely new copy of the array. As a consequence, if the function makes changes to the array elements, those changes may be realized in the calling function. Sometimes you may want this behavior. However, sometimes you may not want the function to make changes to the passed array. Some languages allow you to use various keywords to *prevent* any changes to the passed array.

If a function is designed to return an array, care must be taken to ensure that the correct type of array is returned. Recall that static arrays are allocated on the call stack. It would be inappropriate to return static arrays from a function as the array is part of the stack frame that is destroyed when the function returns control back to the calling function (we discuss this in detail below). Instead, if a function returns an array, it should be an array that has been dynamically allocated (on the heap).

7.2. Static & Dynamic Memory

Recall that arrays can be declared as *static* arrays, meaning that when you declare them, they are allocated and stored on the program's call stack within the stack frame in which they are declared. For example, if a function `foo()` creates a static array of 5 integers, each requiring 4 bytes, then 20 contiguous bytes are allocated on the stack frame for `foo()` to store the static array.

This can cause several potential problems. First, the typical stack space allocated for a program is relatively small. It can be as large as a couple of Megabytes (MBs) or on some older systems or those with limited resources, even on the order of a few hundred Kilobytes (KBs). When dealing with data of any substantial size, you could quickly run out of stack space, resulting in a *stack overflow*.

Another problem arises if we want to return a static array from a function. Recall that when a function returns control to the calling function, its stack frame is popped off the top and goes out-of-scope (see Section 5.2). Since the array is part of the stack frame of the function, it too goes out of scope. Depending on how the system works, the contents of the frame may be completely changed in the "bookkeeping" process of returning from the function. Even if the contents remain unchanged when the function returns, they will almost certainly be overwritten when another function call is made and a new stack frame overwrites the old one. For these reasons, static arrays are of very limited use. They must be kept small and cannot be returned from a function.

Demonstration in C

To make this concept a bit more clear, we'll use a concrete example in the C programming language. Consider the program code in Figure 7.2. Here, we have a function `foo()` that creates a static integer array of size 5, `int b[5];`. This memory is allocated on the stack frame and then assigned values. Printing them in the function will give the expected results,

```
b[0] = 5
b[1] = 10
b[2] = 15
b[3] = 20
b[4] = 25
```

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

int * foo(int n) {
  int i;
  int b[5];
  for(i=0; i<5; i++) {
    b[i] = n*i;
    printf("b[%d] = %d\n", i, b[i]);
  }
  return b;
}

int main(int argc, char **argv) {
  int i, m = 5;
  int *a = foo(m);
  for(i=0; i<5; i++) {
    printf("a[%d] = %d\n", i, a[i]);
  }
  return 0;
}
```

Figure 7.2.: Example returning a static array

However, when the function `foo()` ends execution and returns control back to the `main()` function, (sometimes called unwinding), the contents of `foo()`'s stack frame are altered as part of the process. Some of the contents are the same, but elements have been completely altered. Printing the "returned" contents of the array gives us garbage:

```
a[0] = 1564158624
a[1] = 32767
a[2] = 15
a[3] = 20
a[4] = -626679356
```

This is not an issue when returning primitive types as the return value is placed in a special memory location available to the calling function. Even in our example, the return value is properly communicated to the calling function: its just that the returned value is a *pointer* to the array's location (which happens to be a memory address in the "stale" stack frame). The stack frames are depicted in Figure 7.3.

7.2.1. Dynamic Memory

The solution to the problems presented by static arrays is to use *dynamically allocated arrays*. Such arrays are not allocated and stored in the program call stack, but instead are stored in another area called the *heap*. In fact, because of the shortcomings of static arrays, some languages only allow you to use dynamically allocated arrays even if it is done so implicitly.

Recall that when a program is loaded into memory, its code is placed on the program call stack in memory. On top of that is any static data (global variables or data for example). Subsequent function calls place new stack frames on top of that. In addition, at the "other end" of the program's memory space, the *heap* is allocated. The heap is a "pile" of memory like the stack is, but it is less structured. The stack is one contiguous piece of memory, but the heap may have "holes" in it as various chunks of it are *allocated* and *deallocated* for the program. In general, the heap is larger but allocation and deallocation is a more involved and more costly process while the stack is smaller and faster to allocate/deallocate stack frames from.

Initially, a program is allocated a certain amount of memory for its heap. When a program attempts to dynamically allocate memory, say for a new array of integers, it makes a request to the operating system for a certain amount of memory (a certain number of bytes). The system responds by allocating a chunk of the heap memory which the program can then use to store elements in the array. Usually, the memory space is

stored as a pointer or reference. The reference is stored in a variable in a stack frame, but the actual contents of the array are stored in the heap space.

Depending on the language and system, if a program uses all of its heap space and runs out, the operating system may terminate the program or it may attempt to reserve even more memory for the program.

Memory Management

If a program no longer needs a dynamically allocated memory space, it should "clean up" after itself by deallocating or "freeing" the memory, releasing it back to the heap space so that it can be reused either by the program or some other program or process on the system. The process of allocating and deallocating memory is generally referred to as *memory management*. If a program does not free up memory, it may eventually run out and be forced to terminate. Even if it does not necessarily run out of available memory, its performance may degrade or it may impact system performance.

If a program has poor memory management and fails to deallocate memory when it is no longer needed, the memory "leaks": the available memory gradually runs out because it is not released back to the heap for reallocation. Programs which such poor memory management are said to have a memory leak. Sometimes this is a consequence of a dangling pointer: when a program dynamically allocates a chunk of memory but then due to carelessness, loses the reference to the memory chunk, making it impossible to free up.

Some languages have automatic garbage collection that handle memory management for us. The language itself is able to monitor the dynamically allocated pieces of memory and determine if any variable in the program still references it. If the memory is no longer referenced, it is "garbage" and becomes eligible to be "collected." The system itself then frees the memory and makes it available to the program or operating system. In such "memory managed" languages, we are responsible for allocating memory, but are not (necessarily) responsible for deallocating it.

Even if a language offers automated memory management, it is still possible to have memory leaks and other memory allocation issues. Automated memory management does not solve all of our memory management problems. Moreover, it comes at a cost. The additional resources and overhead required to monitor memory can have a significant performance cost. However, with modern garbage collection systems and algorithms, the performance gap between garbage collected languages and user-managed memory languages has been shrinking. In any case, all program memory is reclaimed by the operating system when the program terminates.

7.2.2. Shallow vs. Deep Copies

In most languages, an array variable is actually a *reference* to the array in memory. We could create an array referred to by a variable A and then create another reference variable B and set it "equal" to A. However, this is simply a shallow copy. Both the reference variables refer to the same data in memory. Consequently, if we change the value of an element in one, the change is realized in both.

Often, we want a completely different copy, referred to as a deep copy. With a deep copy, A and B would refer to different memory blocks. Changes to one would not affect the other. This distinction is illustrated in Figure 7.5.

7.3. Multidimensional Arrays

A normal array is usually one dimensional. One can think an array as a single "row" in a table that contains a certain number of entries. Most programming languages allow you to define *multidimensional* arrays. For example, two dimensional arrays would model having multiple rows in a full table. You can also view two dimensional arrays as matrices in mathematics. A *matrix* is a rectangular array of numbers that have a certain number of *rows* and a certain number of *columns*.

As an example, consider the following 2×3 matrix (it has 2 rows and 3 columns):

$$\begin{bmatrix} 1 & 9 & -8 \\ 2.5 & 3 & 5 \end{bmatrix}$$

In mathematics, entries in a matrix are indexed via their row and column. For example, $a_{i,j}$ would refer to the element in the i-th row and j-th column. Referring to the row first and column second is referred to as *row major* ordering. If the number of rows and the number of columns are the same, the matrix is referred to as a *square matrix*. For example, the following is a square, 10×10 matrix.

$$\begin{bmatrix} 2 & 68 & 9 & 44 & 80 & 79 & 77 & 59 & 27 & 2 \\ 3 & 86 & 22 & 42 & 58 & 24 & 45 & 39 & 7 & 47 \\ 5 & 7 & 17 & 12 & 29 & 56 & 68 & 14 & 65 & 3 \\ 7 & 35 & 64 & 69 & 79 & 56 & 52 & 77 & 82 & 85 \\ 11 & 55 & 36 & 5 & 25 & 6 & 22 & 25 & 72 & 37 \\ 13 & 20 & 37 & 74 & 3 & 53 & 87 & 70 & 3 & 78 \\ 17 & 72 & 68 & 26 & 11 & 6 & 63 & 70 & 29 & 16 \\ 19 & 59 & 6 & 26 & 87 & 18 & 82 & 27 & 75 & 19 \\ 23 & 73 & 30 & 80 & 51 & 14 & 34 & 67 & 59 & 58 \\ 29 & 48 & 2 & 39 & 18 & 21 & 33 & 28 & 40 & 34 \end{bmatrix}$$

We can do something similar in most programming languages. First, languages may vary in how you can create multidimensional arrays, but you usually have to provide a size for each dimension when you create them. Once created, you can index them by providing multiple indices. For example, with a two dimensional array, we could provide two indices each in their own square brackets `arr[i][j]` referring to the i-th row and j-th column. Multidimensional arrays usually use the same 0-indexing scheme as single dimensional arrays.

You can further generalize this and create 3-dimensional arrays, 4-dimensional arrays, etc. However, the use cases for arrays of dimension 3 and certainly for arrays of dimension 4 or greater are rare. If you need to store such data, it may be more appropriate to define a custom, user-defined structure or object (see Chapter 10) instead of a higher dimensional array.

We usually think of 2-dimensional arrays as having the same number of elements in each "row". In the example above, both of the matrix's rows had 3 elements in it. Some languages allow you to create 2-dimensional arrays with a different number of elements in each row. Special care must be taken to ensure that you do not index an element that does not exist.

7.4. Other Collections

Aside from basic arrays, many languages have rich libraries of other *dynamic* collections. Dynamic collections are not the same thing as dynamically allocated arrays. Once an array is created, its size is fixed and cannot, in general, be changed. However, dynamic collections can grow (and shrink) as needed when you add or remove elements from them.

Lists are ordered collections that are essentially dynamic arrays. Lists are ordered and are usually zero-indexed just like arrays. Lists are generally objects and provide methods that can be used to add, remove, and retrieve elements from the list. If you add an element to a list, the list will automatically grow to accommodate it, so its size is not fixed when created. Two common implementations of lists are array-based lists and linked lists. Array-based lists indexarray-based list use an array to store elements. When the array fills up, the list allocates a new, larger array to hold more elements, copying the original contents over to the new array with a larger capacity. Linked lists hold elements in *nodes* that are linked together. Adding a new element simply involves creating a new node and linking it to the last element in the list.

Some languages also define what are called *sets*. Sets allow you to store elements dynamically just like lists, but sets are generally *unordered*. There is no concept of a first, second, or last element in a set. Iterating over the elements in a set could result in a different enumeration of the elements each time. Elements in sets are also usually unique. For example, a set containing integers would only ever contain one instance of

each integer. The value 10, for example, would only ever appear once. If you added 10 to a set that already contained it, the operation would have no effect on the set.

Another type of dynamic array are *associative arrays* (sometimes called *dictionaries*). An associative array holds elements, but may not be restricted in how they are indexed. In particular, a language that supports associative arrays may allow you to use integers or strings as indices, or even any arbitrary object. Further, when using integers to index elements, indices need not be fully defined nor contiguous. In an associative array you could define an element at index 5 and then place the next element at index 10, skipping index 6 through 9 which would remain undefined.

One way to look at associative arrays is as a *map*. A map is a data structure that stores elements as key-value pairs. Both they keys and values could be any arbitrary type (integers or strings) or object depending on the language. You could map account numbers (stored as strings) to account objects, or vice versa. Using a smart data structure like a map can make data manipulation a lot easier and more straightforward.

7.5. Exercises

Exercise 7.1. Write a function to return the *index* of the maximum element in an array of numbers.

Exercise 7.2. Write a function to return the *index* of the minimum element in an array of numbers.

Exercise 7.3. Write a function to compute the mean (average) of an array of numbers.

Exercise 7.4. Write a function to compute the standard deviation of an array of numbers,

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

where μ is the mean of the array of numbers.

Exercise 7.5. Write a function that takes two arrays of numbers that are *sorted* and *merges* them into one array (retuning a new array as a result).

Exercise 7.6. Write a function that takes an integer n and produces a new array of size n filled with 1s.

Exercise 7.7. Write a function that takes an array of numbers are computes returns the *median* element. The median is defined as follows:

- If n is odd, the median is the $\frac{n+1}{2}$-th largest element

- If n is even, the median is the average of the $\frac{n}{2}$ and the $(\frac{n}{2}+1)$-th largest elements

Exercise 7.8. The dot product of two arrays (or vectors) of the same dimension is defined as the sum of the product of each of their entries. That is,

$$\sum_{i=1}^{n} a_i \times b_i$$

Write a function to compute the dot product of two arrays (you may assume that they are of the same dimension)

Exercise 7.9. The *norm* of an n-dimensional vector, $\vec{x} = (x_1, x_2, \ldots, x_n)$ captures the notion of "distance" in a higher dimensional space and is defined as

$$\|\vec{x}\| = \sqrt{x_1^2 + \cdots + x_n^2}$$

Write a function that takes an array of numbers that represents an n-dimensional vector and computes its norm.

Exercise 7.10. Write a function that takes two arrays A, B and creates and returns a new array that is the *concatenation* of the two. That is, the new array will contain all elements a followed by all elements in b.

Exercise 7.11. Write a function that takes an array of numbers A and an element x and returns true/false if A contains x

Exercise 7.12. Write a function that takes an array of numbers A, an element x and two indices i, j and returns true/false if A contains x somewhere between index i and j.

Exercise 7.13. Write a function that takes an array of numbers A and an element x and returns the *multiplicity* of x; that is the number of times x appears in A.

Exercise 7.14. Write a function to compute a *sliding window* mean. That is, it computes the average of the first m numbers in the array. The next value is the average of the values index from 1 to m, then 2 to $m+1$, etc. The last window is the average of the last m elements. Obviously, $m \leq n$ (for $m = n$, this is the usual mean). Since there is more than one value, your function will return a (new) array of means of size $n - m + 1$.

Exercise 7.15. Write a function to compute the *mode* of an array of numbers. The mode is the most *common* value that appears in the array. For example, if the array contained the elements $2, 9, 3, 4, 2, 1, 8, 9, 2$, the mode would be 2 as it appears more than any other element. The mode may not be unique; multiple elements could appear the same, maximal number of times. Your function should simply return *a* mode.

Exercise 7.16. Write a function to find *all* modes of an array. That is, it should find all modes in an array and return a new array containing all the mode values.

Exercise 7.17. Write a function to filter out certain elements from an array. Specifically, the function will create a new array containing only elements that are greater than or equal to a certain threshold δ.

Exercise 7.18. Write a function that takes an array of numbers and creates a new "deep" copy of the array. In addition, the function should take a new "size" parameter which will be the size of the copy. If the new size is less than the original, then the new array will be a *truncated* copy. If the new size is greater then the copy will be *padded* with zero values at the end.

Exercise 7.19. Write a function that takes an array A and two indices i, j and returns a new array that is a subarray of A consisting of elements i through j.

Exercise 7.20. Write a function that takes two arrays A, B and creates and returns a new array that represents the *unique intersection* of A and B. That is, an array that contains elements that are in both A *and* B. However, elements should not be included more than once.

Exercise 7.21. Write a function that takes two arrays A, B and creates and returns a new array that represents the *unique union* of A and B. That is, an array that contains elements that are *either* in A *or* B (or both). However, elements should not be included more than once.

Exercise 7.22. Write a function that takes an array of numbers and returns the sum of its elements.

Exercise 7.23. Write a function that takes an array of numbers and two indices i, j and computes the sum of its elements between i and j inclusive.

Exercise 7.24. Write a function that takes *two* arrays of numbers and determines if they are *equal*: that is, each index contains the same element.

Exercise 7.25. Write a function that takes *two* arrays of numbers and determines if they both contain the same elements (though are not necessarily equal) regardless of their multiplicity. That is, the function should return true even if the arrays' elements appear a different number of times or in a different order. For example $[2, 2, 3]$ would be equal to an array containing $[3, 2, 3, 3, 3, 2, 2, 2]$.

Exercise 7.26. A *suffix array* is a lexicographically sorted array of all suffixes of a string. Suffix arrays have many applications in text indexing, data compression algorithms and in bioinformatics. Write a program that takes a string and produces its suffix array.

For example, the suffixes of `science` are `science`, `cience`, `ience`, `ence`, `nce`, `ce`, and `e`. The suffix array (sorted) would look something like the following.

```
ce
cience
e
ence
ience
nce
science
```

Exercise 7.27. An array of size n represents a *permutation* if it contains all integers $0, 1, 2, \ldots, (n-1)$ exactly once. Write a function to determine if an array is a permutation or not.

Exercise 7.28. The k-th order statistic of an array is the k-th largest element. For our purposes, k starts at 0, thus the minimum element is the 0-th order statistic and the largest element is the $n-1$-th order statistic.

Another way to view it is: suppose we were to *sort* the array, then the k-th order statistic would be the element at index k in the sorted array. Write a function to find the k-th order statistic.

Exercise 7.29. Write a function that takes two $n \times n$ square matrices A, B and returns a new $n \times n$ matrix C which is the product, $A \times B$. The product of two matrices of dimension $n \times n$ is defined as follows:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Where $1 \leq i, j \leq n$ and c_{ij} is the (i, j)-th entry of the matrix C.

Exercise 7.30. We can multiply a matrix by a single scalar value x by simply multiplying each entry in the matrix by x. Write a function that takes a matrix of numbers and an element x and performs scalar multiplication.

Exercise 7.31. Write a function that takes two matrices and determines if they are equal (all of their elements are the same).

Exercise 7.32. Write a function that takes a matrix and an index i and returns a new *array* that contains the elements in the i-th row of the matrix.

Exercise 7.33. Write a function that takes a matrix and an index j and returns a new *array* that contains the elements in the j-th column of the matrix.

Exercise 7.34. Iterated Matrix Multiplication is where you take a square matrix, A and multiply it by itself k times,

$$A^k = \underbrace{A \times A \times \cdots \times A}_{k \text{ times}}$$

Write a function to compute the k-th power of a matrix A.

Exercise 7.35. The *transpose* of a square matrix is an operation that "flips" the matrix along the diagonal from the upper left to the lower right. In particular the values $m_{i,j}$ and $m_{j,i}$ are swapped. Write a function to transpose a given matrix

Exercise 7.36. Write a function that takes a matrix, a row index i, and a number x and adds x to each value in the i-th row.

Exercise 7.37. Write a function that takes a matrix, a row index i, and a number x and multiples each value in the i-th row with x.

Exercise 7.38. Write a function that takes a matrix, and two row indices i, j and *swaps* each value in row i with the value in row j

Exercise 7.39. A special matrix that is often used is the *identity* matrix. The identity matrix is an $n \times n$ matrix with 1s on its diagonal and zeros everywhere else. Write a function that, given n, creates a new $n \times n$ identity matrix.

Exercise 7.40. Write a function to convert a matrix of integers to floating point numbers.

Exercise 7.41. Write a function to determine if a given matrix is a *permutation matrix*. A permutation matrix is a matrix that represents a permutation of the rows of an identity matrix. That is, A is a permutation matrix if every row and every column has exactly one 1 and the rest are zeros.

Exercise 7.42. Write a function to determine if a given square matrix is *symmetric*. A matrix is symmetric if $m_{i,j} = m_{j,i}$ for all i, j.

Exercise 7.43. Write a function to compute the *trace* of a matrix. The trace of a matrix is the sum of its elements along its diagonal.

Exercise 7.44. Write a function that returns a "resized" copy of a matrix. The function takes a matrix of size $n \times m$ (not necessarily square) and creates a copy that is $p \times q$ (p, q are part of the input to the function). If $p < n$ and/or $q < m$, the values are "cut off". If $p > n$ and/or $q > m$, the matrix is padded (to the right and to the bottom) with zeros.

Exercise 7.45. A *submatrix* is a matrix formed by selecting certain rows and columns from a larger matrix. Write a function that constructs a submatrix from a larger matrix. To do so, the function will take a matrix as well as two row indices i, j and two column indices k, ℓ and it will return a new matrix which consists of entries from the i-th row through the j-th row and k-th column through the ℓ-th column.

For example, if A is

$$\mathbf{A} = \begin{bmatrix} 8 & 2 & 4 & 1 \\ 10 & 4 & 2 & 3 \\ 12 & 42 & 1 & 0 \end{bmatrix}.$$

then a call to this function with $i = 1, j = 2, k = 2, \ell = 3$) should result in

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 1 & 0 \end{bmatrix}.$$

Exercise 7.46. The *Kronecker* product (http://en.wikipedia.org/wiki/Kronecker_product) is a matrix operation on two matrices that produces a larger block matrix. Specifically, if A is an $m \times n$ matrix and B is a $p \times q$ matrix, then the Kronecker product $A \otimes B$ is the $mp \times nq$ block matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}$$

more explicitly:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}.$$

Write a function that computes the Kronecker product.

Exercise 7.47. The Hadamard product is an entry-wise product of two matrices of equal size. Let $\mathbf{A}, \mathbf{B}$ be two $n \times m$ matrices, then the Hadamard product is defined as follows.

$$\mathbf{A} \circ \mathbf{B} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix} \circ \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1m}b_{1m} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2m}b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}b_{n1} & a_{n2}b_{n2} & \cdots & a_{nm}b_{nm} \end{pmatrix}$$

Write a function to compute the Hadamard product of two $n \times m$ matrices.

Stack Frame	Variable	Address	Content
		⋮	⋮
	b[4]	0x5c44cb76	25
	b[3]	0x5c44cb72	20
	b[2]	0x5c44cb68	15
	b[1]	0x5c44cb64	10
	b[0]	0x5c44cb60	5
	i	0x5c44cb56	5
foo	n	0x5c44cb52	5
		⋮	⋮
	a	0x5c44cb34	NULL
	m	0x5c44cb30	5
main	i	0x5c44cb26	0

(a) Program stack at the end of the execution of `foo()` prior to returning control back to `main()`.

Stack Frame	Variable	Address	Content
		⋮	⋮
	b[4]	0x5c44cb76	-626679356
	b[3]	0x5c44cb72	20
	b[2]	0x5c44cb68	15
	b[1]	0x5c44cb64	32767
	b[0]	0x5c44cb60	1564158624
	i	0x5c44cb56	5
foo	n	0x5c44cb52	5
		⋮	⋮
	a	0x5c44cb34	0x5c44cb60
	m	0x5c44cb30	5
main	i	0x5c44cb26	0

(b) Upon returning, the stack frame is no longer valid; the pointer variable `a` points to a stack memory address but the frame and its local variables are no longer valid. Some have been overwritten with other values. Subsequent usage or access of the values in `a` are undefined behavior.

Figure 7.3.: Illustration of the pitfalls of returning a static array in C. Static arrays are locally scoped and exist only within the function/block in which they are declared. The program stack frame in which the variables are stored is invalid when the function returns control back to the calling function. Depending on how the system/compiler/language handles this *unwinding* process, values may be changed, unavailable, etc.

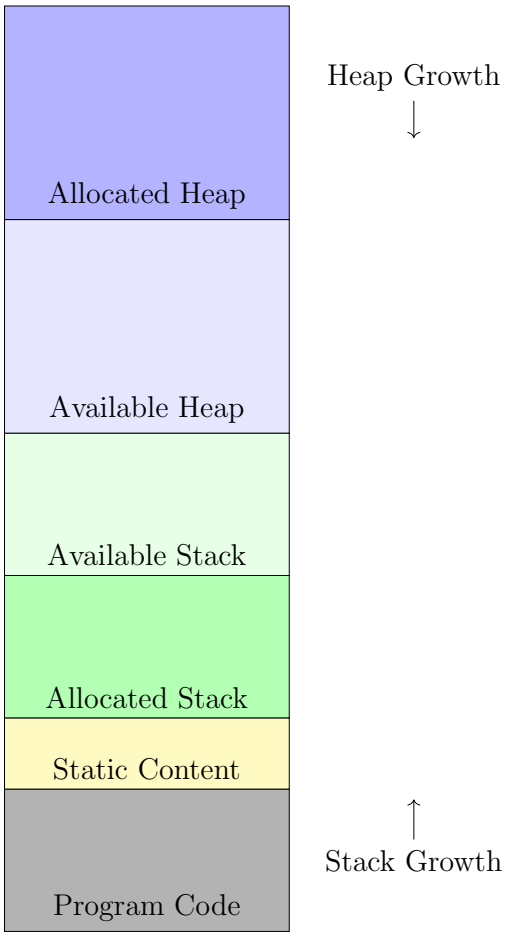
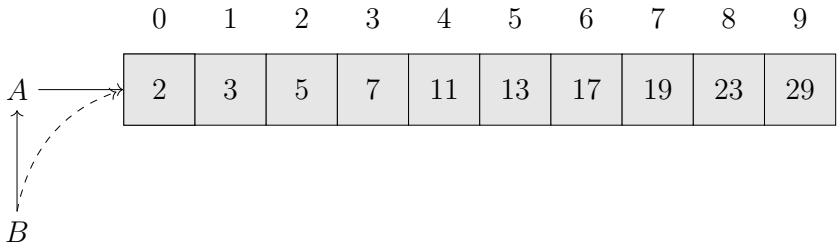
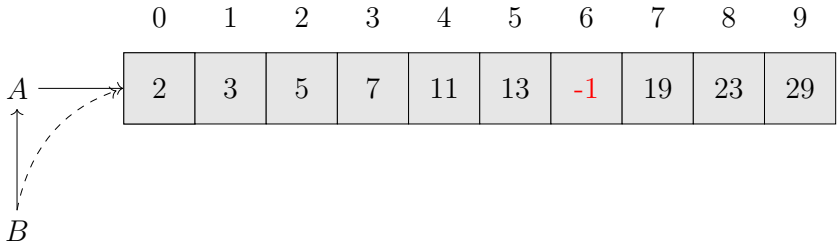


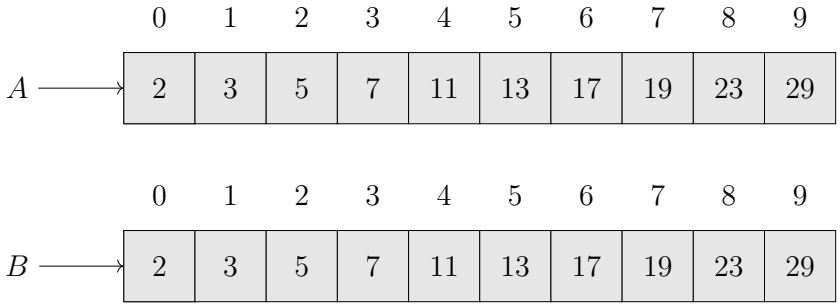
Figure 7.4.: Depiction of Application Memory. The details of how application memory is allocated and how the stack/heap "grow" may vary depending on the architecture. The figure shows stack memory growing "upward" while heap allocation grows "downward." Allocation and deallocation may fragment the heap space though.



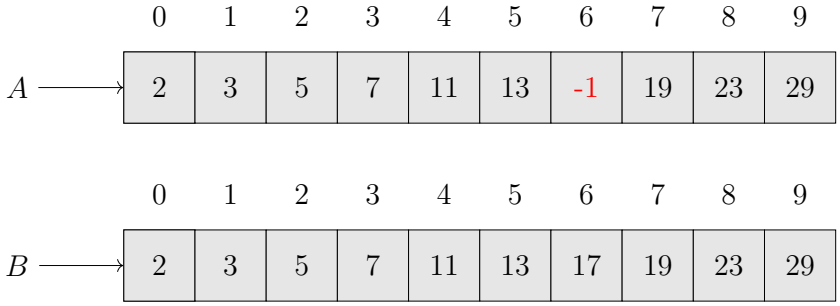
(a) A shallow copy. B refers to A which refers to the array. Thus, B implicitly refers to the same array.



(b) When an element in a shallow copy is changed, `A[6] = -1;`, it is changed from the perspective of both A and B.



(c) A deep copy. B refers to its own copy of the array distinct from A. Both are stored in separate memory locations.



(d) When an element in a deep copy is changed, `A[6] = -1;`, it is changed only in the array A. The element in B is unaffected.

Figure 7.5.: Shallow copies are when two references refer to the same data in memory (a) and (b). Changes to one affect the other. Deep copies (c) and (d) are distinct data in memory, changes to one do not affect the other.

8. Strings

A string is an ordered sequence of characters. We've previously seen string data types as literals. Most languages allow you to define and use static string literals using the double quote syntax. We used strings to specify output formatting using `printf()` -style functions for example. When reading input from a user, we read it as a string and converted the input to numbers. We also described some basic operations on strings including concatenation. We now examine strings in more depth.

Programming languages vary greatly in how they represent string data types. Some languages have string types built-in to the language and others require that you use arrays and yet others treat strings as a special type of object. One issue with string representations is determining where and how the string ends. Some languages use a *length prefix* string representation. The length (the number characters in the string) is stored in a special location at the beginning of a string. Then the string characters are stored as an array. Other Object-Oriented Programming (OOP) languages use a special character, the *null-terminating character* to denote the end of a string. Still other languages store strings as arrays or dynamic arrays and the "bookkeeping" is done internally as part of an object representation.

Other details vary as well. Most languages support the basic ASCII characters, others have full Unicode support or support Unicode through a library. Most languages also provide large libraries of functions and operations that make working with strings easier.

8.1. Basic Operations

Depending on how a language supports strings, it may support various basic operations to create strings and assign them to variables. Usually languages allow you to create and use string literals using the double quote syntax. Modifying a string or copying one string into another may be supported by the built-in assignment operator or it may require the use of a *copy* function. When copying strings, similar issues come into play as with arrays. The "copy" could be a shallow copy or a deep copy (see Section 7.2.2).

Other basic operations may include accessing and/or modifying individual characters in a string. In order to do so, we may also need to know a string's length (so that we do not access invalid characters). A language could provide this as part of the string

itself (usually called a *property* of the string) or through a function call. We can further use such functionality to iterate over the individual characters in a string using an index-controlled for-loop.

More advanced operations on strings include concatenation which is the operation of combining one or more strings to create a new string. Concatenation simply *appends* one string to the end of another string. Another common operation is to extract a *substring* from a string, that is create a new string from a portion of another string. Commonly, this is done via some standard function that may operate by specifying indices and/or the length of the desired substring. Finally, it is also common to deal with collections of strings. Some languages allow you to create arrays of strings or dynamic collections (lists or sets) of strings. for languages in which strings are arrays of characters, an array of strings might be implemented with a 2-dimensional array of characters.

8.2. Comparisons

When processing strings there are several other standard operations. In particular, we often have need to make comparisons between two string variables or between a string variable and a literal. Some languages allow you to use the same operators such as `==` or even `<` to make comparisons between strings. The implied behavior would compare strings for equality (case sensitive) or for lexicographic order. For example "Apple" $<$ "Banana" might evaluate to *true* because "Apple" precedes "Banana" in alphabetic order.

Many languages, however, require that you make string comparisons using a function. Using the equality operator `==` may be correct syntactically, but is usually making a pointer or reference comparison which evaluates to true if and only if the two variables represent the same memory address. Even if two string variables have the same *content*, the equality operator may evaluate to *false* if they are distinct (deep) copies of the same string. Likewise, the inequality operators $<, \leq$, etc. may only be comparing memory addresses which is meaningless for comparing strings.

The solution that many languages provide is the use of a comparator, which is either a function or an object that facilitates the comparison of strings (and more generally, any object). Generally, a comparator function takes two arguments, a, b and compares them, not just for equality, but for their relative order: does a "come before" b or does b "come before" a, or are they equal. To distinguish between these three cases, a comparator returns an integer value with the following general contract: it returns

- Something negative if $a < b$
- Zero if $a = b$
- Something positive if $a > b$

Using this contract we can determine the relative ordering of any two strings. In general we cannot make any assumptions about the actual value that a comparator returns, only that it returns *something* negative or positive. The actual magnitude of the returned value need not be -1 or $+1$, and it may not even have any predefined meaning.

8.3. Tokenizing

It is common to store different pieces of data as a string such that each individual piece of data is demarcated by some *delimiter*. For example, Comma Separated Values (CSV) or Tab Separated Values (TSV) data use commas and tabs to delimit data. For example, the string

```
Smith,Joe,12345678,1985-09-08
```

is a CSV string holding data about a particular person (last name, first name, ID, date of birth). Often we need to process such strings to extract each individual piece of data.

Processing such strings is usually referred to as parsing. In particular, a string is "split" into a collection of individual strings called tokens (thus the process is also sometimes referred to as tokenizing). In the example above, the string would be processed into 4 individual strings, `Smith`, `Joe`, `12345678`, and `1985-09-08`. Each string could further be tokenized if needed, such as parsing the date of birth to extract the year, month, and date.

Most languages provide a function to facilitate tokenization. Some do so by directly returning an array or collection of the resulting tokens (usually with the delimiter removed). Others have a more manual process that requires a loop structure to iterate over each token.

8.4. Exercises

Exercise 8.1. Write functions to reverse a string. If appropriate, write versions to do so by manipulating a given string and returning a new string that is a reversed copy.

Exercise 8.2. Write a function to replace all spaces in a string with *two* spaces.

Exercise 8.3. Write a program to take a phrase (International Business Machines) and "acronymize" it by producing a string that is an upper-cased string of the first letter of each word in the phrase (IBM).

Exercise 8.4. Write a function that takes a string containing a word and returns a *pluralized* version according to the following rules.

1. If the noun ends in "y," remove the "y" and add "ies"
2. If the noun ends in "s," "ch," or "sh," add "es"
3. In all other cases, just add "s"

Exercise 8.5. Write a function that takes a string and determines if it is a *palindrome* or not. A *palindrome* is a word that is spelled exactly the same when the letters are reversed.

Exercise 8.6. Write a function to compute the *longest common prefix* of two strings. For example, the longest common prefix of "global" and "glossary" is "glo". If two strings have no common prefix, then the longest common prefix is the empty string.

Exercise 8.7. Write a function to remove any whitespace from a given string. For example, if the string passed to the function contains `"Hello World How    are you?  "` then it should result in the string `"HelloWorldHowareyou?"`

Exercise 8.8. Write a function that takes a string and flips the case of each alphabetic character in it. For example, if the input string is `"GNU Image Processing Tool-Kit"` then it should output `"gnu iMAGE pROCESSING tOOL-kIT"`

Exercise 8.9. Write a function to validate a variable name according to the rules that it must begin with an alphabetic character, a–z or A–Z but may contain any alphanumeric character a–z, A–Z, 0–9, or underscores `_`. Your function should take a string with a possible variable name and return true or false depending on whether or not it is valid.

Exercise 8.10. Write a function to convert a string that represents a variable name using `under_score_casing` to `lowerCamelCasing`. That is, it should remove all underscores, and replace the first letter of each word with an uppercase (except the first word).

Exercise 8.11. Write a function that takes a string and another character c and counts the number of times that c appears in the string.

Exercise 8.12. Write a function that takes a string and another character c and removes all instances of c from the string. For example, a call to this function on the string `"Hello World"` with c being equal to `'o'` would result in the string `"Hell Wrld"`.

Exercise 8.13. Write a function that takes a string and two characters, c and d and replaces all instances of c with d.

Exercise 8.14. Write a function to determine if a given string s contains a substring t. The function should return true if t appears anywhere inside s and false otherwise.

Exercise 8.15. Write a function that takes a string s and returns a new string that contains the first character of each word in s capitalized. You may assume that words are separated by a single space. For example, if we call this function with the string `"International Business Machines"` it should return `"IBM"`. If we call it with the string `"Flint Lockwood Diatonic Super Mutating Dynamic Food Replicator"` it should return `"FLDSMDFR"`

Exercise 8.16. Write a function that *trims* leading and trailing white space from a string. Inner whitespace should not be modified.

Exercise 8.17. Write a function that splits a string containing a unix path/file into its three components: the directory path, the file base name and the file extension. For example, if the input string is `/usr/home/message.txt` then the three components would be `/usr/home/`, `message` and `txt` respectively. For the purposes of this function, you may assume that the path ends with the *last* forward slash (or is empty if none) and that the extension is always after the *last* period. That is, you should be able to handle inputs such as `../foo/bar/baz.old.txt`.

Exercise 8.18. Write a function that (re)formats a string representing a telephone number. Phone numbers can be written using a variety of formats, for example `1-402-555-1234` or `+4025551234` or `402 555-1234`, etc. Assume that you will only deal with 10 digit US phone numbers. Create a new string that uses the "standard" format of `(402) 555-1234`.

Exercise 8.19. Write a function that takes a string and splits it up to an *array* of strings. The split will be length-based: the function will also take an integer n and will split the given string up into strings of length n. It is possible that the last string will not be of length n.

For example, if we pass `"Hello World, how are you?"` with $n = 3$ then it should return an array of size 9 containing the strings `"Hel"`, `"lo "`, `"Wor"`, `"ld,"`, `" ho"`, `"w a"`, `"re "`, `"you"`, `"?"`

Exercise 8.20. HyperText Markup Language (HTML) (Hypertext Markup Language) is the primary document description language for the World Wide Web (WWW). Certain characters are not rendered in browsers as they are special characters used in HTML; in particular tags which begin and end with the `<` and `>`.

To display such characters correctly they need to be *escaped* (similar to how you need to escape tabs `\t` and endline `\n` characters). Properly escaping these characters is not only important for proper rendering, but there are also security issues involved (Cross-Site Scripting Attacks).

Write a function that takes a string and *escapes* the HTML characters in Table 8.1.

Replace the following	with this
`&`	`&`
`<`	`<`
`>`	`>`
`"`	`"`

Table 8.1.: Replacement HTML Characters

9. File Input/Output

A file is a block of data used for storing information. Normally, we think of a file as something that is stored on a hard drive (or memory stick or other physical media), but the concept of a file is much more general. For example, when a file is loaded ("read") by a program it then exists in main memory. An executable program itself is a file (containing instructions to be executed), both stored on the hard drive and run in memory.

In a typical unix-based system, *everything* is a file. Directories are files, executables are files, devices are files, etc. Even the familiar standard input and standard output are buffers that are treated as files that can be read from or written to.

Data files may be stored as binary data or as plaintext files. Plaintext files are still stored as binary data, but are stored in an encoding using the ASCII text values. Binary files will also have structure, but it depends on the application that produced the file to give meaning to the data. For example, an image file in a Joint Photographic Experts Group (JPEG) format is essentially just binary data but it has a very specific format that an image viewer would be able to process, but, say, a text editor would not. Further, if the binary format is corrupted, the image viewer might not be able to display the image correctly

Typical programs are short lived, anywhere from a few milliseconds to maybe a user "session." We often want data to be saved across multiple runs of a program. We need to save it or *persist* it in some durable storage medium (disk). Files provide a way to achieve data persistence.

9.1. Processing Files

In general, processing data in a file involves three basic steps:

1. Open (or create) the file
2. Read from (or write to) the file
3. Close the file

Depending on the language, the act of opening a file may determine if it will be read from or written to. When read from, the file is referred to as an *input file* while a file that is written to is an *output file*. Languages may also have different a different API or functions to read/write or append to a file.

A file may be read line by line until the end of the end of the file has been reached. Languages usually support this by using a special End Of File (EOF) flag or value to indicate the end of a file has been reached.

A file is a resource just like memory and we need to properly manage it. We need to make sure that we close a file once we are finished processing it. Depending on the language and other factors such as the operating system, failure to close a file may result in corrupted data. Though a file may be closed automatically for us when the program terminates, its still best practice to properly close it.

9.1.1. Paths

When opening a file on a file system, it is necessary to specify *which* file you want to open. This is typically done by specifying at least the name of the file. Often files will have "extensions" which indicate the type of file it is such as `.txt` for text files or `.html` for HTML files. However, file extensions are only for organizational purposes and have no real bearing on what data is stored in the file.

More important is the *path* of the file. Usually, if no path is specified, then implicitly we are opening the file in the Current Working Directory (CWD). For example, if we open the file `data.txt` then we are opening the file in the same directory in which our program is executing. When specifying a path we can either specify an *absolute* path or a *relative* path. An absolute path specifies each and every subdirectory in the file system from the *root* to the directory that the file is located in. The root directory is the top-most directory in the file system. Each subdirectory is separated by some *delimiter*.

Windows systems use a backslash as a directory delimiter while the root directory is specified using a "volume" name such as `C:\`. For example, an absolute path on a Windows system may look something like:

```
C:\applications\users\data\data.txt
```

On a Unix-based system, a forward slash is used as a directory delimiter and the root directory is simply a single forward slash. The same directory structure in a Unix-based system would look like the following.

```
/applications/users/data/data.txt
```

A path may also be relative to the current working directory. In most systems (Windows and Unix-based) the current directory is denoted using a single period, `.`. You can use

this to specify directories deeper in the directory tree from the current directory. For example (in Unix),

```
./app/data/data.txt
```

would refer to the directory `app` in the current working directory, the directory `data` within that, and finally the file `data.txt` within that directory.

We can also refer to files further up the directory tree using the "parent" directory symbol which is two periods, `..`. For example,

```
../../system/data.txt
```

would refer to a file two levels up in the subdirectory `system`.

9.1.2. Error Handling

When dealing with files there are many potential error conditions that may be anticipated and may need to be dealt with. Some types of errors that can occur include the following.

- Permission errors – Your program may not have the proper permissions to read or write to a particular directory or file. For example, user-level processes typically do not have permissions to read system-level files (such as password files) nor can they write to them lest they corrupt critical system data.
- File Not Found errors – Your program may attempt to open a file that does not exist. Sometimes, particularly when writing, the file will be created if it does not exist. However, for reading, this may be a serious error.
- I/O Errors – We may be able to successfully find and open a file, but while processing it something might go wrong with the file system that results in a general input/output error.
- Formatting errors – As previously mentioned, the format of a file is highly dependent on the application that created it (though there are universal data formats such as XML or JavaScript Object Notation (JSON)). If the format is not as expected the file may be corrupted and the program may not be able to successfully read data from it.

Depending on the language, these errors may be fatal or may result in error codes or error values that can be dealt with. Or they may result in exceptions that can be caught and handled.

```
/proc/self/
|-- attr
|-- cwd -> /proc
|-- fd
|   `-- 3 -> /proc/15589/fd
|-- fdinfo
|-- net
|   |-- dev_snmp6
|   |-- netfilter
|   |-- rpc
|   |   |-- auth.rpcsec.context
|   |   |-- auth.rpcsec.init
|   |   |-- auth.unix.gid
|   |   |-- auth.unix.ip
|   |   |-- nfs4.idtoname
|   |   |-- nfs4.nametoid
|   |   |-- nfsd.export
|   |   `-- nfsd.fh
|   `-- stat
|-- root -> /
`-- task
    `-- 15589
        |-- attr
        |-- cwd -> /proc
        |-- fd
        | `-- 3 -> /proc/15589/task/15589/fd
        |-- fdinfo
        `-- root -> /
```

Figure 9.1.: The Linux file system (like most file systems), defines a tree directory structure. Each file and directory is contained in subdirectories all contained within the root directory, `/`. This diagram was generated by the Tree command, http://mama.indstate.edu/users/ice/tree/.

9.1.3. Buffered and Unbuffered

When processing files the input/output may be either *buffered* or *unbuffered*. A buffered input or output "stream" is one in which data that is read/written is actually stored in memory in a "buffer" until such a time as the buffer is "flushed" and the accumulated data is passed to/from the actual file.

For example, in a buffered output file, our program could write several kilobytes of data to the output file, but it might not actually be written to the file right away. Instead, those kilobytes of data are stored in memory until the buffer fills up or some other event takes place to cause the buffer to be flushed. At that point, the data stored in the buffer is emptied and written to the file.

Buffered input/output is used because I/O operations are expensive in terms of system resources and can slow the system down. Because of this, it is better to keep I/O operations as infrequent as possible. Buffers help to reduce the number of I/O operations performed by a program by making them less frequent.

There are some instances in which we want unbuffered I/O. When error messages are written to the standard error output for example, we would prefer to know about errors as soon as possible rather than waiting for error messages to accumulate in a buffer. Using an unbuffered output means that data is written to the standard error (which *is* a file) immediately. However, because errors are (hopefully) infrequent and (likely) fatal, this is not a performance issue.

9.1.4. Binary vs Text Files

As previously mentioned, files can be stored as pure binary data or as plaintext (ASCII). Depending on our application and the nature of the data being written to files, the choice of which to use may be clear. If we want the data in our files to be human-readable, then we need to store them as plaintext. However, in general, we should prefer storing data in a binary format. The reason for this is that binary generally requires less space and is more efficient to process.

Consider as an example, storing a collection of integers in a file. Each integer requires 4 bytes when represented in binary. However, when represented as a string, it requires as many bytes as there are digits in the number. For example, the maximum representable value of such an integer would be $2^{31} - 1 = 2,147,483,647$, requiring 10 ASCII characters and thus 10 bytes to represent, two and a half times as much memory as with the binary representation. Further, if a lot of numbers are stored, each number (as a string) would need to be delimited by yet another character. With a binary representation, no delimiter would be necessary as we would know that each 4 byte block represents a single number.

This disparity can be even more stark with other types such as floating-point numbers.

There are additional performance issues when reading/writing the data and converting binary numbers to their string representations. With binary data no such parsing is necessary. As long as the data does not need to be human-readable, binary formats should be preferred.

9.2. Exercises

Exercise 9.1. Write a function that takes a string representing a file name and opens and processes the file, returning all of its contents as a single string.

Exercise 9.2. Consider an irregular, 2-D simple polygon with n points,

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

The area A of the polygon can be computed as

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

Note, that the initial and end point will be the same, $(x_0, y_0) = (x_n, y_n)$. An example polygon for $n = 5$ can be found in Figure 9.2.

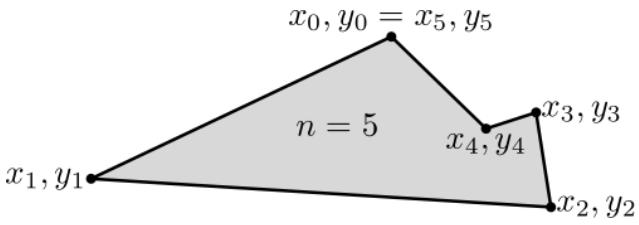


Figure 9.2.: An example polygon for $n = 5$

Write a program to open and process a text file containing n coordinates. In particular, the first line is a single integer n that indicates how many points should be read in. Each line after that has the x, y coordinates of each point separated by a single space.

```
4
1.0 0.0
13.2 1.25
20.5 18.4
16.37 24.54
```

After reading the file in, it will compute the area of the polygon according to the formula above and output it to the user. For example, the output for the above file may be something like `Area of the polygon: 197.9135`

Exercise 9.3. Write a program that processes an input text file and scrubs it of any HTML characters that need to be escaped (see Exercise 8.20 for details). It should produce a new output file with all special characters escaped.

Exercise 9.4. Write a program that spell checks a plain text file. The program will open a text file and process each word separately, checking for proper spelling against a standard dictionary.

You may assume that each word is separated by some whitespace (you may assume that there are no multi-line hyphenated words). However, you should ignore all punctuation (periods, question marks, etc.).

Use a standard American dictionary provided on your unix system which stores words one per line. Your output should include all misspelled or unrecognized words (words not contained in the dictionary file).

Exercise 9.5. A standard word search consists of an $n \times n$ grid in which there are a number of words hidden, some intersecting, with dummy letters filling in the blanks. An example is provided in Figure 9.3.



Figure 9.3.: A Word Search

Write a program to solve a word search. Your program will read in an input file with the following format: the first line will contain a single integer n which is followed by n lines with n characters (not including the end line character) corresponding to the word search.

Once you read in the word search, you will iterate through all possible words running down, right, or diagonally down-right. You will attempt to match each possibility against a standard English dictionary. If the word matches a word in the dictionary, output it to the standard output, otherwise ignore it. To simplify, you may restrict your attention to words that have a length between 3 and 8 (inclusive).

Exercise 9.6. Write a crossword puzzle cheater. The program will take, as input, a "partial" word in a crossword puzzle. That is, some of the letters are known (from other solved clues) while some of the letters are not known. For the purposes of this exercise, we'll use a hyphen as a placeholder for missing letters.

Your program will match the partial word against words in a standard English dictionary and list all possible matches. For example, if the user provided `foo-` as input it might match `food`, `fool`, and `foot`.

Exercise 9.7. Bridge is a four player (2 team) game played with a standard 52-card deck. Prior to play, a round of bidding is performed to determine which team is playing for or against the contract, the trump suit, and at what level. Understanding the rules of the game or the bidding conventions involved are not necessary for this exercise. Instead, write a program to assist players in how they should bid based on the following point system.

A standard 52-card deck is dealt evenly to 4 different hands (Players 1 thru 4, 13 cards each). Each player's hand is worth a number of points based on the following rules:

- Each Ace in the hand is worth 4 points
- Each King is worth 3
- Each Queen is worth 2
- Each Jack is worth 1
- For each suit (Diamond, Spade, Club, Heart) such that the hand has only 2 cards (a "doubleton") an additional point is added
- For each suit that the hand has only 1 card in (a "singleton") two additional points are added
- For each suit that the hand has no cards (a "void") 3 additional points are added.

Write a program that reads in a text file containing a deal. The formatting is as follows: the input file will have 4 lines, one for each player. Each line contains the cards dealt to that player delimited by a single space. The cards are indicated by the rank $(A, K, Q, J, 10, 9, \ldots, 2)$ and the suit (D, S, C, H). An example:

```
3C 3D 7S QD KC AS 6S AC JS 4S JD 7H 6D
5D 8C 7D AH 3H QC 8D JH 5H 9D 7C 9C 4D
2H 10D 8H KS QH 4C 10S 9S 6H 8S KD AD QS
2D 10C 6C 2C 10H 4H 2S 3S 5C 9H KH JC 5S
```

Your program should process the file and output the total number of points each hand represents. You should not make any assumptions about the ordering of the input.

```
Hand 1 Points: 17
Hand 2 Points: 10
Hand 3 Points: 16
Hand 4 Points: 6
```

Exercise 9.8. The game of Sudoku is played on a 9×9 grid in which entries consist of the numbers 1 thru 9. Initially, the board is presented with some values filled in and others blank. The player has to fill in the remaining values until all grid boxes are filled and the following constraints are satisfied.

- In each of the 9 rows, each number, 1–9 must appear exactly once
- In each of the 9 columns, each number 1–9 must appear exactly once
- In each of the 3×3 sub-grids, each number 1–9 must appear exactly once

A full example is presented in Figure 9.4.

8	3	5	4	1	6	9	2	7
2	9	6	8	5	7	4	3	1
4	1	7	2	9	3	6	5	8
5	6	9	1	3	4	7	8	2
1	2	3	6	7	8	5	4	9
7	4	8	5	2	9	1	6	3
6	5	2	7	8	1	3	9	4
9	8	1	3	4	5	2	7	6
3	7	4	9	6	2	8	1	5

Figure 9.4.: A solved Sudoku puzzle

Write a program that processes a text file containing a possible sudoku solution and determine if it is a *valid* or *invalid* solution. The file will have the following format: it will contain 9 lines with 9 numbers on each line delimited by a single space. If the input represents a valid solution, output "Valid Solution", otherwise output at least one reason why the input is not a valid solution.

Exercise 9.9. Write a program that parses and processes a data file containing Comma Separated Values (CSV) and produce an equivalent JSON (JavaScript Object Notation) output file containing the same data.

The input file will have the following format. The first line is a CSV list of column names. Each subsequent line is an individual record with values for each column. The number of columns and rows may vary from file to file. The following is an example containing data about students, which has four columns and 3 records.

```
lastName,firstName,NUID,GPA
Castro,Starlin,11223344,3.48
Rizzo,Anthony,55667788,3.95
Bryant,Chris,01234567,2.7
```

The output file will be formatted in JSON where each "object" (record) is denoted with opening and closing curly brackets, each record is separated by a comma, and all records are enclosed in square brackets (putting them in an array). For each record, each value is denoted with a key (the column name) and a value. For this exercise, treat all values as strings even if they are numbers. For example, the input file above would be formatted as follows.

```
[
  {
    "lastName": "Castro",
    "firstName": "Starlin",
    "NUID": "11223344",
    "GPA": "3.48"
  },
  {
    "lastName": "Rizzo",
    "firstName": "Anthony",
    "NUID": "55667788",
    "GPA": "3.95"
  },
  {
    "lastName": "Bryant",
    "firstName": "Chris",
    "NUID": "01234567",
    "GPA": "2.7"
  }
]
```

Exercise 9.10. Ranked voting elections are elections where each voter *ranks* each candidate rather than just voting for a single candidate. If there are n candidates, then

each voter will rank them 1 (best) through n (worst). Usually, the winner of such an election is determined by a *Condorcet* method (the candidate that would win in by a majority in all head-to-head contests). However, we'll use an alternative method, a *Borda count*.

In a Borda count, points are awarded to each candidate for each ballot. For every number 1 ranking, a candidate receives n points, for every 2 ranking, a candidate gets $n - 1$ points, and so on. For a rank of n, the candidate only receives 1 point. The candidates are then ordered by their total points and the one with the highest point count wins the election. Such a system usually leads to a "consensus" candidate rather than one preferred by a majority.

Implement a Borda-count based ranked voting program. Your program will read in a file in the following format. The first line will contain an ordered list of candidates delimited by commas. Each line after that will represent a single ballot's ranking of the candidates and will contain comma delimited integers 1 through n. The order of the rankings will correspond to the order of the candidates on the first line.

Your program will take an input file name as a command line argument, open the file and process it. It will then report the results including the point total for each candidate (in order) as well as the overall winner. It will also report the total number of ballots. You may assume each ballot is valid and all rankings are provided.

An example input:

```
Alice,Bob,Charlie,Deb
2,1,4,3
3,4,2,1
4,2,3,1
3,2,1,4
3,1,4,2
```

An example output:

```
Election Results
Number of ballots: 5

Candidate  Points
Bob        15
Deb              14
Charlie    11
Alice      10
```

```
Winner is Bob
```

Exercise 9.11. A DNA sequence is a sequence of some combination of the characters `A` (adenine), `C` (cytosine), `G` (guanine), and `T` (thymine) which correspond to the four nucleobases that make up DNA. Given a long DNA sequence, its often useful to compute the frequency of *n-grams*. An n-gram is a DNA subsequence of length n. Since there are four bases, there are 4^n possible n-grams.

Write a program that processes a DNA sequence from a plaintext file and, given n, computes the relative frequency of each n-gram as it appears in the sequence. As an example, consider the sequence in Figure 9.5.

```
GGAAGTAGCAGGCCGCATGCTTGGAGGTAAAGTTCATGGTTCCCTGGCCC
```

Figure 9.5.: A DNA Sequence

To compute the frequency of all $n = 2$ n-grams, we would consider all 16 combinations of length-two DNA sequences. We would then go through the sequence and count up the number of times each 2-gram appears. We then compute the relative frequency (note: if a sequence is length L, then the total number of n-grams in it is $L - (n - 1)$). The relative frequency of each such 2-gram is calculated below.

```
AA  6.1224%
AC  0.0000%
AG 10.2041%
AT  4.0816%
CA  6.1224%
CC 10.2041%
CG  2.0408%
CT  4.0816%
GA  4.0816%
GC 10.2041%
GG 12.2449%
GT  8.1633%
TA  4.0816%
TC  4.0816%
TG  8.1633%
TT  6.1224%
```

Exercise 9.12. Given a long DNA sequence, it is often useful to compute the number of instances of a certain *subsequence*. As an example, if we were to search for the

subsequence GTA in the DNA sequence in Figure 9.5, it appears twice. As another example, in the sequence `CCCC`, the subsequence `CC` appears three times.

Write a program that processes a text file containing a DNA sequence and, given a subequence s, searches the DNA sequence and counts the number of times s appears.

Exercise 9.13. Protein sequencing in an organism consists of a two step process. First the DNA is translated into RNA by replacing each thymine (T) nucleotide with uracil (U). Then, the RNA sequence is translated into a protein according to the following rules. The RNA sequence is processed 3 bases at a time. Each *trigram* is translated into a single amino acid according to known encoding rules. There are 20 such amino acids, each represented by a single letter in $(A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y)$.

The rules for translating trigrams are presented in Figure 9.6. Each triple defines a protein, but we're only interested in the first letter of each protein. Moreover, the trigrams UAA, UAG, and UGA are special markers that indicate a (premature) end to the protein sequencing (there may be additional nucleotides left in the RNA sequence, but they are ignored and the translation ends).

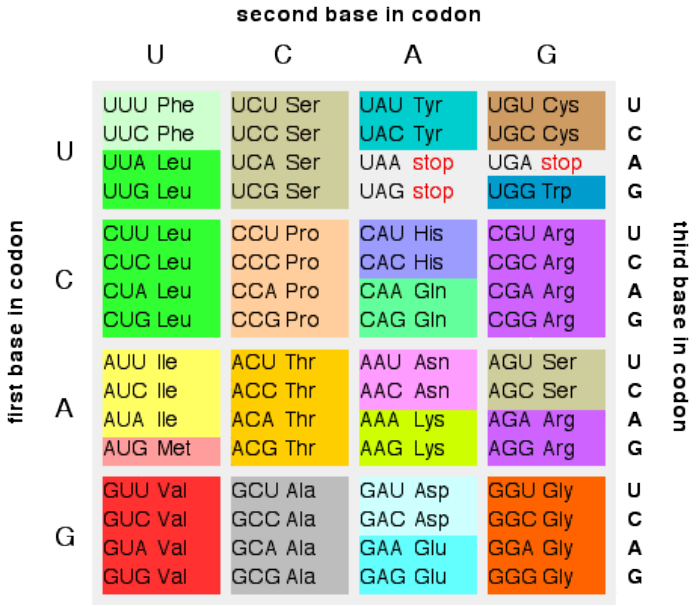


Figure 9.6.: Codon Table for RNA to Protein Translation

As an example, suppose we start with the DNA sequence $AAATTCCGCGTACCC$; it would be encoded into RNA as $AAAUUCCGCGUACCC$; and into an amino acid sequence $KFRVP$.

Write a program that processes a file containing a DNA sequence and outputs the translated proteins (only the first letter of each protein) to an output file.

Exercise 9.14. Recently, researchers have successfully inserted two new artificial nucleases into simple bacteria that successfully reproduced the artificial bases through several generations. The artificial bases d5SICS and dNaM, (X and Y for short) mimic the natural G, and C nucleobases respectively.

Write a program that takes a normal DNA sequence and replace some of its G, C pairs with X, Y respectively. DNA is translated into RNA which is then translated into 20 different amino acids. Each amino acid produced depends on a 3 nucleobase *codon*. For this exercise, we will change G, C pairs with X, Y pairs but only in codons that represent the amino acids Threonine (an essential amino acid) and Alanine (a non-essential amino acid). Table 9.1 contains the codons corresponding to these amino acids and the codons you should translate each one to. All other codons should not be modified.

Your program should open and process a DNA sequence contained in a file and modify the DNA sequence as described above and output the artificial DNA sequence to a new output file.

Amino Acid	Codon	Artificial Codon
Threonine	ACT	AYT
	ACC	AYY
	ACA	AYA
	ACG	AYX
Alanine	GCT	XYT
	GCC	XYY
	GCA	XYA
	GCG	XYX

Table 9.1.: Amino Acid Codons

10. Encapsulation & Objects

One reason we prefer to write programs in high-level programming languages is that we can use syntax that is closer to plain English. Though programming language syntax is a far cry from "natural" language, it is far closer than lower level languages such as assembly or binary machine code. However, from what we've seen so far, when writing programs we are still forced to utilize the primitive variable types built-in to the language we're using, which is still quite limiting.

As a motivating example, suppose we were to write a program that involved organizing the enrollment of students into courses. To model a particular student, we would need a collection of variables, say a first name, last name, GPA, and a unique identification number (likely a lot more, but let's keep it simple). Each of these pieces of data could be modeled by strings, a floating-point number and perhaps an integer.[1] Each of these pieces of data are stored in separate, unrelated variables even though they represent a single *entity*.

Even worse, suppose that we needed to keep track of a collection of students. Each piece of data would need to be stored in a separate array. If we wanted to rearrange the data (say, sort it), we would need to do a lot of manual bookkeeping to make sure that the separate pieces of data that represented a single entity were all aligned at the same index in each of the arrays. If we wanted to pass the data around to functions, we'd be forced to pass multiple arrays to each function. This becomes all the more complex when we attempt to model entities with more pieces of data.

The solution is to *encapsulate* the pieces of data into one logical entity, sometimes referred to as an *object*. More formally, *encapsulation* is a mechanism by which pieces of data can be grouped together along with the functions that operate on that data. Encapsulation may also provides a means to *protect* that data by controlling the visibility of that data from code outside the object.

Contrast this with an array which is also a collection of data. However, an array usually contains pieces of similar data (all elements are integers or all elements are floating point numbers) while an object may collect pieces of dissimilar data that make up a larger entity. It is like the difference between rows and columns in a table. Consider the student data in Table 10.1. Each row represents a record while each column represents a

[1] Depending on the identification number, it may be more appropriately modeled with a string. Social Security Numbers for example are not purely numeric; they include dashes and may begin with zeros.

First Name	Last Name	ID	GPA
Tom	Baker	74	3.75
Christopher	Eccleston	5	3.5
David	Tennant	10	4.0
Matt	Smith	29	3.2
Peter	Capaldi	13	2.9

Table 10.1.: Student Data

collection of data from each record. A single column is comparable to an array while each row is comparable to an object. In this example, each object has four pieces of data encapsulated in it, a first name, last name, an ID, and a GPA.

To represent this data in code without objects we would need at least 4 separate arrays, more if we wanted to model more data for a student. Moreover, data in separate arrays or collections have no real logical relationship to each other. The solution that most programming languages provide is allowing you to *define* an object or structure that collects pieces of data into one logical unit, allowing you to *name* the object (say "Student") so that you can deal with the data in a more abstract way. With objects, we can treat each row in the table as a single, distinct entity allowing us to collect Student objects into a single array or collection rather than many separate ones.

10.1. Objects

Though languages differ in how they support objects, they all have some commonalities. A language needs to provide ways to define objects, create *instances* of objects, and to use them in code.

10.1.1. Defining

Most object oriented programming languages such as C++ and Java are *class-based* languages. Meaning that they allow you to define objects by declaring and defining a "class." A class is essentially a blueprint for what the object *is* and how it is defined. Generally, a class declaration allows you to specify member variables and member methods which are part of the class. Further, full encapsulation is achieved by using *visibility* keywords such as `public` or `private` to either allow or restrict access to variables and methods from code *outside* the object.

Non-object-oriented languages may not support full encapsulation. Instead they may allow you to define *structures* which support the grouping of data, but make it difficult

or impossible to achieve the other two aspects of encapsulation (the grouping of methods that act on that data and the protection of data).

In either case, a language allows you to define the member variables and to *name* the class or structure so that instances can be referred to by that *type*. Built-in types such as numbers or strings already have a type name defined by the language. However, an object is a *user-defined* type that is not built-in to the language. Once defined, however, the class or structure *can* be referred to just like any built-in variable type.

It is not unusual to create objects that are made of other objects. For example, a student object may be defined by using two strings for its first and last name. In the language, a string may also be an object. As a more complex example, suppose that we wanted an additional member variable to model a student's date of birth. A date may itself be an object as it consists of several pieces of information (a year, month, and date at least). When an object "owns" an instance of another object it is referred to as *composition* as the object is *composed* of other objects. Further, an object may consist of a *collection* of other objects (suppose that a student object owned an array of course objects representing their schedule). This is a form of composition known as *aggregation* (multiple objects have been aggregated by the object).

10.1.2. Creating

Once a blueprint for an object (or structure) has been declared and defined, we need a way to *create* instances of the object. The concept of an "object" is general and abstract. It is more like the *idea* of a student. Only once we have created an entity that exists in memory do we have an actual *instance* of the class. Creating instances of an object is usually referred to as *instantiation*.

Languages may be able to automatically create instances of your object with *default* values. After all, your object is likely composed of built-in types. The student example above for example could be modeled with two strings, an integer, and a floating point number. The language/compiler/interpreter "knows" how to deal with these built-in types, so it can extend that knowledge to create instances of your object which are essentially just collections of types that it already knows how to deal with.

Object-oriented languages usually provide a special method for you to be able to specify the details of how to create an instance. These are called *constructor* methods. Sometimes you can define multiple constructors methods that take different number(s) of arguments and/or have different behavior. Constructor methods typically have special syntax or have the same name as the class.

In other languages that do not fully support object-oriented programming, you must define utility functions that can be used to create instances of your object. Sometimes these are referred to as *factory* functions as they are responsible for "manufacturing"

instances of your object.

10.1.3. Using Objects

After defining and creating an object, you can usually use it like any regular variable. In a strongly typed language you would declare a variable whose type matches the declared class or structure. The variable type can usually be passed and returned from functions, assigned to other variables, etc.

A language also provides ways to access the member variables or methods that are *visible* to the outside world. Languages usually allow you to do this through the "dot operator" or the "arrow operator." Suppose we have an instance of a student object stored in a variable `s`. To access the first name of this instance, we may be able to use either `s.firstName` or `s->firstName`. We can access and invoke visible methods likewise.

The dot/arrow operators are how code *outside* the object interacts with the object. Outside code is able to do this because it holds a reference, `s` to the object. However, *inside* the object, we may not have a reference (the variable `s` was ostensibly declared and used outside the object and so is not in scope inside the object). However, we still have need to reference member variables or methods from inside the object. Many languages use open recursion , a mechanism by which we can write code so that an instance is able to refer to itself. Languages use keywords such as `this` or `self` or something similar. The keyword is essentially a self-reference to the object itself so that you can refer to "this" object from within the object.

10.2. Design Principles & Best Practices

Using objects in your code follows more of a bottom-up design rather than a top-down design approach. In a top-down design, a program is designed by breaking a program or problem down into smaller and smaller components. Bottom-up design approaches a problem differently. First, real-world entities involved with the problem are modeled by defining objects. Then objects are used as building blocks that can be combined and made to interact to solve a problem.

Object design is usually a straightforward process. Typically an object is modeling a real-world entity, so it is simple enough to decompose the entity into its constituent components. We do this until the component can either be modeled by a built-in type such as a string or number or by an existing object. In general, you want to keep things as simple as possible. Any time you need to associate pieces of data together into one logical unit, it is appropriate to encapsulate them into an object.

A good design principle is to utilize composition as much as possible. If you have multiple

pieces of data that define a logical entity or unit, it is good design to create another object. For example, suppose a student object needs to model a mailing address; think about what an address is: it is a street address, city, state, zip, etc. Rather than having these as member fields to your object, it is probably more appropriate to define an "address" object, especially if such an object would be useful elsewhere in a program.

10.3. Exercises

Exercise 10.1. A *complex* number consists of two real numbers: a real component a and an imaginary component bi where b is a real number and $i = \sqrt{-1}$. Define an object or structure to model a complex number. Write functions to:

- Create a complex number
- Print a complex number
- Perform basic arithmetic operations on two complex numbers including addition, subtraction and multiplication as defined by:

$$c_1 + c_2 = (a_1 + b_1 i) + (a_2 + b_1 i) = (a_1 + a_2) + (b_1 + b_2)i$$

$$c_1 - c_2 = (a_1 + b_1 i) - (a_2 + b_1 i) = (a_1 - a_2) + (b_1 - b_2)i$$

$$c_1 \cdot c_2 = (a_1 + b_1 i) \cdot (a_2 + b_1 i) = (a_1 a_1 - b_1 b_2) + (a_1 b_2 + b_1 a_1)i$$

Exercise 10.2. Design an object (or structure) that models an album. Include at least the album title, artist (or band) and release year. Include any other data that you think is relevant and write functions to support your object.

Exercise 10.3. Design an object (or structure) that models a bank savings account. Include at least the balance, APR, an account "number" and customer information (which may be another object or structure). Include any other data that you think is relevant and write functions to support your object.

Exercise 10.4. Design an object (or structure) that models a sports stadium. Include at least the stadium name, the team that plays there, its city, state, and year built. Include both latitude and longitude data. Include any other data that you think is relevant and write functions to support your object. Write a parser to process a flat file data of all stadiums (in your chosen sport) and build instances of all of them.

Exercise 10.5. Design an object (or structure) that models a network-connected electronic device. Include at least a unique ID, a human-readable name for the device, a Media Access Control (MAC) address and Internet Protocol (IP) address as well as the device's bandwidth in megabits per second. Include any other data that you think is relevant and write functions to support your object.

Exercise 10.6. Design an object (or structure) that models an airport. Include at least the name, FAA designation, its city, state, and latitude/longitude data. Include any other data that you think is relevant and write functions to support your object.

11. Recursion

Suppose we wanted to write a simple program that performed a countdown, printing 10, 9, 8, . . . , 2, 1 and when it reached zero it printed a "Happy New Year" message. Likely our first instinct would be to write a very simple for loop using an increment variable. But suppose we lived in a world *without* the usual loop control structures that we are now familiar with. How might we write such a program?

After thinking about it for a while, we might think: well, we don't have loops, but we still have functions. In particular what if we had a function that took the "current" value of our counter variable and decremented it, passing it to another function, which did the same thing. For example, we could pass 10 to such a function, which would then subtract 1, passing 9 to another function and so on. A check could be made to see if the value was zero, in which case we print our special message and no longer call any more functions.

In fact, we would not need to define 10 different functions to do so. Instead, we could define one function that *called itself*. It might look something like Algorithm 11.1.

```
   INPUT  : An integer n ≥ 0
   OUTPUT : A countdown of integers n, ... 0
1  IF n = 0 THEN
2  |  output "Happy New Year!!!"
3  ELSE
4  |  output n
5  |  CountDown(n − 1)
6  END
```

Algorithm 11.1: Recursive CountDown(n) Function

The function in this case is called CountDown(). In Line 5 the function calls itself on a decremented value. When a function calls itself, it is a *recursive* function. When a language allows functions to call themselves they support *recursion*.

This is not that odd of a concept. We've seen many examples where functions invoke (call) other functions. Each function call simply creates a new stack frame on the program stack. There is nothing particularly special about which functions call which other functions, so there is little difference when a function calls itself.

This was not just a toy example. There are many programming languages in which recursion is used as a matter of course. Functional programming languages tend to avoid control structures like loops and even (mutable) variables. Instead, control flow is defined by evaluating a series of functions, making recursion a fundamental technique.

Recursion is extensively used in mathematics. Recurrence relations or recursive functions are common. The Fibonacci sequence is a common, if not overused[1] example. It has a simple definition: the next value in the sequence is simply the sum of the two previous values. The sequence starts with the *initial values* of 1. The first few terms in the sequence:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

The more formal mathematical definition can be stated as follows.

$$F_n = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

The Fibonacci sequence is the cliche example for recursion. We can define an algorithmic function to compute the n-th Fibonacci number as follows.

```
   INPUT  : An integer n ≥ 0
   OUTPUT : The n-th Fibonacci number, F_n
1  IF n ≤ 1 THEN
2  |  output 1
3  ELSE
4  |  output FIBONACCI(n − 1) + FIBONACCI(n − 2)
5  END
```

Algorithm 11.2: Recursive FIBONACCI(n) Function

Though hackneyed, it does provide a good example for how recursive functions work. We'll also utilize it as an example of *why you should avoid recursion in practice*. We use it to illustrate how the problems with recursion can be mitigated or avoided altogether.

11.1. Writing Recursive Functions

When writing a recursive function, there are several key elements that we need to take care of to ensure that it executes correctly. In particular, every recursive function requires

[1] The Fibonacci sequence is nothing special; its simply a second order linear homogenous recurrence relation with coefficients of 1. The near reverence that so many people attribute to it borders on mysticism.

at least one *base case* or base condition which serves as a terminating condition for the recursion. A base case is a condition which, instead of making a recursive call, processes and returns a value. Without a base case, the recursion would continue unbounded: the function would call itself over and over again, creating new stack frame after stack frame until we run out of stack space. If a program makes too many function calls and runs out of stack memory, it may lead to a stack overflow and the termination of the program. Even if we don't have unbounded recursion, it is still possible to run out of stack space even with simple recursion.

The other key element that we need is to ensure that every recursive call *makes progress* toward one of the terminating conditions. If no progress is made, then again we may have an unbounded recursion. In the Fibonacci example in Algorithm 11.2, the base case can be found in the first if-statement: when n reaches 1 or less, no recursive calls are made. In the else-statement, we make two recursive calls, but both of them make progress toward this base case. The first decrements n by 1 and the second by 2, eventually reaching $n = 1$.

11.1.1. Tail Recursion

Making many function calls can be costly in terms of stack space. One optimization that can be made is to use *tail recursion*. The last operation that a function executes is referred to as the *tail* operation. If a function invokes another function as its tail operation, its a *tail call*. For example, consider the following snippet of code:

```
1 int foo(int x) {
2   ...
3   return bar(x-1) + 1;
4 }
```

Here, `foo()` calls `bar()` but it is *not* the last operation before it returns. Instead, it invokes `bar()`, takes the result and adds one *then* returns to the calling function. Note that decrementing `x` is performed *before* the invocation of `bar()`. In contrast, consider the following modified code:

```
1 int foo(int x) {
2   ...
3   return bar(x-1);
4 }
```

Here, the invocation of `bar()` is the last operation performed by `foo()`. Thus, this is a tail call.

Tail calls have the advantage that a language or compiler can generally optimize the function call with respect to the stack frame. Since the function `foo()` is essentially done with its computation, its stack frame is no longer needed. The system, therefore, can reuse the stack frame. Tail recursion is such an important optimization, some languages require it or "guarantee" it in other ways.

11.2. Avoiding Recursion

Recursion is not essential; some languages do not even support recursion. In fact, any recursive function can be rewritten to not use recursion. Usually, you can write an equivalent loop structure or use an in-memory stack data structure to replace the recursion. So why use it? Proponents would argue that recursion allows you to write simple code that more closely matches mathematical functions and expressions. Recursion is also a natural way to think about certain problem solving techniques such as divide-and-conquer (see Chapter 12). It is also a natural way to code in functional programming languages.

These arguments, however, are *subjective*. One person's "cleaner" or "more understandable" code is another person's spaghetti code hack. What is "natural" for one person may be "weird" and "odd" for another. However, there are many other arguments against recursion, many of which are *objective* reasons: that recursion is more expensive and can easily lead to inefficient, exponential algorithms.

In general, recursion requires lots of function calls which requires creating and removing lots of stack frames. This usually results in a lot of overhead and resources being used to perform the computation. Unless you are using a language in which recursion is optimized and made to be more efficient (such as functional programming languages), this is a lot more expensive than using simple loops and iteration.

Another reason to avoid recursion is that it can lead to a lot of extraneous re-computations. The cliched example of the Fibonacci recursion is a prime example of this.[2] Consider the computation of FIBONACCI(5). This results in two recursive calls, each of those calls results in 2 recursive calls and so on as depicted in Figure 11.1.

As depicted in the figure, several function calls are repeated: FIBONACCI(3) is called twice, FIBONACCI(2) is called 3 times, etc. 15 total function calls are made to compute

[2] Its overuse as an example of recursion is even less explicable as it solves a problem that no one cares about.

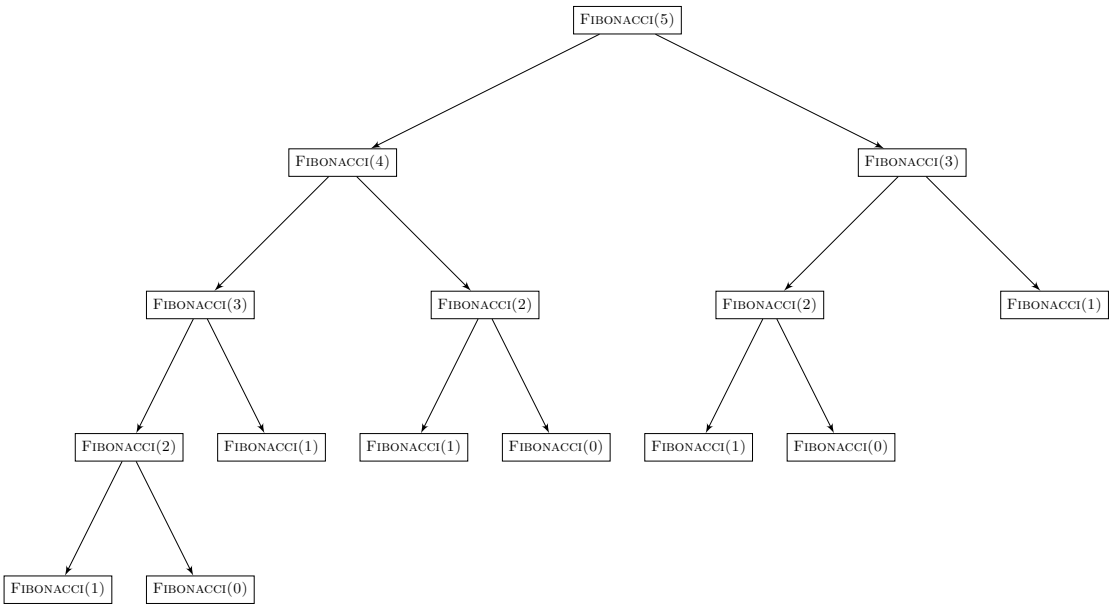


Figure 11.1.: Recursive Fibonacci Computation Tree

FIBONACCI(5). In general, the computation of FIBONACCI(n) will result in an *exponential* number of function calls. The number of function calls to compute F_n with this recursive solution will be equal to

$$F_{n-2} + \sum_{i=0}^{n-1} F_i$$

That is more than the first $n-1$ Fibonacci numbers combined! It should come as no surprise that the Fibonacci sequence grows *exponentially* and thus so would the number of operations with this recursive solution.

To put this in perspective, consider computing $F_{45} = 1,836,311,903$ ($n = 45$), the maximum representable value for a 32-bit signed two's complement integer. Executing a C implementation of this recursive algorithm took about 8 seconds[3] and required 3,672,623,805 function calls!

What if we wanted to compute $F_{100} = 573,147,844,013,817,084,101$ (573 quintillion) it would result in 1,146,295,688,027,634,168,201 (1.146 sextillion) function calls. Using the same hardware, at 4.59×10^8 (459 million) function calls per second, it would take 2.497×10^{12} seconds to compute. That would be more than 79,191 *years*! Even if we performed these (useless) calculations on hardware that was 1 million times faster than my laptop, it would still take over 4 *weeks*!

[3]On a 2.7GHz Intel Core i7.

11.2.1. Memoization

The inefficiency in the example above comes from the fact that we make the same function calls on the same values over and over. One way to avoid recomputing the same values is to *store* them into a table (or *tableau* if you prefer being fancy). Then, when you need to compute a value, you look at the table to see if it has already been computed. If it has, we reuse the value stored in the table, otherwise we compute it by making the appropriate recursive calls. Once computed, we place the value into the table so that it can be looked up on subsequent function calls. This approach is usually referred to as memoization.

The "table" in this scenario is very general: it can be achieved using a number of different data structures including simple arrays, or even maps (mapping input value(s) to output values). The table is essentially serving as a cache for the previously computed values. An illustration of how this might work can be found in Algorithm 11.3. Here, the recursion only occurs if the value F_n is not yet defined.

```
   INPUT  : An integer n ≥ 0, a global map M that maps n values to F_n
   OUTPUT : The n-th Fibonacci number, F_n
1  IF F_n is defined in M THEN
2  |  output M(n)
3  ELSE
4  |  a ← FIBONACCI(n − 1)
5  |  b ← FIBONACCI(n − 2)
6  |  Define M(n) = a + b
7  |  output (a + b)
8  END
```

Algorithm 11.3: Recursive FIBONACCI(n) Function With Memoization

In many functional programming languages, memoization is implicit and provided by the language itself to ensure that we do not have the same problems with recomputing values as we observed above. In languages such as C and Java, memoization becomes our responsibility and is not an optimization provided by the language itself.

However, if we are filling in a table of values anyway, we really don't need to make recursive calls at all. We simply need to figure out in what order to fill out the values in the table. This is actually the basis of a powerful programming technique known as dynamic programming which is a "bottom-up" approach to solving problems by combining solutions to smaller ones.

11.3. Exercises

Exercise 11.1. The binomial coefficients, $C(n, k)$ or $\binom{n}{k}$ ("n choose k"), are defined as the number of ways you can select k distinct items from a collection of n items. A direct combinatorial definition is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

An alternative is Pascal's identity, which gives a recurrence to compute this value:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Where $\binom{n}{0} = 1$ for any n and for all $k > n$, $\binom{n}{k} = 0$. Finally, $\binom{n}{1} = n$.

1. Write a recursive function using Pascal's identity to compute $\binom{n}{k}$. Benchmark its performance.
2. Write a recursive version that uses memoization to avoid recomputing values
3. Modify your functions to utilize an arbitrary precision numeric type so that you can compute arbitrarily large values.

Exercise 11.2. The Jacobsthal sequence is very similar to the Fibonacci sequence in that it is defined by its two previous terms. The difference is that the second term is multiplied by two.

$$J_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ J_{n-1} + 2J_{n-2} & \text{otherwise} \end{cases}$$

1. Write a recursive function that computes the n-th Jacobsthal number. Benchmark its performance.
2. Write a recursive version that uses memoization to avoid recomputing values
3. Modify your functions to utilize an arbitrary precision numeric type so that you can compute arbitrarily large values.

12. Searching & Sorting

Searching and sorting are two fundamental operations when dealing with collections of data. Both operations are not only important in and of themselves, but they also form the basis of many algorithms and other more complex operations. These operations are so essential that a wide variety of algorithms and techniques have been developed to solve them, each with their own advantages and disadvantages. This variety provides a good framework from which to study the relative efficiency and complexity of algorithms through algorithm analysis.

12.1. Searching

Searching is a very basic operation. Given a collection of data, we wish to find a particular element or elements that match a certain criteria. More formally, we have the following.

Problem 1 (Searching)**.**
Given: a collection of elements, $A = \{a_1, a_2, \ldots, a_n\}$ and a *key* element e_k
Output: The element a_i in A such that $a_i = e_k$

The "equality" in this problem statement is not explicitly specified. In fact, this is a very general, abstract statement of the basic search problem. We didn't specify that the "collection" was an array, a list, a set, or any other particular data structure. Nor did we specify what type of elements were in the collection. They could be numbers, they could be strings, they could be objects.

There are many variations of this general search problem that we could consider. For example, we could generalize it to find the "first" or "last" such element if our collection is ordered. We could find *all* elements that match our criteria. Some basic operations that we've already considered such as finding the minimum or maximum (*extremal* elements), or median element are also variations on this search problem.

When designing a solution to any of these variations additional considerations must be made. We may wish our search to be index-based (that is, output the index i rather than the element a_i). We may need to think about how to handle *unsuccessful* searches (return `null`? A special flag value? Throw an exception?, etc.).

index	0	1	2	3	4	5	6	7	8
contents	42	4	9	4	102	34	12	2	0

Figure 12.1.: Array of Integers

When implementing a solution in a programming language, we of course will need to be more specific about the type of collection being searched and the type of elements in the collection. However, we will still want to keep our solution as general as possible. As we'll see, most programming languages facilitate some sort of *generic* programming so that we do not need to reimplement the solution for *each* type of collection or for *each* type of variable. Instead, we can write one solution, then *configure* it to allow for comparisons of any type of variable (numeric, string, object, etc.).

12.1.1. Linear Search

The first solution that we'll look at is the *linear search* algorithm (also known as sequential search). This is a basic, straightforward solution to the search problem that works by simply iterating through each element a_i, testing for equality, and outputting the first element that matches the criteria. The pseudocode is presented as Algorithm 12.1.

```
   INPUT  : A collection of elements A = {a_1, ..., a_n} and a key e_k
   OUTPUT : An element a in A such that a = e_k according to some criteria; φ if no
            such element exists
1  FOREACH a_i in the collection A DO
2  |  IF a_i = e_k THEN
3  |  |  output a_i
4  |  END
5  END
6  output φ
```

Algorithm 12.1: Linear Search

To illustrate, consider the following example searches. Suppose we wish to search the 0-indexed array of integers in Figure 12.1.

A search for the key $e_k = 102$ would start at the first element. $42 \neq 102$ so the search would continue; it would compare it against 4, then 9, then 5, and finally find 102 at index $i = 4$, making a total of 5 comparisons (including the final comparison to the matched element).

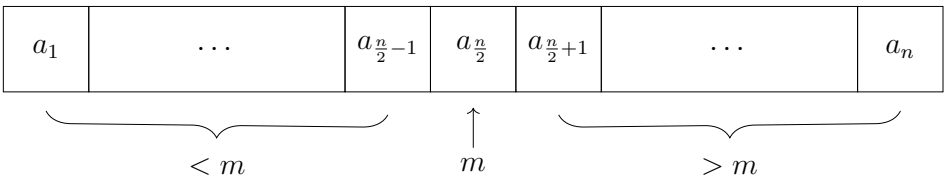


Figure 12.2.: When an array is sorted, all elements in the left half are less than the middle element m, all elements in the right half are greater than m.

A search for the key $e_k = 42$ would get lucky. It would find it after only one comparison as the first element is a match. A search for the element 20 would result in an unsuccessful search with a total of 10 comparisons being made. Finally a search for $e_k = 4$ would only require two comparisons as we find 4 at the second index. There is a duplicate element at index 3, but the way we've defined linear search is to find the "first" such element. Again, we could design any number of variations on this solution. We give a more detailed analysis of this algorithm below.

12.1.2. Binary Search

An alternative search algorithm is *binary search*. This is a clever algorithm that requires that the array being searched is *sorted* in ascending order. Though it works on any type of data, let's again use an integer array as an example. Suppose we're searching for the key element e_k. We start by looking at the element in the *middle* of the array, call it m.

Since the array is sorted, everything in the left-half of the array is $< m$ and everything in the right-half of the array is $> m$.[1] We will now make one comparison between e_k and m. There are three cases to consider.

1. If $e_k = m$, then we've found *an* element that matches our key and search criteria and we are done. We can output m and stop the algorithm.

2. If $e_k < m$ then we know that if a matching element exists, it must lie in the left-half of the list. This is because all elements in the right-half are $> m$.

3. If $e_k > m$ then we know that if a matching element exists, it must lie in the right-half of the list. This is because all elements in the left-half are $< m$.

In either of the second two cases, we have essentially cut the array in half, halving the number of elements we need to consider. Suppose that the second case applies. Then we can consider elements indexed from 1 to $\frac{n}{2} - 1$ (we need not consider $a_{\frac{n}{2}}$ as the first case would have applied if we found a match). We can then do the same trick: check the middle element among the remaining elements and determine which half to cutout and

[1] If duplicate elements are in the array, then elements in the left/right half *could* be less than or equal to and greater than or equal to m, but this will not affect how our algorithm works.

which half to consider. We repeat this process until we've either found the element we are looking for or the range in which we are searching becomes "empty" indicating an unsuccessful search.

This description suggests a recursive solution. Given two indices l, r, we can compute the index of the middle element, $m = \frac{l+r}{2}$ and make one of two recursive calls depending on the cases identified above. Of course, we will need to make sure that our base case is taken care of: if the two indices are *invalid*, that is if the left is greater than the right, $l > r$, then we know that the search was unsuccessful.

```
INPUT  : A sorted collection of elements A = {a_1, ..., a_n}, bounds 1 ≤ l, r ≤ n,
         and a key e_k
OUTPUT : An element a in A such that a = e_k according to some criteria; φ if no
         such element exists
1  IF l > r THEN
2  |  output φ
3  END
4  m ← ⌊(l+r)/2⌋
5  IF a_m = e_k THEN
6  |  output a_m
7  ELSE IF a_m < e_k THEN
8  |  BINARYSEARCH(A, m + 1, r, e_k)
9  ELSE
10 |  BINARYSEARCH(A, l, m − 1, e_k)
11 END
```

Algorithm 12.2: Recursive Binary Search Algorithm, BINARYSEARCH(A, l, r, e_k)

As discussed in Chapter 11, non-recursive solutions are generally better than recursive ones. We can design a straightforward iterative version of binary search using a while loop. We initialize two index variables, l, r and update them on each iteration depending on the three cases above. The loop stops when we've found our element or $l > r$ resulting in an unsuccessful search. The iterative version is presented in Algorithm 12.3, an example

run of the algorithm is shown in Figure 12.3.

```
INPUT  : A sorted collection of elements A = {a_1, ..., a_n} and a key e_k
OUTPUT : An element a ∈ A such that a = e_k according to some criteria; φ if no
         such element exists
1  l ← 1
2  r ← n
3  WHILE l ≤ r DO
4  |  m ← ⌊(l+r)/2⌋
5  |  IF a_m = e_k THEN
6  |  |  output a_m
7  |  ELSE IF a_m < e_k THEN
8  |  |  l ← (m + 1)
9  |  ELSE
10 |  |  r ← (m − 1)
11 |  END
12 END
13 output φ
```

Algorithm 12.3: Iterative Binary Search Algorithm, BINARYSEARCH(A, e_k)

12.1.3. Analysis

When algorithms are implemented and run on a computer, they require a certain amount of *resources*. In general, we could consider a lot of different resources such as computation time and memory. Algorithm analysis involves quantifying how many resource(s) an algorithm requires to execute with respect to the size of the input it is run on.

When analyzing algorithms, we want to keep the analysis as abstract and general as possible, independent of any particular language, framework or hardware. We could always update the hardware on which we run our implementation, but that does not necessarily make the *algorithm* faster. The algorithm would still execute the same *number* of operations. Faster machines just mean that more steps can be performed in less time. In fact, the concept of an algorithm itself is a mathematical concept that predates modern computers by thousands of years. One of the oldest algorithms, for example, Euler's GCD (greatest common divisor) algorithm dates to 300 BCE. Whether or not you're "running" it on a piece of papyrus 2,300 years ago or on a modern supercomputer, the same number of divisions and subtractions are performed.

To keep things abstract, we analyze an algorithm by identifying an *elementary operation*.

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

(a) Initially, $l = 0, r = 10$ and so we examine the middle element at index $m = 5$ which is 12.

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

(b) Since $64 > 12$, we update our left index variable l to $m + 1$, thus $l = 6$ and we've eliminated the left half of the list from consideration.

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

(c) Our new middle index is $m = \frac{l+r}{2} = \frac{6+10}{2} = 8$, corresponding to the element 102.

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

(d) Since $64 < 102$, we update the right index variable r to $m - 1 = 7$, eliminating the right half of the subarray.

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

(e) Here, $l = 6, r = 7$, and so our new middle index is $m = \lfloor \frac{6+7}{2} \rfloor = 6$. Since $64 > 34$, we update our left index variable l to $m + 1 = 7$

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

(f) Since $64 > 42$ we again update our left index variable l to $m + 1 = 8$.

index	0	1	2	3	4	5	6	7	8	9	10
contents	-3	2	4	4	9	12	34	42	102	157	180

(g) Since $l = 8$ and $r = 7$, $l > r$ and the loop terminates, resulting in an unsuccessful search.

Figure 12.3.: The worst case scenario for binary search, resulting in an unsuccessful search for $e_k = 64$. This example is run on a 0-indexed array with an array of integers of size 11.

This is generally the most common or most "expensive" operation that the algorithm performs. Sometimes there may be more than one reasonable choice for an elementary operation which may give different results in our analysis. However, we generally do *not* consider basic operations that are necessary to the *control flow* of an algorithm. For example, variable assignments or the iteration of index variables.

Once we have identified an elementary operation, we can quantify the complexity of an algorithm by analyzing the number of times the elementary operation is executed with respect to the *input size*. For a collection, the input size is generally the number of elements in the collection, n. We can then characterize the number of elementary operations and thus the complexity of the algorithm itself as a *function* of the input size. We illustrate this process by analyzing and comparing the two search algorithms.

Linear Search Analysis

When considering the linear search algorithm, the input size is clearly the number of elements in the collection, n. The best candidate for the elementary operation is the comparison (Line 2, Algorithm 12.1). To analyze this algorithm, we need to determine how many comparisons are made with respect to the size of the collection, n.

As we saw in the examples, the number of comparisons made by linear search can vary depending on the element we're searching and the configuration of the collection being searched. Because of this variability, we can analyze the algorithm in one of three ways: by looking at the best case scenario, worst case scenario, and average case scenario.

The best case scenario is when the number of operations is *minimized*. For linear search, the best case scenario happens when we get lucky and the first element that we examine matches our criteria, requiring only a single comparison operation. In general, it is not reasonable to assume that the best case scenario will be commonly encountered.

The worst case scenario is when the number of operations is *maximized*. This happens when we get "unlucky" and have to search the entire collection finding a match at the last element or not finding a match at all. In either case, we make n comparisons to search the collection.

A formal average case analysis is not difficult, but is a bit beyond the scope of the present analysis. However, informally, we could expect to make about $\frac{n}{2}$ comparisons for successful searches if we assume that all elements have a uniform probability of being searched for.

Both the worst-case and average-case are reasonable scenarios from which to analyze the linear search algorithm. In the end, however, the only difference between the two analyses is a constant factor. Both analyses result in two linear functions,

$$f_1(n) = n \quad f_2(n) = \frac{1}{2}n$$

The only difference being the constant factor $\frac{1}{2}$. In fact, this is why the algorithm is called *linear* search. The number of comparison operations performed by the algorithm *grows linearly* with respect to the input size. For example, if we were to double the input size from n to $2n$, then we would expect the number of comparisons to search the collection would also double (this applies in either the worst-case and average-case scenarios). This is what is most important in algorithm analysis: quantifying the complexity of an algorithm by the rate of growth of the operations (and thus resources) required to execute the algorithm as the input size n grows.[2]

Binary Search Analysis

Like linear search, the input size is the size of the collection n and the elementary operation is the comparison. As presented in the pseudocode, binary search would seem to perform *two* comparisons (Lines 5 and 7 in Algorithm 12.3). However, to make the analysis simpler, we will instead count one comparison operation per iteration of the while loop. This is a reasonable simplification; in practice the comparison operation would likely involve a single function call, after which distinguishing between the three cases is a simple matter of control flow. Further, even if we were to consider both comparisons, it would only contribute a constant factor of 2 to the final analysis.

Since we perform one comparison for each iteration of the while loop, we need to determine how many times the while loop executes. In the worst case, the number of iterations is maximized when we fail to find an element that does not match our criteria. However each iteration essentially cuts the array in half each time. That is, if we start with an array of size n, then after the first iteration, it is of size $\frac{n}{2}$. After the second iteration we have cut it in half again, so it is of size $\frac{n}{4}$, after the third iteration it is of size $\frac{n}{8}$ and so on. More generally, after k iterations, the size of the array is

$$\frac{n}{2^k}$$

The loop terminates one iteration *after* the the index variables are equal, $l = r$. Equivalently, when $l = r$, the size of the subarray under consideration is 1. That is, the algorithm stops when the array size has been cut down to 1:

$$\frac{n}{2^k} = 1$$

Solving for k gives us the number of iterations:

$$k = \log_2(n)$$

Adding one additional comparison for the final iteration gives us a total of

$$k = \log_2(n)$$

[2] This is the basis of Big-O analysis, something that we will not discuss in detail here, but is of prime importance when analyzing algorithms.

comparisons (see Table 12.1). Thus, binary search performs a logarithmic number of comparisons in the worst case. As we will see, this is *exponentially* better than linear search.

Iteration	Array Size	Comparisons
1	$\frac{n}{2}$	1
2	$\frac{n}{4}$	1
3	$\frac{n}{8}$	1
4	$\frac{n}{16}$	1
$\vdots$	$\vdots$	$\vdots$
k	$\frac{n}{2^k}$	1
$\vdots$	$\vdots$	$\vdots$
$\log_2(n)$	1	1
$\log_2(n)+1$	0	1
Total		$\log_2(n)+1$

Table 12.1.: Number of comparisons and array size after the iteration during the execution of binary search.

Comparative Analysis

Binary search presents a clear advantage over linear search. There is an *exponential* difference between a linear function, $\frac{n}{2}$ and a logarithmic function, $\log_2(n)$. To put this in perspective, consider searching a *moderately* large database of 1 trillion (10^{12}) records.[3] Using linear search, even the average-case scenario would require about

$$\frac{10^{12}}{2} = 5 \times 10^{11}$$

or about 500 billion comparisons. However, using binary search would only require at most

$$\log_2(10^{12}) = 12 \cdot \log_2(10) < 40$$

comparisons to search. This is a *huge* difference in performance.

As another comparison, let's consider how each algorithm's complexity grows as we increase the size of the collection being searched. As observed earlier, if we double the

[3] In the era of "big data," 1 trillion records only qualifies as *moderately* large.

input size, $n \to 2n$, we would expect the number of comparisons performed by linear search to also double. However, if we double the input size for binary search, we get the following.

$$\log_2(2n) = \log_2(2) + \log(n) = \log(n) + 1$$

That is, only a single additional comparison is necessary to search an array of twice the size.

The difference between these two algorithms shows up in many different instances. Figure 12.4 contains a screen shot of the search feature in Windows 7. When searching for particular files or content in particular files, the search can be greatly increased if the files have been *indexed*, that is, sorted. As the dialog indicates, non-indexed (unsorted) records will take far longer to search.

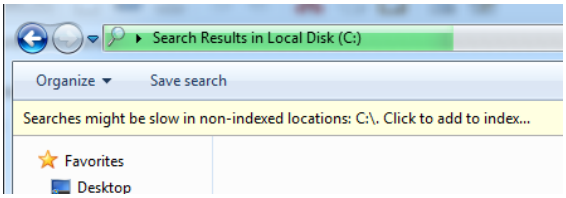


Figure 12.4.: Example of the benefit of ordered (indexed) elements in Windows 7

Though binary search presents a clear advantage over linear search, it only works if the collection has been sorted.[4] Thus, we now turn our attention to the problem of sorting a collection.

12.2. Sorting

Sorting a collection of data is another fundamental data operation. It is conceptually simple, but is ubiquitous. There are a large variety of algorithms, data structures and applications built around the problem of sorting. As we've already seen, being able to sort a collection provides a huge speed up when searching for a particular element. Sorting provides a natural way to store and organize data.

Problem 2 (Sorting)**.**
Given: a collection of *orderable* elements, $A = \{a_1, a_2, \ldots, a_n\}$
Output: A, sorted in ascending order

The requirement that the collection be made of "orderable" elements can be a bit technical[5], but essentially we need to be guaranteed that given two elements, a, b in the

[4] Binary search also only works when searching an array with random access to its elements. The performance of binary search cannot generally be realized with data structures such as linked lists or unordered sets.

[5] We require that A be a *total order* a partially ordered binary relation such that all pairs are comparable.

collection, we can determine whether $a < b$, $a = b$ or $a > b$. If such a determination cannot be made, then sorting is impossible.

Again, we can consider variations on this problem. We may want our collection to be sorted in descending order instead of ascending.[6] We may also want the collection itself to be *permuted* (that is, reordered) or we may instead want a *copy* of the collection to be created and sorted so that the original is unchanged.

We will examine several standard sorting algorithms. Though there are dozens of sorting algorithms, we will only focus on a few of the more common ones. As with searching, we can analyze a sorting algorithm based on the number of comparisons it makes in the worst, best, or average cases. We may also look at alternative resources or operations: how many swaps does the algorithm make? How much extra memory is required? Etc.

Though we will examine, analyze and compare several sorting algorithms, most programming languages provide standard functionality to sort a collection of elements. It is generally preferable to use the functionality built into whatever language you're using rather that reimplementing your own. Typically, these functions are well-designed, well-tested, optimized and more efficient than any custom alternatives.

12.2.1. Selection Sort

The first sorting algorithm we'll examine is *Selection Sort* which is similar to the way a human would sort a collection of objects. The basic idea is that you search through the collection and find the minimal element (we say minimal and not minimum because there could be elements with the same value). Once we've found a minimal element, we can swap it with the "first" element, a_1, placing the minimal element where it belongs in the collection.

We repeat this process for the remaining elements, $a_2, \ldots, a_n$, finding the minimal element among these and swapping with a_2. In general, if we have already sorted the first $i - 1$ elements, $a_1, \ldots, a_{i-1}$ then we only need to examine the elements $a_i, \ldots, a_n$ for the minimal element, swapping it with a_i. We end this process after we have sorted the first $n - 1$ elements, $a_1, \ldots, a_{n-1}$ since the last element a_n will already be where it needs to be. We present Selection Sort as Algorithm 12.4. We illustrate the execution of Selection

[6] Technically, these are referred to as *non-decreasing* and *non-increasing* respectively. This is because the collection could contain duplicate elements and not lead to a strictly increasing or strictly decreasing ordering.

Sort in Figure 12.5.

```
   INPUT  : A collection A = {a_1, ..., a_n}
   OUTPUT : An array A' containing all elements of A in nondecreasing order
1  FOR i = 1, ..., (n − 1) DO
2  |  a_min ← a_i
3  |  FOR j = (i + 1), ..., n DO
4  |  |  IF a_min > a_j THEN
5  |  |  |  min ← a_j
6  |  |  END
7  |  END
8  |  swap a_min and a_i
9  END
```

Algorithm 12.4: Selection Sort

Analysis

We now analyze Selection Sort to determine how complex it is. First, the elementary operation is the comparison on line 4. We need to determine how many times this line is executed with respect to the size of the input, n. Observe that on the i-th iteration, the first $i-1$ elements are sorted and we need not make any comparisons among them. We start by assuming that the a_i is the minimum element and compare it to the remaining $n-i$ elements, requiring $n-i$ comparisons, to find the minimal element. For example, in the first iteration we make $n-1$ comparisons, the second we make $n-2$, etc. The last iteration we make only 1 comparison. Totaling these all up gives us

$$(n-1)+(n-2)+(n-3)+\cdots+3+2+1$$

Which can be rewritten as

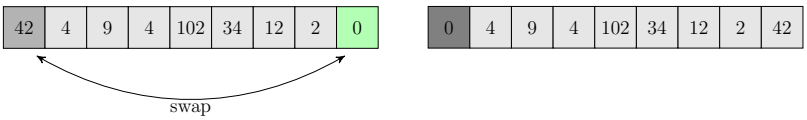
$$\sum_{i=1}^{n-1} i = 1+2+3+\cdots+(n-2)+(n-1)$$

We can use Gauss's Formula to solve this summation.

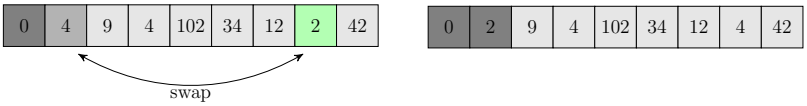
Theorem 1 (Gauss's Formula)**.** The sum of integers 1 up to n can be written as follows.

$$\sum_{i=1}^{n} i = 1+2+3+\cdots+(n-1)+n = \frac{n(n+1)}{2}$$

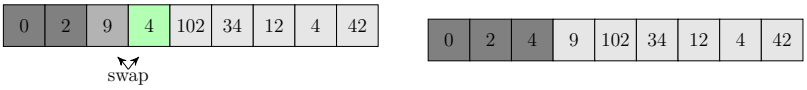
In Selection Sort, the number of comparisons doesn't sum up to n, only $n-1$. Substituting this in gives us



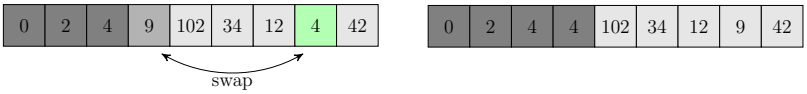
(a) First iteration. We find the minimal element, 0, at the last index, swapping it with the first element. At this point, the first element is sorted.



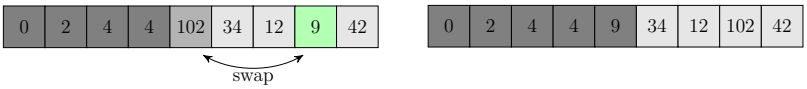
(b) Second Iteration. Now starting with the second element, the minimal element among the remaining is found at the second to last element. 4 and 2 are swapped. At this point, the first two elements are sorted.



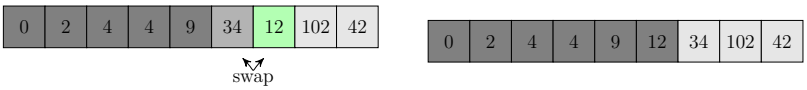
(c) Third iteration. Since we are using the strictly-less than comparison, the first 4 is the minimal element and swapped with 9.



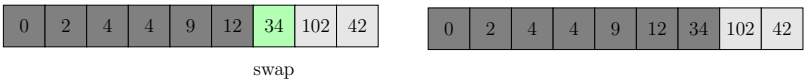
(d) Fourth iteration. At this point, the first 3 elements are sorted. We find the minimal element (the other 4) and swap it with 9. At the end of this iteration, the first 4 elements are sorted.



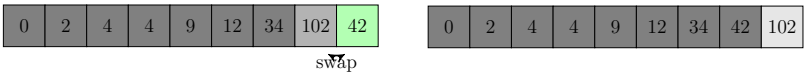
(e) Fifth iteration. The 9 is swapped with 102, sorting the first 5 elements.



(f) Sixth iteration. 12 is swapped with 34.



(g) Seventh iteration. 34 ends up being the minimal element and we essentially swap it with itself. Even though the "current" element was also the minimal element, we still had to compare the current element with all other elements; in this case we made 2 comparisons.



(h) Eighth iteration. This is the final iteration, we swap 42 and 102. After this iteration, the final element, 102 is already where it belongs.

Figure 12.5.: Example execution of Selection Sort.

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Another way to analyze the code is to count the number of comparisons with respect to the for loop index variables. In particular, there is one comparison made on line 4. Line 4 itself is executed once for each time the inner for loop on line 3 executes which executes for j running from $i+1$ up to n. Line 3 and the entire inner for loop executes once for each time the outer for loop executes, that is for i running from 1 up to $n-1$. This gives us the following summation.

$$\underbrace{\sum_{i=1}^{n-1} \underbrace{\sum_{j=i+1}^{n} \underbrace{1}_{\text{line 4}}}_{\text{line 3}}}_{\text{line 1}}$$

Solving this summation gives us:

$$\begin{aligned}
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n-1} n - i \\
&= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\
&= n(n-1) - \frac{n(n-1)}{2} \\
&= \frac{n(n-1)}{2}
\end{aligned}$$

This illustrates that Selection Sort is a *quadratic* sorting algorithm, requiring roughly n^2 comparisons to sort an array of n elements. We analyze this further below.

12.2.2. Insertion Sort

Another basic sorting algorithm is *Insertion Sort* which works slightly differently than Selection Sort. Rather than finding a minimal element and swapping with the "current" element, the current element is *inserted* in the list where it belongs.

If we consider just the first element, the collection is sorted by definition. Now consider the second element: either its greater than the first element and so is already sorted, or it is less than the first element and needs to be swapped. In either case, the first

two elements are now sorted. If we continue this, then on the i-th iteration, the first i elements, $a_1, \ldots, a_i$ are sorted (just as with Selection Sort). Now consider the $(i+1)$-th element: we will *insert* it amongst the elements $a_1, \ldots, a_i$ where it needs to be.

We insert the "current" element, a_{i+1} by comparing it to a_i: if $a_i > a_{i+1}$, then we swap them. We keep comparing the current element to the one immediately to its left, shifting the current element down the collection until we find an element that is less than or equal to the current element at which point we stop. Now the elements $a_1, \ldots, a_{i+1}$ are now sorted. In contrast to Selection Sort, we *do* need to process the last element as it might not be where it needs to be. The full pseudocode is presented in Algorithm 12.5. An example of an execution of Insertion Sort can be found in Figure12.6.

```
   INPUT  : A collection A = {a_1, ..., a_n}
   OUTPUT : An array A' containing all elements of A in nondecreasing order
1  FOR i = 2, ..., n DO
2  |   x ← a_i
3  |   j ← i
4  |   WHILE j > 1 and a_{j-1} > x DO
5  |   |   a_j ← a_{j-1}
6  |   |   decrement j
7  |   END
8  |   a_j ← x
9  END
```

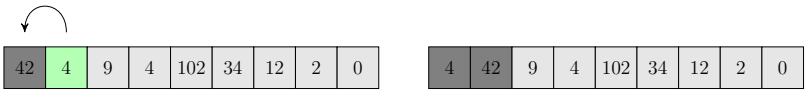
Algorithm 12.5: Insertion Sort

Analysis

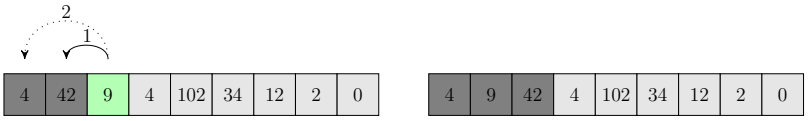
As we can see in the example run of Insertion Sort, not every iteration needs to make comparisons with all elements in the sorted part of the collection. For example, in iteration 4 we only had to make one comparison and we were done. In other iterations such as the last two, we had to make comparisons to *every* element in the sorted part of the collection. This illustrates that Insertion Sort is adaptive and may have a different complexity depending on the structure of the collection it sorts.

Let's consider the best case in which the number of comparisons is minimized. Suppose, for example, we ran Insertion Sort on a collection that was already sorted. Each iteration would only need one comparison to determine that the element was already where it needed to be. As there are only $n-1$ iterations, in the best case, Insertion Sort makes $n-1$ comparisons.

In contrast, the worst case would occur if the list was already sorted, but in reverse order.



(a) First iteration. We insert 4 in front of 42, requiring 1 comparison.



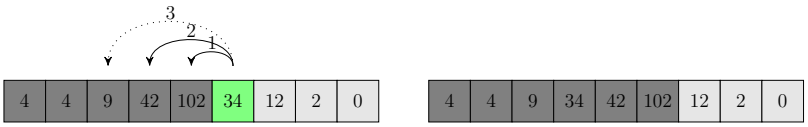
(b) Second iteration. The first two elements are sorted, we insert 9 by making two comparisons: to find that it is less than 42, but greater than 4. At the end of the iteration, the first three elements are sorted.



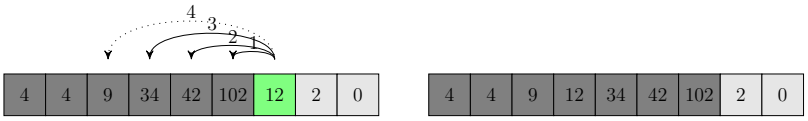
(c) Third iteration. The first three elements are sorted, we insert the second four by making 3 comparisons.



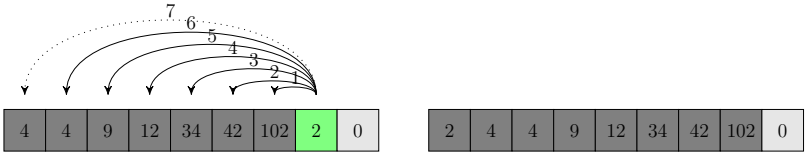
(d) Fourth iteration. Here, only one comparison is necessary to find that 102 is already sorted.



(e) Fifth iteration. Here, 3 comparisons are necessary to insert 34 between 9 and 42.



(f) Sixth iteration. Here, 4 comparisons are necessary to insert 12 between 9 and 34.



(g) Seventh iteration. Here, the number of comparisons is maximized as we shift 2 all the way to the front of the array.

Figure 12.6.: Example execution of Insertion Sort. Each iteration depicts the comparisons to previous elements; the last comparison is dashed indicating a comparison was made, but not a swap. The final iteration is omitted for space, but would require 8 comparisons to insert 0 at the front of the collection.

The i-th iteration would require i comparisons to move the current element all the way to the front of the collection. Again, this gives us a summation:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

matching the complexity of Selection Sort.

We could also analyze Insertion Sort with respect to average case. On average, we would expect to make about $\frac{1}{2}$ of the total possible comparisons on each iteration. That is,

$$\sum_{i=1}^{n-1} \frac{i}{2}$$

Moving the constant outside the summation and applying Gauss's Formula would give us an expected number of comparisons to be

$$\frac{n(n-1)}{4}$$

This is still a quadratic function, but the constant involved and the fact that Insertion Sort is more adaptive to the input make Insertion Sort a much better algorithm in practice than Selection Sort. In fact, Insertion Sort is very efficient on "small" arrays in practice and is used in many hybrid algorithm implementations (see Section 12.2.5).

12.2.3. Quick Sort

Selection Sort and Insertion Sort are simple algorithms, but inefficient, making a quadratic number of comparisons in even the average case. A more sophisticated and efficient algorithm is Quick Sort, first developed by Tony Hoare in the early 1960s [17, 18]. Quick Sort is a divide-and-conquer style algorithm that attempts to solve the sorting problem by splitting a collection up into two parts, then *recursively* sorting each part.

The basic idea is as follows. First, choose a *pivot* element p in the collection. We will then *partition* all the elements in the collection around this pivot element by placing all elements less than p in the "left" partition and all elements greater than p in the "right" partition. This concept is similar to binary search. After partitioning all the elements, we place p between them so that p is where it needs to be. We then repeat this process on the two partitions recursively. The recursion stops when the size of the sub-collection is trivially sorted (it is empty or consists of a single element).

Quick Sort is presented as Algorithm 12.6 with a partitioning subroutine presented in Algorithm 12.7. The partitioning chooses the first element in the sub-collection as the pivot element. Further, the partitioning is done in-place: we maintain two index variables,

i, j and increment/decrement them respectively to find a pair that are both in the wrong partitions, swapping them. The partitioning continues until the two index variables meet each other. As a final step, the pivot element is placed between the two partitions and the index at which it is placed is returned to the main QuickSort routine so that it can make two recursive calls on the two partitions.

We depict a few examples of the partitioning algorithm. Figure 12.7 depicts the first partitioning on the same example array while Figures 12.8 and 12.9 depict subsequent

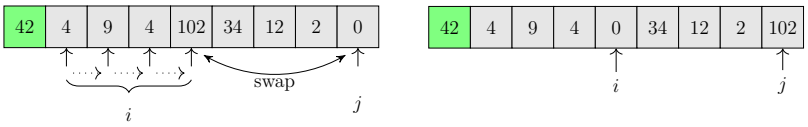
partitioning operations on subarrays as part of the recursion.

```
INPUT  : A collection A = {a_1, ..., a_n}, indices l, r
OUTPUT : A, sorted in ascending order
IF l < r THEN
    p ← PARTITION(A, l, r)
    QUICKSORT(A, l, p − 1)
    QUICKSORT(A, p + 1, r)
END
```

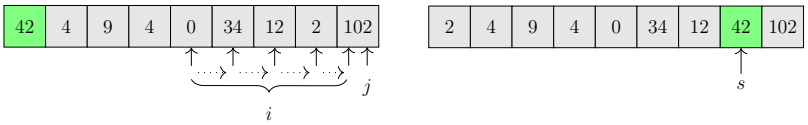
Algorithm 12.6: QUICKSORT

```
INPUT  : A collection A = {a_1, ..., a_n}, indices l, r
OUTPUT : An index s such that the sub-collection A from l to r has been
         partitioned around s so that all elements a_l, ... a_{s−1} are less than a_s
         and all elements a_{s+1}, ..., a_r are greater than a_s
pivot ← a_l
i ← (l + 1)
j ← r
WHILE i < j DO
    WHILE i < j AND a_i ≤ pivot DO
        i ← (i + 1)
    END
    WHILE i < j AND a_j ≥ pivot DO
        j ← (j − 1)
    END
    swap a_i, a_j
END
//Swap the pivot
IF a_i ≤ pivot THEN
    swap pivot, a_i
    output i
ELSE
    swap pivot, a_{i−1}
    output (i − 1)
END
```

Algorithm 12.7: In-Place PARTITION

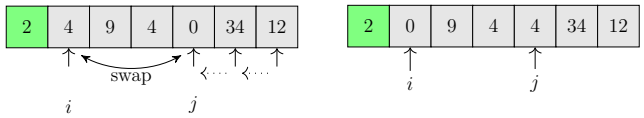


(a) First iteration. 42 is chosen as the pivot element. The index variable i moves over to 102, the first element that is greater than 42 and on the "wrong" side of the partition. The j index variable does not move as 0 is less than the pivot element and on the wrong side; these are swapped.

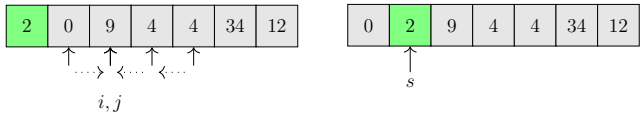


(b) Second iteration. The index variable i moves all the way to the right as all remaining elements are less than the pivot, 42. The index variable j remains at 102 as i is now equal to j. The pivot, 42 is placed between the two partitions and its resulting index, s is returned.

Figure 12.7.: Example execution of the PARTITION subroutine in Quick Sort. A total of 8 comparisons are made, 9 if you count the last swap outside the while loop. The partition returns s as the pivot position; the right-partition is sorted as it only consists of 1 element.

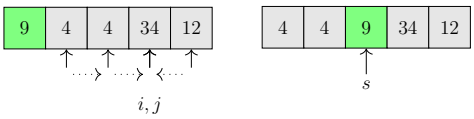


(a) First iteration. In this partitioning, 2 is the pivot element. The index variable i is not incremented as 4 is larger than the pivot. The index variable j decrements to 0 and they are swapped.



(b) Second iteration. The index variable i is incremented once, while j is decremented to meet it. After the while loop the second case applies as $a_i = 9 > 2 = pivot$, and so we swap $a_{i-1} = 2$ with the pivot.

Figure 12.8.: Example execution of the PARTITION subroutine in Quick Sort on the first recursive call on the left partition. A total of 6 comparisons are made, 7 if you count the last swap outside the while loop. The partition returns s as the pivot position; in this case the left-partition is sorted as it only consists of 1 element.



(a) First (and only) iteration. In this partitioning, 9 is the pivot. The index variable i is incremented to 34 while j decrements to match. 34 is swapped with itself. After this iteration, the second condition applies and the pivot is swapped with $a_{i-1} = 4$

Figure 12.9.: Example execution of the PARTITION subroutine in Quick Sort on the second recursive call on the right partition. A total of 5 comparisons are made, 6 if you count the last swap outside the while loop. The partition returns s as the pivot position.

Analysis

We can easily analyze how many comparisons are made by the PARTITION subroutine. Suppose that we are given a (sub)array of n elements. Since the pivot element must be compared to every other element in the (sub)array, it must make $n - 1$ comparisons to partition the elements around the pivot. If we count the last comparison to determine where to place the pivot, it would be n comparisons (but the difference is trivial).

The analysis of Quick Sort itself is a bit more involved and is highly dependent on how well the PARTITION subroutine splits the array. In the worst case, our pivot choice will always partition the array into one "empty" subarray and one subarray with $n - 1$ elements (we do not count the pivot element). One such example of this would be if our collection is already sorted. In this case, one of the recursive calls would result in no comparisons and the other would result in $n - 1$ comparisons. This lopsided recursion would result in the following number of comparisons:

$$n + (n - 1) + (n - 2) + \cdots + 3 + 2 + 1$$

which is exactly Gauss's Formula, meaning that Quick Sort would perform

$$\frac{n(n + 1)}{2}$$

in the worst case.

However, in the sorting scenario, we cannot always assume the worst case. If we are given random data, then the likelihood that we will always have such an extremely lopsided partitioning is extremely small. A more reasonable analysis would involve the average case: where each partition is roughly an equal size, *about* $\frac{n}{2}$. This happens when our pivot choice is the median (or *close* to the median) element.

To analyze the number of comparisons in this case, we can setup a *recurrence relation*:

$$C(n) = 2C\left(\frac{n}{2}\right) + n$$

Here, $C(n)$ represents the number of comparisons made by Quick Sort on an array of size n. The first term on the right hand side represents the fact that we make 2 recursive calls on subarrays of size (roughly) $\frac{n}{2}$. The second term captures the number of comparisons we make in the PARTITION subroutine. Solving this recurrence is a bit beyond the scope of the current discussion. However, it can be shown that this is roughly equivalent to

$$n \log(n)$$

comparisons in the best case.

The difference between the best and worst case scenarios is in our pivot choice. Though the ideal pivot choice is the median element, to find the median the usual strategy is to sort the collection first. This puts the cart before the horse. If the data we are sorting is in more-or-less random order, then the choice of the first element as a pivot is good enough as it is unlikely that we will always have a bad paritioning. However, there are other strategies.

One strategy is to randomly choose a pivot element. A random choice would make the worst-case scenario very unlikely over many runs of the algorithm. Another strategy is to choose the median of three elements (either fixed or randomized), ensuring that neither partition will ever be empty. This strategy can be taken further to find the median-of-three amongst each third of the array and then take the median of these (or the "ninther"). The more effort you put into finding the median the more the performance degrades in practice.[7] A more sophisticated analysis of these strategies yields a complexity similar to the best case [9].

12.2.4. Merge Sort

Another "fast" sorting algorithm is Merge Sort, due to John von Neumann, 1945 (as reported by Knuth [27]). The performance is similar to Quick Sort's best/average case, making

$$n \log(n)$$

comparisons. However, Merge Sort's performance does not depend on the structure of the input array or a pivot choice, guaranteeing this performance in the best/average/worst case.[8]

[7] For example, there does exist a median-finding algorithm that runs in linear time which would guarantee the theoretically best running time, but the algorithm is recursive and has a large overhead, making its use slow in practice.

[8] Still, Quick Sort is still more common in practice because it has less memory requirements and the pivot choice strategies can mitigate the risk of the worst-case scenario.

Merge Sort works by *first* dividing the list into two (roughly) equal partitions. It then recursively sorts each partition. The recursion stops when the subarray is of size ≤ 1 just as with Quick Sort. The difference, however, is what Merge Sort does after the recursion. After having sorted the left partition, L and the right partition R, Merge Sort *merges* the two sorted partitions into one.

The MERGE subroutine works by maintaining two index variables, i, j, one for each partition. Suppose that the index variables correspond to the elements L_i and R_j in the left/right partition. If $L_i \leq R_j$, we add L_i to the end of a temporary array and increment i. Otherwise we place R_j into the temporary array and increment j. We continue until we have examined every element in one or both partitions (if one partition still has elements in it, we can simply copy the rest over in order). This merge operation works because each subarray is sorted.

Merge Sort is presented as Algorithm 12.8 with the MERGE subroutine as Algorithm 12.9.

```
   INPUT  : A (sub)collection A = {a_l, ..., a_r}, indices l, r
   OUTPUT : A collection A' which is A sorted
1  IF l < r THEN
2  |  m ← ⌊(l+r)/2⌋
3  |  L ← MERGESORT(A, l, m)
4  |  R ← MERGESORT(A, m+1, r)
5  |  output MERGE(L, R)
6  ELSE
7  |  output A
8  END
```

Algorithm 12.8: MERGESORT

```
   INPUT  : Two sorted collections, L, R of size n, m respectively.
   OUTPUT : A sorted collection A consisting of all elements of L and R
 1 A ← a new, empty collection
 2 i ← 1
 3 j ← 1
 4 k ← 1 //index variable for A
 5
 6 WHILE i ≤ n AND j ≤ m DO
 7 |   IF L_i ≤ R_i THEN
 8 |   |   A_k ← L_i
 9 |   |   i ← (i + 1)
10 |   ELSE
11 |   |   A_k ← L_j
12 |   |   j ← (j + 1)
13 |   END
14 |   k ← (k + 1)
15 END
   //At least one collection is empty, we can blindly copy the other
16 WHILE i ≤ n DO
17 |   A_k ← L_i
18 |   i ← (i + 1)
19 |   k ← (k + 1)
20 END
21 WHILE j ≤ m DO
22 |   A_k ← R_j
23 |   j ← (j + 1)
24 |   k ← (k + 1)
25 END
26 output A
```

Algorithm 12.9: MERGE

We present an example run of Merge Sort in Figure 12.10 (here, we have made the collection of size 8 to emphasize the even split). An example run of the MERGE subroutine on the last merge operation of this algorithm is presented in Figure 12.11.

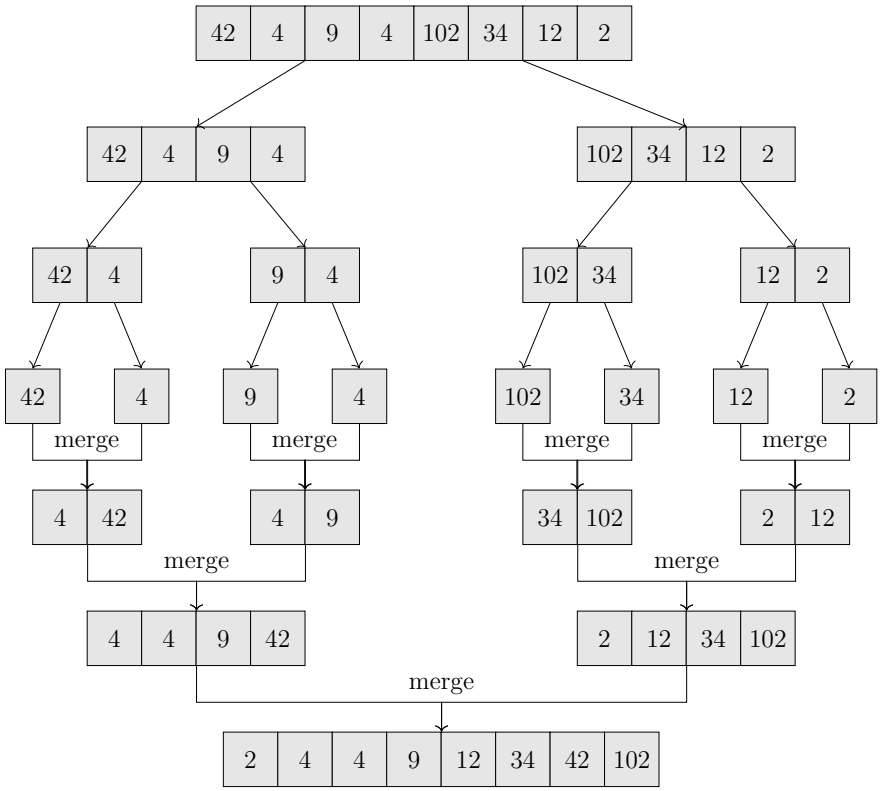
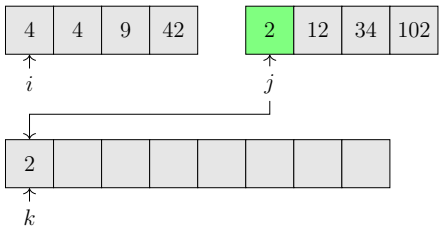
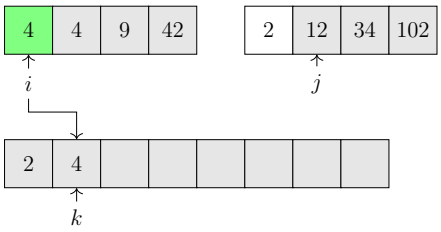


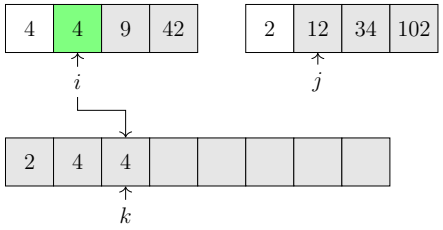
Figure 12.10.: Illustration of Merge Sort's recursion and merge operations.



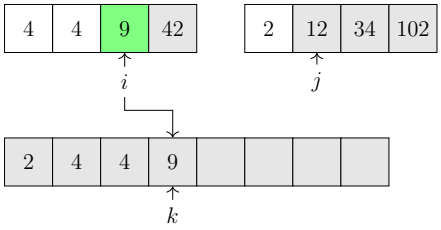
(a) Iteration One. $L_1 = 4 > R_1 = 2$, so the element in the right partition is copied into A_1. Both j, k are incremented.



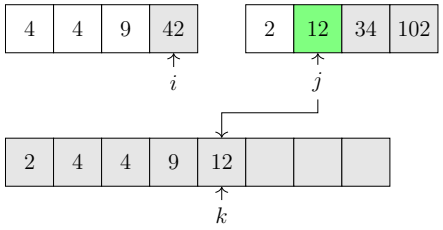
(b) Iteration Two. Now the element in the left partition is lesser and is copied, incrementing i, k



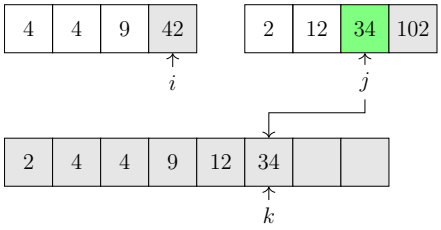
(c) Iteration Three. Again the element in the left partition is less.



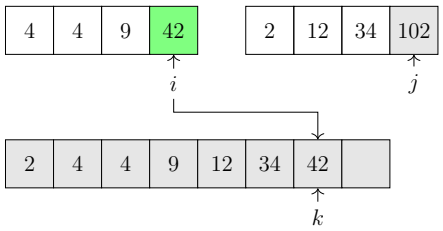
(d) Iteration Four. And so on.



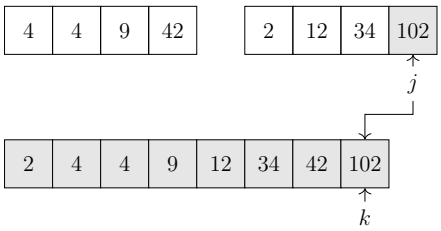
(e) Iteration Five



(f) Iteration Six



(g) Iteration Seven



(h) Final Copy. After one sub-collection has been exhausted the rest (in this case only one) of the elements in the other are blindly copied over.

Figure 12.11.: Demonstration of the merge operation in Merge Sort. Here we depict the final MERGE subroutine invocation from the previous example.

Analysis

Because Merge Sort divides the list *first*, an even split is *guaranteed*. After the recursion, the MERGE subroutine requires at most $n-1$ comparisons to merge the two collections. This leads to a recurrence relation similar to Quick Sort,

$$C(n) = 2C\left(\frac{n}{2}\right) + (n-1)$$

A similar analysis yields a complexity of $n \log(n)$.

12.2.5. Other Sorts

There are dozens of other sorting algorithms and many more variations on sorting algorithms, each with some unique properties and advantages. For example, Heap Sort, due to J. W. J. Williams, 1964 [38], uses a data structure called a *heap*. A heap stores elements in a *tree* structure and offers efficient insertion of elements and retrieval of the minimal (or maximal for a "max-heap") element. The sorting algorithm then simply inserts each element into the heap, and retrieves each element out of the heap. Because of the the heap data structure, elements come out of the heap in order.

Other sorting algorithms are variations on those we've already seen or *hybrid* sorting algorithms. For example, we've already noted that Insertion Sort is efficient on "small" arrays. One common technique is to use a fast sorting algorithm such as Quick Sort or Merge Sort, but then switch over to Insertion Sort at some point in the recursion rather than recursing to the point that the collection is of size 1 (or 0). This switch point is usually tuned using empirical experiments.

A more recent sorting algorithm is Tim Sort, due to Tim Peters in 2002 [34] who originally developed it as the default sorting algorithm for the Python language. It has since been adopted in Java (version 7 for arrays of non-primitive types). Tim Sort is a sort of "bottom-up" Merge Sort/Insertion Sort hybrid. Instead of recursing, it looks for subsequences of elements that are already in order or nearly in order, then merges them. It was designed to have many of the desirable properties of a sorting algorithm including *stability* (see Section 12.3.6).

12.2.6. Comparison & Summary

To illustrate just how much more efficient an $n \log(n)$ sorting algorithm is over a naive n^2 algorithm consider the same scenarios as with searching. Suppose we want to sort a huge collection of 1 trillion, 10^{12}, elements. Doing so with Selection Sort or Insertion Sort would require about

$$n^2 = (10^{12})^2 = 10^{24}$$

or 1 septillion comparisons. The same sorting operation using either Quick Sort or Merge Sort would require only

$$n \log(n) = 10^{12} \log 10^{12} \approx 4 \times 10^{13}$$

or just under 40 trillion comparisons. This is 25 *billion* times fewer operations.

As another example, suppose that we sort a collection with n elements and then want to double the size of our collection to $2n$. How do each of these types of algorithms perform? For the slow, n^2 algorithms we have

$$(2n)^2 = 4n^2$$

Doubling the input size *quadruples* the number of operations Selection Sort and Insertion Sort perform. However, for Quick Sort and Merge Sort, we only have

$$2n \log(2n) = 2n(\log(2) + \log(n)) = 2n \log(n) + 2n$$

That is, only about twice as many operations (with an additive term of $2n$). Table 12.2 gives a summary of the complexity and other properties of the sorting algorithms we've seen.

Algorithm	Complexity			Stable?	Notes
	Best	Average	Worst		
Selection Sort	$\sim n^2$	$\sim n^2$	$\sim n^2$	No	
Insertion Sort	$\sim n$	$\sim n^2$	$\sim n^2$	Yes	Best in practice for small collections
Quick Sort	$\sim n \log n$	$\sim n \log n$	$\sim n^2$	No	Performance depends on pivot choices
Merge Sort	$\sim n \log n$	$\sim n \log n$	$\sim n \log n$	Yes	Requires extra space

Table 12.2.: Summary of Sorting Algorithms. See Section 12.3.6 for a discussion on stability.

12.3. Searching & Sorting In Practice

12.3.1. Using Libraries and Comparators

In practice you do not write your own searching and sorting algorithm implementations. Instead, you use the functionality provided by the language, a framework, or a library. First, there is rarely a good reason to "reinvent the wheel" and write your own if the functionality already exists. Second, the algorithms and code provided by a language or library are typically optimized and have been well-tested. Established libraries have

been developed with thousands of man-hours and have proven themselves over millions of computing hours.

Typically, searching and sorting functions in a language are made to be *generic*: they don't *just* search a collection of numbers or strings. Instead, they accept collections of *any type*. The basic algorithms for searching and sorting are generic themselves (after all we presented them as pseudocode). The only real difference between an implementation for, say, numbers versus a sorting algorithm for strings is *how* pairs of elements are compared and ordered.

This genericness extends to objects. It would be extremely inefficient, development-wise, to have to rewrite Quick Sort for a collection of student objects to sort them by name, then another to sort them by GPA, then another for ID, and repeat them all again for a reversed ordering. Instead, a single, generic sorting algorithm is implemented that can be *configured* by passing in a *comparator*.

A comparator is a function or object that provides functionality to determine how to order two elements. In general, a comparator takes two arguments, a, b which are the elements to be compared and returns an integer value with the following "contract." It returns:

- something negative, < 0 if a comes before b, $a < b$
- zero if a and b are equivalent, $a = b$
- something positive, > 0 if a comes after b, $a > b$

We've previously seen this type of functionality when comparing strings in Chapter 8. Using comparators allows us to reuse the generic search and sorting algorithms. Now for each sorting that we want, we only need to create a comparator to define the *ordering* rather than reimplementing the whole algorithm every single time. This relationship is depicted in Figure 12.12.

12.3.2. Preventing Arithmetic Errors

Recall that in binary search, we need to compute the index of the middle element.[9] Given a left l and right r index variable, we computed the middle index as

$$m \leftarrow \left\lfloor \frac{l + r}{2} \right\rfloor$$

Though mathematically correct, when implementing an expression like this, care must be taken. We may initially translate this pseudocode into a line that looks something like the following.

[9] Some implementations of Quick Sort will do something similar when choosing the "middle" element as a pivot.

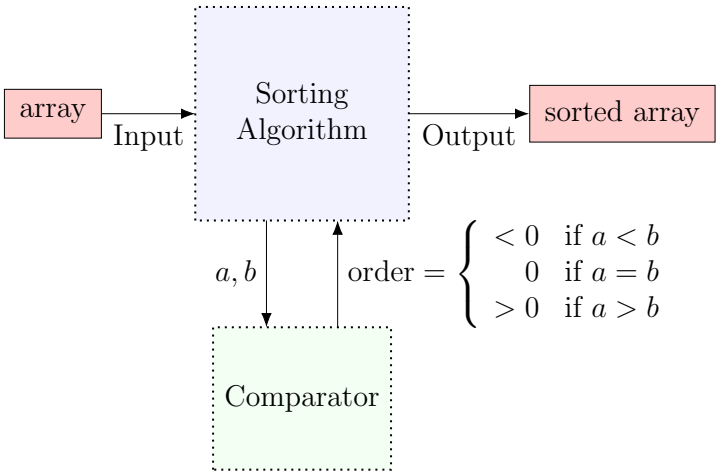


Figure 12.12.: Generalized Sorting with a Comparator. A sorting algorithm doesn't need to know *what* it is sorting or how they are *ordered* as long as it has access to a *comparator* that *does* know how to order elements. By using a comparator, the sorting function can be kept general and generic so that one implementation can be used for any type of data.

```
int middle_index = (left + right) / 2;
```

However, this is prone to arithmetic errors in certain situations, particularly when dealing with large arrays. If the variables `left` and `right` have a sum that exceeds the maximum value that a signed 32-bit integer can hold ($2^{31} - 1 = 2,147,483,647$), then overflow will occur before the division by 2, leading to a (potentially) negative index value and thus an invalid out-of-bounds error or exception.

One solution would be to use a variable type that supports larger values. For example, a 64-bit signed integer would be able to handle arrays of 9.223 quintillion elements. However, depending on the language, you may not be able to use such variable types as index values. Another solution is to use operations that do not introduce this potential for overflow. For example,

$$\frac{l+r}{2} = l + \frac{(r-l)}{2}$$

but the expression on the right hand side will not be prone to overflow. Thus the code,

```
int middle_index = left + (right - left) / 2;
```

would be safer to use. In fact, this bug is quite common [32] and was in the Java implementation of binary search, unreported for nearly a decade [10].

12.3.3. Avoiding the Difference Trick

Another issue related to arithmetic overflow is a common "trick" used in comparators when ordering integer values. Consider the following example: we want to order integer values in ascending order, the basic logic would look something like the following.

```
1 IF a < b THEN
2 |  output −1
3 ELSE IF a = b THEN
4 |  output 0
5 ELSE
6 |  output +1
7 END
```

For example, if $a = 5$ and $b = 10$ then our comparator would output -1. If the values were $a = 10, b = 5$ it would output $+1$, and if they were equal, $a = b = 10$ then it would output zero. A common "trick" is to instead compute the difference between these values, $a - b$. Observe:

a	b	$a - b$
5	10	-5
10	5	5
5	5	0

The *sign* of the difference in each of these examples matches the logic of our if-else-if statement. The lazy programmer may be tempted to write a one liner, "output $(a - b)$" instead of the if-else-if statement. However, this would fail for certain values of a and b. Suppose both of them are 32-bit 2s complement integers. And suppose that $a = 2^{31} - 1$, the maximum representable value, and $b = -10$. In the logic above, b would come before a and so we would expect a positive result. However, our arithmetic trick would give the following result:

$$(2^{31} - 1) - (-10) = 2^{31} + 9$$

which, mathematically, is positive, but exceeds the maximum representable value, leading to overflow. In most systems, the result of this arithmetic is $-2,147,483,639$, a negative value. There are many other input values that could cause arithmetic overflow. Only if you are *absolutely* sure that no arithmetic overflow is possible should you even consider using this "trick."

Another issue with this trick is when comparing floating point values. GPAs for example: suppose that $a = 4.0$ and $b = 3.9$. Their difference would be $a - b = 0.1$. However, a comparator returns an *integer* value. In some languages this result would be casted to

an integer, truncating the fractional value, so that $0.1 \to 0$, meaning that a GPA of 3.9 is "equivalent" to the 4.0. Given the potential for errors, it is best to avoid this trick altogether.

12.3.4. Importance of a Total Order

In many applications it is important to design your comparator to provide a *total order*. For example, suppose we have students "Gary Busey" with an ID of 1234 and another student with the same name, "Gary Busey" but with an ID of 4321. It is entirely possible (and common) for different people to have the same names. However, a comparator that ordered student objects based *only* on the name fields would interpret these as the same person.

The distinction is especially important when using a comparator in a sorted collection (such as a Binary Search Tree, or Ordered Hash Map data structure) that *maintains* an ordering on the elements. Many of these data structures do not allow "duplicate" elements where duplication is determined by the comparator. Attempting to insert both of these students will likely result in only one of them being in the collection as the second will be interpreted as a a duplicate. In such scenarios it is important to design your comparator to *break ties* by using some *unique* data field of an object such as an ID.

12.3.5. Artificial Ordering

Many languages recognize what is called a "natural ordering" on certain elements. For example, numbers have a natural ordering: from least to greatest. Strings also have a natural ordering: lexicographic ordering. However, we often wish to order elements in an artificial ordering. For example, suppose we wish to include the year of a student, Freshman, Sophomore, Junior, or Senior. If we modeled these as strings, the natural ordering would be alphabetical:

$$Freshman, Junior, Senior, Sophomore$$

However, if we order students by year, we would want the artificial order of Freshman, Sophomore, Junior, or Senior.

To do this, we might have a whole series of if-else-if statements to order these elements

```
   INPUT  : Two student objects, a, b
1  IF a.year = b.year THEN
2  |  output 0
3  ELSE IF a.year = "Freshman" THEN
4  |  output −1
5  ELSE IF b.year = "Freshman" THEN
6  |  output 1
7  ELSE IF a.year = "Sophomore" THEN
8  |  output −1
9  ...
```

However, this logic is complex and does not provide a good solution if we want to then add support for "Pre-freshman" or "Graduate", etc.

Another solution would be to model the year using a data type that has a natural ordering. For example, we could define an enumerated type and associate each of the years with 0, 1, 2, 3; giving them a natural ordering. Alternatively, we could use a data structure, such as a map, to model the artificial ordering. For example, we could map "Freshman" to 0, "Sophomore" to 1, etc. Then, when we wanted to order two elements, we could look up the "natural value" via the map and use their natural ordering to order our elements.

12.3.6. Sorting Stability

One desirable property of sorting algorithms is *stability*. A sorting algorithm is *stable* if the relative order of "equal" elements is preserved. For example, suppose we have the following integer values:

$$10, 2_a, 5, 2_b$$

The subscripts on the two 2 values are for demonstration purposes only. A stable sorting algorithm would sort the elements as:

$$2_a, 2_b, 5, 10$$

preserving the relative ordering of the two 2 elements. The element 2_a came before 2_b prior to sorting and remains so after sorting. However, an unstable sorting algorithm may instead produce

$$2_b, 2_a, 5, 10$$

The collection is still sorted, but the two equal elements have been reversed from their original ordering.

Sorting stability is often desirable for data presentation. A user can typically sort table data by clicking on a column header. Suppose we sorted a table of students first by GPA then by year. We would expect that all Freshman would be grouped together and within that group, would be ordered by GPA (because the first ordering would be preserved). An unstable sorting algorithm would result in all Freshman being grouped together, but within that grouping, the students would not necessarily be ordered by GPA.

As indicated in Table 12.2, some algorithms (as presented) are stable and others are not. For example, Selection Sort is not stable. Consider the following input:

$$2_a, 2_b, 1, 5$$

In the first iteration, 1 and 2_a would be swapped, resulting in a sorted collection and no further swaps, but with 2_b preceding 2_a.

Insertion Sort, however, is stable: it only moves an element down the collection if it is *strictly* less than its adjacent element. Likewise, Merge Sort is stable since we "prefer" elements from the left partition when equal.

Quick Sort is not stable as the partitioning can easily place things out of their original relative ordering. Usually, sorting algorithms can be *made* to be stable by doing some extra work, but it may impact the performance.

12.4. Exercises

Exercise 12.1. Give an input example input demonstrating that the Quick Sort algorithm, as presented, is unstable. Run through the algorithm to demonstrate how it results in an unstable sort.

Exercise 12.2. Implement the sorting algorithms described in this chapter in the programming language of your choice. Then setup experiments: run each on randomly ordered collections many times to get an average run time. Graph these empirical results and see if they match the theoretical results of our analysis. Be sure to also include any built-in sorting functionality your language provides as a benchmark.

Exercise 12.3. Ternary search is similar to binary search in that it searches a sorted array. However, instead of splitting the array in two, it splits it into three partitions and makes two comparisons to determine which third the element lies in (if it is in the collection). Implement ternary search and benchmark it against a binary search implementation.

Exercise 12.4. Data has a "natural" ordering: numbers are ordered in nondecreasing order, strings are ordered lexicographically (according to the ASCII text table). However, in many situations, the natural ordering isn't the *expected* ordering. For example, class

years are usually ordered Freshman, Sophomore, Junior, Senior whereas the natural ordering would order them Freshman, Junior, Senior, Sophomore.

Write a program to sort a collection of strings according to an arbitrary artificial ordering. That is, instead of the A–Z alphabetic ordering, we will order them according to an alternative ordering of the English alphabet. As an example, one possible ordering would be:

```
n e d c r h a l g k m z f w j o b v x q y i p u s t
```

Your job will be to *sort* a list of English language words using this artificial ordering of the English alphabet. Specifically, you will read in an input file in the following format. The first line is the new ordering of English letters, each separated by a space. Each line after that contains a single string. You will read in this file, process it and produce a reordered list of the words sorted according to the new ordering.

An example input:

```
n e d c r h a l g k m z f w j o b v x q y i p u s t
vcawufotrb
laencfuesw
gvtkwekfom
vrsfqictqc
wmcvmjmtet
qetegyqelu
newaxdtjlt
nfrfrwkknj
fzqrvgblov
gkkmgwwwpa
```

The result ordering:

```
Artificial Ordering:
newaxdtjlt
nfrfrwkknj
laencfuesw
gkkmgwwwpa
gvtkwekfom
fzqrvgblov
wmcvmjmtet
vcawufotrb
vrsfqictqc
```

```
qetegyqelu
```

For simplicity, you can assume that all words will be lower case and no non-alphabetic characters are used. However, you may not assume that all words will be the same length. Words of a shorter length that are a prefix of another word should be ordered first. For example, "newax" should come before "newaxn" in the ordering above.

Exercise 12.5. Write comparators for all the member fields of the album object in Exercise 10.2.

Exercise 12.6. Write comparators for all the member fields of the savings account object in Exercise 10.3.

Exercise 12.7. Write comparators for all the member fields of the stadium object in Exercise 10.4.

Exercise 12.8. Write comparators for all the member fields of the network device object in Exercise 10.5.

Exercise 12.9. Write comparators for all the member fields of the airport object in Exercise 10.6.

13. Graphical User Interfaces & Event Driven Programming

This chapter may appear in a future version.

14. Introduction to Databases & Database Connectivity

This chapter may appear in a future version.

Part I.

The C Programming Language

15. Basics

The C programming language is a relatively old language, but still widely used. It is universal in that nearly every system, platform, and operating system has a C compiler that produces machine code for that system. C is used extensively in systems programming for operating system kernels, embedded systems, microcontrollers, and supercomputers. It is generally an "imperative" language which is a paradigm that characterizes computation in terms of executable statements and functions that change a program's state (variable values). C has been highly influential in the design and syntax of other languages including C++, Objective-C, C#, Java, and PHP, among many others. Many languages have adopted the basic syntactic elements and structured programming approach of the C language.

C was originally developed by Dennis Ritchie while at AT&T Bell Labs 1969–1972. C was born out of the need for a new language for the PDP-11 minicomputer that used the Unix operating system (written by Ken Thompson). From its inception, C has had a close relation to Unix; in fact the operating system was subsequently rewritten in C, making it the first OS to be written in a language other than assembly. The language was dubbed "C" as its predecessor was named "B," a simplified version of BCPL (Basic Combined Programming Language). The first formal specification was published as *The C Programming Language* by Kernighan and Ritchie (1978) [26] often referred to as "The K&R Book" which would later become the American National Standards Institute (ANSI) C standard.

C gained in popularity and directly influenced object-oriented variations of it. Bjarne Stroustrup developed C++ while at Bell Labs from 1979–1983. Brad Cox and Tom Love developed Objective-C from 1981–1983 at their company Stepstone. Subsequent standards of the C language have added and extended features. In 1990, the International Organization for Standardization (ISO)/IEC 9899:1990 standard, referred to as C89 or C90, was adopted. About every 10 years since, a new standard has been adopted; ISO/IEC 9899:1999 (referred to as C99) in 1999 and ISO/IEC 9899:2011 (C11) in 2011.

15.1. Getting Started: Hello World

The hallmark of an introduction to a new programming language is the *Hello World!* program. It consists of a simple program whose only purpose is to print out the message

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

/**
 * Basic Hello World program in C
 * Prints "Hello World" to the standard output and exits
 */
int main(int argc, char **argv) {

  printf("Hello World\n");

  return 0;
}
```

Code Sample 15.1.: Hello World Program in C

"Hello World!" to the user in some manner. The simplicity of the program allows the focus to be on the basic syntax of the language. It is also typically used to ensure that your development environment, compiler, runtime environment, etc. are functioning properly with a Minimum Working Example (MWE). The Hello World! program is generally attributed to Brian Kernighan who used it as an example of programming in C in 1974 [25]. A basic Hello World! program in C can be found in Code Sample 15.1.

We will not focus on any particular development environment, code editor, or any particular operating system, compiler, or ancillary standards in our presentation. However, as a first step, you should be able to write, compile, and run the above program on the environment you intend to use for the rest of this book. This may require that you download and install a basic C compiler/development environment (such as GCC, the GNU Compiler Collection on OSX/Unix/Linux, cygwin or MinGW for Windows) or a full IDE (such as Xcode for OSX, or Code::Blocks, http://www.codeblocks.org/ for Windows).

15.2. Basic Elements

Using the Hello World! program as a starting point, we will now examine the basic elements of the C language.

15.2.1. Basic Syntax Rules

C is a highly influential programming language. Many modern programming languages have adopted syntactic elements that originated in C. Usually such languages are referred to as "C-style syntax" languages. These elements include the following.

- C is a statically typed language so variables must be declared along with their types before using them.
- Strings are delimited with double quotes. Single characters, including special escaped characters are delimited by single quotes; `"this is a string"`, and these are characters: `'A'`, `'4'`, `'$'` and `'\n'`
- Executable statements are terminated by a semicolon, `;`
- Code blocks are defined by using opening and closing curly brackets, `{ ... }`. Moreover, code blocks can be *nested*: code blocks can be defined within other code blocks.
- Variables are scoped to the code block in which they are declared and are only valid within that code block.
- In general, whitespace between coding elements is ignored.

Though not a syntactic requirement, the proper use of whitespace is important for good, readable code. Code inside code blocks is indented at the same indentation level. Nested code blocks are indented further. Think of a typical table of contents or the outline of a formal paper or essay. Sections and subsections or points and subpoints all follow proper indentation with elements at the same level at the same indentation. This convention is used to organize code and make it more readable.

15.2.2. Preprocessor Directives

The lines,

```
1 #include <stdlib.h>
2 #include <stdio.h>
```

are *preprocessor directives*. Preprocessor directives are instructions to the compiler to *modify* the source code before it starts to compile it. These particular lines are "including" standard libraries of functions so that the program can use functionality that has already

been implemented for us.

The first, `stdlib.h` represents the C standard (`std`) library (`lib`). This library is so essential that many compilers will automatically include it even if you do not explicitly do so in your program. Still, it is best practice to include it in your code.

The second, `stdio.h` is the standard (`std`) input/output (`io`) library which contains basic input/output (I/O) functions that we can use. In particular, the standard output function `printf()` is part of this library.

Failure to include a library means that you will not be able to use the functions it provides in your program. Using the functions without including the library may result in a compiler error. The `.h` in the library names stands for "header"; function declarations are typically contained in a header file while their definitions are placed in a source file of the same name. We'll explore this convention in detail when we look at functions in C (Chapter 18).

There are many other important standard libraries that we'll touch on as needed; of immediate interest is the standard mathematics library, `math.h`. It includes many useful functions to compute common mathematical functions such as the square root and natural logarithm. Table 15.1 highlights several of these functions. To use them you'd include the math library in your source file `#include <math.h>` and then "call" them by providing input and assigning the output value to a variable. For example:

```
1 double x = 1.5;
2 double y, z;
3 y = sqrt(x); //y now has the value √x = √1.5
4 z = sin(x); //z now has the value sin(x) = sin(1.5)
```

In both of the function calls above, the value of the variable `x` is "passed" to the math function which computes and "returns" the result which then gets assigned to another variable.

Macros

Another preprocessor directive establishes *macros* using the `#define` keyword. A macro is a single instruction that specifies a more complex set of instructions. The macro can be used to define constants to be used throughout your program. To illustrate, consider the following example.

Function	Description
`abs(x)`	Absolute value for `int` variables, $\lvert x \rvert$[a]
`fabs(x)`	Absolute value for `double` variables
`ceil(x)`	Ceiling function, $\lceil 46.3 \rceil = 47.0$
`floor(x)`	Floor function, $\lfloor 46.3 \rfloor = 46.0$
`cos(x)`	Cosine function[b]
`sin(x)`	Sine function[b]
`tan(x)`	Tangent function[b]
`exp(x)`	Exponential function, e^x, $e = 2.71828\ldots$
`log(x)`	Natural logarithm, $\ln(x)$[c]
`log10(x)`	Logarithm base 10, $\log_{10}(x)$[c]
`pow(x,y)`	The power function, computes x^y
`sqrt(x)`	Square root function[c]

Table 15.1.: Several functions defined in the C standard math library. [a]The absolute value function is actually in the standard library, `stdlib.h`. [b]all trigonometric functions assume input is in *radians*, **not** degrees. [c]Input is assumed to be positive, $x > 0$.

```c
#define MILES_PER_KM 1.609
```

The macro defines an "alias" for the `MILES_PER_KM` identifier as the value 1.609. Essentially, the C preprocessor will go through the code and any instance of `MILES_PER_KM` will be replaced with `1.609`. The advantage of using a macro like this is that we can use the identifier `MILES_PER_KM` throughout our program instead of magic numbers whose meaning and intent may not be immediately clear. Moreover, if we want to change the definition (say make it more precise, using 1.60934 instead) then we only need to change the macro instead of making the same change throughout our program.

As a stylistic note: macro constants in C are usually associated with uppercase underscore casing as in our example. Also, the math standard library defines several macros for common mathematical constants such as π, e, and $\sqrt{2}$ (`M_PI`, `M_E`, and `M_SQRT2` respectively) among others.

15.2.3. Comments

Comments can be written in a C program either as a single line using two forward slashes, `//comment` or as a multiline comment using a combination of forward slash and asterisk: `/* comment */`. With a single line comment, everything on the line *after* the forward slashes is ignored. With a multiline comment, everything between the forward slash/asterisk is ignored. Comments are ultimately ignored by the compiler so the amount of comments do not have an effect on the final executable code. Consider the following example.

```c
//this is a single line comment
int x;  //this is also a single line comment, but after some code

/*
  This is a comment that can
  span multiple lines to format the comment
  message more clearly
*/
double y;
```

Most code editors and IDEs will present comments in a special color or font to distinguish them from the rest of the code (just as our example above does). Failure to close a

multiline comment will likely result in a compiler error but with color-coded comments its easy to see the mistake visually.

15.2.4. The `main()` Function

Every executable program starts its execution somewhere. In C, the starting point is the `main()` function. When a program is compiled to an executable and the program is invoked, the code in the `main()` function starts executing. In our example, the code between the two curly brackets defines the `main()` function. At the end of the function we have a `return` statement. We'll examine `return` statements in detail when we examine functions. The program is "returning" a zero value to the operating system. The convention in C is that zero indicates "no error occurred" while a non-zero value is used to indicate that "some" error occurred (the value used is determined by system standards such as the Portable Operating System Interface (POSIX) standard).

In addition, our `main()` function has two *arguments*: `argc` and `argv` which serve to communicate any command line arguments provided to the program (review Section 2.4.4 for details). The first, `argc` is an integer that indicates the *number* of arguments provided *including* the executable file name itself. The second, `argv` actually stores the arguments as strings. We will understand the syntax later on, but for now we can at least understand how we might convert these arguments to different types such as integers and floating point numbers.

Recall that `argv` is the argument *vector*: it is an array (see Chapter 20) of the command line arguments. To access them, you can *index* them starting at zero, the first being `argv[0]`, the second `argv[1]`, etc. (the last one of course would be at `argv[argc-1]`). The first one is always the name of the executable file being run. The remaining are the command line arguments provided by the user.

To convert them you can use two different functions, `atoi()` and `atof()` which are short for **a**lphanumeric **to** **i**nteger and **f**loafing-point number respectively. An example:

```
1 //prints the first command line argument:
2 printf("%s\n", argv[0]);
3 //converts the "second" command line argument to an integer
4 int x = atoi(argv[1]);
5 //converts the "third" command line argument to a double:
6 double y = atof(argv[2]);
```

15.3. Variables

As previewed, the three primary primitive types supported in C are `int`, `double`, and `char` which support integers, floating point numbers, and single ASCII characters.

Integer (`int`) types are only guaranteed to "be at least" 16 bytes by the C standard but are usually 32-bit signed integers on most modern systems.[1] With a 32-bit signed `int` we can represent integers in the range $-2,147,483,648$ to 2,147,483,647. Doubles (`double`) types are double-precision floating point numbers as per the IEEE 754 standard and provide about 16 digits of precision.

C provides a `float` (single precision floating point number) type and there are various modifiers such as `short`, `long`, `unsigned` and `signed` that can be used, but these are either system-dependent or rely on later versions of the C standard (such as C99). We will restrict our focus to more portable, interoperable code and stick with the basic `int` and `double` types in our code.

Finally, the `char` type is typically a single byte that represents a single ASCII character. For all intents and purposes a `char` can be treated as an integer in the range 0 to 127 (or 255) as defined by the ASCII text table (see Table 2.4).

15.3.1. Declaration & Assignment

C is a statically typed language meaning that all variables must be declared before you can use them or refer to them. In addition, when declaring a variable, you must specify both its type and its identifier. For example:

```
1 int numUnits;
2 double costPerUnit;
3 char firstInitial;
```

Each declaration specifies the variable's type followed by the identifier and ending with a semicolon. The identifier rules are fairly standard: a name can consist of lower and uppercase alphabetic characters, numbers, and underscores but may *not* begin with a numeric character. We adopt the modern camelCasing naming convention for variables in our code.

The assignment operator is a single equal sign, `=` and is a right-to-left assignment. The

[1] You may have to deal with 16-bit `int` types in legacy systems/compilers or in modern embedded systems.

variable that we wish to assign the value to appears on the left-hand-side while the value (literal, variable or expression) is on the right-hand-size. Using our variables from before, we can assign them values:

```
numUnits = 42;
costPerUnit = 32.79;
firstInitial = 'C';
```

An important thing to understand and to keep in mind is: if you declare a variable but do not assign it a value, its value is *undefined*. If we code something like `int a;`, the value of the variable `a` is *not* necessarily zero; depending on the system, it could contain a special value that indicates "uninitialized memory" or it could contain garbage, or it *could* have the value zero. The C standard does *not* specify default values for variables. The default value of variables is highly system dependent–on the compiler, the libraries, and even the operating system. You should not make any assumptions on the initial or default values of variables. If you need such assumptions, then values must be assigned.

For brevity, C allows you to declare a variable and immediately assign it a value on the same line. Our example could be written more compactly written as follows.

```
int numUnits = 42;
double costPerUnit = 32.79;
char firstInitial = 'C';
```

As another shorthand, we can declare multiple variables on the same line by delimiting them with a comma. However, they *must* be of the same type. We can also use an assignment with them.

```
int numOrders, numUnits = 42, numCustomers = 10, numItems;
double costPerUnit = 32.79, salesTaxRate;
```

Another convenient keyword is `const`, short for "constant." We can apply it to any variable to indicate that it is a read-only variable. Of course, we *must* assign it a value at declaration. For example:

```
const int secret = 42;
const double salesTaxRate = 0.075;
```

Any attempt to reassign the values of `const` variables will result in a compiler error.

15.4. Operators

C supports the standard arithmetic operators for addition, subtraction, multiplication, and division using `+`, `-`, `*`, and `/` respectively. Each of these operators is a binary operator that acts on two operands which can be literals, variables or expressions and follow the usual rules of arithmetic when it comes to order of precedence (multiplication and division before addition and subtraction).

```
int a = 10, b = 20, c = 30, d;
d = a + 5;
d = a + b;
d = a - b;
d = a + b * c;
d = a * b;
d = a / b; //integer division and truncation!  See below

double x = 1.5, y = 3.4, z = 10.5, w;
w = x + 5.0;
w = x + y;
w = x - y;
w = x + y * z;
w = x * y;
w = x / y;

//you can do arithmetic with both types:
w = a + x;
d = b + y; //truncation also occurs here!
```

Special care must be taken when dealing with `int` types. For all four operators, if both operands are integers, the result will be an integer. For addition, subtraction, and multiplication this isn't a big deal, but for division it means that when we divide, say

`10 / 20`, the result is not 0.5 as expected. The number 0.5 is a floating point number. As such, the fractional part gets truncated (cut off and thrown out) leaving only zero. In the code above, `d = a / b;` the variable `d` ends up getting the value zero because of this.

Similarly, attempting to assign a floating point number to an integer also results in truncation because an `int` type cannot handle the fractional part. In the line `d = b + y` above, `b + y` correctly evaluates as $20 + 3.4 = 23.4$, but when assigned to the `int` variable `d` the .4 gets truncated and `d` is assigned the value 23. Assigning an `int` value to a `double` variable is not a problem as the integer 2 implicitly becomes the floating point number 2.0.

A solution to this problem is to use explicit type casting to force at least one of the operands in an integer division to become a `double` type. For example:

```
int a = 10, b = 20;
double x;

x = (double) a / b;
```

results in `x` getting the "correct" value of 0.5. This works because the `(double)` code forces the `int` variable `a` to *temporarily* be treated as a double variable (in this case 10.0) for the purposes of division (so that truncation does *not* occur).

C also supports the integer remainder operator using the `%` symbol. This operator gives the remainder of the result of dividing two integers. Examples:

```
int x;

x = 10 % 5; //x is 0
x = 10 % 3; //x is 1
x = 29 % 5; //x is 4
```

15.5. Basic I/O

The standard I/O library (`stdio.h`) contains many functions that facilitate input and output including `printf()` for standard output and `scanf()` (**scan f**ormatted input)

for standard input.

The `printf()` function works exactly as discussed in Section 2.4.3. The `scanf()` function works using similar placeholders as `printf()`. To illustrate how it works, consider the following lines of code:

```
1 int a;
2 printf("Please enter a number: ");
3 scanf("%d", &a);
```

The `printf()` statement prompts the user for an input. The `scanf()` then executes and the program *waits* for the user to enter input. The user is free to start typing. When the user is done, they hit the enter key at which point the program resumes and reads the input from the standard input buffer, converts the value entered by the user into an integer and places the result in the variable `a` where it can now be used by the remainder of the program.

A few points of interest. First, the same placeholder as `printf()` was used, `%d` for `int` values. However, when we "passed" the variable `a` to `scanf()` we placed an ampersand, `&` in front of it. This is passing the variable *by reference* and we'll explore that concept further in Chapter 18, but for now just know that when using variables with `scanf()`, an ampersand is required. Failure to place an ampersand in front of a variable with `scanf()` will likely result in a *segmentation fault* (an illegal memory access).

You can use the same placeholder, `%c` with `scanf()` to read in a single character as well. However, for floating point numbers, in particular `double` types, the placeholder `%lf` must be used (which stands for long float, a double precision number). Failure to use the correct placeholder may result in garbage results as the input will be interpreted incorrectly. Another example:

```
1 double x;
2 printf("Please enter a fractional number: ");
3 scanf("%lf", &x);
```

Another potential problem is that `scanf()` expects a certain *format* (thus its name). If we prompt the user for a number but they just start mashing the keyboard giving non-numerical input, we may get incorrect results. `scanf()` will interpret the input as zero. It may be very difficult to distinguish between the case of a user actually entering

in zero as a legitimate input versus bad input. In general, `scanf()` is not a good mechanism for reading input (and in fact can be very dangerous), but it does provide a good starting point.

15.6. Examples

15.6.1. Converting Units

Let's write a program that will prompt the user to enter a temperature in degrees Fahrenheit and convert it to degrees Celsius using the formula

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

We begin with the basic program outline which will include preprocessor directives to bring in the standard library and the standard input/output library (we'll need to prompt for input and print the result as output to the user). Further, we want our program to be executable, so we need to put our code into the `main()` function. Finally, we'll document our program to indicate its purpose.

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

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */
int main(int argc, char **argv) {

  //TODO: implement this

  return 0;
}
```

It is common for programmers to use a comment along with a `TODO` note to themselves as a reminder of things that they still need to implement in a program.

Let's first outline the basic steps that our program will go through:

1. We'll first prompt the user for input, asking them for a temperature in Fahrenheit

2. Next we'll read the user's input, likely into a floating point number as degrees can be fractional
3. Once we have the input, we can calculate the degrees Celsius by using the formula above
4. Lastly, we will want to print the result to the user to inform them of the value

Sometimes its helpful to write an outline of such a program directly in the code using comments to provide a step-by-step process. For example:

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

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */
int main(int argc, char **argv) {

  //TODO: implement this
  //1. Prompt the user for input in Fahrenheit
  //2. Read the Fahrenheit value from the standard input
  //3. Compute the degrees Celsius
  //4. Print the result to the user

  return 0;
}
```

As we read each step it becomes apparent that we'll need a couple of variables: one to hold the Fahrenheit (input) value and one for the Celsius (output) value. It also makes sense that each of these should be `double` variables as we want to support fractional values. So at the top of our `main()` function, we'll add the variable declarations:

```c
double fahrenheit, celsius;
```

Each of the steps is now straightforward; we'll use a `printf()` statement in the first step to prompt the user for input:

```c
printf("Please enter degrees in Fahrenheit: ");
```

In the second step, we'll use the standard input to read the `fahrenheit` variable value from the user. Recall that we use the placeholder `%lf` for reading `double` values and

use an ampersand when using `scanf()`:

```
scanf("%lf", &fahrenheit);
```

We can now compute `celsius` using the formula provided:

```
celsius = (fahrenheit - 32) * (5 / 9);
```

Finally, we use `printf()` again to output the result to the user:

```
printf("%f Fahrenheit is %f Celsius\n", fahrenheit, celsius);
```

Try typing and running the program as defined above and you'll find that you don't get correct answers. In fact, you'll find that no matter what values you enter, you get zero. This is because of the calculation using `5 / 9`: recall what happens with integer division: truncation! This will *always* end up being zero.

One way we could fix it would be to pull out our calculators and find that $\frac{5}{9} = 0.55555\ldots$ and replace `5/9` with `0.555555`. But, how many fives? It may be difficult to tell how accurate we can make this floating point number by hardcoding it ourselves. A much better approach would be to let the compiler take care of the optimal computation for us by making at least one of the numbers a `double` to prevent integer truncation. That is, we should instead use `5.0 / 9`. The full program can be found in Code Sample 15.2.

15.6.2. Computing Quadratic Roots

Some programs require the user to enter multiple inputs. The prompt-input process can be repeated. In this example, consider asking the user for the coefficients, a, b, c to a quadratic polynomial,

$$ax^2 + bx + c$$

and computing its roots using the quadratic formula,

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

As before, we can create a basic program with a `main()` function and start filling in the details. In particular, we'll need to prompt for the input a, then read it in; then prompt for b, read it in and repeat for c. We'll also need several variables: three for the coefficients a, b, c and *two* more; one for each root. Thus, we have

```
1 double a, b, c, root1, root2;
2
3 printf("Please enter a: ");
4 scanf("%lf", &a);
```

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

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */
int main(int argc, char **argv) {

  double fahrenheit, celsius;

  //1. Prompt the user for input in Fahrenheit
  printf("Please enter degrees in Fahrenheit: ");

  //2. Read the Fahrenheit value from the standard input
  scanf("%lf", &fahrenheit);

  //3. Compute the degrees Celsius
  celsius = (fahrenheit - 32) * 5.0/9;

  //4. Print the result to the user
  printf("%f Fahrenheit is %f Celsius\n", fahrenheit, celsius);

  return 0;
}
```

Code Sample 15.2.: Fahrenheit-to-Celsius Conversion Program in C

```c
printf("Please enter b: ");
scanf("%lf", &b);
printf("Please enter c: ");
scanf("%lf", &c);
```

Now to compute the roots: we need to take care that we correctly adapt the formula so it accurately reflects the order of operations. We also need to use the standard math library's square root function (unless you want to write your own![2] Carefully adapting

[2] We *will* write several square root methods as exercises later!

the formula leads to

```
root1 = (-b + sqrt(b*b - 4*a*c) ) / (2*a);
root2 = (-b - sqrt(b*b - 4*a*c) ) / (2*a);
```

Finally, we print the output using `printf()`. The full program can be found in Code Sample 15.3.

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

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */
int main(int argc, char **argv) {

  double a, b, c, root1, root2;

  printf("Please enter a: ");
  scanf("%lf", &a);
  printf("Please enter b: ");
  scanf("%lf", &b);
  printf("Please enter c: ");
  scanf("%lf", &c);

  root1 = (-b + sqrt(b*b - 4*a*c) ) / (2*a);
  root2 = (-b - sqrt(b*b - 4*a*c) ) / (2*a);

  printf("The roots of %fx^2 + %fx + %f are: \n", a, b, c);
  printf("  root1 = %f\n", root1);
  printf("  root2 = %f\n", root2);
  return 0;
}
```

Code Sample 15.3.: Quadratic Roots Program in C

This program was interactive. As an alternative, we could have read all three of the

inputs as command line arguments, converting them to floating point numbers. Lines 12–17 in the program could have been changed to

```
1 a = atof(argv[1]);
2 b = atof(argv[2]);
3 c = atof(argv[3]);
```

Finally, think about the possible inputs a user could provide that may cause problems for this program. For example:

- What if the user entered zero for a?
- What if the user entered some combination such that $b^2 < 4ac$?
- What if the user entered non-numeric values?
- For the command line argument version, what if the user provided less than three arguments? Or more?

How might we prevent the consequences of such bad inputs? How might we handle the event that a user enters bad input and how do we communicate these errors to the user? To begin to resolve these issues, we'll need conditionals.

16. Conditionals

C supports the basic if, if-else, and if-else-if conditional structures as well as switch statements. Logical statements are built using the standard logical operators for numeric comparisons as well as logical operators such as negation, AND, and OR. However, there are a few idiosyncrasies that need to be understood.

16.1. Logical Operators

C has no built-in Boolean type, nor does it have any keywords associated with *true* and *false*. Instead, C uses numeric types as implicit Boolean types with the convention that zero is associated with *false* and *any* non-zero value is associated with *true*. So the values 1, 2.5, and even negatives, -1, -123.2421, etc. are all interpreted as *true* when used in logical statements.

The standard numeric comparison operators are also supported. Consider the following code snippet:

```
1 int a = 10;
2 int b = 20;
3 int c = 10;
4 int d = 0;
```

The six standard comparison operators are presented in Table 16.1 using these variables as examples. The comparison operators are the same when used with `double` types as well and `int` types can be compared with each other without type casting. The three basic logical operators are also supported as described in Table 16.2 using the same code snippet variable values as examples.

Name	Operator Syntax	Examples	Value
Equals	`==`	`a == 10`	*true*
		`b == 10`	*false*
		`a == b`	*false*
		`a == c`	*true*
Not Equals	`!=`	`a != 10`	*false*
		`b != 10`	*true*
		`a != b`	*true*
		`a != c`	*false*
Strictly Less Than	`<`	`a < 15`	*true*
		`a < 5`	*false*
		`a < b`	*true*
		`a < c`	*false*
Less Than Or Equal To	`<=`	`a <= 15`	*true*
		`a <= 5`	*false*
		`a <= b`	*true*
		`a <= c`	*true*
Strictly Greater Than	`>`	`a > 15`	*false*
		`a > 5`	*true*
		`a > b`	*false*
		`a > c`	*false*
Greater Than Or Equal To	`>=`	`a >= 15`	*false*
		`a >= 5`	*true*
		`a >= b`	*false*
		`a >= c`	*true*

Table 16.1.: Comparison Operators in C

Operator	Operator Syntax	Examples	Values
Negation	`!`	`!a`	*false*
		`!d`	*true*
And	`&&`	`a && b`	*true*
		`a && d`	*false*
Or	`||`	`a || b`	*true*
		`!a || d`	*false*

Table 16.2.: Logical Operators in C

	Operator(s)	Associativity	Notes
Highest	`++`, `--`	left-to-right	increment operators
	`-`, `!`	right-to-left	unary negation operator, logical not
	`*`, `/`, `%`	left-to-right	
	`+`, `-`	left-to-right	addition, subtraction
	`<`, `<=`, `>`, `>=`	left-to-right	comparison
	`==`, `!=`	left-to-right	equality, inequality
	`&&`	left-to-right	logical AND
	`||`	left-to-right	logical OR
Lowest	`=`, `+=`, `-=`, `*=`, `/=`	right-to-left	assignment and compound assignment operators

Table 16.3.: Operator Order of Precedence in C. Operators on the same level have equivalent order and are performed in the associative order specified.

16.1.1. Order of Precedence

At this point it is worth summarizing the order of precedence of all the operators that we've seen so far including assignment, arithmetic, comparison, and logical. Since all of these operators could be used in one statement, for example,

```
(b*b < 4*a*c || a == 0 || argc != 4)
```

it is important to understand the order in which each one gets evaluated. Table 16.3 summarizes the order of precedence for the operators seen so far. This is not an exhaustive list of C operators.

16.1.2. Comparing Strings and Characters

The comparison operators in Table 16.1 can also be used for *single characters* because of the nature of the ASCII text table (see Table 2.4). Each alphanumeric character, including the various symbols and whitespace characters, is associated with an integer 0–127. We can therefore write statements like `('A' < 'a')`, which is *true* since uppercase letters are ordered before lowercase letters in the ASCII table (`'A'` is 65 and `'a'` is 97 and so $65 < 97$ is *true*). Several more examples can be found in Table 16.4.

Numeric comparison operators *cannot* be used to compare strings in C. For example, we *could* write `("aardvark" < "zebra")` which would be valid C, and it would even have a result. However, that result wouldn't necessarily be *true* or *false*. The reason for this is that strings in C are actually represented as arrays, which in turn are represented as *memory locations*. We'll explore these issues in greater depth later on, but for now

Comparison Example	Result
`('A' < 'a')`	*true*
`('A' == 'a')`	*false*
`('A' < 'Z')`	*true*
`('0' < '9')`	*true*
`('\n' < 'A')`	*true*
`(' ' < '\n')`	*false*

Table 16.4.: Character comparisons in C

understand that you can write this code, it will compile, and it will even run. However, the results will not be as expected.

16.2. If, If-Else, If-Else-If Statements

Conditional statements in C utilize the keywords `if`, `else`, and `else if`. Conditions are placed inside parentheses immediately after the `if` and `else if` keywords. Examples of all three can be found in Code Sample 16.1.

Some observations about the syntax: the statement, `if(x < 10)` does not have a semicolon at the end. This is because it is a conditional statement that determines the flow of control and *not* an executable statement. Therefore, no semicolon is used. Suppose we made a mistake and *did* include a semicolon:

```
1 int x = 15;
2 if(x < 10); {
3   printf("x is less than 10\n");
4 }
```

Some compilers may give a warning, but this is valid C; it will compile and it will run. However, it will end up printing `x is less than 10`, even though $x = 15$! Recall that a conditional statement *binds* to the executable statement or code block *immediately* following it. In this case, we've provided an *empty* executable statement ended by the semicolon. The code is essentially equivalent to

```
1 int x = 15;
2 if(x < 10) {
3 }
```

```c
//example of an if statement:
if(x < 10) {
  printf("x is less than 10\n");
}

//example of an if-else statement:
if(x < 10) {
  printf("x is less than 10\n");
} else {
  printf("x is 10 or more \n");
}

//example of an if-else-if statement:
if(x < 10) {
  printf("x is less than 10\n");
} else if(x == 10) {
  printf("x is equal to ten\n");
} else {
  printf("x is greater than 10\n");
}
```

Code Sample 16.1.: Examples of Conditional Statements in C

```
4  printf("x is less than 10\n");
```

This is obviously not what we wanted. The semicolon ended up binding to the empty executable statement. The code block containing the print statement immediately followed, but it was *not* bound to the conditional statement which is why the print statement executed regardless of the value of x.

Another convention that we've used in our code is where we have placed the curly brackets. First, if a conditional statement is bound to only one statement, the curly brackets are not necessary. However, it is best practice to include them even if they are not necessary and we'll follow this convention. Second, the opening curly bracket is on the same line as the conditional statement while the closing curly bracket is indented to the same level as the start of the conditional statement. Moreover, the code inside the code block is indented. If there were more statements in the block, they would have all been at the same indentation level.

16.3. Examples

16.3.1. Computing a Logarithm

The logarithm of x is the exponent that some *base* must be raised to get x. The most common logarithm is the natural logarithm, $\ln(x)$ which is base $e = 2.71828\ldots$. But logarithms can be in any base $b > 1$.[1] What if we wanted to compute $\log_2(x)$? Or $\log_\pi(x)$? Let's write a program that will prompt the user for a number x and a base b and computes $\log_b(x)$.

Arbitrary bases can be computed using the change of base formula:

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$$

If we can compute *some* base a, then we can compute any base b. Fortunately we have such a solution. Recall that the standard library provides a function to compute the natural logarithm, `log()`). This is one of the fundamentals of problems solving: if a solution already exists, use it. In this case, a solution exists for a different, but similar problem (computing the natural logarithm), but we can *adapt* the solution using the change of base formula. In particular, if we have variables `b` (base) and `x`, we can compute $\log_b(x)$ using

[1] Bases can also be $0 < b < 1$, but we'll restrict our attention to increasing functions only.

`log(x) / log(b)`

But wait: we have a problem similar to the examples in the previous section. The user could enter invalid values such as $b = -10$ or $x = -2.54$ (logarithms are undefined for non-positive values in any base). We want to ensure that $b > 1$ and $x > 0$. With conditionals, we can now do this. Once we have read in the input from the user we can make a check for good input using an `if` statement.

```
1 if(x <= 0 || b <= 1) {
2   printf("Error: bad input!\n");
3   exit(1);
4 }
```

This code has something new: `exit(1)`. The `exit()` function immediately terminates the program regardless of the rest of the code that it may remain. The argument passed to `exit` is an integer that represents an *error code*. The convention is that zero indicates "no error" while non-zero values indicate some error. This is a simple way of performing *error handling*: if the user provides bad input, we inform them and quit the program, forcing them to run it again and provide good input. By prematurely terminating the program we avoid any illegal operation that would give a bad result.

Alternatively, we could have split the conditions into two statements and given more descriptive error messages. We use this design in the full program which can be found in Code Sample 16.2. The program also takes the input as command line arguments. Now that we have conditionals, we can actually check that the correct number of arguments was provided by the user and quit in the event that they don't provide the correct number.

16.3.2. Life & Taxes

Let's adapt the conditional statements we developed in Section 3.6.4 into a full C program. The first thing we need to do is establish the variables we'll need and read them in from the user. At the same time we can check for bad input (negative values) for both the inputs.

```
1 double income, baseTax, numChildren, credit, totalTax;
2
3 printf("Please enter your Adjusted Gross Income: ");
4 scanf("%lf", &income);
5
```

```
6  printf("How many children do you have?");
7  scanf("%d", &numChildren);
8
9  if(income < 0 || numChildren < 0) {
10   printf("Invalid inputs");
11   exit(1);
12 }
```

Next, we can code a series of if-else-if statements for the income range. By placing the ranges in increasing order, we only need to check the upper bounds just as in the original example.

```
1  if(income <= 18150) {
2    baseTax = income * .10;
3  } else if(income <= 73800) {
4    baseTax = 1815 + (income - 18150) * .15;
5  } else if(income <= 148850) {
6  ...
7  } else {
8    baseTax = 127962.50 + (income - 457600) * .396;
9  }
```

Next we compute the child tax credit, taking care that it does not exceed $3,000. A conditional based on the number of children suffices as at this point in the program we already know it is zero or greater.

```
1  if(numChildren <= 3) {
2    credit = numChildren * 1000;
3  } else {
4    credit = 3000;
5  }
```

Finally, we need to ensure that the credit does not exceed the total tax liability (the credit is non-refundable, so if the credit is greater, the tax should only be zero, not negative).

```
if(baseTax - credit >= 0) {
  totalTax = baseTax - credit;
} else {
  totalTax = 0;
}
```

The full program is presented in Code Sample 16.3.

16.3.3. Quadratic Roots Revisited

Let's return to the quadratic roots program we previously designed that uses the quadratic equation to compute the roots of a quadratic polynomial by reading coefficients a, b, c in from the user. One of the problems we had previously identified is if the user enters "bad" input: if $a = 0$, we would end up dividing by zero; if $b^2 - 4ac < 0$ then we would have complex roots. With conditionals, we can now check for these issues and exit with an error message.

Another potential case we might want to handle differently is when there is only one distinct root ($b^2 - 4ac = 0$). In that case, the quadratic formula simplifies to $\frac{-b}{2a}$ and we can print a different, more specific message to the user. The full program can be found in Code Sample 16.4.

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

/**
 * This program computes the logarithm base b (b > 1)
 * of a given number x > 0
 */
int main(int argc, char **argv) {

  double b, x, result;
  if(argc != 3) {
    printf("Usage: %s b x \n", argv[0]);
    exit(1);
  }

  b = atof(argv[1]);
  x = atof(argv[2]);

  if(x <= 0) {
    printf("Error: x must be greater than zero\n");
    exit(1);
  }
  if(b <= 1) {
    printf("Error: base must be greater than one\n");
    exit(1);
  }

  result = log(x) / log(b);
  printf("log_(%f)(%f) = %f\n", b, x, result);
  return 0;
}
```

Code Sample 16.2.: Logarithm Calculator Program in C

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

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

  double income, baseTax, credit, totalTax;
  int numChildren;

  //prompt for income from the user
  printf("Please enter your Adjusted Gross Income: ");
  scanf("%lf", &income);

  //prompt for children
  printf("How many children do you have?");
  scanf("%d", &numChildren);

  if(income < 0 || numChildren < 0) {
    printf("Invalid inputs");
    exit(1);
  }

  if(income <= 18150) {
    baseTax = income * .10;
  } else if(income <= 73800) {
    baseTax = 1815 + (income  -18150) * .15;
  } else if(income <= 148850) {
    baseTax = 10162.50 + (income - 73800) * .25;
  } else if(income <= 225850) {
    baseTax = 28925.00 + (income - 148850) * .28;
  } else if(income <= 405100) {
    baseTax = 50765.00 + (income - 225850) * .33;
  } else if(income <= 457600) {
    baseTax = 109587.50 + (income - 405100) * .35;
  } else {
    baseTax = 127962.50 + (income - 457600) * .396;
  }

  if(numChildren <= 3) {
    credit = numChildren * 1000;
  } else {
    credit = 3000;
  }

  if(baseTax - credit >= 0) {
    totalTax = baseTax - credit;
  } else {
    totalTax = 0;
  }

  printf("AGI:           $%10.2f\n", income);
  printf("Tax:           $%10.2f\n", baseTax);
  printf("Credit:        $%10.2f\n", credit);
  printf("Tax Liability: $%10.2f\n", totalTax);

  return 0;
}
```

Code Sample 16.3.: Tax Program in C

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

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */
int main(int argc, char **argv) {

  double a, b, c, root1, root2;

  if(argc !=4) {
    printf("Usage: %s a b c\n", argv[0]);
    exit(1);
  }

  a = atof(argv[1]);
  b = atof(argv[2]);
  c = atof(argv[3]);

  if(a == 0) {
    printf("Error: a cannot be zero\n");
    exit(1);
  } else if(b*b < 4*a*c) {
    printf("Error: cannot handle complex roots\n");
    exit(1);
  } else if(b*b == 4*a*c) {
    root1 = -b / (2*a);
    printf("Only one distinct root: %f\n", root1);
  } else {
    root1 = (-b + sqrt(b*b - 4*a*c) ) / (2*a);
    root2 = (-b - sqrt(b*b - 4*a*c) ) / (2*a);

    printf("The roots of %fx^2 + %fx + %f are: \n", a, b, c);
    printf("  root1 = %f\n", root1);
    printf("  root2 = %f\n", root2);
  }
    return 0;
}
```

Code Sample 16.4.: Quadratic Roots Program in C With Error Checking

17. Loops

C supports while loops, for loops, and do-while loops using the keywords `while`, `for`, and `do` (along with another `while`). Continuation conditions for loops are enclosed in parentheses, `(...)` and the blocks of code associated with the loop are enclosed in curly brackets.

17.1. While Loops

Code Sample 17.1 contains an example of a basic while loop in C. Just as with conditional statements, our code styling places the opening curly bracket on the same line as the `while` keyword and continuation condition. The inner block of code is also indented and all lines in the block are indented to the same level.

In addition, the continuation condition does *not* contain a semicolon since it is not an executable statement. Just as with an if-statement, if we *had* placed a semicolon it would have led to unintended results. Consider the following:

```
while(i <= 10); {
  //perform some action
  i++; //iteration
}
```

```
int i = 1; //Initialization
while(i <= 10) {  //continuation condition
  //perform some action
  i++; //iteration
}
```

Code Sample 17.1.: While Loop in C

```
int i = 1;
int flag = 1;
while(flag) {
  //perform some action
  i++; //iteration
  if(i>10) {
    flag = 0;
  }
}
```

Code Sample 17.2.: Flag-controlled While Loop in C

A similar problem occurs: the `while` keyword and continuation condition bind to the next executable statement or code block. As a consequence of the semicolon, the executable statement that gets bound to the while loop is *empty*. What happens is even worse: the program will enter an infinite loop. To see this, the code is essentially equivalent to the following:

```
while(i <= 10) {
}
{
  //perform some action
  i++; //iteration
}
```

In the while loop, we never increment the counter variable `i`, the loop does nothing, and so the computation will continue on forever! Some compilers may warn you about this, others will not. It is valid C and it will compile and run, but obviously won't work as intended. Avoid this problem by using proper syntax and good style.

Another common use case for a while loop is a flag-controlled loop in which we use a Boolean flag rather than an expression to determine if a loop should continue or not. Recall that in C, zero is treated as *false* and any non-zero numeric value is treated as *true*. We can thus create an implicit Boolean flag by using an integer variable and setting it to 1 for *true* and 0 for *false* (when we want the loop to terminate). An example can be found in Code Sample 17.2.

17.2. For Loops

For loops in C use the familiar syntax of placing the initialization, continuation condition, and iteration on the same line as the keyword `for`. An example can be found in Code Sample 17.3.

```
int i;
for(i=1; i<=10; i++) {
  //perform some action
}
```

Code Sample 17.3.: For Loop in C

Semicolons are placed at the end of the initialization and continuation condition, but *not* the iteration statement. Just as with while loops, the opening curly bracket is placed on the same line as the `for` keyword. Code within the loop body is indented, all at the same indentation level.

Another observation is that we declared the index variable `i` *prior* to the for loop. Some languages allow you to declare the index variable in the initialization statement, for example `for(int i=1; i<=10; i++)`. Doing so *scopes* the index variable to the loop and so `i` would be out-of-scope before and after the loop body. This is a nice convenience and is generally good practice. However, C89 and prior standards do not allow you to do this; the variable must be declared prior to the loop structure. C99 and newer standards do allow you to do this and some compilers will be somewhat forgiving when you use the newer syntax (by supporting their own non-standard extensions to C). For maximum portability, we'll follow the older convention.

17.3. Do-While Loops

C supports do-while loops. Recall that the difference between a while loop and a do-while loop is when the continuation condition is checked. For a while loop it is *prior* to the beginning of the loop body and in a do-while loop it is at the *end* of the loop. This means that a do-while always executes *at least once*. An example can be found in Code Sample 17.4.

Note the syntax and style: the opening curly bracket is again on the same line as the keyword `do`. The `while` keyword and continuation condition are on the same line as the closing curly bracket. In a slight departure from prior syntax, a semicolon *does*

```c
int i;
do {
  //perform some action
  i++;
} while(i<=10);
```

Code Sample 17.4.: Do-While Loop in C

appear at the end of the continuation condition even though it is not an executable statement.

17.4. Other Issues

C does not support a traditional foreach loop. When iterating over a collection like an array, you iterate an index variable, typically with a for loop. However, there are some syntactic tricks that you can use to get the same effect. Some of this will be a preview of Chapter 20 where we discuss arrays in C, but in short, you can assign a variable to an element in an array in iteration statement:

```c
double arr[] = {1.41, 2.71, 3.14};
int n = 3;
int i = 0;
double x;
for(x=arr[0]; i<n; x=arr[++i]) {
  //x now holds the i-th element in arr
}
```

The initialization sets the variable `x` to the first element in the array, `arr[0]`. The loop continues for as many elements as there are in the array, `n`. The iteration does two things: it assigns `x` to the next element in the array while at the same time incrementing the index variable using the prefix increment operator (see Section 2.3.6).

17.5. Examples

17.5.1. Normalizing a Number

Let's revisit the example from Section 4.1.1 in which we *normalize* a number by continually dividing it by 10 until it is in the range $[1, 10)$. The code in Code Sample 17.5 specifically refers to the value 32145.234 but would work equally well with any non-negative value of `x`.

```
double x = 32145.234;
int k = 0;
while(x > 10) {
  x = x / 10; //or: x /= 10;
  k++;
}
```

Code Sample 17.5.: Normalizing a Number with a While Loop in C

17.5.2. Summation

Let's revisit the example from Section 4.2.1 in which we computed the sum of integers $1 + 2 + \cdots + 10$. The code is presented in Code Sample 17.6

```
int i;
int sum = 0;
for(i=1; i<=10; i++) {
  sum += i;
}
```

Code Sample 17.6.: Summation of Numbers using a For Loop in C

Of course we could easily have generalized the code somewhat. Instead of computing a sum up to a particular number, we could have written it to sum up to another variable `n`, in which case the for loop would instead look like the following.

```
for(i=1; i<=n; i++) {
  sum += i;
}
```

17.5.3. Nested Loops

Recall that you can write loops within loops. The inner loop will execute fully *for each* iteration of the outer loop. An example of two nested of loops in C can be found in Code Sample 17.7.

```
int i, j;
int n = 10;
int m = 20;
for(i=0; i<n; i++) {
  for(j=0; j<m; j++) {
    printf("(i, j) = (%d, %d)\n", i, j);
  }
}
```

Code Sample 17.7.: Nested For Loops in C

The inner loop executes for $j = 0, 1, 2, \ldots, 19 < m = 20$ for a total of 20 iterations. However, it executes 20 times *for each* iteration of the outer loop. Since the outer loop executes for $i = 0, 1, 2, \ldots, 9 < n = 10$, the total number of times the `printf()` statement execute is $10 \times 20 = 200$. In this example, the sequence $(0, 0), (0, 1), (0, 2), \ldots, (0, 19), (1, 0), \ldots, (9, 19)$ will be printed.

17.5.4. Paying the Piper

Let's adapt the solution for the loan amortization schedule we developed in Section 4.7.3. First, we'll read the principle, terms, and interest as command line inputs. Adapting the formula for the monthly payment and using the standard math library's `pow()` function, we get

```
double monthlyPayment = (monthlyInterestRate * principle) /
                        (1 - pow( (1 + monthlyInterestRate), -n));
```

However, recall that we may have problems due to accuracy. The monthly payment could come out to be a fraction of a cent, say $43.871. For accuracy, we need to ensure that all of the figures for currency are rounded to the nearest cent. The standard math library does have a `round()` function, but it only rounds to the nearest whole number, not the nearest 100th.

However, we can *adapt* the "off-the-shelf" solution to fit our needs. If we take the number, multiply it by 100, we get (say) 4387.1 which we can now round to the nearest whole number, giving us 4387. We can then divide by 100 to get a number that has been rounded to the nearest 100th! In C, we could simply do the following.

```
monthlyPayment = round(monthlyPayment * 100.0) / 100.0;
```

We can use the same trick to round the monthly interest payment and any other number expected to be whole cents. To output our numbers, we use `printf()` and take care to align our columns to make it look nice. To finish our adaptation, we handle the final month separately to account for an over/under payment due to rounding. The full solution can be found in Code Sample 17.8.

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

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

  if(argc != 4) {
    printf("Usage: %s principle apr terms\n", argv[0]);
    exit(1);
  }

  double principle = atof(argv[1]);
  double apr = atof(argv[2]);
  int n = atoi(argv[3]);

  double balance = principle;
  double monthlyInterestRate = apr / 12.0;
  int i;

  //monthly payment
  double monthlyPayment = (monthlyInterestRate * principle) /
          (1 - pow( (1 + monthlyInterestRate), -n));
  //round to the nearest cent
  monthlyPayment = round(monthlyPayment * 100.0) / 100.0;

  printf("Principle: $%.2f\n", principle);
  printf("APR: %.4f%%\n", apr*100.0);
  printf("Months: %d\n", n);
  printf("Monthly Payment: $%.2f\n", monthlyPayment);

  //for the first n-1 payments in a loop:
  for(i=1; i<n; i++) {
    //  compute the monthly interest, rounded:
    double monthlyInterest =
            round( (balance * monthlyInterestRate) * 100.0) / 100.0;
    //  compute the monthly principle payment
    double monthlyPrinciplePayment = monthlyPayment - monthlyInterest;
    //  update the balance
    balance = balance - monthlyPrinciplePayment;
    //  print i, monthly interest, monthly principle, new balance
    printf("%d\t$%10.2f  $%10.2f  $%10.2f\n", i, monthlyInterest,
            monthlyPrinciplePayment, balance);
  }

  //handle the last month and last payment separately
  double lastInterest = round( (balance * monthlyInterestRate) * 100.0) / 100.0;
  double lastPayment = balance + lastInterest;

  printf("Last payment = $%.2f\n", lastPayment);

  return 0;
}
```

18. Functions

As a procedural-style language, functions are essential in C programming. As we've already seen, C provides a large library of standard functions to perform basic input/output, math, and many other functions. C also provides the ability to define and use your own functions.

When you define functions in C, careful thought must be made as to the naming of your functions. This is because C does *not* support function overloading. When you name a function, that is the *only* function that can have that name. Consequently, you cannot, in general, use the same function names as defined in the standard libraries or any other 3rd party library that you would like to use in your programs.

C supports both call by value and call by reference using *pointers* (see Section 18.2). C also supports vararg functions (`printf()` being a prime example) and allows you to define vararg functions, but we will not cover them in depth here. Finally, parameters are not, in general, optional. For modern versions of C the omission of parameters is a syntax error. For older versions, complex rules dictate what happens when arguments are omitted, but doing so usually results in garbage.

18.1. Defining & Using Functions

For modern C, defining functions is a two step process. First, you *declare* a function and its signature using a *prototype*. Then you *define* the function by providing a function body that defines what the function does.

18.1.1. Declaration: Prototypes

Just as with variables, functions in C must be *declared* before they can be used. The modern way to declare a function in C is to use a *prototype* declaration which specifies the function's signature including its return type, identifier, and parameters. However, a prototype does *not* include the actual body of function. Instead, the function's *definition*, which includes the function body is included later in the program. Consequently, a prototype always ends with a semicolon.

Typically, the documentation for functions is included with the prototype but is *not* repeated with the function definition. This is a principle known as Don't Repeat Yourself (DRY). Consider the following examples. In these examples we use a commenting style known as "doc comments." This style was originally developed for Java but has since been adopted by many other languages.

```
/**
 * Computes the sum of the two arguments.
 */
int sum(int a, int b);

/**
 * Computes the Euclidean distance between the 2-D points,
 * (x1,y1) and (x2,y2).
 */
double getDistance(double x1, double y1, double x2, double y2);

/**
 * Computes a monthly payment for a loan with the given
 * principle at the given APR (annual percentage rate) which
 * is to be repaid over the given number of terms (usually
 * months).
 */
double getMonthlyPayment(double principle, double apr, int terms);
```

In each of these, the return type is the first thing specified. The function identifier (name) is then specified. Function names must follow the same naming rules as variables: they must begin with an alphabetic character and may contain alphanumeric characters as well as underscores. Using modern coding conventions we usually name functions using lower camel casing.

Each prototype ends with a semicolon. Further, prototypes do *not* specify what the function does, they only specify its signature. Later in the program, we can provide the actual definition of each function by using the following syntax. We repeat the signature, but instead of using a semicolon, we provide a code block, enclosed using opening/closing curly brackets, that specifies the function body. Here are the definitions from the prototype examples above:

```
int sum(int a, int b) {
  return (a + b);
}
```

```
4
5  double getDistance(double x1, double y1, double x2, double y2) {
6    double xDiff = (x1-x2);
7    double yDiff = (y1-y2);
8    return sqrt( xDiff * xDiff + yDiff * yDiff);
9  }
10
11 double getMonthlyPayment(double principle, double apr, int terms) {
12   double rate = (apr / 12.0);
13   double payment = (principle * rate) / (1-pow(1+rate, -terms));
14   return payment;
15 }
```

The keyword `return` is used to specify the value that is returned to the calling function.

18.1.2. Void Functions

The keyword `void` can be used in C to indicate a function does *not* return a value, in which case it is called a "void function." Though it is not necessary, it is still good practice to include a `return` statement.

```
1  //prototype:
2  void printCopyright();
3
4  //definition:
5  void printCopyright() {
6    printf("(c) Bourke 2015\n");
7    return;
8  }
```

In the example above, we've also illustrated how to define a function that has no inputs. Some sources may include an explicit `void` keyword as a parameter to indicate the function takes no parameters as in `void printCopyright(void);`.

18.1.3. Organizing Functions

The separation of a function declaration (prototype) and a function definition provides a natural way to organize functions in C. We place prototypes into a *header* file which has a file extension `.h` and then place the corresponding function definitions into a *source* file with the file extension, `.c`.

We've seen this before with the standard libraries: we use `#include <math.h>` to "include" the math library's header file in our code. This essentially brings in the math library function prototypes so that we can write calls to functions like `sqrt()` or `sin()`. Only when we compile do we actually need to *link* our code to the function definitions.

When we separate prototypes into header files and definitions into source files we also need to "include" the prototypes in our source file just as we would need to include them in any other file in which we use one of the functions. Suppose our functions above have their prototypes in a file named `utils.h` and their definitions in a file named `utils.c`. In the `utils.c` source file we would typically use the following syntax to include the header file:

```
#include "utils.h"
```

We use the double quote syntax with user-defined libraries while the usual less-than/greater-than syntax is used with standard libraries. With the less-than/greater-than syntax, the compiler will attempt to look for the header file(s) in a specified system directory which it will fail to find if it is a user-defined library. Furthermore, other elements are usually included in header files such as preprocessor directives and other declarations (such as enumerated types and structures which we introduce later).

18.1.4. Calling Functions

Once a function has been defined, or at least a prototype has been brought into scope via an `#include` statement, you can write code to call your function(s). The syntax for doing so is to simply provide the function name followed by parentheses containing values or variables to pass to the function. Some examples:

```
1 int a = 10, b = 20;
2 int c = sum(a, b); //c contains the value 30
3
4 //invoke a function with literal values:
5 double dist = getDistance(0.0, 0.0, 10.0, 20.0);
6
7 //invoke a function with a combination:
8 double p = 1500.0;
```

```
double r = 0.05;
double monthlyPayment = getMonthlyPayment(p, r, 60);
```

By default, all primitive types including `int`, `double`, and `char` are passed by value. To be able to pass arguments by reference, we need to use *pointers*.

18.2. Pointers

Consider the following line of C code.

```
int a = 10;
```

This line creates an integer variable and sets it equal to 10. In more detail, this line creates a spot in memory (typically 32 bits) and stores a binary representation of the value 10 at that location. In many instances, we don't care *where* the variable is stored in memory. However, we may have need to communicate that memory location to other functions. To do so, we can use *pointers*.

A pointer in C is a reference to a memory location. Because different types (`int`, `double`, `char`) take a different amount of memory, it is necessary to have a pointer for each type. That is, a pointer that points to a memory location that stores an `int` or a pointer that points to a memory location that stores a `double`, etc.

The syntax for declaring a pointer is to use an asterisk.

```
//regular variable declarations
int a;
double b;

//pointer variable declarations
int *ptrA;
double *ptrB;
```

If `ptrA` represents a memory location, what values can it take on? A memory location is just a number, so you *could* do something like the following.

```
ptrA = 10;
```

Though syntactically this makes sense (and generally the compiler will let you do this with at most a warning), it is not really what you want. This assigns to the pointer variable `ptrA` the value 10, which will be interpreted as the *memory address* 10. This memory address may not belong to your program, or it may not even exist as a valid memory address. Attempts to access the value stored at an arbitrary memory location may be illegal and may result in the operating system killing the program with a segmentation fault or similar error.

There are many reasons why a program should not be allowed access to arbitrary memory locations, but one of the prime reasons is security. Imagine if the operating system allowed a program access to any part of memory; in particular memory that contained sensitive information such as passwords or secret Secure Sockets Layer (SSL) keys. To prevent this, operating systems generally only allow a program to access its own memory.

Referencing Operator

So how do we assign a valid value to a pointer? We do so using a *referencing operator*. Given a normal variable as above, we place an ampersand in front of it to get the memory address of the variable. For example:

```
1 ptrA = &a;
2 ptrB = &b;
```

The operation `&a` results in the memory address of the variable `a` and we can assign it to a pointer value. The pointer type and variable type should match; an integer pointer should point to an `int`, a double pointer should point to a `double`. Making a `double` pointer point to an `int` variable type such as

```
ptrB = &a;
```

is valid syntax, but since the two types use different amounts of memory you may get garbage results.

There is a special value used in C called `NULL` which is a (case-sensitive) keyword used for an uninitialized, undefined, empty or otherwise invalid or meaningless value. In the context of memory locations, `NULL` "points" to nothing. As with regular variables, its best practice to initialize pointer values to `NULL`. For example,

```
int *ptrA = NULL;
```

Without an initialization, the pointer may point to a random memory address which may be dangerous to attempt to access. You can also test whether or not a pointer points to

`NULL` using the usual equality operator.

```
int *ptrA = NULL;
...
if(ptrA == NULL) {
  printf("Error: invalid memory location\n");
}
```

Dereferencing Operator

Once we have a valid pointer to a memory location, we may want to manipulate the contents of the memory it references. To do this we use the inverse operation, the *dereferencing operator* which again uses an asterisk. Given a pointer variable `ptrA`, we apply an asterisk in front of it to turn it into a regular variable. Consider the following example.

```
//declare a normal integer variable
int a = 10;
//declare a pointer and initialize it to NULL
int *ptrA = NULL;

//point ptrA to a's memory location
ptrA = &a;

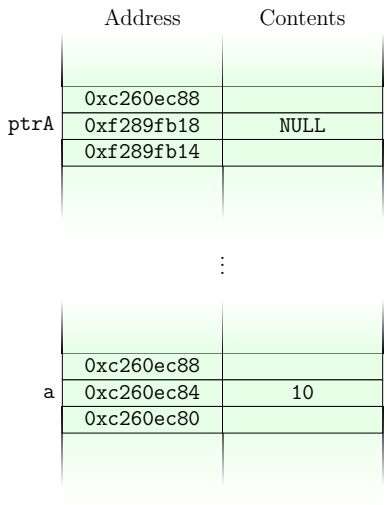
//change the value of the variable a using its pointer
*ptrA = 20;

//now the variable a has a value of 20:
printf("a = %d\n", a); //prints "a = 20"
```

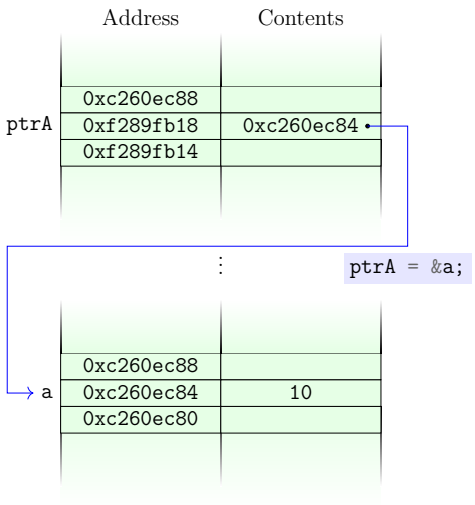
Figure 18.1 depicts how these lines of code operate in memory.

18.2.1. Passing By Reference

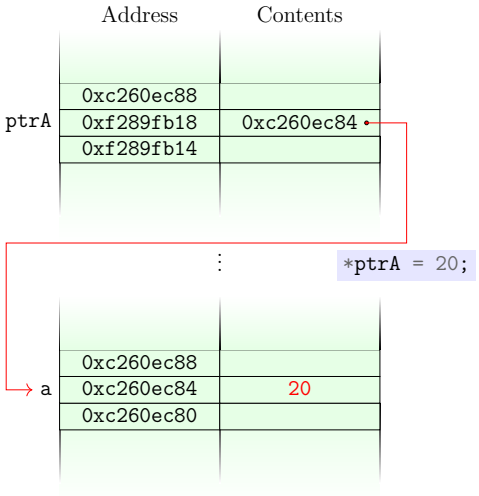
Now that we have the ability to reference a memory location using pointers, we can write functions that pass variables by reference. To do so, we use the same asterisk syntax



(a) After the first two lines memory has been dedicated for the variable `a` and the pointer variable `ptrA` and their values have been initialized.



(b) Making `ptrA` point to the variable `a`'s memory location. The value stored in the variable `ptrA` is a memory address.



(c) Dereferencing `ptrA` and assigning a value changes the value stored at what it *points* to.

Figure 18.1.: Pointer Operations. Pointers can be made to point to other variable's memory locations. You can manipulate/access values of variables via their pointers using dereferencing.

used with pointer variables.

```
//prototypes
/**
 * This function sums the first two variables (passed by
 * value) and places the result into the third variable
 * (passed by reference).
 */
void sum(int a, int b, int *c);

/**
 * This function swaps the values stored in the
 * two variables passed by reference.
 */
void swap(int *a, int *b);
```

In the function definitions, we can use the dereferencing operator to access or modify the value stored in the variable pointed to by the pointer.

```
void sum(int a, int b, int *c) {
  int x = a + b;
  *c = x;
  return;
}

void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  return;
}
```

To invoke these functions, we need to pass pointers to the functions appropriately. We could do this by creating pointer variables or using the referencing operator directly.

```
int x = 10;
int y = 20;
int c;
```

```
4  int *ptrC = &c;
5  sum(x, y, ptrC);
6  //at this point c contains the value 30
7
8  swap(&x, &y);
9  //at this point, the values in x and y have been swapped
10 // x contains 20 and y contains 10
```

This should look familiar. We saw this same syntax when we used `scanf()` to read input from the standard input. We needed to place an ampersand in front of each variable in order to pass the variable by reference so that `scanf()` could place the results into the respective memory locations. If the variables had been passed by value, then `scanf()` would not have been able to manipulate their values.

You can also specify functions to *return* pointers which we discuss in detail in Chapter 20.

18.2.2. Function Pointers

Functions are just pieces of code that reside somewhere in memory just as variables do. Since we can create pointers that point to variables, it makes sense to be able to create variables that point to functions too! These are referred to as *function pointers*.

The syntax for declaring function pointers is similar to variable pointers. However, since a function's signature involves a return type and parameter list, these need to be specified. For example, suppose we wanted to create a function pointer that could point to the math library's `sqrt()` function which takes a single `double` parameter and returns a `double` value.

```
double (*ptrToSqrt)(double) = NULL;
```

The above line creates a function pointer that can point to any function that takes a single `double` parameter and returns a `double` value. As is good practice, we've initialized it to point to `NULL`. The function pointer itself is named `ptrToSqrt`. To make it point to the `sqrt()` function we can use the following syntax.

```
ptrToSqrt = sqrt;
```

This is because a function's identifier acts as a pointer as well! Once we have a pointer to a function, we can invoke the function via its pointer as we would any other function call.

```
double x = ptrToSqrt(2.0);
```

Some more examples:

```
//this pointer can point any function that takes
//three arguments: an int, double, and a char
//and returns an int value
int (*ptrToFunc)(int, double, char)= NULL;

double x;
double (*ptr)(double) = NULL;
//we can make it point to sqrt:
ptr = sqrt;
x = ptr(2.0); //x contains 1.4142...
//or we can make it point to fabs
ptr = fabs;
x = ptr(-10.5); //x contains 10.5
```

You generally want to create and use function pointers when passing and returning functions as arguments to other functions as callbacks. We discuss this in further detail in Chapter 25.

18.3. Examples

18.3.1. Generalized Rounding

Recall that the standard math library provides a `round()` function that rounds a number to the nearest whole number. We've had need to round to cents as well. We now have the ability to write a function to do this for us. Before we do, however, let's think more generally. What if we wanted to round to the nearest tenth? Or what if we wanted to round to the nearest 10s or 100s place? Let's write a general purpose rounding function that allows us to specify *which* decimal place to round to.

The most natural input values would be to specify the place using an integer exponent. That is, if we wanted to round to the nearest tenth, then we would pass it -1 as $0.1 = 10^{-1}$, -2 if we wanted to round to the nearest 100th, etc. In the other direction, passing in 0 would correspond to the usual round function, 1 to the nearest 10s spot, and so on. Moreover, we could demonstrate good code reuse (as well as procedural abstraction) by *scaling* the input value and reusing the functionality already provided

in the math library's `round()` function. We could further define a `roundToCents()` function that used our generalized round function.

Let's also think about organization. We could place the prototypes into a `round.h` header file and the corresponding definitions in a `round.c` source file. The contents of these two files are presented here:

```
/**
 * Rounds to the nearest digit specified by the place
 * argument.  In particular to the (10^place)-th digit
 */
double roundToPlace(double x, int place);

/**
 * Rounds to the nearest cent
 */
double roundToCents(double x);
```

```
#include <math.h>
#include "round.h"

double roundToPlace(double x, int place) {
  double scale = pow(10, -place);
  double rounded = round(x * scale) / scale;
  return rounded;
}

double roundToCents(double x) {
  return roundToPlace(x, -2);
}
```

Observe that neither of these files contains a `main()` function. By themselves they would not be able to be compiled into an executable program. We've essentially built a small *library* of rounding functions. We could compile them though into a binary *object* file using gcc (something like `gcc -c round.c`). We could then link into the object file when compiling an executable program that uses these functions.

18.3.2. Quadratic Roots

Another advantage of passing variables by reference is that we can "return" multiple values with one function call. Functions are limited in that they can only return at most one value. But if we pass multiple parameters by reference, the function can manipulate the contents of them, thereby communicating (though not strictly returning) multiple values.

Consider again the problem of computing the roots of a quadratic equation,

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

using the quadratic formula,

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

Since there are two roots, we may have to write two functions, one for the "plus" root and one for the "minus" root both of which take the coefficients, a, b, c as arguments. However, if we wrote a single function that took the coefficients as parameters by value as well as two other parameters by reference, we could compute *both* root values and place each one in two pass by reference variables.

```
1  void quadraticRoots(double a, double b, double c,
2                      double *root1, double *root2) {
3    double discriminant = sqrt(b*b - 4*a*c);
4    *root1 = (-b + discriminant) / (2*a);
5    *root2 = (-b - discriminant) / (2*a);
6    return;
7  }
```

By using pass by reference variables, we avoid multiple functions. We also note that the return value in this case is unused since we are "returning" the root values in the two pass by reference variables. This frees up the return value to be used to communicate *errors* to the calling function. Recall that there could be several "bad" inputs to this function. The roots could be complex values, the coefficient a could be zero, etc. And now that we are dealing with pointers, the pointers could be invalid (point to `NULL`). In the next chapter, we examine how we can use the return value to communicate different errors to the calling function, letting it *handle* those errors.

19. Error Handling

The C language does not support exceptions or exception handling. Instead, the usual method of error handling is done through defensive programming. As a user, it is your responsibility to write code that checks for invalid or unsafe operations before executing them and handle the error appropriately.

One way that we've already seen to handle errors in C is to use the `exit()` function in the standard library. This function immediately terminates the execution of our program and "returns" an integer valued "exit code." This exit code can be used to communicate errors between different processes. The process that executed the program can use it to determine if the program exited with or without an error. By convention, 0 indicates an error while a non-zero value indicates an error. Using the `exit()` function should only be done when an error should be considered *fatal* or severe enough that the program should immediately terminate.

19.1. Language Supported Error Codes

In the standard libraries, some functions are designed to indicate an error by returning a special "flag" value such as -1 or `NULL`. In general, though, the standard library functions communicate an error code represented as an integer value. As previously mentioned, C uses the convention that 0 indicates "no error" while a non-zero value indicates an error.

Error codes are communicated through a special global variable called `errno` which is defined in the `errno.h` standard header file. This header file also defines several macros that are identified with various error codes. The C standard actually only requires the following three error codes to be supported.

- `EDOM` indicates an error in the *domain* of a function; that is, an error with respect to the function's input value(s). For example, calling the math library's `sqrt(-1)` on a negative value results in an `EDOM` error as C does not support complex numbers.
- `ERANGE` indicates an error in the *range* of a function; that is, an error with respect to the function's output value(s). For example, calling the math library's `log(0)`

with a zero value results in an `ERANGE` error as C does not support $-\infty$ as an actual number.

- `EILSEQ` indicates an illegal byte sequences in characters on systems that use UTF-8.

All three of these are defined in the `errno.h` header file. Depending on the system, additional error codes may also be defined and supported (see POSIX Error Codes below).

When an error occurs, a function will set the global variable `errno` to one of these error code values. Upon returning from a function, you can check for these error codes. Since these error codes are represented as integers, you simply use the numerical comparison operator, `==`. You can check for no error by making a comparison to zero.

In addition, the standard string library, defined in the header file, `string.h` provides a function, `char * strerror(errno)` that can be used to map the value in `errno` to a human-readable error message. We discuss strings in detail later on, but we can see how to use this function in Code Sample 19.1. The output of this program is as follows.

```
result: 1.4142, error: 0
result: -nan, error: 33
it was an EDOM error
Error Message: Numerical argument out of domain
result: -inf, error: 34
it was an ERANGE error
Error Message: Numerical result out of range
```

For this particular system, the `EDOM` and `ERANGE` error codes were associated with the integer values 33 and 34 respectively. These numbers are not necessarily the same on all systems so comparisons must be made against the macro names for portability.

19.1.1. POSIX Error Codes

POSIX is an IEEE set of standards for maintaining compatibility between various operating systems. It defines several rules and expectations that operating systems must adhere to in order to be POSIX *compliant*. Such standards allow developers to develop code that should be interoperable among a collection of different operating systems, reducing the need for rewrites and reimplementations for different systems.

In particular, POSIX compliant systems define many other error codes (see [3]) that

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

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

  double a = -1, b = 2, c = 0.0;
  double x;

  //okay
  x = sqrt(b);
  printf("result: %.4f, error: %d\n", x, errno);

  //NaN and EDOM error...
  x = sqrt(a);
  printf("result: %.4f, error: %d\n", x, errno);

  //make a comparison
  if(errno == EDOM) {
    printf("it was an EDOM error\n");
  }

  //error messages can be accessed via:
  char *errMsg = strerror(errno);
  printf("Error Message: %s\n", errMsg);

  //ERANGE error
  x = log(c);
  printf("result: %.4f, error: %d\n", x, errno);

  if (errno == ERANGE) {
    printf("it was an ERANGE error\n");
  }

  //error messages can be accessed via:
  errMsg = strerror(errno);
  printf("Error Message: %s\n", errMsg);

  return 0;

}
```

Code Sample 19.1.: Using the `errno.h` library

can be used beyond the three mentioned above. For example, the `ENOENT` error code corresponds to "No such file or directory" and `EACCES` corresponds to a "Permission denied" error.

19.2. Error Handling By Design

In our own code we *could* communicate errors to calling functions by setting the `errno` variable, however, we may run into compatibility issues with the standard error codes or POSIX error codes. Instead, it may be more appropriate to do error handling by utilizing the return value of a function to communicate an error code. We could design our functions to always return an `int` value to indicate an error: 0 for no error and some non-zero value to indicate various different types of errors. Of course, as a consequence any value that needs to be "returned" to the calling function would need to be done so via a pass by reference variable.

As an example, let's revisit the quadratic roots example in Section 18.3.2. By returning the two output values via pass by reference variables, we freed up the return value. We can now modify our function to return an integer indicating an error code instead. Previously we had identified several different types of errors: division by zero (if $a = 0$), complex roots (if $b^2 - 4ac < 0$) and a `NULL` pointer error if the variables passed by reference were `NULL`. We can now modify our function to check for these errors and return an appropriate error code. We return zero in the event that no error was encountered.

```
1  int quadraticRoots(double a, double b, double c,
2                     double *root1, double *root2) {
3    if(a == 0) {
4      return 1;
5    } else if( b*b - 4*a*c < 0 ) {
6      return 2;
7    } else if(root1 == NULL || root2 == NULL) {
8      return 3;
9    }
10   double discriminant = sqrt(b*b - 4*a*c);
11   *root1 = (-b + discriminant) / (2*a);
12   *root2 = (-b - discriminant) / (2*a);
13   return 0;
14 }
```

Now when a function invokes our `quadraticRoots()` it can check to see what kind of

error code it returned and handle the error in whatever way it wants.

There is still an issue, however. The usage of the integers 1, 2, 3 to indicate the various errors was arbitrary. These are essentially magic numbers that the calling function would have to deal with by making comparisons with various integers. The numbers themselves are meaningless and someone using them would need to constantly refer to documentation to understand which integer corresponded to which error condition.

It would be much better if we could follow the strategy of the `errno` and define human-readable identifiers for each error code. We could accomplish this by defining macros, but another solution is to use an *enumerated type*.

19.3. Enumerated Types

C allows you to define enumerated types which allow you to define a fixed list of possible values. For example, the days of the week or months of the year are possible values used in a date. However, they are more-or-less fixed (no one will be adding a new day of the week or month any time soon). An enumerated type allows us to define these values using human-readable keywords (much like the `#define` macro).

To declare an enumerated type we use the keywords `typedef enum` (short for type definition and enumeration). We then provide a comma delimited list of keywords inside a code block. At the end of the code block we provide the type's identifier. That is, the name of the type itself. For example, the names of integer and floating-point types in C are `int` and `double` respectively. This identifier gives our type a name that we can later use to declare variables of that type. Consider the following example.

```
1 typedef enum {
2   SUNDAY,
3   MONDAY,
4   TUESDAY,
5   WEDNESDAY,
6   THURSDAY,
7   FRIDAY,
8   SATURDAY
9 } DayOfWeek;
```

In this example we've defined an enumeration of the days of the week. The name of the type itself is `DayOfWeek` and we can now declare variables of this type. The possible

values it can take are `SUNDAY`, `MONDAY`, etc. and we can use these keywords in our program. For example,

```
DayOfWeek today = MONDAY;

if(today == SUNDAY || today == SATURDAY) {
  printf("It is the weekend!\n");
}
```

Note the modern naming conventions: the type identifier uses upper camel casing while the enumerated values follow an uppercase underscore convention. Though our example does not contain a value with multiple words, if it had, we would have used an underscore to separate them. Furthermore, enumerated type declarations are usually placed in separate header files along with function prototype declarations.

Care must be taken when using enumerated types, however. Internally, C simply associates integers with the values. Thus, in our example, `SUNDAY` is actually `0`, `MONDAY` is `1`, and `SATURDAY` is `6`. When we do assignments or equality comparisons, we're actually just comparing integers. Consequently, a `DayOfWeek` variable *may* be assigned values that do not correspond to our enumeration. For example,

```
DayOfWeek today = 1000;
```

is valid code and will not (in general) result in any compiler errors or warnings, even though it is assigning an invalid value to the variable. You should only assign valid values to an enumerated type variable. Proper error checking should also be done.

Despite this limitation, using enumerated types in C provides an obvious advantage. Without an enumerated type we'd be forced to use a collection of magic numbers to indicate values. Even for something as simple as the days of the week we'd be constantly trying to remember: which day is Wednesday again? I forget, does our week start with Monday or Sunday? Etc. By using an enumerated type these questions are mostly moot as we can use the more human-readable keywords and eliminate the guess work.

19.4. Using Enumerated Types for Error Codes

Let's apply enumerated types to our `quadraticRoots()` function example from before. First, we define our enumerated type which includes all types of errors including a `NO_ERROR` type. Note that we start with `NO_ERROR` as its value will be zero following our convention.

```
typedef enum {
  NO_ERROR,
  DIV_BY_ZERO_ERROR,
  COMPLEX_ROOT_ERROR,
  NULL_POINTER_ERROR
} ErrorCode;
```

Now in the `quadraticRoots()` function, we can return the appropriate error code as an enumerated type value.

```
ErrorCode quadraticRoots(double a, double b, double c,
                         double *root1, double *root2) {
  if(a == 0) {
    return DIV_BY_ZERO_ERROR;
  } else if( b*b - 4*a*c < 0 ) {
    return COMPLEX_ROOT_ERROR;
  } else if(root1 == NULL || root2 == NULL) {
    return NULL_POINTER_ERROR;
  }
  double discriminant = sqrt(b*b - 4*a*c);
  *root1 = (-b + discriminant) / (2*a);
  *root2 = (-b - discriminant) / (2*a);
  return NO_ERROR;
}
```

20. Arrays

C allows you to declare and use arrays. Since C is statically typed, arrays must also be typed when they are declared and may only hold that particular type of element. C supports the use of both static arrays and dynamic arrays through standard library calls.

20.1. Basic Usage

To declare a static array, you use syntax similar to declaring a regular variable, but you use square brackets to designate the variable as an array. Within the square brackets you indicate the size (number of elements) in the array. For example,

```
1 int arr[5];
2 double values[10];
```

The two declarations above create arrays of size 5 and 10 respectively. The array types are also defined using the usual keywords. In this case, `arr` can only hold `int` values and `values` can only hold `double` values. Arrays follow the same naming rules and conventions as regular variables. Many times, identifiers are made plural as an array naturally holds more than one value.

Another way to declare an array is to use the compound declaration and assignment syntax whereby you can initialize an array to hold a certain list of values.

```
1 int a[] = { 2, 3, 5, 7, 11 };
```

Using this syntax we do not need to specify the size of the array as the compiler is smart enough to count the number of elements we've provided. The elements themselves are denoted inside curly brackets and delimited with commas.

Variable Length Arrays

C99 introduced Variable Length Arrays (VLAs, which are also supported in GNU C89) which allow you to declare a static array whose size is determined by a variable. For example,

```
int n = 5;
int arr[n];
```

or within a function,

```
void foo(int n) {
  int arr[n];
  ...
}
```

In either case, care must be taken as static arrays are allocated on the stack. The stack is generally small and allocating even a moderately large array on the stack may lead to a stack overflow. In addition, VLAs are not supported in any C++ standard and should be avoided if portable code is desired.

Indexing

Once an array has been created, its elements can be accessed using indexing. C uses the standard 0-indexing scheme so the first element is at index 0, the second at index 1, etc. Indexing an element involves using the square bracket notation and providing an index. Once indexed, an array element can be treated as a normal variable and can be used with other operators such as the assignment operator or comparison operators.

```
arr[0] = 42;
if(arr[4] < 0) {
  printf("negative!\n");
}
printf("arr[1] = %d\n", arr[1]);
```

Recall that an index is actually an offset. The compiler and system know exactly how many bytes each `int` element takes and so an index `i` calculates exactly how many bytes from the first element the i-th element is located at. Consequently it is possible to index elements that are beyond the range of the array. For example, `arr[-1]` or `arr[5]` would attempt to access an element immediately *before* the first element and immediately *after* the last element. Obviously, these elements are not part of the array.

If these out-of-bound elements represent a memory space that does not belong to our program, then it is likely that a segmentation fault will occur and terminate our program. However, it is also likely that the memory space surrounding our array *does* belongs to our program. Still, accessing those blocks of memory may not give us meaningful values and modifying them could corrupt other variable values or generally lead to *undefined behavior*. It is our responsibility as programmers to write code that does not go beyond the bounds of an array.

How can we keep track of the size of the array so that we do not access elements after the last element? With C the responsibility again falls to us. We must always have secondary variables that store the size of an array. If we pass an array to a function (see below), we must also pass a "size" parameter so that the function knows how big the array is.

Iteration

C provides no foreach loop to iterate over an array. Instead, the most natural way to iterate over the elements is a normal for loop that increments an index variable.

```
1 int i, n = 10;
2 int arr[n];
3 for(i=0; i<n; i++) {
4   arr[i] = 5 * i;
5 }
```

The for loop above initializes the variable `i` to zero, corresponding to the first element in the array. The continuation condition is specifies that the loop continues while `i` is strictly less than the size of the array. This iteration for loop is idiomatic when dealing with arrays. An alternative was presented in Section 17.4.

20.2. Dynamic Memory

Recall that static arrays have many shortcomings (see Section 7.2). In general they should be avoided since stack space is limited and they cannot be returned from functions. Fortunately, C provides several standard library functions that facilitate the creation and management of dynamically allocated arrays.

The primary function to allocate memory is the **m**emory **alloc**ation function, `malloc()`:

```
void * malloc(size_t size);
```

which takes a single parameter, the number of bytes that you wish to allocate. The type, `size_t` is an alias for an unsigned integer type, so you can think of it as an integer. Thus, `malloc(200)` would allocate 200 bytes and return a pointer to the memory space. In some instances, the allocation may fail. For example, if the program or system has run out of available memory or you simply request too much. In the event of a failure, `malloc()` returns a `NULL` pointer. The returned value can thus be checked to see if the allocation was successful or not.

We don't have to manually calculate how many bytes we need for an array of a particular size. The macro `sizeof()` can be used to determine the number of bytes any type of variable requires on a system. For example, `sizeof(int)` gives the number of bytes an `int` takes while `sizeof(double)` gives the number of bytes for a `double`, etc. Thus, if we want to allocate an array of 100 integers, we could call `malloc()` as

```
malloc(100 * sizeof(int));
```

Using `sizeof()` is actually preferable as some systems may use a different number of bytes for various types.

Finally, note the return type of `malloc()`: it is a *void pointer*. The `malloc()` function simply allocates chunks of memory. It doesn't care that you intend to use the memory to store integers or floating-point numbers. Thus, `malloc()` returns a "generic" pointer, simply an address in memory. Once we have that pointer we can treat it as an integer pointer, `int *` or a floating-point pointer, `double *` depending on what we want to store.

One way of doing this is to explicitly *cast* the void pointer as the pointer that we want. Some examples:

```
1 int *arr = NULL;
2 double *values = NULL;
3
4 arr = (int *) malloc(sizeof(int) * 10);
```

```
5  values = (double *) malloc(sizeof(double) * 100);
```

The pointer cast is just like when we casted `int` types as `double` types so that we could perform division without truncation. In this case, we convert the returned generic void pointer into a `int` pointer and `double` pointer respectively.[1]

Once created, a dynamic array can be used just like a static array. You use the array's identifier as well as an index to access or modify each element. The same rules and pitfalls apply, so care must be taken to not access elements outside the bounds of the array. Finally, when using `malloc()` it is important to understand that the memory that is allocated is uninitialized. Just as with variables you cannot make any assumptions as to the contents of the memory space that is allocated. It may contain garbage values, it may contain the contents that occupied the memory last time it was used, etc. A full example:

```
1  int n = 100;
2  int *arr = NULL;
3  arr = (int *) malloc(sizeof(int) * n);
4  if(arr == NULL) {
5    fprintf(stderr, "unable to allocate memory!\n");
6    exit(1);
7  }
8  for(i=0; i<n; i++) {
9    arr[i] = (i+1) * 10;
10   printf("a[%d] = %d\n", i, arr[i]);
11 }
```

There are other functions that can be used to allocate memory in C. For example, `calloc()` is similar to `malloc()` but initializes the contents to zero (null bytes). `realloc()` can be used to resize an existing memory space (though it may fail if it cannot be expanded).

[1] Performing an explicit pointer cast is actually not necessary in C as the type system will do an implicit cast for us. Whether explicit or implicit types casts are "better" can evoke debates akin to nerd holy wars. Though there are advantages and disadvantages to both [15], we perform explicit pointer casts in this book as clarity and intent is more important than brevity. Even more important is that explicit pointer casts are *necessary* in C++ so doing so in our C code makes our code more portable.

Deallocation

Once dynamically allocated memory is no longer needed, we should release it so that it can be reused by the program or the operating system. The `free()` function in the standard library does this for us. All we need to do is provide the pointer to `free()` and it deallocates the memory block.

```
free(arr);
```

Once freed, accessing old memory pointed to by the `arr` pointer is undefined behavior and may lead to unexpected or fatal results.

20.3. Using Arrays with Functions

Since arrays are represented using pointers, we can pass and return arrays to and from functions simply by passing in a pointer. When passing an array to a function, we also need to make sure that we communicate to the function the size of the array as there is no reliable way for the function to determine this. By passing both a pointer and an integer representing the size, we are implicitly indicating that we are passing an array and not just a single value.

As an example, the following function takes an array of integers and computes their sum.

```
/**
 * This function computes the sum of elements in the
 * given array which contains n elements
 */
int computeSum(int *arr, int n) {
  int i;
  int sum = 0;
  for(i=0; i<size; i++) {
    sum += arr[i];
  }
  return sum;
}
```

In this example we had no need to make changes to any of the elements in the array. However, the array was still passed by reference, meaning we could have. When passing arrays, we can use the keyword `const` (short for constant) to explicitly indicate that no changes will be made to the array elements. For example,

```
int computeSum(const int *arr, int n)
```

This is enforced by the compiler: if we do attempt to make changes to a `const` array, it will be a compiler error.

We can also create an array in a function and return it as a value. As previously discussed, we will need to do so by creating a dynamically allocated array. For example, the following function creates a deep copy of the given integer array. That is, a completely new array that is a distinct copy of the old array. In contrast, a shallow copy would be if we simply made one reference point to another reference.

```
1  /**
2   * This function creates a new copy of the given integer
3   * array which contains n elements and returns a pointer
4   * to the new copy.
5   */
6  int * makeCopy(const int *a, int n) {
7    int *copy = (int *)malloc(sizeof(int) * n);
8    int i;
9    for(i=0; i<n; i++) {
10     copy[i] = a[i];
11   }
12   return copy;
13 }
```

The function returns an integer pointer. Here we have a similar problem with respect to the size of the array. We only have one return value, which must be the pointer. In this particular example the calling function knows how big the array is because it specified the size by passing in `n`. In general we could have communicated the size of the returned array through another pass by reference integer variable.

20.4. Multidimensional Arrays

C supports multidimensional arrays both static and dynamically allocated; though we'll only focus on dynamically allocated 2-dimensional arrays. Conceptually, a 2-dimensional

array of, say, integers can be modeled as an array of pointers that point to an array of integers. That is, a pointer to pointers, for example, `int **`. The initial pointer points to an array of integer pointers, `int *` and each integer pointer points to an array of `int` variables.

To understand this better, let's look at the details of allocation. Consider the following code.

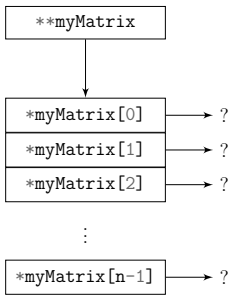
```
1 int i;
2 int **myMatrix = NULL;
3 myMatrix = (int **)malloc(n * sizeof(int*));
4 for(i=0; i<n; i++) {
5   myMatrix[i] = (int *)malloc(n * sizeof(int));
6 }
```

Line 3 invokes `malloc()` to create an array of n integer pointers, that is `int *` types. We then go into a loop to iterate over each of these pointers and invoke `malloc()` again to setup each array. This process is visualized in Figure 20.1. Note the syntax: when we invoke `malloc()` to create an array of pointers, we use `sizeof(int*)` to determine how many bytes each integer *pointer* takes. We also do an explicit type cast of `(int **)` to match our pointer-to-pointer(s) variable `myMatrix`.

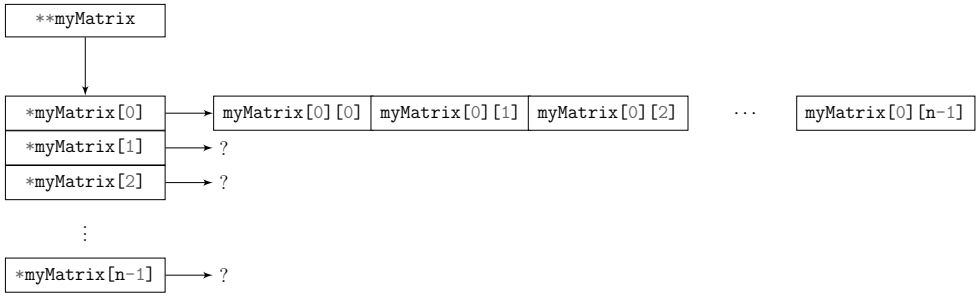
Once the allocation has completed, we can treat `myMatrix` as a normal 2-dimensional array and index each element with two index variables.

```
1 int i, j;
2 for(i=0; i<n; i++) {
3   for(j=0; j<n; j++) {
4     myMatrix[i][j] = 0;
5   }
6 }
```

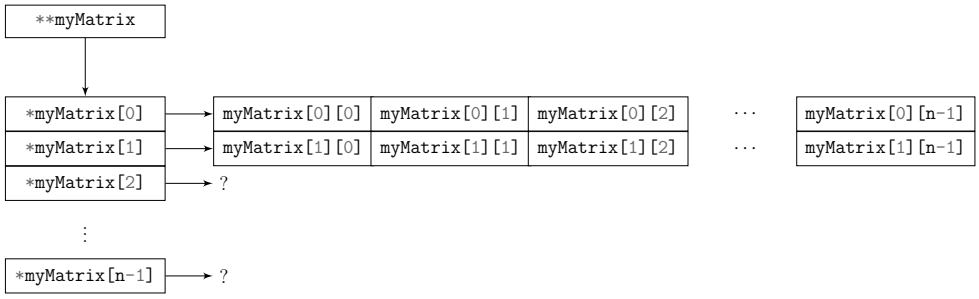
To deallocate and free multidimensional arrays, we need to work backwards. If we immediately freed the pointer-to-pointers, `free(myMatrix);`, we would lose all references to the individual array "rows" and would not be able to free them, resulting in a memory leak. We need to free up each row first, *then* we can free the pointer-to-pointers.



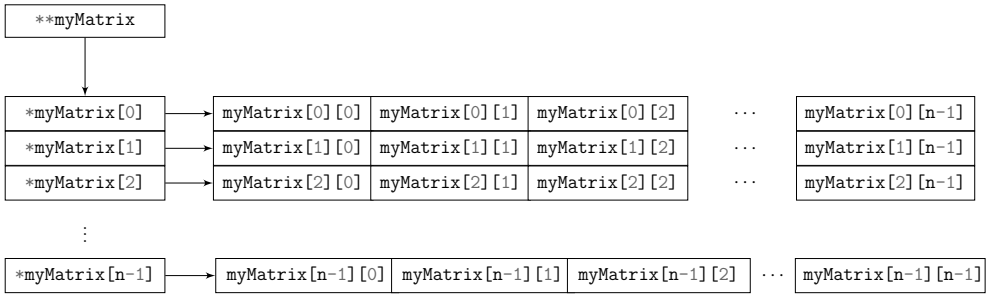
(a) Initialization of the pointer-to-pointers. The first invocation of `malloc()` sets up an array of integer pointers, `int *` that are uninitialized (where they point to is undefined).



(b) Initialization of the first pointer. On the first iteration of the `for` loop when `i = 0`, the first "row" is initialized when `malloc()` is invoked.



(c) Initialization of the second pointer. On the second iteration of the `for` loop when `i = 1`, the second "row" is initialized.



(d) After the termination of the `for` loop, each row has been initialized and the pointer-to-pointers can be treated like a 2-dimensional array.

Figure 20.1.: The process of dynamically allocating a 2-dimensional array of integers using `malloc()` in a for-loop.

```
for(i=0; i<n; i++) {
  free(myMatrix[i]);
}
free(myMatrix);
```

20.4.1. Contiguous 2-D Arrays

The example depicted in Figure 20.1 constructs a two dimensional array. However, each "row" of the array was created using an independent call to `malloc()` which may result in non-contiguous memory blocks (each row may not be located in the memory address immediately following the previous row). In general, this is not a problem for most situations (we can expect most implementations of `malloc()` to be efficient). However, it might be better in some scenarios if the two dimensional array were all one big block of contiguous memory.

We can achieve this by a single call to `malloc()` and some clever pointer manipulation. We do this very similar to the previous example. Our first call to `malloc()` to set up the pointer-to-pointers is the same. However, instead of calling `malloc()` over and over in a for loop, we make one single call asking for a number of bytes to accommodate the entire $n \times m$ array. We then store the result at the "beginning" of the array (at index zero). In the following example, we will create a 4×3 sized matrix.

```
int **m  = (int **) malloc(sizeof(int *) * 4);
m[0] = (int *) malloc(sizeof(int) * (4 * 3));
```

We're not done yet, however. We still need to initialize all of the other pointers. To do so, we dereference the array (once) to get the address of the start of the memory block and then compute an *offset* from this beginning on where the next "row" should be. Since each row has 3 elements, this arithmetic is simple.

```
for(i=1; i<4; i++) {
  arr[i] = (*arr + (3 * i));
}
```

Now we can treat the array like we would any other two dimensional array by specifying two indices.

```
for(i=0; i<4; i++) {
  for(j=0; j<3; j++) {
    arr[i][j] = 10 * i + j;
  }
}
```

Which would result in an array that, conceptually, looks something like the following.

```
[   0   1   2  ]
[  10  11  12  ]
[  20  21  22  ]
[  30  31  32  ]
```

Which are all stored in one large chunk of memory. To see this, we can print out the memory address of each "row" during a particular run of the program:

```
arr[0] = 0x7fe51b403270
arr[1] = 0x7fe51b40327c
arr[2] = 0x7fe51b403288
arr[3] = 0x7fe51b403294
```

Note that each hexadecimal value differs by `0x0c` (that is, 12 bytes):

`0x...70 + 0x0c = 0x...7c`

`0x...7c + 0x0c = 0x...88`

`0x...88 + 0x0c = 0x...94`

This is exactly how many bytes the 3 integers (4 bytes each) take in memory in each "row." The pointer setup is depicted in Figure 20.2.

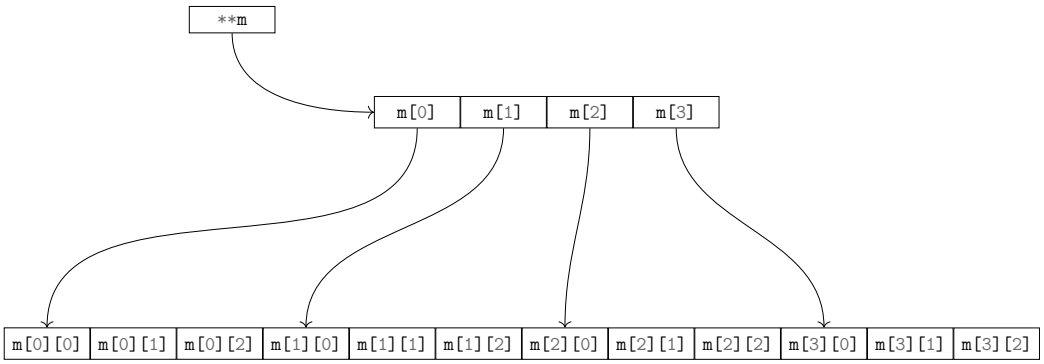


Figure 20.2.: Contiguous Two Dimensional Array. The initial pointer-to-pointers `m` is initialized to an array of pointers, `m[0]..m[3]`. A 4×3 contiguous array is created and each of the pointers is initialized to the proper location so it can be treated as a two-dimensional array.

20.5. Dynamic Data Structures

The C standard library does not provide a variety of advanced, dynamic data structures such as linked lists or sets. As of C89 and POSIX.1 the C standard library does provide hash tables and binary search trees, but does not require implementations of other dynamic data structures. Instead, you must either implement your own or utilize a third-party library such as glibc, the GNU C library (`http://www.gnu.org/software/libc/`).

21. Strings

C has no built-in string type. Instead, strings are represented as arrays of `char` elements. They differ from, say arrays of `int` or `double` types, however, in that they are *null terminated* arrays. The end of the string must *always* be denoted with a null-terminating character, `'\0'` (the 0 valued character in the ASCII table). Failure to properly null-terminate a string may lead to undefined behavior or fatal errors.

C provides a good variety of functions in its standard string library (included in the `string.h` header). All of these functions operate under the assumption that the strings passed to it are null-terminated. It is your responsibility to ensure that they are. Moreover, some of the functions that operate on strings will inset the null-terminating character for us, but others will not. It is important to understand the expectations and guarantees of each function.

21.1. Character Arrays

We've previously used string literals when using the standard input and output functions, `printf()` and `scanf()`. You can also create static strings (as static arrays) using the standard assignment operator. Some examples:

```
1 char firstName[] = "Tom";
2 char lastName[] = "Waits";
```

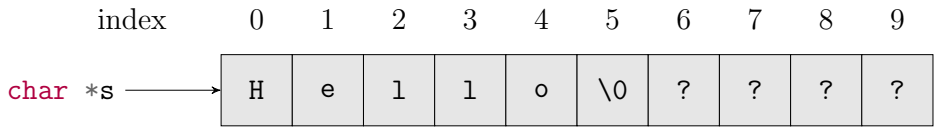


Figure 21.1.: A string in C is achieved by using a `char` array. However, the string is terminated by a null-terminating character, `\0`. Though an array may have space for additional characters, they are irrelevant if the null terminator precedes them.

This syntax can only be used when creating static strings (they are allocated on the stack and locally scoped). The compiler is able to scan the string literal and determine how many characters are needed and even inserts the null-terminating character for us. Thus, the length of the two strings in the example are 3 and 5 respectively, but the array size created for them will be 4 and 6 respectively to accommodate the null-terminating character.

Static strings have the same limitations as static arrays. Since they are allocated on the stack, they cannot be returned from a function. Just as with `int` and `double` types, however, we can dynamically allocate memory to hold `char` types using `malloc()`. The important difference being that we need to always allocate at least one more character to accommodate the null-terminating character.

```
1 char *fullName = NULL;
2 fullName = (char *) malloc(sizeof(char) * 10);
```

This example creates a dynamically allocated `char` array that is able to hold *9* characters (since one will be needed for the null-terminating character). However, once we have created the array, we cannot simply assign an entire string to it using the usual assignment operator.

```
1 //THIS IS *WRONG*:
2 fullName = "Tom Waits";
```

This will compile, but doesn't give us what we want. Recall that `fullName` is a character pointer. Using the assignment operator simply makes it *point* to the static, literal string `"Tom Waits"`. In fact, we lose our reference to the dynamically allocated array that was created with `malloc()`, resulting in a memory leak.

Instead, we need to use a function in the standard library to *copy* a string. The function in the standard library that allows you to copy strings is as follows.

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

The name, `strcpy` is short for "string copy."[1] Both arguments are character pointers. In keeping with the use of the assignment operator, the second argument ("source") is copied into the first, "destination" (just as with an assignment operator, the value

[1]The abbreviations are mostly historic: in the 70s and 80s when memory was measured in kilobytes, saving a few characters made a significant difference.

on the right-hand-side is copied into the variable on the left-hand-side). Moreover, the second argument has been marked as `const` indicating that it will not be changed. The contents of the first argument *will* be changed since we are copying a string into it, erasing whatever contents it had prior. Finally, the function returns a pointer to the `dest` argument, mostly so that it can be used in nested function calls (though we'll avoid such confusingly terse "tricks"). From our example:

```
//"assign" (copy) "Tom Waits" into the string fullName:
strcpy(fullName, "Tom Waits");
```

In addition, we can access and modify individual characters in a string using the usual indexing and assignment operator with `char` literals.

```
//access individual characters:
char firstInitial = fullName[0]; //'T'
char lastInitial = fullName[4]; //'W'

//printing:
printf("First Initial: %c\n", fullName[0]);

//modifying:
firstInitial[0] = 't';
firstInitial[4] = 'w';
//fullName is now "tom waits"
```

21.2. String Library

The standard string library provides many other convenient functions that allow you to process and modify strings. We highlight a few of the more common ones here. A full list of supported functions can be found in standard documentation.

Length

As with regular arrays, we are responsible for ensuring that we do not access characters outside the character array of a string. Since a string is null-terminated, there is a nice

function provided by the string library to determine its length,

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

Recall that `size_t` can essentially be treated as an integer, indicating the number of bytes in the passed string. Since a character is a single byte, this function tells us how many character are in the given string *not including* the null-terminating string. This function is an abbreviation for "string length." Using this function we can easily iterate over each character in a string.

```
1 int i;
2 for(i=0; i<strlen(fullName); i++) {
3   printf("fullName[%d] = %c\n", i, fullName[i]);
4 }
```

Concatenation

A concatenation function is provided that allows you to append one string to the end of another.

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

Similar to the `strcpy()` function, `strcat()` appends the "source" (`src`) string to the end of the "destination" (`dest`) string. It is *your* responsibility to ensure that the destination string is large enough to accommodate the source string. If it is not, then it could lead to undefined behavior as `strcat()` overwrites memory after the end of the destination string. Further, `strcat()` *will* copy the null-terminating character for you so that the resulting string (now stored in `dest`) is a valid null-terminated string.

```
1 char *formattedName = (char *)malloc(11 * sizeof(char));
2 //copy "Waits" into formattedName
3 strcpy(formattedName, lastName);
4 //append a ", " to formattedName
5 strcat(formattedName, ", ");
6 //append the firstName to the formattedName
7 strcat(formattedName, firstName);
8 //formattedName now contains "Waits, Tom"
```

Byte-Limited Versions

C also provides several *byte-limited* versions of the copy and concatenation functions:

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

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

They work similarly in that they copy/concatenate the source, `src` string into the destination, `dest` string. However, there is a third parameter, `n` which specifies *at most* how many bytes to copy/concatenate. The parameter, `n` allows you to limit the number of characters that the operation uses. If either of these functions encounters the null-terminating character before `n` bytes have been copied/concatenated, they stop and copy the null-terminating character for us.

However, these functions do not always handle the null-terminating character for us. If the null-terminating character appears in the first `n` bytes, it will be copied for us, but if it is not, we need to be sure to handle it ourselves. For example,

```
1 char email[] = "twaits@cse.unl.edu";
2 char *login = (char *) malloc(7 * sizeof(char));
3
4 //only copy the first 6 characters:
5 strncpy(login, email, 6);
6 //at this point, login contains "twaits" but is
7 //not null-terminated, so we manually terminate it
8 login[6] = '\0';
```

Computing a Substring

The byte-limited versions of the copy function can be used to compute a substring of another string. The parameter `n` can be used to limit the length of the string, but how might we specify where the substring should start?

For example, in the string `"Thomas Alan Waits"` we may want to get the substring representing his middle name, `"Alan"`. The length is 4 and we can specify that the copying should start by using an index. In this case, we want the copying to start at index `7`, the 8th character. If the string is stored in an array named `name`, this would be `name[7]`. However, indexing an array like this results in a single character and `strncpy()` expects a string (a character *pointer*). Fortunately, we know how to do this:

using the referencing operator, we can turn the 8th character into a character pointer, `&name[7]`. A full example:

```
1 char name[] = "Thomas Alan Waits";
2 char *middleName = (char *) malloc(sizeof(char) * 5);
3 strncpy(middleName, &name[7], 5);
4 middleName[4] = '\0';
5 //middleName now holds "Alan"
```

In the call to `malloc()`, we included space for a null-terminating character. Furthermore, `strncpy()` does not insert the null-terminating character for us in this case. In line 4 we manually insert it.

21.3. Arrays of Strings

We often need to deal with collections of strings. An array of strings can be viewed as a 2-dimensional character array. Indeed, we've seen this before. In the typical C program's `main()` function, command line arguments are passed as the second parameter:

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

which is represented as a double character pointer, `char **`. Conceptually, each row is a string, `char *`. The first parameter, `argc` indicates how many rows there are. The `main()` function doesn't need to be told how big each "row" is since strings are null-terminated.

We can create our own arrays of strings similar to how we created 2-dimensional arrays of `int` and `double` types.

```
1  //create an array that can hold 5 strings
2  char **names = (char **) malloc(5 * malloc(sizeof(char*)));
3  int i;
4  for(i=0; i<5; i++) {
5    //each string can hold at most 19 characters
6    names[i] = (char *) malloc(sizeof(char) * 20);
7  }
8  strcpy(names[0], "Margaret Hamilton");
9  strcpy(names[1], "Ada Lovelace");
10 strcpy(names[2], "Grace Hopper");
```

```
11 strcpy(names[3], "Marie Curie");
12 strcpy(names[4], "Hedy Lamarr");
```

21.4. Comparisons

When comparing strings in C, we cannot use the numerical comparison operators such as `==`, or `<`. Because strings are represented as arrays, using these operators actually compares the variable's *memory addresses*.

```
1 char *a = (char *) malloc(sizeof(char) * 13);
2 char *b = (char *) malloc(sizeof(char) * 13);
3 strcpy(a, "Hello World!");
4 strcpy(b, "Hello World!");
5
6 if(a == b) {
7   printf("strings match!\n");
8 }
```

The code above will not print anything even though the strings `a` and `b` have the same content. This is because `a == b` is comparing the memory address of the two variables. Since they point to different memory addresses (created by two separate calls to `malloc()`) they are not equal.

The C string library provides a standard comparator function to compare strings based on their content:

```
int strcmp(const char *a, const char *b);
```

The function takes two strings and returns an integer based on the lexicographic ordering of `a` and `b`. If `a` precedes `b`, `strcmp()` returns something negative. It returns zero if `a` and `b` have the same content. Otherwise it returns something positive if `b` precedes `a`. Some examples:

```
1 int x;
2 x = strcmp("apple", "banana"); //x is negative
3 x = strcmp("zelda", "mario");  //x is positive
```

```
4  x = strcmp("Hello", "Hello");  //x is zero
5
6  //shorter strings precede longer strings:
7  x = strcmp("apple", "apples"); //x is negative
8  //uppercase precede lowercase:
9  x = strcmp("Apple", "apple");  //x is negative
```

In the last example, `"Apple"` precedes `"apple"` since uppercase letters are ordered before lowercase letters according to the ASCII table. We can also make comparisons ignoring case if we need to using the alternative:

```
int strcasecmp(const char *s1, const char *s2);
```

which is a case-insensitive version. Here, `strcasecmp("Apple", "apple")` will return zero as the two strings are the same ignoring the cases.

The comparison functions also have byte-limited versions,

```
int strncmp(const char *s1, const char *s2, size_t n);
```

and

```
int strncasecmp(const char *s1, const char *s2, size_t n);
```

Both will only make comparisons in the first `n` bytes of the strings. Thus, the comparison, `strncmp("apple", "apples", 5)` will result in zero as the two strings are equal in the first 5 bytes.

21.5. Conversions

We've previously examined the functions `atoi()` and `atof()` that allow you to convert strings that hold numeric values to `int` and `double` values respectively. Another way to convert strings to numbers is to use a function similar to the familiar `scanf()` function which reads its string from the standard input. The `sscanf()` function reads its "input" from a string instead of the standard input.

```
1  char s[] = "103212.3214";
2  double x;
3  sscanf(s, "%lf", &x);
4  //x now has the value 103212.3214
```

The `sscanf()` function differs in its first argument: the string that contains the value you want to parse. Otherwise, the second two arguments are as in `scanf()`: the format (as a string) and the variable(s) that the results should be stored in (passed by reference).

Likewise, there is a companion `sprintf()` which is similar to the `printf()` function, but instead of printing to the standard output, it "prints" to the string. That is, the result is placed in a string.

```
int x = 10;
double y = 3.14;
char *s = (char *) malloc(sizeof(char) * 50);
sprintf(s, "The value of x is %d, y = %f.", x, y);
//s now contains "The value of x is 10, y = 3.140000."
```

21.6. Tokenizing

Recall that *tokenizing* is the process of splitting up a string along some *delimiter*. For example, the comma delimited string, `"Smith,Joe,12345678,1985-09-08"` contains four pieces of data delimited by a comma. Our aim is to split this string up into four separate strings so that we can process each one. The C string library provides a tokenizing function:

```
char *strtok(char *str, const char *delim);
```

which works as follows. The first argument is the string that you want to tokenize and the second contains the delimiter that you want to split along. The second argument is actually a string and allows you to specify more than one delimiter, but we'll restrict our attention to single character delimiters. The function returns a pointer to the first token in the string. To get the second and all subsequent tokens, we call `strtok()` again, but we pass it `NULL` as the first argument to continue parsing the same string.

If we pass a new string as the first argument to `strtok()` the tokenization process will start over on the new string. Note that the first argument does not have the `const` keyword. This is because `strtok()` *will make changes* to the string during the tokenization process. If the string needs to be preserved, tokenization should be performed on a deep copy of the string.

When there are no more tokens in the string, `strtok()` returns `NULL` to indicate no

more tokens are in the string. This logic can be used to write a while loop to iterate over each token. Consider the following example.

```
char data[] = "Smith,Joe,12345678,1985-09-08";
char *token = NULL;
//make the initial call to strtok:
token = strtok(data, ",");
while(token != NULL) {
  printf("token: %s\n", token);
  //get the next token:
  token = strtok(NULL, ",");
}
```

This outputs the following:

```
token: Smith
token: Joe
token: 12345678
token: 1985-09-08
```

It should be noted that `strtok()` is *not* reentrant. It can only work on one string at a time. In a multithreaded application, two threads cannot both use `strtok()` or they would end up processing each other's strings. Even in a non-threaded application, we need to be careful. For example, we cannot process a string using `strtok()` in one function and then call another function that does the same. C does provide a reentrant version, `strtok_r()` that can be used.

22. File I/O

C provides several functions to manipulate and process files. Like other I/O functions, these are all defined in the standard input/output library, `stdio.h`. Writing binary or plaintext data is determined by which functions you use. Whether or not a file input/output stream is buffered or unbuffered is determined by the system configuration. There are some ways in which this can be changed, but we will not cover them in detail.

22.1. Opening Files

Files are represented in C by a `FILE *` pointer type defined in the standard input/output library. As a pointer, it essentially points to the file stored in memory. To open a file, you use the `fopen()` function (short for **f**ile **open**) which takes two arguments and returns a `FILE *` pointer:

```
FILE *fopen(const char *path, const char *mode);
```

The first argument is the file path/name that you want to open for processing. The second argument is a string representing the "mode" that you want to open the file in. There are several supported modes, but the two we will be interested in are reading, in which case you pass it `"r"` and writing in which case you pass it `"w"`. The path can be an absolute path, relative path, or may be omitted if the file is in the current working directory.

```
1  //open a file for reading (input):
2  FILE *input = fopen("/user/apps/data.txt", "r");
3  //open a file for writing (output):
4  FILE *output = fopen("./results.txt", "w");
5
6  if(input == NULL) {
7    fprintf(stderr, "Unable to open input file");
8    exit(1);
9  }
10
11 if(output == NULL) {
```

```
12    fprintf(stderr, "Unable to open output file");
13    exit(1);
14  }
```

The two checks above check that the file opened successfully. If the file opening failed, `fopen()` returns `NULL`. Opening a file can fail for a number of reasons. On POSIX systems for example, additional information can be obtained by accessing the standard error number, `errno` (see Section 19.1). Some errors that can result:

- `ENOENT` – No such file or directory
- `EACCES` – Permission denied
- `ENOMEM` – Insufficient storage space is available

among many other possibilities. These error codes can be used to implement more specific error handling code if desired.

22.2. Reading & Writing

When a file is opened, the file pointer returned by `fopen()` initially points to the beginning of the file. As you read from it or write to it, the pointer advances through the file content.

22.2.1. Plaintext Files

To process a plaintext file we use two functions, `fprintf()` for output and `fscanf()` for input. These two functions should look familiar. They work almost exactly the same as `printf()` and `scanf()` that work with the standard output and standard input respectively. In fact, the standard in/out are both *files* so it makes sense that they can be read from/written to. The `f` prefixed versions are simply generalizations that can be used for any file. The only difference is that the first argument for these two functions is the file you want to output to or read from.

```
1  int x = 10;
2  double y = 3.14;
3
4  //write to a plaintext file
```

```
fprintf(output, "Hello World!\n");
fprintf(output, "x = %d, y = %f\n", x, y);

//read from a plaintext file
fscanf(input, "%d", &x);
fscanf(input, "%lf", &y);

//these are equivalent to printf, scanf:
fprintf(stdout, "Please enter an integer:");
fscanf(stdin, "%d", &x);
```

Using `fscanf()` for arbitrary string input is potentially dangerous as there is limited bounds checking. We must store the input value from the file into a string (character array), but if the file or line contains more characters than the array can accommodate we may have a buffer overflow.

A better way to read input is to use `fgets()` which allows us to limit the number of bytes that are read.

```
char *fgets(char *s, int size, FILE *stream);
```

The first argument is the string that the input data will be read into. The second parameter is how we limit the number of characters that will be read. It actually reads in one fewer character, `size-1` to account for the null-terminating character which `fgets()` automatically inserts for us. The last argument is the file pointer that we wish to read from.

The behavior of `fgets()` is that it reads *up to* `size-1` characters from the input file and places the results into `s`. If `fgets()` encounters either an `EOF` symbol or an endline character, `'\n'` it stops reading. In the case of an endline character, it is included in the result and may need to be chomped out (that is, removed).

The `fgets()` function can be used to process a file line by line until the end of the file is reached. Each line can be processed (perhaps tokenized) individually to extract particular pieces of data. This is typically how a CSV or similar file may be processed. To determine if the end of the file has been reached, you can use the return value of the function: it returns `NULL` when no more characters have been read.

```
FILE *input = fopen("data.txt", "r");
//this assumes that no line is more than 999 characters
char line[1000];
```

```
4  //read the first line
5  char *s = fgets(line, 1000, input);
6  while(s != NULL) {
7
8    //chomp the endline character from line:
9    line[strlen(line)-1] = '\0';
10
11   //process the current buffer
12   //for demonstration, we simply print it:
13   printf("line = %s\n", line);
14
15   //read the next line
16   s = fgets(line, 1000, input);
17 }
```

A similar function, `int fgetc(FILE *stream)` allows us to get a single character from the input file (returned as an `int`) if we prefer to read character by character.

22.2.2. Binary Files

To read and write binary data to files we use two different functions:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

for reading from a file and

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

to write to a file. Both functions have the exact same arguments:

1. `ptr` is a pointer to the item that is being read/written (possibly an array of elements)
2. `size` is the number of bytes that each element stored at `ptr` take, usually determined by using `sizeof()`
3. `nmemb` is the number of elements to be written/read
4. `stream` is the file stream to be read from/written to

These functions allow you to read/write an arbitrary amount of binary data. When reading, it is your responsibility, as usual, to ensure that the `ptr` is large enough to

accommodate the data you are reading into it.

```
int x = 10;

FILE *binaryOutputFile = fopen("demo.bin", "w");

//write a single int to a file:
fwrite(&x, sizeof(int), 1, binaryOutputFile);

FILE *binaryInputFile = fopen("input.bin", "r");

//read a single int from the file:
fread(&x, sizeof(int), 1, binaryInputFile);

//read an entire array from the file:
int *a = (int *) malloc(10 * sizeof(int));
fread(a, sizeof(int), 10, binaryInputFile);
```

22.3. Closing Files

Once you are done processing a file, you should close it using the `fclose()` function:

```
int fclose(FILE *fp);
```

which takes a single argument, the `FILE` pointer of the file you wish to close. Closing an invalid file may result in an error or segmentation fault. In any case, after a file has been closed, it cannot be read from or written to, doing so is undefined behavior. Failure to close a file *may* result in a corrupted file.

23. Structures

Strictly speaking, C is not an object-oriented programming language, it is an *imperative* (or, relatedly a *structured* or *procedural*) programming language. This means that C can be characterized as a language that changes a program's state through statements and the use of function calls. Though C does not have objects, it does support a "weak" form of encapsulation through the use of *structures*. Structures are a user-defined type that collects multiple pieces of data together into one logical unit. Once defined, structures can be used in a program just like any other variable type; structure variables can be declared, passed and returned from functions, pointers to structures can be used, etc. Structures form a "weak" form of encapsulation in that they only provide the grouping of data. The protection of data through visibility keywords is not supported. The grouping of functions that act on that data is also not readily supported.[1] However, even this weak form provides a very useful and convenient way to collect related pieces of ?ata.

23.1. Defining Structures

To define a structure, we use the keyword `struct` along with the previously introduced keyword `typedef`. To motivate an example, let's reconsider a student entity, which has an ID (integer), first and last name (strings), and a GPA (a floating point number). Each of these pieces of data can be encapsulated into a structure as follows.

```
1 typedef struct {
2   int id;
3   char *firstName;
4   char *lastName;
5   double gpa;
6 } Student;
```

Note the syntax and naming conventions:

[1] You *can* define a structure with function pointers as elements, so structures could technically include "methods" but this is not really what most people think of when considering object-oriented paradigms.

- The elements of the structure (also referred to as components or *members*) are included inside curly brackets, delimited using semicolons (in contrast to an enumerated type which is a list, these elements do *not* constitute a list).
- A structure may contain any number of elements of any type.
- The name (identifier) of the structure is provided at the end, ended with a semicolon.
- We use a modern naming convention: each element is named using lower camel casing while the name of the structure itself uses upper camel casing.

In addition, structure declarations are generally placed in a header file along with any any function prototypes that use the structure as a parameter or a return type. Once you have defined a structure you can use it as you would a built-in variable type. For example,

```
Student s;
```

declares a `Student` structure with the variable name `s`.

23.1.1. Alternative Declarations

In some examples and code bases you may find an alternative way of declaring a structure that looks like the following.

```
1 struct Student {
2   ...
3 };
```

This syntax omits the keyword `typedef` and places the structure name at the beginning. The difference is a bit technical (the original syntax creates an anonymous structure with the `Student` identifier placed in the global namespace, while this declaration places the `Student` identifier in the structure namespace), but one of the consequences of using this type of declaration is that the `Student` identifier is not in the global scope, so to declare a variable of the type `Student` you need to further specify that it is a structure using the following syntax.

```
struct Student s;
```

Further, you may also see another style,

```
typedef struct Student {
  ...
} Student;
```

Which places the `Student` identifier in both the global space *and* in the "structure" space. Which style of declaration you use depends on several factors, but for simplicity we'll stick with the first style.

In addition, you may see some older naming conventions use lowercase underscore naming for structure names. In particular, POSIX systems use names that end with `_t` (indicating a structure **t**ype). You should avoid such naming conventions to avoid conflicts.

Finally, though C does not provide a mechanism for the protection of data, many libraries and code bases will begin certain member variables with underscore(s) to indicate that they are "internal" variables used by the library and should *not* be accessed or modified. For example, `_someVariable` or `__anotherVariable` would indicate "private" variables that should not be tampered with. The C language itself would still allow you access and modify the variables, but doing so may violate assumptions and expectations of the library code, leading to undefined behavior.

23.1.2. Nested Structures

Consider adding another member variable to the `Student` structure to model a student's date of birth. How might we model a date? Unix systems model time as a single integer value that represents the number of milliseconds that have passed since the Unix *epoch*, January 1st, 1970 (also known as Coordinated Universal Time (UTC)). For example, the New Horizons space probe was launched on January 19th, 2006 at 19:00:00 UTC. This corresponds to 1,137,697,200, roughly 1.137 billion milliseconds since the epoch. Another way to model time is as a string. There are several standards for time representations, but the most common is ISO standard 8601 [22] which specifies time using a format that includes a date, time and an offset from Greenwich Mean Time (GMT). For example, the New Horizons launch date/time would be represented as `"2006-01-19T19:00:00+00:00"`.

To keep things simple, we'll model our date using three numbers: a year, a month and a date. But how should we implement this? Conceptually, a date is a single entity, separating these three numbers doesn't make much sense. This is a perfect opportunity to define another structure.

```
typedef struct {
  int id;
  char *firstName;
  char *lastName;
  double gpa;
  Date dateOfBirth;
} Student;
```

Code Sample 23.1.: A `Student` structure declaration

```
typedef struct {
  int year;
  int month;
  int date;
} Date;
```

Once we have defined a structure we can use it as we would a normal variable, so it makes sense that we could include it in another structure.

This is a good illustration of a form of *composition* where one structure may be *composed* of other structures. Since the `Student` structure "owns" an instance of the `Date` structure, it is necessary to ensure that the `Date` structure is declared before the `Student` structure (just as we need to declare variables before we use them.

23.2. Usage

23.2.1. Declaration & Initialization

Once we have defined a structure (and included the header file it has been defined in), we can create instances using the usual syntax.

```
Student s;
Student t;
```

These static declarations will allocate enough space on the stack to hold all of the data associated with the structures (the two `char *` pointers, `int`, and `double` and the three `int` variables in the `Date` structure). However, the values stored in each of the structure's member variables are undefined. With this style of declaration, C does not define default values.

Another way to declare instances of our structures is to use the following syntax.

```
Student s = {};
Student t = {
  12345678,
  "Grace",
  "Hopper",
  4.0,
  {
    1906,
    1,
    1
  }
};
```

The first declaration creates a `Student` structure with default values (zero for any numeric types, `null` for any pointers). The second creates a `Student` structure initialized with the values provided in the curly brackets. The order matters here and will match the ordering of the original structure declaration. Since the `dateOfBirth` is a structure itself, a nested set of values within curly brackets is necessary. One draw back to this type of declaration is that the character pointers are statically declared. The strings `"Grace"` and `"Hopper"` are initialized in a *read-only* segment of memory, any attempts to change the *contents* of these strings are undefined behavior. The pointers themselves, however, can be reassigned.

This static declaration allocates the structure on the stack. Since structures consist of multiple pieces of data, their memory footprint is larger. Allocating larger and larger structures on the stack runs the risk of running out of stack memory resulting in a stack overflow. To solve this, we can instead use dynamically allocated structures.

To dynamically allocate a structure, we use a pointer to a structure and a call to `malloc()` to allocate enough space for the structure. Even though our structure is user-defined, we can still use the `sizeof()` macro to determine the number of bytes required by the structure. The compiler and macro are "smart" enough to look at the structure declaration and determine the size of each of its variables and add up the total number of bytes required.

```
Student *s = (Student *) malloc(sizeof(Student) * 1);
```

The multiplication by 1 in this example is not strictly necessary, but emphasizes the fact that we are allocating space for one structure and not an array of structures. Initializing a dynamically allocated structure like this does *not* initialize any of its variables as there are no default values defined by C. Moreover, it does *not* initialize memory for any pointer member variables. For example the `firstName` and `lastName` pointers need to be manually initialized with additional `malloc()` calls.

23.2.2. Selection Operators

Once we have a declared structure, we need to access its member variables and perhaps update them. If we have a statically declared structure, such as

```
Student s;
```

we can use the *direct component selector operator*, which is simply just a period followed by the member variable we wish to access. Commonly, this is referred to simply as the *dot operator*.

```
1 Student s;
2
3 //set values:
4 s.id = 87654321;
5 s.gpa = 3.9;
6
7 //access values:
8 printf("Name: %s, %s\n", s.lastName, s.firstName);
9 printf("GPA: %.2f\n", s.gpa);
```

When structures are nested, we can use the dot operator multiple times to access members variables of member variables.

```
1 s.dateOfBirth.year = 2525;
2 s.dateOfBirth.month = 12;
3 s.dateOfBirth.date = 25;
```

When we have a pointer to a structure, we cannot directly use the dot operator. Instead,

we have to change the pointer into a "normal" structure by dereferencing it, then we can use the dot operator. However, the dot operator has a higher order of precedence than the dereferencing operator, thus parentheses are required:

```
Student *s = ...;
(*s).id = 87654321;
```

This can be a bit unwieldy, so C provides a convenience operator, the *indirect component selector operator*, or more commonly, the *arrow operator* that allows us to select a member variable with a single operator that resembles a right-pointing arrow.

```
Student *s = (Student *) malloc(sizeof(Student));

//set values:
s->id = 87654321;
s->gpa = 3.9;

//initialize the string variables:
s->firstName = (char *) malloc(sizeof(char) * 6);
strcpy(s->firstName, "Grace");
s->lastName = (char *) malloc(sizeof(char) * 7);
strcpy(s->lastName, "Hopper");

//access values:
printf("Name: %s, %s\n", s->lastName, s->firstName);
printf("GPA: %.2f\n", s->gpa);

s->dateOfBirth.year = 2525;
```

Note the last line: the `dateOfBirth` member variable was another structure, but not a pointer to a structure, so we use the dot operator to access the `year` member variable. Alternatively, we could have defined `dateOfBirth` to be a pointer, `Date *dateOfBirth;` in the `Student` structure. If we had, then to access the `year` member variable using two arrow operators, `s->dateOfBirth->year`.

23.3. Arrays of Structures

Just as we can create arrays of built-in types such as integers, we can also create arrays of our user-defined structures. As an example, the following creates an array of 10 `Student` structures. Once created, we can treat them like any other array.

```
Student *roster = (Student *) malloc(sizeof(Student) * 10);

...

double sum = 0.0;
for(i=0; i<10; i++) {
  sum += roster[i].gpa;
}
double averageGpa = sum / 10;
```

As in the example, we can index each element in the array, `roster`. Once indexed, each element is a regular structure and so we use the dot operator to access each of its member variables. As with any other array, each element takes up a number of bytes, equal to `sizeof(Student)`. We can swap and reassign each element just like any other variable. For example, the following code swaps the first two elements using a temporary variable.

```
Student temp = roster[0];
roster[0] = roster[1];
roster[1] = temp;
```

Each of these operations copies over every byte that makes up the structure. For small structures, this isn't that big of a deal. However, for larger structures, this may become an issue, especially if we do this often or pass structures around to functions.

As an alternative, we could instead deal indirectly with structures by creating an array of *pointers* to structures. Swapping elements then involves only copying pointer values rather than every byte that makes up the structure. To do this, we use the familiar pointer-to-pointer syntax.

```
Student **roster = (Student **) malloc(sizeof(Student *) * 10);
for(i=0; i<10; i++) {
```

```
  roster[i] = (Student *) malloc(sizeof(Student) * 1);
}

//access each as pointers and use the arrow operator
roster[0]->id = 87654321;
roster[0]->gpa = 4.0;

//swap the first two *pointers*:
Student *temp = roster[0];
roster[0] = roster[1];
roster[1] = temp;
```

As in the example above, if each element in the array is a pointer to a structure, then we use the arrow operator to access each member variable.

The differences between these two approaches is illustrated in Figures 23.1 and 23.2. As presented, each `Student` structure takes 40 bytes.[2] With the first approach (as in Figure 23.1, each structure instance is stored contiguously in memory. Swapping two records involves copying entire blocks of 40 bytes. In contrast, using pointers to structures as in Figure 23.2 means that structures may be stored non-contiguously in different memory locations. Swapping two structures is done indirectly by swapping pointers (only 8 bytes). The difference in this particular example is not that great (8 vs. 40 bytes), but with larger structures it can become an issue. Each approach has its own advantages and disadvantages and one may be more appropriate than the other in different situations.

Hybrid Approach

It is also possible to take a "hybrid" approach to storing structures in arrays that stores them both contiguously, but also allows you to reference them indirectly through a pointer array. This is essentially what we did in Section 20.4.1 when we created a contiguous 2-dimensional array of integers. We made a single call to `malloc` and then had to setup the pointers to each "row." To do this with structures, we can do something similar. First, we declare and allocate a dynamic array as before.

```
int n = 10;
Student *rosterData = (Student *) malloc(n * sizeof(Student));
```

[2]This is just an estimate and may vary on different systems and compilers. Usually, compilers use *alignment* and may pad structures with extra bytes in order to make it more efficient to store in memory.

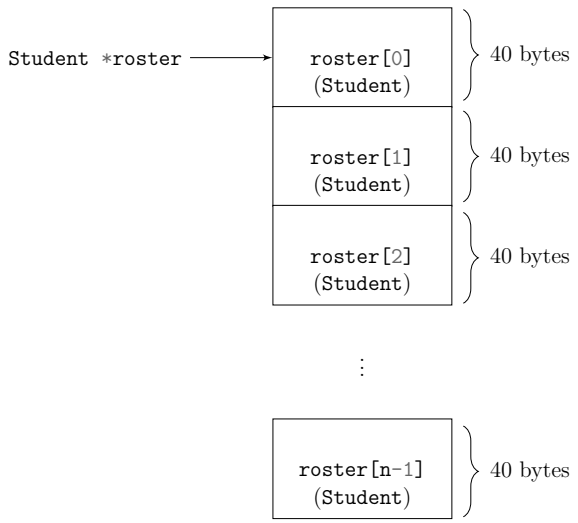


Figure 23.1.: An array of structures. Each record is stored in a contiguous manner one after the other.

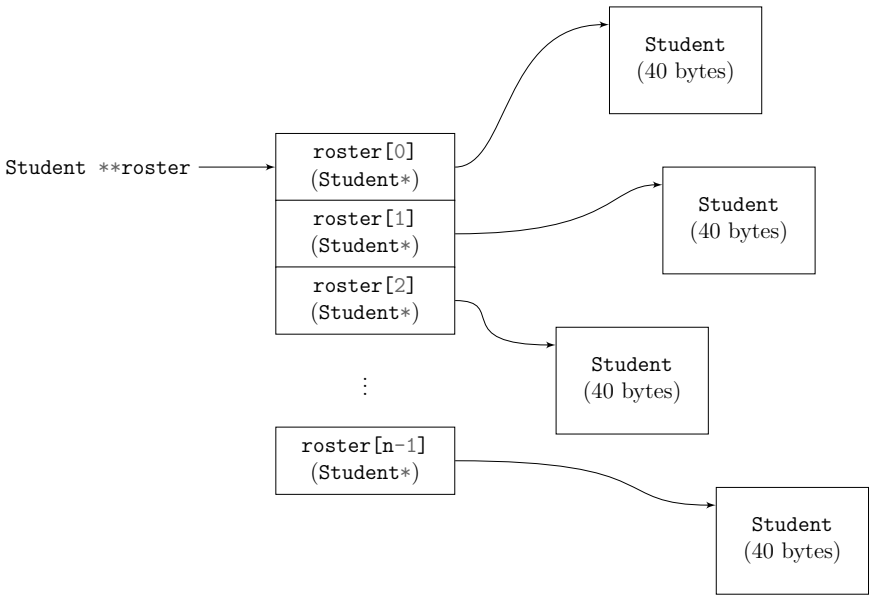


Figure 23.2.: An array of structure pointers. Each record is a pointer that refers to a structure which may be stored non-contiguously in completely different memory locations.

Then, we can declare an array of `Student` pointers as well.

```
Student **roster = (Student **) malloc(n * sizeof(Student *));
```

But now we need to make each `roster[i]` pointer point to the i-th record in `rosterData`. Each record, `rosterData[i]` is a regular structure, but we need a pointer to it. In order to get a pointer, we use the referencing operator, `&rosterData[i]` and make each pointer point to it.

```
for(i=0; i<n; i++) {
  roster[i] = &rosterData[i];
}
```

This approach is illustrated in Figure 23.3. Now we can indirectly reference each record in `rosterData` via a pointer in the `roster` array.

```
roster[0]->nuid = 12345678;
```

If we wanted to "swap" two records, we simply swap their pointers instead of their data. Indeed, care must be taken not to swap the data in `rosterData` otherwise the pointers would become invalid and out-of-synch with the data array.

23.4. Using Structures With Functions

As with built-in types, we can use structures as parameters to functions as well as the return type of functions. Consider the following examples.

```
void foo(Student s);
Student bar();
```

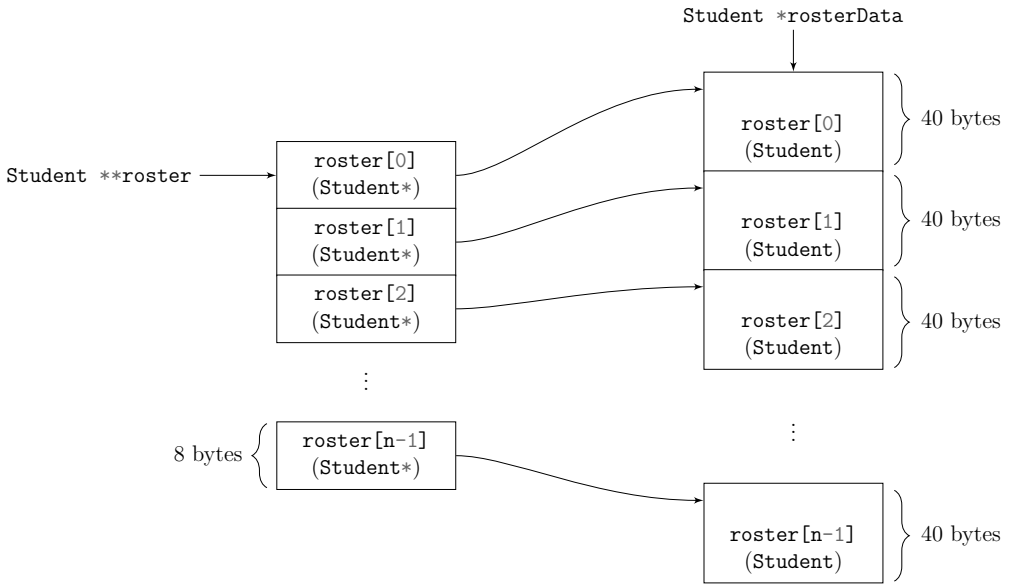


Figure 23.3.: Hybrid Array of Structures. The `rosterData` is an array of contiguous structures. The `roster` array is an array of pointers that refer to each record. Accessing elements in `rosterData` is done indirectly through a pointer in `roster`

In the first prototype, we would pass a `Student` structure *by value* to the function. As we've already seen, this would mean that every byte of the structure would be copied onto the stack. For larger structures, we risk a stack overflow while for even smaller structures this can be very inefficient. Likewise, the second prototype would return a structure. When assigned to a variable, every byte is copied. A better solution is to use dynamically allocated structures, passing them by reference to functions, and returning pointers to dynamically allocated instances as return types. For example, the functions above would be better implemented as follows.

```
1 void foo(const Student *s);
2 Student * bar();
```

In the first example, we've used the `const` keyword which prevents any changes to the structure's values (we may omit this if we need to design a function that changes its values). We will now consider several idiomatic examples of using structure with functions.

23.4.1. Factory Functions

Properly creating and initializing structure instances can be a complex and tedious task. However, it is likely that we will need to repeat this operation over and over. We can simplify our task if we write a utility function that creates a structure instance for us. We provide the function the values we want initialized to. Such functions are sometimes referred to as *factory* functions as they can be used to manufacture as many instances as we want (in object-oriented programming languages, such methods are called constructors).

We will need to take care that we make *deep* copies of any dynamically allocated elements such as strings. Shallow copies where references are shared may lead to unexpected behavior as changes to one string may affect multiple structures.

```
1  /**
2   * This function creates a new student structure with the
3   * given values.
4   */
5  Student * createStudent(const char *firstName,
6                          const char *lastName,
7                          int id,
8                          double gpa) {
9
10   Student *s = NULL;
11   s = (Student *) malloc(sizeof(Student) * 1);
12
13   //make a deep copy of the strings
14   s->firstName = (char *)malloc(sizeof(char)*(strlen(firstName)+1));
15   strcpy(s->firstName, firstName);
16   s->lastName = (char *)malloc(sizeof(char)*(strlen(lastName)+1));
17   strcpy(s->lastName, lastName);
18
19   s->id = id;
20   s->gpa = gpa;
21
22   return s;
23 }
```

Another common operation is to create a copy of a given structure. We will want to again ensure that everything is a *deep* copy so that the two structures are not sharing references. We can easily do this by reusing the functionality we just wrote in the factory

function.

```
/**
 * This function creates a new, deep copy of a Student
 * structure .
 */
Student * copyStudent(const Student *s) {
  return createStudent(s->firstName, s->lastName, s->id, s->gpa);
}
```

23.4.2. To String Functions

Another common operation with structures is to output their data as a human-readable string representation. We could write a function that output a structure's member variables to the standard output using `printf()`. However, it would more useful if we created a general function that returned a string representation of the structure. That way, we could decide what to do with the string: output it to the standard output, to a file, etc. Not all structure instances will require the same length string. Some students for example may have long names, while others have short. We can handle this by first calculating the length of the string necessary for whatever formatting we've chosen.

```
/**
 * Returns a string representation of the given
 * Student structure.
 */
char * studentToString(const Student *s) {

  int n = strlen(s->firstName) +
          strlen(s->lastName) +
          8 + //id, assumed to always be at most 8 digits
          4 + //gpa, assumed to be 0.0 - 4.0
          19; //other formatting characters

  char *str = (char *) malloc(sizeof(char) * n);

  sprintf(str, "%s, %s, ID = %d (GPA = %.2f)",
                s->lastName, s->firstName, s->id, s->gpa);

  return str;
```

```
19 }
```

Here, we've utilized a variation on the familiar `printf()` function, `sprintf()` which "prints" the result not to the standard output or a file, but to a string, specified as the first argument. This function would end up returning a string similar to the following for our previous example: `"Hopper, Grace, ID = 87654321 (GPA = 3.90)"`

23.4.3. Passing Arrays of Structures

We often also have need to pass arrays of structures to functions. As an example, consider passing an entire roster of students to a function in order to compute the average GPA. If we have an array of structures, we would have a function that looked something like the following.

```
1  /**
2   * Computes the average GPA of the Student structures in
3   * the given roster (which is of size n).
4   */
5  double computeAverageGpa(const Student *roster, int n) {
6    double sum = 0.0;
7    int i;
8    for(i=0; i<n; i++) {
9      sum += roster[i].gpa;
10   }
11   return sum / n;
12 }
```

When we pass in the array of structures, it is not passed by value. That is, the total number of bytes for each student is *not* copied onto the call stack. Nevertheless, as we've previously seen it is sometimes preferable to maintain an array of pointers to structures. If we had an array of pointers, we would have a function that looks something like the following.

```
1  /**
2   * Computes the average GPA of the Student structures in
3   * the given roster (which is of size n).
```

```
 */
double computeAverageGpa(const Student **roster, int n) {
  double sum = 0.0;
  int i;
  for(i=0; i<n; i++) {
    sum += roster[i]->gpa;
  }
  return sum / n;
}
```

The only difference here is in how we access the `gpa` member variable using the arrow operator instead of the dot operator.

24. Recursion

C supports recursion with no special syntax necessary. However, as a structured, procedural language, recursion is generally expensive and iterative or other non-recursive solutions are generally preferred. We present a few examples to demonstrate how to write recursive functions in C.

The first example of a recursive function we gave was the toy count down example. In C it could be implemented as follows.

```
1 void countDown(int n) {
2   if(n==0) {
3     printf("Happy New Year!\n");
4   } else {
5     printf("%d\n", n);
6     countDown(n-1);
7   }
8 }
```

As another example that actually does something useful, consider the following recursive summation function that takes an array, its size and an index variable. The recursion works as follows: if the index variable has reached the size of the array, it stops and returns zero (the base case). Otherwise, it makes a recursive call to `recSum()`, incrementing the index variable by 1. When the function returns, it adds its result to the i-th element in the array. To invoke this function we would call it with an initial value of 0 for the index variable: `recSum(arr, n, 0)`.

```
1 int recSum(const int *arr, int size, int i) {
2   if(i == size) {
3     return 0;
4   } else {
5     return recSum(arr, size, i+1) + arr[i];
6   }
7 }
```

This example was not tail-recursive as the recursive call was not the final operation (the sum was the final operation). To make this function tail recursive, we can carry the summation through to each function call ensuring that the summation is done prior to the recursive function call.

```
int recSumTail(const int *arr, int size, int i, int sum) {
  if(i == size) {
    return sum;
  } else {
    return recSumTail(arr, size, i+1, sum + arr[i]);
  }
}
```

As a final example, consider the following C implementation of the naive recursive Fibonacci sequence. An additional condition has been included to check for "invalid" negative values of n for which zero is returned.

```
int fibonacci(int n) {
  if(n < 0) {
    return 0;
  } else if(n <= 1) {
    return 1;
  } else {
    return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

C is not a language that provides implicit memoization. Instead, we need to explicitly keep track of values using a table. In the following example, the table is passed to the function as an argument.

```
int fibonacciMemoization(int n, int *table) {
  if(n < 0) {
    return 0;
  } else if(n <= 1) {
    return 1;
```

```
  } else if(table[n] > 0) {
    return table[n];
  } else {
    int a = fibonacciMemoization(n-1, table);
    int b = fibonacciMemoization(n-2, table);
    int result = (a + b);
    table[n] = result;
    return result;
  }
}
```

It is the responsibility of the calling function to ensure that the `table` array is large enough to accommodate all values. In this case should be at least of size $(n + 1)$ to compute the n-th Fibonacci number.

25. Searching & Sorting

The standard C library provides several functions to search and sort arrays of *any* type of element including `int`, `double`, or even user-defined structures such as our `Student` example from Chapter 23. These functions are able to operate on any type of array because they take generic `void *` pointers. However, these functions still need a way to *order* the generic elements. To understand how this all works we need to understand both *comparator* functions and *function pointers*.

25.1. Comparator Functions

Let's consider a "generic" Quick Sort algorithm as was presented in Algorithm 12.6. The algorithm itself specifies how to sort elements, but it doesn't specify how they are *ordered*. The difference is subtle but important. Essentially, Quick Sort needs to know when two elements, a, b are in order, out of order, or equivalent in order to decide which partition each element goes in. However, it doesn't "know" anything about the elements a and b themselves. They could be numbers, they could be strings, they could be user-defined objects.

A sorting algorithm still needs to be able to determine the proper ordering in order to sort. In C this is achieved through a *comparator function*, which is a function that is responsible for comparing two elements and determining their proper order. A comparator function has the following signature and behavior:

```
int cmp(const void *a, const void *b);
```

- The function takes two generic `void *` pointers which refer to the elements being compared.
- Moreover, the `const` keyword is used to indicate that no changes will be made to the elements.
- The function returns an integer indicating the relative ordering of the two elements:
  - It returns something negative, `< 0` if `a` comes before `b` (that is, $a < b$)
  - It returns zero if `a` and `b` are equal ($a = b$)

– It returns something positive, `> 0` if `a` comes after `b` (that is, $a > b$)

Note that there is no guarantee on the value's magnitude, it does *not* necessarily return -1 or $+1$; it just returns *something* negative or positive. We've previously seen this pattern when comparing strings. The standard string library provides a function, `strcmp()` that has the same basic contract: it takes two strings and returns something negative, zero or something positive depending on the lexicographic ordering of the two strings. Strictly speaking, however, `strcmp()` is *not* a comparator function. It is defined to take two `const char *` parameters, not `const void *` pointers.

The C language (and thus the compiler) "knows" how to compare built-in primitive types like `int` and `double` using the built-in comparison operators. However, to generalize the comparison operation, we use `void *` pointers so that we can write and use general comparator functions that can be used in generic searching and sorting functions. As an example, let's write a comparator function that orders integers in ascending order. We'll call our function `cmpInt()` and use the signature above.

```
1 int cmpInt(const void *a, const void *b) {
2
3 }
```

But how might we implement this function? Obviously we want to compare the values stored in the pointer variables, so we need to dereference them. However, the comparison, `(*a < *b)` for example is not comparing integer values. The variables `a` and `b` are `void` pointers, not `int` pointers. In general, you cannot dereference a `void` pointer. Dereferencing an `int *` or `double *` is possible because the compiler knows how many bytes each type takes. However, the number of bytes a `void` type takes is undefined. Thus, the first step is to *make* them `int` pointers by doing an explicit type cast:

```
1 const int *x = (const int *) a;
2 const int *y = (const int *) b;
```

Now the variables `x` and `y` *are* `int` pointers which can be dereferenced and compared (we preserve the keyword `const` to ensure we do not make changes to the variables).

```
1 int cmpInt(const void *a, const void *b) {
2   const int *x = (const int *) a;
3   const int *y = (const int *) b;
```

```
4     if(*x < *y) {
5       return -1;
6     } else if(*x == *y) {
7       return 0;
8     } else {
9       return 1;
10    }
11  }
```

What if we wanted to order integers in the opposite order? We could write another comparator in which the comparisons or values are reversed. Even simpler, we could reuse the comparator above and "flip" the sign by multiplying by -1 (the purpose of writing functions is code reuse, after all). Even simpler, we could flip the arguments we pass to `cmpInt()` to reverse the order.

```
1  int cmpIntDesc(const void *a, const void *b) {
2    return cmpInt(b, a);
3  }
```

The examples above illustrate the standard implementation pattern of comparator functions:

1. Make the general `const void *` pointers into a `const` pointer to the specific type you want to compare by making an explicit type cast.
2. Use the state of the type (one or more components if you are comparing structures) to determine their order.
3. Return an integer that expresses the desired order of elements.

The cautious observer may be asking: what happens if I call a comparator and pass it pointers to mismatched elements? For example, what if I pass two `double` pointers to `cmpInt()`? Obviously, the code will not work as intended (it will end up comparing the upper 32 bits of the `double` values, which leads to unintended behavior). A comparator function is designed with certain expectations. Namely, that a user will always pass it valid pointers to `int` values. If a user violates these expectations, it is their fault, not ours. This is no different from many other functions in the standard library. Passing non-null terminated strings to most string functions for example is undefined behavior.

To illustrate some more examples, consider the `Student` structure we defined in Code

Sample 23.1. The following code samples demonstrate various ways of ordering `Student` structures based on one or more of their components.

```c
/**
 * A comparator function to order Student structures by
 * last name/first name in alphabetic order
 */
int studentByNameCmp(const void *s1, const void *s2) {
  const Student *a = (const Student *)s1;
  const Student *b = (const Student *)s2;
  int result = strcmp(a->lastName, b->lastName);
  if(result == 0) {
    return strcmp(a->firstName, b->firstName);
  } else {
    return result;
  }
}
```

```c
/**
 * A comparator function to order Student structures by
 * last name/first name in reverse alphabetic order
 */
int studentByNameCmpDesc(const void *s1, const void *s2) {
  return studentByNameCmp(s2, s1);
}
```

```c
/**
 * A comparator function to order Student structures by
 * id in ascending numerical order
 */
int studentIdCmp(const void *s1, const void *s2) {
  const Student *a = (const Student *)s1;
  const Student *b = (const Student *)s2;
  if(a->nuid < b->nuid) {
    return -1;
  } else if(a->nuid == b->nuid) {
    return 0;
```

```
  } else {
    return 1;
  }
}
```

```
/**
 * A comparator function to order Student structures by
 * GPA in descending order
 */
int studentGpaCmp(const void *s1, const void *s2) {
  const Student *a = (const Student *)s1;
  const Student *b = (const Student *)s2;
  if(a->gpa > b->gpa) {
    return -1;
  } else if(a->gpa == b->gpa) {
    return 0;
  } else {
    return 1;
  }
}
```

25.2. Function Pointers

Now that we have comparator functions to order elements, we need a way of passing a comparator to a generic search or sort function so that it can be *used* by that function. To achieve this in C, we use *function pointers*.

Recall that a pointer is simply a reference to some memory location. As we've already seen, pointers can point to variables of simple data types such as `int` or `double`, arrays of these types or even user-defined structures. We've also seen that pointers can be generic using the `void` keyword. The `malloc()` function for example, returned a generic void pointer, `void *`, to allow it to be used to allocate memory for *any* type.

Generic void pointers simply point to the start of a memory block, not necessarily to a memory location of any particular type of variable. As we've also already seen, the program code itself lives in memory (the stack). It thus makes sense that we could

reference a memory location that, instead of containing variables, contains executable code, in particular a *function*. This is what a function pointer does: it points to a memory location where the code for the function is stored.

To declare a function pointer, we need to specify more information than a typical `int *` or `double *` pointer. Since it points to a function, we need to specify the function's signature: its return type and parameter list. Consider the following example:

```
int (*ptrToFunc)(int, double, char) = NULL;
```

In this declaration, we've created a function pointer named `ptrToFunc`. This pointer is capable of pointing to any function whose return type is `int` (indicated by the first keyword) and which takes *three* parameters: an `int`, a `double` and a `char`. The assignment operation has initialized this pointer to `NULL`.

Consider another example: let's declare a function pointer that is capable of pointing to the math library's `sqrt()` function. The `sqrt()` function returns a `double` and takes a single `double` parameter. Adapting the syntax above, we would create this pointer as follows.

```
double (*ptrToSqrt)(double) = NULL;
```

Again, we have initialized it to `NULL`; it does not yet reference the `sqrt()` function. To make it reference this function, we need to assign it a value. Recall that to get the memory location of a regular variable, say `int x;`, we use the referencing operator, `&x`. Similarly, with functions we *can* use the referencing function, in this case `&sqrt` to get the memory address of the `sqrt()` function, however this is not necessary. Similar to arrays, the identifier (name) of a function serves as its memory address! The identifier `sqrt` itself is the memory location of the function. Thus to assign `ptrToSqrt` to point to `sqrt()`, we simply need to do the following:

```
ptrToSqrt = sqrt;
```

Note the difference: usually we *invoke* `sqrt()` by writing parentheses and providing an argument. When referencing the function itself, we omit the parentheses. Additional examples of this syntax can be found in Code Sample 25.1.

Callbacks

The main use for function pointers is so that references to functions can be passed as parameters to other functions. The passed function is known as a callback. This gives us the ability to write a more abstract and generic function. For example, in GUI programming, we frequently need to associate a particular function with a particular

event. Suppose we create a button; we need to be able to specify what happens when that button gets clicked. We do so by providing a function as a callback to a *registration* function that associates the "click" event with the provided function. Thus, whenever a user clicks the button, the callback is invoked ("called back").

Let's illustrate some syntax usage with another example. Suppose we want to create a `getMax()` function. We *could* write one function for arrays of integers, another for arrays of `double`s, another for `Student` arrays that gets the student with the maximum GPA, then another for the ID, then another for the name and so on. Or, we could program a *generic* `getMax()` function that could be used for *any* type by taking a comparator function as a callback. To illustrate, consider the following non-generic function for integers.

```
1  int getMax(const int *arr, int n) {
2    int i, maxIndex = 0;
3    for(i=1; i<n; i++) {
4      if(arr[maxIndex} < arr[i] {
5        //we've found something larger, update the maxIndex:
6        maxIndex = i;
7      }
8    }
9    return maxIndex;
10 }
```

This simple function iterates through the array, keeping track of the maximum value found "so far" and updating it when it finds something larger. Because we've specified that `arr` is an array if `int` values, we can use the less-than comparison operator (line 4). Now let's make it more generic: rather than taking an array of `int` values, it will now take a generic `void` array. Further, we will pass a function pointer to this function that references a generic comparator function that can be used to replace the less-than comparison operator.

```
1  int getMax(const void *arr, int n,
2             int(*cmp)(const void *, const void *)) {
3    int i, maxIndex = 0;
4    for(i=1; i<n; i++) {
5      ...
6    }
7    return maxIndex;
8  }
```

There are a couple of issues here that we have to deal with. When working with generic `void *` pointers in C and using arrays, you cannot simply index using the usual 0, 1, 2, etc. indices. Recall that when elements are stored in an array, the index represents an *offset* of a memory address. If the array is an array of integers or `double` or some other built-in type, the compiler knows how large each one is and is able to compute the appropriate offset given the usual 0, 1, 2, etc. indices.

However, when dealing with `void *` elements, a function must be told how many bytes each element takes. C uses an unsigned integer type, `size_t` to indicate a size in bytes, so we'll use it as well. We modify the function signature above to pass in a `size_t size` parameter.

We now need a way to access each element in the array. The array itself is generic. We cannot simply use indices such as `arr[i]` to access elements. Since it is a `void *` pointer, we cannot simply dereference it using an index. The compiler doesn't know how many bytes each element takes, so it cannot compute an offset using the index variable `i`. Instead, we need to do the pointer arithmetic ourselves.

We could use the size parameter, say `arr[i*size]` to compute the offset of the i-th element, but this still dereferences a void pointer, which we generally do not want to do. Instead, we can use the pointer `arr` and do some simple arithmetic; `arr` is the starting memory location, so if we simply *add* `i*size` to it, that gives us the memory location of the i-th element *as a void pointer*!

```
1 void *first  = arr + 0 * size; //first element
2 void *second = arr + 1 * size; //second element
3 void *third  = arr + 2 * size; //third element
4 ...
5 void *x      = arr + i * size; //i-th element
```

We can use this in our `getMax()` function to iterate over each element in the array and pass it to our comparator. The comparator expects generic void pointers, and that is what our pointer arithmetic is computing. Passing two memory addresses to the comparator determines which is the larger of two elements in the array. To do this, we call the comparator on the maximum element we've found so far and the i-th element in the loop. If it returns something negative, then we know that the "max" element is less than the i-th element and so update our `maxIndex` variable. Making these changes results in the this final version.

```
int getMax(const void *arr, int n, size_t size,
           int(*cmp)(const void *, const void *)) {
  int i, maxIndex = 0;
  for(i=1; i<n; i++) {
    if(cmp(arr + maxIndex * size, arr + i * size) < 0) {
      //we've found something larger, update the max_index:
      maxIndex = i;
    }
  }
  return maxIndex;
}
```

Suppose we have an array of integers and the `cmpInt()` comparator function above. We can use our `getMax()` function as follows.

```
int arr[] = {8, 2, 9, 10, 4, 2, 2, 5, 6, 7};
int maxIndex = getMax(arr, 10, sizeof(int), cmpInt);
printf("maximum value: %d\n", arr[maxIndex]);
```

In this example, the `getMax()` function would return the index 3 and print the maximum value stored there, 10. The `getMax()` function returns the index corresponding to the "maximum" element *according to the comparator* used. If instead we had used `cmpIntDesc()` the "maximum" element would have been the least element, (2 in the example above) because it would have been the element ordered "last" by the *descending* comparator.

Consider another example with our `Student` structure.

```
int n = 10;
Student *roster = (Student *) malloc(sizeof(Student) * n);
...
int maxIndex = getMax(roster, n, sizeof(Student), studentByNameCmp);
```

Since `studentByNameCmp()` orders by lexicographic ordering, a student with the last name "Zadora" would have been "larger" than someone with a last name "Anderson."

Thus the code above will return an index corresponding to the *last* student in lexicographic ordering of their name. Similarly, if we had used the `studentGpaCmp()` comparator instead, `getMax()` would have returned an index for the student with the *lowest* GPA as this comparator ordered highest to lowest.

Another variation on this function would be to return a *pointer* to the maximum element rather than an index. The same logic would have applied, but the return type would be a `void *` pointer and the return statement would return the memory address of the maximum element instead. To avoid warnings, in the final return statement we cast `arr` as a void pointer (instead of a `const` void pointer) to make the return value compatible with the return type.

```
void * getMax(const void *arr, int n, size_t size,
              int(*cmp)(const void *, const void *)) {
  int i, maxIndex = 0;
  for(i=1; i<n; i++) {
    if(cmp(arr + maxIndex * size, arr + i * size) < 0) {
      //we've found something larger, update the max_index:
      maxIndex = i;
    }
  }
  return (void *)arr + maxIndex * size;
}
```

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

int function01(int a, double b);
void function02(double x, char y);

void runAFunction(int (*theFunc)(int, double));

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

  int i = 5;
  double d = 3.14;
  char c = 'Q';

  //calling a function normally...
  int j = function01(i, d);
  function02(d, c);

  //function pointer declaration
  int (*pt2Func01)(int, double) = NULL;
  void (*pt2Func02)(double, char) = NULL;

  //assignment
  pt2Func01 = function01;
  //or:
  pt2Func01 = &function01;
  pt2Func02 = &function02;

  //you can invoke a function using a pointer to it:
  j = pt2Func01(i, d);
  pt2Func02(d, c);

  //alternatively, you can invoke a function by dereferencing it:
  j = (*pt2Func01)(i, d);
  (*pt2Func02)(d, c);

  //With function pointers, you can now pass entire functions as arguments to another function!
  printf("Calling runAFunction...\n");
  runAFunction(pt2Func01);
  //we should not pass in the second pointer as it would not match the signature:
  //syntactically okay, compiler warning, undefined behavior
  //runAFunction(pt2Func02);
}

void runAFunction(int (*theFunc)(int, double)) {

  printf("calling within runAfunction...\n");
  int result = theFunc(20, .5);
  printf("the result was %d\n", result);

  return;
}

int function01(int a, double b) {
  printf("You called function01 on a = %d, b = %f\n", a, b);
  return a + 10;
}

void function02(double x, char y) {

  printf("You called function02 on x = %f, y = %c\n", x, y);

}
```

Code Sample 25.1.: C Function Pointer Syntax Examples

25.3. Searching & Sorting

We now turn our attention to the search and sorting functions provided by the standard library. Each function is a generic implementation that takes advantage of function pointers and comparator functions.

25.3.1. Searching

Linear Search

The C search library, `search.h` provides a linear search function to search arrays, named `lfind()` (linear find). This function does not require that the array be sorted and performs a linear search algorithm, returning a *pointer* to the first element such that the comparator returns 0 (indicating equality). The full signature of the function is as follows.

```
void *lfind(const void *key,
            const void *base,
            size_t *nmemb,
            size_t size,
            int(*compar)(const void *, const void *));
```

Each argument represents:

- `key` – a "dummy" structure instance matching the element you are searching for. For example, if you are searching for a `Student` with the last name `"Smith"` then you can construct an instance with the same last name, ignoring the value of all other components.
- `base` – a pointer to the array to be searched
- `nmemb` – the size of the array (number of "members")
- `size` – the size, in bytes, of each element in the array (generally you use use `sizeof()` to determine this)
- `compar` – a pointer to a comparator function to use in the search.

As with our `getMax()` example, `lfind()` returns a pointer to the element it finds. If no such element is found, this function will return `NULL`.

In the same library, there is another linear search function:

```
void *lsearch(const void *key,
              void *base,
              size_t *nmemb,
              size_t size,
              int(*compar)(const void *, const void *));
```

It differs in that if it does not find a matching element, it still returns `NULL` but also attempts to *insert* the element being searched for at the end of the array. The function will assume there is enough room at the end of the array to accommodate the inserted element (if not, the behavior is undefined). This behavior is hinted at by the fact that `base` is not `const`. Moreover, `nmemb` is passed by reference. If the function inserts the element, it will also increment `nmemb` variable to reflect this new element. It is your responsibility to ensure that there is enough valid memory to accommodate any inserts when using `lsearch()`.

Binary Search

In the standard library (`stdlib.h`) there is an additional binary search function that can be used to more efficiently search a *sorted* array. The prototype:

```
void *bsearch(const void *key,
              const void *base,
              size_t nmemb,
              size_t size,
              int (*compar)(const void *, const void *));
```

All parameters are exactly as with `lfind()` as is the behavior: it returns a pointer to the first element that it finds (though first does not necessarily mean the first in the order of the array) and `NULL` if no matching element is found. It is an essential requirement that the array be sorted *with the same comparator* as was used to sort the array or `NULL` may be returned erroneously.

25.3.2. Sorting

The standard library also provides a generic sorting function, `qsort()`. Though the name suggests a Quick Sort implementation, it does not necessarily have to be (it was when the function was originally designed). Modern implementations of `qsort()` may implement alternatives such as Merge Sort or non-recursive hybrid Quick Sort algorithms.

The prototype and parameters are similar to the search functions but do not include a key. The array is also not `const` indicating that it *will* be changed (which is the whole point of calling the function). The function will sort elements in ascending order according to the provided comparator function.

```
1 void qsort(void *base,
2             size_t nmemb,
3             size_t size,
4             int(*compar)(const void *, const void *))
```

- `base` – pointer an array of elements
- `nmemb` – the size of the array (number of members)
- `size` – the size (in bytes) of each element (use `sizeof()`)
- `compar` – a comparator function used to order elements

The advantages to using `qsort()` (as well as `lfind()` and `bsearch()`) should be clear. There is no need to write a new function that reimplements the same algorithm for every possible ordering of every possible user-defined structure. We need only to create a comparator function and pass it to `qsort()`. There is less code, and less chance of bugs. The `qsort()` function is well-designed, optimized, and most importantly well-tested and proven.

These functions represent a sort of "weak" form of polymorphic behavior found in more modern object-oriented programming languages and other languages that support "generic programming." Polymorphism is the characteristic that the same code can be executed on different types, greatly reducing the need for duplicate code.

25.3.3. Examples

We illustrate the usage of these functions in Code Samples 25.2 and 25.3.

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

#include "student.h"

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

  int n = 0;
  Student *roster = loadStudents("student.data", &n);
  int i;
  size_t numElems = n;


  printf("Roster: \n");
  printStudents(roster, n);

  /* Searching */
  Student *castro = NULL;
  Student *castroKey = NULL;
  Student *sandberg = NULL;
  char *str = NULL;

  castro = linearSearchStudentByNuid(roster, 10, 131313);
  str = studentToString(castro);
  printf("castro: %s\n", str);
  free(str);

  //create a key that will match according to the NUID
  int nuid = 23232323;
  Student * key = createEmptyStudent();
  key->nuid = nuid;

  //use lfind to find the first such instance:
  //sandberg =
  sandberg = lfind(key, roster, &numElems, sizeof(Student), studentIdCmp);
  str = studentToString(sandberg);
  printf("sandberg: %s\n", str);
  free(str);

  //create a key with only the necessary fields
  castroKey = createStudent("Starlin", "Castro", 0, 0.0);
  //sort according to a comparator function
  qsort(roster, n, sizeof(Student), studentLastNameCmp);

  castro = bsearch(castroKey, roster, n, sizeof(Student), studentLastNameCmp);
  str = studentToString(castro);
  printf("castro (via binary search): %s\n", str);
  free(str);

  //create a key with only the necessary fields
  castroKey = createStudent(NULL, NULL, 131313, 0.0);
  //sort according to a comparator function
  qsort(roster, n, sizeof(Student), studentIdCmp);

  castro = bsearch(castroKey, roster, n, sizeof(Student), studentIdCmp);
  str = studentToString(castro);
  printf("castro (via binary search): %s\n", str);
  free(str);

  return 0;
}
```

.

Code Sample 25.2.: C Search Examples

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

#include "student.h"


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

  int n = 0;
  Student *roster = loadStudents("student.data", &n);
  int i;
  size_t numElems = n;

  printf("Roster: \n");
  printStudents(roster, n);

  printf("\n\n\nSorted by last name/first name: \n");
  qsort(roster, numElems, sizeof(Student), studentLastNameCmp);
  printStudents(roster, n);

  printf("\n\n\nSorted by ID: \n");
  qsort(roster, numElems, sizeof(Student), studentIdCmp);
  printStudents(roster, n);

  printf("\n\n\nSorted by ID, descending: \n");
  qsort(roster, numElems, sizeof(Student), studentIdCmpDesc);
  printStudents(roster, n);

  printf("\n\n\nSorted by GPA: \n");
  qsort(roster, numElems, sizeof(Student), studentGPACmp);
  printStudents(roster, n);

  return 0;
}
```

.

Code Sample 25.3.: C Sort Examples

25.4. Other Considerations

25.4.1. Sorting Pointers to Elements

Recall that it is sometimes preferable to maintain an array of pointers to structures rather than an array of structures. Sorting is a scenario where this is particularly true. When sorting an array of structure elements, the entire structure is copied back and forth as elements are swapped. Depending on the number of bytes of a structure, this can be quite expensive. It is generally more efficient to sort an array of pointers to structures instead. A prime example of this is sorting an array of strings.

An array of strings can be thought of as a 2-dimensional array of `char`s. Specifically, an array of strings is a `char **` type. That is, an array of pointers to pointers of `char`s. We may be tempted to use `strcmp()` in the standard string library, passing it to `qsort()`. Unfortunately this will not work. `qsort()` requires two `const void *` types, while `strcmp()` takes two `const char *` types. This difference is subtle but important.[1] The recommended way of doing this is to define a different comparator function as in Code Sample 25.4.

```
1 /* compare strings via pointers */
2 int pstrcmp(const void *p1, const void *p2)
3 {
4   return strcmp(*(char * const *)p1, *(char * const *)p2);
5 }
```

Code Sample 25.4.: C Comparator Function for Strings

Observe the behavior of this function: it uses the standard `strcmp()` function, but makes the proper explicit type casting before doing so. The `*(char * const *)` casts the generic void pointers as pointers to strings (or pointers to pointers to characters), then dereferences it to be compatible with `strcmp()`.

Another case is when we wish to sort user-defined structures. The `Student` structure presented earlier is "small" in that it only has a few fields. When structures are stored in an array and sorted, there may be many *swaps* of individual elements which involves a lot of memory copying. If the structures are small this is not too bad, but for "larger" structures this could be potentially expensive. Instead, it may be preferred to have an array of *pointers* to structures. Swapping elements involves only swapping pointers

[1] A full discussion can be found on the c-faq, http://c-faq.com/lib/qsort1.html.

instead of the entire structure. This is far cheaper as a memory address is likely to be far smaller than the actual structure it points to. This is essentially equivalent to the string scenario: we have an array of pointers to be sorted, our comparator function then needs to deal with pointers to pointers.[2] An example appears in Code Sample 25.5.

```
/**
 * Orders two Student pointers according to the last name/first name
 */
int studentPtrLastNameCmp(const void *s1, const void *s2) {
  //we receive a pointer to an individual element in the array
  //but individual elements are POINTERS to students thus we cast
  //them as (const Student **) then dereference to get a pointer
  //to a Student!
  const Student *a = *(const Student **)s1;
  const Student *b = *(const Student **)s2;
  int result = strcmp(a->lastName, b->lastName);
  if(result == 0) {
    return strcmp(a->firstName, b->firstName);
  } else {
    return result;
  }
}



Student **roster = (Student **) malloc(sizeof(Student *) * n);
...
qsort(roster, n, sizeof(Student *), studentPtrLastNameCmp);
```

Code Sample 25.5.: Sorting Structures via Pointers

Another issue when sorting arrays of pointers is that we may now have to deal with `NULL` elements. When sorting arrays of elements this is not an issue as a properly initialized array will contain non-null elements (though elements could still be uninitialized, the memory space will be valid).

How we handle `NULL` pointers is more of a design decision. We could ignore it and any attempt to access a `NULL` structure will result in undefined behavior (or segmentation faults, etc.). Or we could give `NULL` values an explicit ordering with respect to other

[2] Again, a full discussion can be found on c-faq, http://c-faq.com/lib/qsort2.html.

elements. That is, we could order all `NULL` pointers *before* non-`NULL` elements (and consider all `NULL` pointers to be equal). An example with respect to our `Student` structure is given in Code Snippet 25.6.

```
int studentPtrLastNameCmpWithNulls(const void *s1, const void *s2) {
  const Student *a = *(const Student **)s1;
  const Student *b = *(const Student **)s2;
  if(a == NULL && b == NULL) {
    return 0;
  } else if(a == NULL && b != NULL) {
    return -1;
  } else if (a != NULL && b == NULL) {
    return 1;
  }
  int result = strcmp(a->lastName, b->lastName);
  if(result == 0) {
    return strcmp(a->firstName, b->firstName);
  } else {
    return result;
  }
}
```

Code Sample 25.6.: Handling Null Values

Part II.

The Java Programming Language

26. Basics

The Java programming language was developed in the early 1990s at Sun Microsystems by James Gosling, Mike Sheridan, and Patrick Naughton. Its original intention was to enable cable box sets to be more interactive. By the mid-90s, Java was retargeted toward the WWW. The first public release came on May 23, 1995 with the first Java Development Kit (JDK), Java 1.0 on January 23rd, 1996. A new, updated release has come about every other year. As of 2014, Java 8 is the current stable version.

Today, Java is one of the most popular programming languages, consistently ranked as one of the top 2 languages (see `http://www.tiobe.com`). It is now owned and maintained by Oracle, but there are many open source tools, compilers and runtime environments available. Java is used in everything from mobile devices (Android) and desktop applications to enterprise application servers.

From its inception, Java was designed with 5 basic principles:

1. Simple, Object-oriented, familiar
2. Robust and secure
3. Architecture-neutral and portable
4. High performance
5. Interpreted, threaded and dynamic

Java offers many key features that have made it popular. It is unique in that it is not entirely compiled nor interpreted. Instead, Java source code is compiled into an intermediate form, called Java bytecode. This bytecode is not directly runnable on a processor. Instead, a JVM, an application that was written and compiled for a particular system, interprets the bytecode and runs the application. This added layer of abstraction means that Java source code can be written once (and compiled once) and then run anywhere on any device that has a JVM. The added layer of abstraction makes development easier, but comes at a cost in performance. However, the most recent JVMs have offered performance that is comparable to native machine code in many applications.

Another key feature is that Java has its own automated garbage collection. Some languages require manual memory management, meaning that requesting, managing,

```java
package unl.cse;  //package declaration

//imports would go here

/**
 * A basic hello  world program in Java
 */
public class HelloWorld {

  //static main method
  public static void main(String args[]) {
    System.out.println("Hello World!");
  }

}
```

Code Sample 26.1.: Hello World Program in Java

and freeing memory is part of the code that you write as a developer. Failure to handle memory management properly can lead to wasted resources (memory leaks), poor or unstable performance, and even more serious security issues (buffer overflows). In Java, there is no manual memory management. The JVM handles the allocation and clean up of memory automatically.

In following with the five design principles, Java is similar in syntax to C (called "C-style syntax"). Executable statements are terminated by semicolons, code blocks are defined by opening/closing curly brackets, etc. Java is also fundamentally a class-based OOP language. With the exception of a few primitive types, in Java everything is a class or belongs to an class.

26.1. Getting Started: Hello World

The hallmark of an introduction to a new programming language is the *Hello World!* program. It consists of a simple program whose only purpose is to print out the message "Hello World!" to the user. The simplicity of the program allows the focus to be on the basic syntax of the language. It is also typically used to ensure that your development environment, compiler, runtime environment, etc. are functioning properly with a minimal example. A basic Hello World! program in Java can be found in Code Sample 26.1.

We will not focus on any particular development environment, code editor, or any particular operating system, compiler, or ancillary standards in our presentation. However, as a first step, you should be able to write, compile, and run the above program on the environment you intend to use for the rest of this book. This may require that you download and install a JDK and IDE. Eclipse (`http://eclipse.org/`) is the industry standard, though IntelliJ (`https://www.jetbrains.com/idea/`) and NetBeans (`https://netbeans.org/`) are also popular.

26.2. Basic Elements

Using the Hello World! program as a starting point, we will now examine the basic elements of the Java language.

26.2.1. Basic Syntax Rules

Java's syntax is adopted from C, referred to as "C-style syntax." These elements include the following.

- Java is a statically typed language so variables must be declared along with their types before using them.
- Strings are delimited with double quotes. Single characters, including special escaped characters are delimited by single quotes; `"this is a string"`, and these are characters: `'A'`, `'4'`, `'$'` and `'\n'`
- In addition, Java uses Unicode (UTF-16 encoding) to represent characters. This is fully back-compatible with ASCII, but also allows you to specify Unicode characters using special escape sequences and hexadecimal encodings. For example, `'\u4FFA'` represents a Japanese character:

  俺

  The string `"\u4FFA\u306F\u6700\u9AD8\u3060\u305C\uFF01"` represents the phrase

  俺は最高だぜ！

- Executable statements are terminated by a semicolon, `;`
- Code blocks are defined using opening and closing curly brackets, `{ ... }`. Moreover, code blocks can be *nested*: code blocks can be defined within other code

blocks.

- Variables are scoped to the code block in which they are declared and are only valid within that code block.
- In general, whitespace between coding elements is ignored.

Though not a syntactic requirement, the proper use of whitespace is important for good, readable code. Code inside code blocks is indented at the same indentation. Nested code blocks are indented further. Think of a typical table of contents or the outline of a formal paper or essay. Sections and subsections or points and subpoints all follow proper indentation with elements at the same level at the same indentation. This convention is used to organize code and make it more readable.

26.2.2. Program Structure

Classes

In Java, everything is a *class* or belongs to a class. A class is an extensible program or blueprint for creating objects. Objects are an integral part of OOP that we'll explore later. However, to start out our programs will be simple enough that they can be contained in a single class. To declare a class, you use the following syntax.

```
public class HelloWorld { ... }
```

where the contents of the class are placed between the opening/closing curly brackets. In addition, a class must be placed in a Java source file with the same name. In our previous example, the class must be in a file named `HelloWorld.java`.

When naming classes, the most commonly accepted naming convention is to use uppercase camel casing (also called PascalCase) in which each word in the name is capitalized including the first, `Employee`, `SavingsAccount`, `ImageFile`, etc. Class names (as well as their source file names) are case sensitive.

Packages

Java code is organized into modules called *packages*. Packages are essentially directories (or folders) which follow a directory tree structure which allows subdirectories and separate directories at the same level. It all starts at the *root* directory called the "default" package.

Within a source file, we declare which package the file belongs to using the keyword `package` followed by a *fully qualified package declaration* which is essentially just the

names of the directories that the file is located in, separated by a period. The declaration is terminated by a semicolon. For example, the package declaration,

```
package unl.cse;
```

would indicate that the file belongs in the directory `cse` which is a subdirectory of the directory `unl`. The absence of a package declaration will mean that the file is associated with the default directory.

Packages allow you to organize source files and code functionality. Mathematics related classes can be placed in one package while image related classes can be placed in another, etc. It also provides separate name spaces for classes. There could be 3 or 4 different classes named `List` for example. They could not be located in the same directory; if you wanted to use one which one would you be referring to? By separating them into packages, they can all exist without conflict. When we want to use a particular one, we *import* that class with its fully qualified package name.

Imports

An `import` statement essentially "brings in" another class so that its methods and functionality can be used. For example, there is a class named `Scanner` (located in the package `java.util`) that makes it easy to read input from the standard input. To include it in our program so that we can use its functionality, we would need[1] to import it:

```
import java.util.Scanner;
```

Classes in the package `java.lang` (such as `String` and `Math`) are considered standard and are imported by default without an explicit `import` statement.

You may see some code that uses a *wildcard* like `import java.util.*;` which ends up importing *every* class in that package. This is generally considered bad practice. In general, code should be intentional and specific, importing every class even if they are not used goes against this principle.

When naming packages, you must follow the general naming rules for identifiers (see below). Package names cannot begin with a number, no whitespace, etc. Moreover, the general convention for package names is to use lowercase underscore casing, `here_is_an_example`. Moreover, packages and subpackages follow the same convention as directories: the top most directory is the most general and subdirectories are more and more specific.

In many of our examples we'll use `unl.cse` (UNL, University of Nebraska–Lincoln; CSE,

[1] You can still use it without importing it, but you'd need to use a fully qualified path name at declaration/instantiation.

Function	Description
`Math.abs(x)`	Absolute value function, $\lvert x \rvert$[a]
`Math.ceil(x)`	Ceiling function, $\lceil 46.3 \rceil = 47.0$
`Math.floor(x)`	Floor function, $\lfloor 46.3 \rfloor = 46.0$
`Math.cos(x)`	Cosine function[b]
`Math.sin(x)`	Sine function[b]
`Math.tan(x)`	Tangent function[b]
`Math.exp(x)`	Exponential function, e^x, $e = 2.71828\ldots$
`Math.log(x)`	Natural logarithm, $\ln(x)$[c]
`Math.log10(x)`	Logarithm base 10, $\log_{10}(x)$[c]
`Math.pow(x,y)`	The power function, computes x^y
`Math.sqrt(x)`	Square root function[c]

Table 26.1.: Several methods defined in the Java `Math` library. [a]There are several versions of the absolute value method, one for each numeric type but all with the same name. [b]all trigonometric functions assume input is in *radians*, **not** degrees. [c]Input is assumed to be positive, $x > 0$.

Department of Computer Science & Engineering) which illustrates this general-to-specific organization.

There are many other important classes and packages provided by the standard JDK that we'll examine as needed. Of immediate interest is the `Math` class and its library of common mathematical functions such as the square root and the natural logarithm. Table 26.1 highlights several of these functions. To use them you'd need to invoke them by using the class name and dot operator. For example:

```
1 double x = 1.5;
2 double y, z;
3 y = Math.sqrt(x); //y now has the value √x = √1.5
4 z = Math.sin(x); //z now has the value sin(x) = sin(1.5)
```

In both of the method calls above, the value of the variable `x` is "passed" to the math function which computes and "returns" the result which then gets assigned to another variable.

26.2.3. The `main()` Method

Every executable program has to have a beginning: a point at which the program starts to execute. In Java, a class may contain many variables and methods, but a class is only *executable* if it contains a `main()` method. When a Java class is compiled and the JVM is started, the JVM loads the class into memory and starts executing code contained in the `main()` method.

In addition, our `main()` method takes an *array* of `String` types which serve to communicate any command line arguments provided to the program (review Section 2.4.4 for details). The array, `args` stores the arguments as strings. The number of arguments provided can be determined using the `length` property of the array. Specifically, `args.length` is an integer indicating how many arguments were provided. This does *not* include the name of the program (class) itself as that is already be known to the programmer.

To access any one argument, it will be necessary to *index* the array. The index for the first argument is zero, thus the first argument is `args[0]`, the second is `args[1]`, etc. The last one would be at `args[args.length-1]`.

If a user is expected to provide numbers as input, they'll need to be converted as the `args` array are only `String` types. To convert the arguments you can use parsing methods provided by the `Integer` and `Double` classes. An example:

```
1 //converts the "first" command line argument to an integer
2 int x = Integer.parseInt(args[0]);
3 //converts the "third" command line argument to a double:
4 double y = Double.parseDouble(args[1]);
```

26.2.4. Comments

Comments can be written in a Java program either as a single line using two forward slashes, `//comment` or as a multiline comment using a combination of forward slash and asterisk: `/* comment */`. With a single line comment, everything on the line *after* the forward slashes is ignored. With a multiline comment, everything in between the forward slash/asterisk is ignored. Comments are ultimately ignored so the amount of comments do not have an effect on the final executable code. Consider the following example.

```
1 //this is a single line comment
2 int x;  //this is also a single line comment, but after some code
3
4 /*
5   This is a comment that can
6   span multiple lines to format the comment
7   message more clearly
8 */
9 double y;
```

Most code editors and IDEs will present comments in a special color or font to distinguish them from the rest of the code (just as our example above does). Failure to close a multiline comment will likely result in a compiler error but with color-coded comments its easy to see the mistake visually.

Another common comment style convention is the Javadoc (Java Documentation) style of comments. Javadoc style comments are multiline comments that begin with `/**`. The Javadoc framework allows you to *markup* your comments with tags and links so that documentation can be automatically generated and published. We will sometimes use this style, but we will not cover the details.

26.3. Variables

Java has 8 built-in primitive types supporting numbers (integers and floating point numbers), Booleans, and characters. Table 26.2 contains a complete description of these types. Each of these primitive types also has a corresponding *wrapper* class defined in the `java.lang` package. Wrapper classes provide *object* versions of each of these classes. The object versions have many utility methods that can be used in relation to their type. For example, the aforementioned `Integer.parseInt()` method is part of the `Integer` wrapper class.

The wrapper classes, however, are different. These are objects, so when a reference is declared for them, by default, that reference refers to `null`. The keyword `null` is used to indicate a special memory address that represents "nothing." In fact, the default value for *any* object type is `null`. Care must be taken when mixing primitive types and their wrapper classes (see below) as `null` references may result in a `NullPointerException`. Finally, instances of the wrapper classes are immutable. Once they are created, they cannot be changed. References can be made to refer to a different object, but the object's value cannot be changed.

Type	Description	Wrapper Class
`byte`	8-bit signed 2s complement integer	`Byte`
`short`	16-bit signed 2s complement integer	`Short`
`int`	32-bit signed 2s complement integer	`Integer`
`long`	64-bit signed 2s complement integer	`Long`
`float`	32-bit IEEE 754 floating point number	`Float`
`double`	64-bit floating point number	`Double`
`boolean`	may be set to `true` or `false`	`Boolean`
`char`	16-bit Unicode (UTF-16) character	`Character`

Table 26.2.: Primitive types in Java

26.3.1. Declaration & Assignment

Java is a statically typed language meaning that all variables must be declared before you can use them or refer to them. In addition, when declaring a variable, you must specify both its type and its identifier. For example:

```
1 int numUnits;
2 double costPerUnit;
3 char firstInitial;
4 boolean isStudent;
```

Each declaration specifies the variable's type followed by the identifier and ending with a semicolon. The identifier rules are fairly standard: a name can consist of lowercase and uppercase alphabetic characters, numbers, and underscores but may *not* begin with a numeric character. We adopt the modern camelCasing naming convention for variables in our code. In general, variables *must* be assigned a value before you can use them in an expression. You do not have to immediately assign a value when you declare them (though it is good practice), but some value must be assigned before they can be used or the compiler will issue an error.[2]

The assignment operator is a single equal sign, `=` and is a right-to-left assignment. That is, the variable that we wish to assign the value to appears on the left-hand-side while the value (literal, variable or expression) is on the right-hand-size. Using our variables from before, we can assign them values:

[2]Instance variables, that is variables declared as part of an object *do* have default values. For objects, the default is `null`, for all numeric types, zero is the default value. For the `boolean` type, `false` is the default, and the default `char` value is `\0`, the null-terminating character (zero in the ASCII table).

```
numUnits = 42;
costPerUnit = 32.79;
firstInitial = 'C';
isStudent = true;
```

For brevity, Java allows you to declare a variable and immediately assign it a value on the same line. So these two code blocks could have been more compactly written as:

```
int numUnits = 42;
double costPerUnit = 32.79;
char firstInitial = 'C';
boolean isStudent = true;
```

As another shorthand, we can declare multiple variables on the same line by delimiting them with a comma. However, they *must* be of the same type. We can also use an assignment with them.

```
int numOrders, numUnits = 42, numCustomers = 10, numItems;
double costPerUnit = 32.79, salesTaxRate;
```

Another useful keyword is `final`. Though it has several uses, when applied to a variable declaration, it makes it a read-only variable. After a value has been assigned to a `final` variable, its value cannot be changed.

```
final int secret = 42;
final double salesTaxRate = 0.075;
```

Any attempt to reassign the values of `final` variables will result in a compiler error.

26.4. Operators

Java supports the standard arithmetic operators for addition, subtraction, multiplication, and division using `+`, `-`, `*`, and `/` respectively. Each of these operators is a binary operator that acts on two operands which can either be literals or other variables and follow the usual rules of arithmetic when it comes to order of precedence (multiplication and division before addition and subtraction).

```
int a = 10, b = 20, c = 30, d;
d = a + 5;
d = a + b;
d = a - b;
d = a + b * c;
d = a * b;
d = a / b; //integer division and truncation!  See below

double x = 1.5, y = 3.4, z = 10.5, w;
w = x + 5.0;
w = x + y;
w = x - y;
w = x + y * z;
w = x * y;
w = x / y;

//you can do arithmetic with both types:
w = a + x;

//however you CANNOT assign a double to an integer:
d = b + y; //compilation error
//but you can do so with an explicit type cast:
d = (int) (b + y);  //though truncation occurs, d is 23
```

In addition, you can *mix* the wrapper classes with their primitive types. You must be careful though. The wrapper classes are object references which can be `null`. If a `null` reference is used in an arithmetic expression, it will result in a `NullPointerException` which can be caught and handled (see Chapter 30). If not caught, it will end up being a fatal error. Some examples:

```
int a = 10, c;
Integer b = 20;
```

```
3
4  //int and Integer can be mixed:
5  c = a + b;
6
7  double x = 3.14, z;
8  Double y = 2.71;
9  //double and Double can be mixed:
10 z = x + y;
11
12 //all types can be mixed:
13 double u = a + x + b + y;
14
15 //Be careful:
16 Integer d = null;
17 c = a + d; //NullPointerException
```

This works because of a mechanism called *autoboxing* (or *autounboxing* in this case). The wrapper class is acting like a "box": it is an object that stores the value of a primitive type. When it gets used in an arithmetic expression, it gets "unboxed" and converted to a primitive type so that the arithmetic operation is performed on compatible primitive types. This is all done by the compiler and is completely transparent to us. However, that is the reason that we may get a `NullPointerExcpetion`. Our code actually gets converted from `c = a + d;` to `c = a +d.doubleValue();`. The `doubleValue()` method returns a `double` primitive value. However, if `d` is `null`, you can't call a method on it; thus the `NullPointerException` is thrown as a consequence.

Special care must be taken when dealing with `int` types. For all four operators, if both operands are integers, the result will be an integer. For addition, subtraction, and multiplication this isn't an issue, but for division it means that when we divide, say `(10 / 20)`, the result is not 0.5 as expected. The number 0.5 is a floating point number. As such, the fractional part gets truncated (cut off and thrown out) leaving only zero. In the code above, `d = a / b;` the variable `d` ends up getting the value zero because of this.

A solution to this problem is to use explicit type casting to force at least one of the operands in an integer division to become a `double` type. For example:

```
1 int a = 10, b = 20;
2 double x;
3
4 x = (double) a / b;
```

Assigning a floating point number to an integer is not allowed in Java and attempting to do so will be treated as a compiler error. This is because Java does not support *implicit* type casts. However, you *can* do so if you provide an *explicit* type cast as in the code above,

```
d = (int) (b + y);
```

In this code, `b + y` is correctly computed as $20 + 3.4 = 23.4$, but the explicit type cast (down to an integer) results in truncation. The .4 gets cutoff and `d` gets the value 23. Assigning an `int` value to a `double` variable is not a problem as the integer 2 becomes the floating point number 2.0.

Java also supports the integer remainder operator using the `%` symbol. This operator gives the remainder of the result of dividing two integers. Examples:

```
1 int x;
2
3 x = 10 % 5; //x is 0
4 x = 10 % 3; //x is 1
5 x = 29 % 5; //x is 4
```

26.5. Basic I/O

Java provides several ways to perform input and output operations with the standard input/output as part of the `System` class. The `System` class contains a standard output stream which can be accessed using `System.out`. It also has a standard input stream which can be accessed using `System.in`.

For output, there are several methods that can be used but we'll focus on `System.out.println()` and `System.out.printf()`. The first is for printing strings on a single line. The `ln` at the end indicates that the method will insert an endline character, `\n` for you. The second is a `printf`-style output method that takes placeholders like `%d` and `%f` (review Section 2.4.3 for details).

The easiest way to read input is through the `Scanner` class. You can create a new instance of the `Scanner` class and associate it with the standard input using the following

```java
Scanner s = new Scanner(System.in);
int a;
System.out.println("Please enter a number: ");
a = s.nextInt();
System.out.printf("Great, you entered %d\n", a);
```

Code Sample 26.2.: Basic Input/Output in Java

code.

```java
Scanner s = new Scanner(System.in);
```

The variable `s` is now active and can be read from. You can get specific values from the `Scanner` by calling various methods such as `s.nextInt()` to get an `int`, `s.nextDouble()` to get a `double`, etc. When these methods are called, the program *blocks* until the user enters her input and presses the enter/return key. The conversion to the type you requested is automatic. A full example is depicted in Code Sample 26.2.

One potential problem with using `Scanner` is that the methods cannot force a user to enter good input. In the example above, if the user, instead of entering a number, entered `"Hello"`, the conversion to a number would fail and result in a `InputMismatchException`.

26.6. Examples

26.6.1. Converting Units

Let's write a program that will prompt the user to enter a temperature in degrees Fahrenheit and convert it to degrees Celsius using the formula

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

We begin with the basic program outline which will include a package and class declaration. We'll also need to read from the standard input, so we'll import the `Scanner` class. We'll want want our class to be executable, so we need to put a `main()` method in our class. Finally, we'll document our program to indicate its purpose.

```java
package unl.cse;

import java.util.Scanner;

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */
public class TemperatureConverter {

  public static void main(String args[]) {

    //TODO: implement this

  }
}
```

It is common for programmers to use a comment along with a `TODO` note to themselves as a reminder of things that they still need to do with the program.

Let's first outline the basic steps that our program will go through:

1. We'll first prompt the user for input, asking them for a temperature in Fahrenheit
2. Next we'll read the user's input, likely into a floating point number as degrees can be fractional
3. Once we have the input, we can calculate the degrees Celsius by using the formula above
4. Lastly, we will want to print the result to the user to inform them of the value

Sometimes it is helpful to write an outline of such a program directly in the code using comments to provide a step-by-step process. For example:

```java
package unl.cse;

import java.util.Scanner;

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
```

```
 */
public class TemperatureConverter {

  public static void main(String args[]) {

    //TODO: implement this
    //1. Prompt the user for input in Fahrenheit
    //2. Read the Fahrenheit value from the standard input
    //3. Compute the degrees Celsius
    //4. Print the result to the user

  }
}
```

As we read each step it becomes apparent that we'll need a couple of variables: one to hold the Fahrenheit (input) value and one for the Celsius (output) value. It also makes sense that each of these should be `double` variables as we want to support fractional values. At the top of our `main()` method, we'll add the variable declarations:

```
double fahrenheit, celsius;
```

We'll also need a `Scanner`, initialized to read from the standard input:

```
Scanner s = new Scanner(System.in);
```

Each of the steps is now straightforward; we'll use a `System.out.println()` statement in the first step to prompt the user for input:

```
System.out.println("Please enter degrees in Fahrenheit: ");
```

In the second step, we'll use our `Scanner` to read in a value from the user for the `fahrenheit` variable. Recall that we use the method `s.nextDouble()` to read a `double` value from the user.

```
fahrenheit = s.nextDouble();
```

We can now compute `celsius` using the formula provided:

```
celsius = (fahrenheit - 32) * (5 / 9);
```

Finally, we use `System.out.printf()` to output the result to the user:

```
System.out.printf("%f Fahrenheit is %f Celsius\n", fahrenheit, celsius);
```

Try typing and running the program as defined above and you'll find that you don't get correct answers. In fact, you'll find that no matter what values you enter, you get zero. This is because of the calculation using `(5 / 9)`: recall what happens with integer division: truncation! This will *always* end up being zero.

One way we could fix it would be to pull out our calculators and find that $\frac{5}{9} = 0.55555\ldots$ and replace `(5 / 9)` with `0.555555`. But, how many fives? It may be difficult to tell how accurate we can make this floating point number by hardcoding it ourselves. A much better approach would be to let the compiler take care of the optimal computation for us by making at least one of the numbers a `double` to prevent integer truncation. That is, we should instead use `5.0 / 9`. The full program can be found in Code Sample 26.3.

```
1  package unl.cse;
2
3  import java.util.Scanner;
4
5  public class TemperatureConverter {
6
7    public static void main(String args[]) {
8
9      double fahrenheit, celsius;
10     Scanner s = new Scanner(System.in);
11
12     //1. Prompt the user for input in Fahrenheit
13     System.out.println("Please enter degrees in Fahrenheit: ");
14
15     //2. Read the Fahrenheit value from the standard input
16     fahrenheit = s.nextDouble();
17
18     //3. Compute the degrees Celsius
19     celsius = (fahrenheit - 32) * 5.0 / 9;
20
21     //4. Print the result to the user
22     System.out.printf("%f Fahrenheit is %f Celsius\n",
23             fahrenheit, celsius);
24
25   }
26 }
```

Code Sample 26.3.: Fahrenheit-to-Celsius Conversion Program in Java

26.6.2. Computing Quadratic Roots

Some programs require the user to enter multiple inputs. The prompt-input process can be repeated. In this example, consider asking the user for the coefficients, a, b, c to a quadratic polynomial,

$$ax^2 + bx + c$$

and computing its roots using the quadratic formula,

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

As before, we can create a basic program with a `main()` method and start filling in the details. In particular, we'll need to prompt for the input a, then read it in; then prompt for b, read it in and repeat for c. We'll also need several variables: three for the coefficients a, b, c and *two* more; one for each root.

```
1 double a, b, c, root1, root2;
2 Scanner s = new Scanner(System.in);
3
4 System.out.println("Please enter a: ");
5 a = s.nextDouble();
6 System.out.println("Please enter b: ");
7 b = s.nextDouble();
8 System.out.println("Please enter c: ");
9 c = s.nextDouble();
```

Now to compute the roots: we need to adapt the formula so it accurately reflects the order of operations. We also need to use the standard math library's square root method.

```
1 root1 = (-b + Math.sqrt(b*b - 4*a*c) ) / (2*a);
2 root2 = (-b - Math.sqrt(b*b - 4*a*c) ) / (2*a);
```

Finally, we print the output using `System.out.printf()`. The full program can be found in Code Sample 26.4.

This program was interactive. As an alternative, we could have read all three of the inputs as command line arguments, taking care that we need to convert them to floating point numbers. Lines 16–21 in the program could have been changed to

```
a = Double.parseDouble(args[0]);
b = Double.parseDouble(args[1]);
c = Double.parseDouble(args[2]);
```

Finally, think about the possible inputs a user could provide that may cause problems for this program. For example:

- What if the user entered zero for a?
- What if the user entered some combination such that $b^2 < 4ac$?
- What if the user entered non-numeric values?
- For the command line argument version, what if the user provided less than three arguments? Or more?

How might we prevent the consequences of such bad input? How might we handle the event that a user enters bad input and how do we communicate these errors to the user? To start to resolve these issues, we'll need conditionals.

```java
package unl.cse;

import java.util.Scanner;

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */
public class QuadraticRoots {

  public static void main(String args[]) {

    double a, b, c, root1, root2;
    Scanner s = new Scanner(System.in);

    System.out.println("Please enter a: ");
    a = s.nextDouble();
    System.out.println("Please enter b: ");
    b = s.nextDouble();
    System.out.println("Please enter c: ");
    c = s.nextDouble();

    root1 = (-b + Math.sqrt(b*b - 4*a*c) ) / (2*a);
    root2 = (-b - Math.sqrt(b*b - 4*a*c) ) / (2*a);

    System.out.printf("The roots of %fx^2 + %fx + %f are: \n",
            a, b, c);
    System.out.printf("  root1 = %f\n", root1);
    System.out.printf("  root2 = %f\n", root2);

  }

}
```

Code Sample 26.4.: Quadratic Roots Program in Java

27. Conditionals

Java supports the basic if, if-else, and if-else-if conditional structures as well as switch statements. Java has Boolean types and logical statements are built using the standard logical operators for numeric comparisons as well as logical operators such as negations, And, and Or that can be used with Boolean types.

27.1. Logical Operators

Recall that Java has Boolean types built-in to the language using either the primitive type, `boolean` or its wrapper class `Boolean`. Moreover, the keywords `true` and `false` can be used to assign and check values. Because Java has Boolean types, it does not allow you to mix logical operators with numeric types. That is, code like the following is invalid.

```
1 int a = 10;
2 boolean b = true;
3 boolean result = (a || b); //compilation error
```

The standard numeric comparison operators are also supported. Consider the following code snippet:

```
1 int a = 10;
2 int b = 20;
3 int c = 10;
4 boolean x = true;
5 boolean y = false;
```

The six standard comparison operators are presented in Table 27.1 using these variables as examples. The comparison operators are the same when used with `double` types as

Name	Operator Syntax	Examples	Value
Equals	`==`	`a == 10`	*true*
		`b == 10`	*false*
		`a == b`	*false*
		`a == c`	*true*
Not Equals	`!=`	`a != 10`	*false*
		`b != 10`	*true*
		`a != b`	*true*
		`a != c`	*false*
Strictly Less Than	`<`	`a < 15`	*true*
		`a < 5`	*false*
		`a < b`	*true*
		`a < c`	*false*
Less Than Or Equal To	`<=`	`a <= 15`	*true*
		`a <= 5`	*false*
		`a <= b`	*true*
		`a <= c`	*true*
Strictly Greater Than	`>`	`a > 15`	*false*
		`a > 5`	*true*
		`a > b`	*false*
		`a > c`	*false*
Greater Than Or Equal To	`>=`	`a >= 15`	*false*
		`a >= 5`	*true*
		`a >= b`	*false*
		`a >= c`	*true*

Table 27.1.: Comparison Operators in Java

well and `int` types and can be compared with each other without type casting.

Furthermore, because of autoboxing and unboxing, the wrapper classes for numeric types can be compared using the same operators. For example:

```
1 int a = 10;
2 Integer b = 20;
3 Double x = 3.14;
4 boolean r;
5 r = (a < b);
6 r = (a >= b);
7 r = (x == 2.71);
```

Operator	Operator Syntax	Examples	Values
Negation	`!`	`!x`	*false*
		`!y`	*true*
And	`&&`	`x && true`	*true*
		`x && y`	*false*
Or	`||`	`x || false`	*true*
		`x || y`	*true*
		`!x || y`	*false*

Table 27.2.: Logical Operators in Java with `x = true` and `y = false` both being `Boolean` variables.

	Operator(s)	Associativity	Notes
Highest	`++`, `--`	left-to-right	postfix increment operators
	`-`, `!`	right-to-left	unary negation operator, logical not
	`*`, `/`, `%`	left-to-right	
	`+`, `-`	left-to-right	addition, subtraction
	`<`, `<=`, `>`, `>=`	left-to-right	comparison
	`==`, `!=`	left-to-right	equality, inequality
	`&&`	left-to-right	logical And
	`||`	left-to-right	logical Or
Lowest	`=`, `+=`, `-=`, `*=`, `/=`	right-to-left	assignment and compound assignment operators

Table 27.3.: Operator Order of Precedence in Java. Operators on the same level have equivalent order and are performed in the associative order specified.

The three basic logical operators are also supported as described in Table 27.2.

27.1.1. Order of Precedence

At this point it is worth summarizing the order of precedence of all the operators that we've seen so far including assignment, arithmetic, comparison, and logical. Since all of these operators could be used in one statement, for example,

```
(b*b < 4*a*c || a == 0 || args.length != 4)
```

It is important to understand the order in which each one gets evaluated. Table 27.3 summarizes the order of precedence for the operators seen so far. This is not an exhaustive list of Java operators.

27.1.2. Comparing Strings and Characters

The comparison operators in Table 27.1 can also be used for *single characters* because of the nature of the ASCII text table (see Table 2.4). Each alphanumeric character, including the various symbols and whitespace characters, is associated with an integer 0–127. We can therefore write statements like `('A' < 'a')`, which is *true* since uppercase letters are ordered before lowercase letters in the ASCII table (`'A'` is 65 and `'a'` is 97 and so $65 < 97$ is *true*). Several more examples can be found in Table 27.4.

Comparison Example	Result
`('A' < 'a')`	*true*
`('A' == 'a')`	*false*
`('A' < 'Z')`	*true*
`('0' < '9')`	*true*
`('\n' < 'A')`	*true*
`(' ' < '\n')`	*false*

Table 27.4.: Character comparisons in Java

Numeric comparison operators *cannot* be used to compare strings in Java. For example, we could *not* code something like `("aardvark" < "zebra")`. The Java compiler would not allow you to do this because the comparison operator is for numeric types *only*. However, the following code *would* compile and run:

```
1 String s = "aardvark";
2 String t = "zebra";
3 boolean b = (s == t);
```

but it wouldn't necessarily give you what you want. To understand why this is okay, recall that a `String` is an object; the `s` and `t` variables are *references* to that object in memory. When we use the equality comparison, `==` we're asking if `s` and `t` are the *same memory address*. In this case, likely they are not and so the result is `false`. However, similar code,

```
1 String s = new String("liger");
2 String t = new String("liger");
3 boolean b = (s == t);
```

```java
//example of an if statement:
if(x < 10) {
  System.out.println("x is less than 10");
}

//example of an if-else statement:
if(x < 10) {
  System.out.println("x is less than 10");
} else {
  System.out.println("x is 10 or more");
}

//example of an if-else-if statement:
if(x < 10) {
  System.out.println("x is less than 10");
} else if(x == 10) {
  System.out.println("x is equal to ten");
} else {
  System.out.println("x is greater than 10");
}
```

Code Sample 27.1.: Examples of Conditional Statements in Java

would also result in `false` because `s` and `t` represent different strings in memory, even though they have the same sequence of characters. We'll explore how to properly compare strings later. For now, avoid using the equality operators with strings.

27.2. If, If-Else, If-Else-If Statements

Conditional statements in Java utilize the keywords `if`, `else`, and `else if`. Conditions are placed inside parentheses immediately after the `if` and `else if` keywords. Examples of all three can be found in Code Sample 27.1.

The statement, `if(x < 10)` does not have a semicolon at the end. This is because it is a conditional statement that determines the flow of control and *not* an executable statement. Therefore, no semicolon is used. Suppose we made a mistake and *did* include a semicolon:

```
int x = 15;
if(x < 10); {
  System.out.println("x is less than 10");
}
```

Some compilers may give a warning, but this is valid Java; it will compile and it will run. However, it will end up printing `x is less than 10`, even though $x = 15$! Recall that a conditional statement *binds* to the executable statement or code block *immediately* following it. In this case, we've provided an *empty* executable statement ended by the semicolon. The code is essentially equivalent to

```
int x = 15;
if(x < 10) {
}
System.out.println("x is less than 10");
```

Which is obviously not what we wanted. The semicolon ended up binding to the empty executable statement, and the code block containing the print statement immediately followed, but was *not* bound to the conditional statement which is why the print statement executed regardless of the value of x.

Another convention that we've used in our code is where we have placed the curly brackets. First, if a conditional statement is bound to only one statement, the curly brackets are not necessary. However, it is best practice to include them even if they are not necessary and we'll follow this convention. Second, the opening curly bracket is on the same line as the conditional statement while the closing curly bracket is indented to the same level as the start of the conditional statement. Moreover, the code inside the code block is indented. If there were more statements in the block, they would have all been at the same indentation level.

27.3. Examples

27.3.1. Computing a Logarithm

The logarithm of x is the exponent that some *base* must be raised to get x. The most common logarithm is the natural logarithm, $\ln(x)$ which is base $e = 2.71828\ldots$. But

logarithms can be in any base $b > 1$.[1] What if we wanted to compute $\log_2(x)$? Or $\log_\pi(x)$? Let's write a program that will prompt the user for a number x and a base b and computes $\log_b(x)$.

Arbitrary bases can be computed using the change of base formula:

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$$

If we can compute *some* base a, then we can compute any base b. Fortunately we have such a solution. Recall that the standard library provides a function to compute the natural logarithm, `Math.log()`. This is one of the fundamentals of problems solving: if a solution already exists, use it. In this case, a solution exists for a different, but similar problem (computing the natural logarithm), but we can *adapt* the solution using the change of base formula. In particular, if we have variables `b` (base) and `x`, we can compute $\log_b(x)$ using

```
Math.log(x) / Math.log(b)
```

However, we have a problem similar to the examples in the previous section. The user could enter invalid values such as $b = -10$ or $x = -2.54$ (logarithms are undefined for non-positive values in any base). We want to ensure that $b > 1$ and $x > 0$. With conditionals, we can now do this. Once we have read in the input from the user we can make a check for good input using an `if` statement.

```
1 if(x <= 0 || b <= 1) {
2   System.out.println("Error: bad input!");
3   System.exit(1);
4 }
```

This code has something new: `System.exit(1)`. The `exit()` function immediately terminates the program regardless of the rest of the code that it may remain. The argument passed to `exit()` is an integer that represents an *error code*. The convention is that zero indicates "no error" while non-zero values indicate some error. This is a simple way of performing *error handling*: if the user provides bad input, we inform them and quit the program, forcing them to run it again and provide good input. By prematurely terminating the program we avoid any illegal operation that would give a bad result.

Alternatively, we could have split the conditions into two statements and given a more descriptive error message. We use this design in the full program which can be found in

[1] Bases can also be $0 < b < 1$, but we'll restrict our attention to increasing functions only.

Code Sample 27.2. The program also takes the input as command line arguments. Now that we have conditionals, we can actually check that the correct number of arguments was provided by the user and quit in the event that they don't provide the correct number.

27.3.2. Life & Taxes

Let's adapt the conditional statements we developed in Section 3.6.4 into a full Java program. The first thing we need to do is establish the variables we'll need and read them in from the user. At the same time we can check for bad input (negative values) for both the inputs.

```
Scanner s = new Scanner(System.in);
double income, baseTax, numChildren, credit, totalTax;

System.out.println("Please enter your Adjusted Gross Income: ");
income = s.nextDouble();

System.out.println("How many children do you have?");
numChildren = s.nextDouble();

if(income < 0 || numChildren < 0) {
  System.out.println("Invalid inputs");
  System.exit(1);
}
```

Next, we can code a series of if-else-if statements for the income range. By placing the ranges in increasing order, we only need to check the upper bounds just as in the original example.

```
if(income <= 18150) {
  baseTax = income * .10;
} else if(income <= 73800) {
  baseTax = 1815 + (income - 18150) * .15;
} else if(income <= 148850) {
...
} else {
  baseTax = 127962.50 + (income - 457600) * .396;
}
```

Next we compute the child tax credit, taking care that it does not exceed $3,000. A conditional based on the number of children should suffice as at this point in the program we already know it is zero or greater.

```
if(numChildren <= 3) {
  credit = numChildren * 1000;
} else {
  credit = 3000;
}
```

Finally, we need to ensure that the credit does not exceed the total tax liability (the credit is non-refundable, so if the credit is greater, the tax should only be zero, not negative).

```
if(baseTax - credit >= 0) {
  totalTax = baseTax - credit;
} else {
  totalTax = 0;
}
```

The full program is presented in Code Sample 27.3.

27.3.3. Quadratic Roots Revisited

Let's return to the quadratic roots program we previously designed that uses the quadratic equation to compute the roots of a quadratic polynomial by reading coefficients a, b, c in from the user. One of the problems we had previously identified is if the user enters "bad" input: if $a = 0$, we would end up dividing by zero; if $b^2 - 4ac < 0$ then we would have complex roots. With conditionals, we can now check for these issues and exit with an error message.

Another potential case we might want to handle differently is when there is only one distinct root ($b^2 - 4ac = 0$). In that case, the quadratic formula simplifies to $\frac{-b}{2a}$ and we can print a different, more specific message to the user. The full program can be found in Code Sample 27.4.

```java
/**
 * This program computes the logarithm base b (b > 1)
 * of a given number x > 0
 */
public class Logarithm {

  public static void main(String args[]) {

    double b, x, result;
    if(args.length != 2) {
      System.out.println("Usage: b x");
      System.exit(1);
    }

    b = Double.parseDouble(args[0]);
    x = Integer.parseInt(args[1]);

    if(x <= 0) {
      System.out.println("Error: x must be greater than zero");
      System.exit(1);
    }
    if(b <= 1) {
      System.out.println("Error: base must be greater than one");
      System.exit(1);
    }

    result = Math.log(x) / Math.log(b);
    System.out.printf("log_(%f)(%f) = %f\n", b, x, result);

  }

}
```

Code Sample 27.2.: Logarithm Calculator Program in Java

```java
import java.util.Scanner;

public class Taxes {

  public static void main(String args[]) {

    Scanner s = new Scanner(System.in);
    double income, baseTax, totalTax, numChildren, credit;

    System.out.println("Please enter your Adjusted Gross Income: ");
    income = s.nextDouble();

    System.out.println("How many children do you have?");
    numChildren = s.nextDouble();

    if(income < 0 || numChildren < 0) {
      System.out.println("Invalid inputs");
      System.exit(1);
    }

    if(income <= 18150) {
      baseTax = income * .10;
    } else if(income <= 73800) {
      baseTax = 1815 + (income  -18150) * .15;
    } else if(income <= 148850) {
      baseTax = 10162.50 + (income - 73800) * .25;
    } else if(income <= 225850) {
      baseTax = 28925.00 + (income - 148850) * .28;
    } else if(income <= 405100) {
      baseTax = 50765.00 + (income - 225850) * .33;
    } else if(income <= 457600) {
      baseTax = 109587.50 + (income - 405100) * .35;
    } else {
      baseTax = 127962.50 + (income - 457600) * .396;
    }

    if(numChildren <= 3) {
      credit = numChildren * 1000;
    } else {
      credit = 3000;
    }

    if(baseTax - credit >= 0) {
      totalTax = baseTax - credit;
    } else {
      totalTax = 0;
    }

    System.out.printf("AGI:           $%10.2f\n", income);
    System.out.printf("Tax:           $%10.2f\n", baseTax);
    System.out.printf("Credit:        $%10.2f\n", credit);
    System.out.printf("Tax Liability: $%10.2f\n", totalTax);

  }

}
```

Code Sample 27.3.: Tax Program in Java

```java
/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */
public class Roots {

  public static void main(String args[]) {
    double a, b, c, root1, root2;

    if(args.length != 3) {
      System.err.println("Usage: a b c\n");
      System.exit(1);
    }

    a = Double.parseDouble(args[0]);
    b = Double.parseDouble(args[1]);
    c = Double.parseDouble(args[2]);

    if(a == 0) {
      System.err.println("Error: a cannot be zero");
      System.exit(1);
    } else if(b*b < 4*a*c) {
      System.err.println("Error: cannot handle complex roots\n");
      System.exit(1);
    } else if(b*b == 4*a*c) {
      root1 = -b / (2*a);
      System.out.printf("Only one distinct root: %f\n", root1);
    } else {
      root1 = (-b + Math.sqrt(b*b - 4*a*c) ) / (2*a);
      root2 = (-b - Math.sqrt(b*b - 4*a*c) ) / (2*a);

      System.out.printf("The roots of %fx^2 + %fx + %f are: \n",
              a, b, c);
      System.out.printf("  root1 = %f\n", root1);
      System.out.printf("  root2 = %f\n", root2);
    }
  }

}
```

Code Sample 27.4.: Quadratic Roots Program in Java With Error Checking

28. Loops

Java supports while loops, for loops, and do-while loops using the keywords `while`, `for`, and `do` (along with another `while`). Continuation conditions for loops are enclosed in parentheses, `(...)` and the blocks of code associated with the loop are enclosed in curly brackets.

28.1. While Loops

Code Sample 28.1 contains an example of a basic while loop in C. Just as with conditional statements, our code styling places the opening curly bracket on the same line as the `while` keyword and continuation condition. The inner block of code is also indented and all lines in the block are indented to the same level.

In addition, the continuation condition does *not* have a semicolon since it is not an executable statement. Just as with an if-statement, if we *had* included a semicolon it would have led to unintended results. Consider the following:

```
while(i <= 10); {
  //perform some action
  i++; //iteration
}
```

```
int i = 1; //Initialization
while(i <= 10) {  //continuation condition
  //perform some action
  i++; //iteration
}
```

Code Sample 28.1.: While Loop in Java

```java
int i = 1;
boolean flag = true;
while(flag) {
  //perform some action
  i++; //iteration
  if(i>10) {
    flag = false;
  }
}
```

Code Sample 28.2.: Flag-controlled While Loop in Java

A similar problem occurs: the `while` keyword and continuation condition bind to the next executable statement or code block. As a consequence of the semicolon, the executable statement that gets bound to the while loop is *empty*. What happens is even worse: the program will enter an infinite loop. To see this, the code is essentially equivalent to the following:

```java
while(i <= 10) {
}
{
  //perform some action
  i++; //iteration
}
```

In the while loop, we never increment the counter variable `i`, the loop does nothing, and so the computation will continue on forever! Some compilers will warn you about this, others will not. It is valid Java and it will compile and run, but obviously won't work as intended. Avoid this problem by using proper syntax.

Another common use case for a while loop is a flag-controlled loop in which we use a Boolean flag rather than an expression to determine if a loop should continue or not. An example can be found in Code Sample 28.2.

28.2. For Loops

For loops in Java use the familiar syntax of placing the initialization, continuation condition, and iteration on the same line as the keyword `for`. An example can be found in Code Sample 28.3.

```
1 for(int i=1; i<=10; i++) {
2   //perform some action
3 }
```

Code Sample 28.3.: For Loop in Java

Semicolons are placed at the end of the initialization and continuation condition, but *not* the iteration statement. Just as with while loops, the opening curly bracket is placed on the same line as the `for` keyword. Code within the loop body is indented, all at the same indentation level.

Another observation: the declaration of the counter variable `i` was done in the initialization statement. This scopes the variable to the loop itself. The variable `i` is valid inside the loop body, but will be out-of-scope *after* the loop body. It is possible to declare the variable prior to the loop, but the variable `i` would have a much larger scope. It is best practice to limit the scope of variables only to where they are needed. Thus, we will write our loops as above.

28.3. Do-While Loops

Finally, Java supports do-while loops. Recall that the difference between a while loop and a do-while loop is when the continuation condition is checked. For a while loop it is *prior* to the beginning of the loop body and in a do-while loop it is at the *end* of the loop. This means that a do-while always executes *at least once*. An example can be found in Code Sample 28.4.

The opening curly bracket is again on the same line as the keyword `do`. The `while` keyword and continuation condition are on the same line as the closing curly bracket. In a slight departure from previous syntax, a semicolon *does* appear at the end of the continuation condition even though it is not an executable statement.

```
int i;
do {
  //perform some action
  i++;
} while(i <= 10);
```

Code Sample 28.4.: Do-While Loop in Java

28.4. Enhanced For Loops

Java also supports foreach loops (which were introduced in JDK 1.5.0) which Java refers to as "Enhanced For Loops." Foreach loops allow you to iterate over each element in a collection without having to define an index variable or otherwise "get" each element. We'll revisit these concepts in detail in Chapter 31, but let's take a look at a couple of examples.

An enhanced for loop in Java still uses the keyword `for` but uses different syntax for its control. The example in Code Sample 28.5 illustrates this syntax: `(int a : arr)`. The last element of this syntax is a reference to the collection that we want to iterate over. The first part is the *type* and local reference variable that the loop will use.

```
int arr[] = {10, 20, 8, 42};
int sum = 0;
for(int a : arr) {
  sum += a;
}
```

Code Sample 28.5.: Enhanced For Loops in Java Example 1

The code `(int a : arr)` should be read as "for each integer element `a` in the collection `arr`..." Within the enhanced for loop, the variable `a` will be automatically updated for you on each iteration. Outside the loop body, the variable `a` is out-of-scope.

Java allows you to use an enhanced for loop with any array or collection (technically, anything that implements the `Iterable` interface). One example is a `List`, an ordered collection of elements. Code Sample 28.6 contains an example.

```java
List<Integer> list = Arrays.asList(10, 20, 8, 42);
int sum = 0;
for(Integer a : list) {
  sum += a;
}
```

Code Sample 28.6.: Enhanced For Loops in Java Example 2

28.5. Examples

28.5.1. Normalizing a Number

Let's revisit the example from Section 4.1.1 in which we *normalize* a number by continually dividing it by 10 until it is less than 10. The code in Code Sample 28.7 specifically refers to the value 32145.234 but would work equally well with any value of `x`.

```java
double x = 32145.234;
int k = 0;
while(x > 10) {
  x = x / 10; //or: x /= 10;
  k++;
}
```

Code Sample 28.7.: Normalizing a Number with a While Loop in Java

28.5.2. Summation

Let's revisit the example from Section 4.2.1 in which we computed the sum of integers $1 + 2 + \cdots + 10$. The code is presented in Code Sample 28.8

We could have easily generalized the code somewhat. Instead of computing a sum up to a particular number, we could have written it to sum up to another variable `n`, in which case the for loop would instead look like the following.

```java
int sum = 0;
for(int i=1; i<=10; i++) {
  sum += i;
}
```

Code Sample 28.8.: Summation of Numbers using a For Loop in Java

```java
for(int i=1; i<=n; i++) {
  sum += i;
}
```

28.5.3. Nested Loops

Recall that you can write loops within loops by nesting them. The inner loop will execute fully *for each* iteration of the outer loop. An example of two nested of loops in Java can be found in Code Sample 28.9.

```java
int i, j;
int n = 10;
int m = 20;
for(i=0; i<n; i++) {
  for(j=0; j<m; j++) {
    System.out.printf("(i, j) = (%d, %d)\n", i, j);
  }
}
```

Code Sample 28.9.: Nested For Loops in Java

The inner loop execute for $j = 0, 1, 2, \ldots, 19 < m = 20$ for a total of 20 iterations. However, it executes 20 times *for each* iteration of the outer loop. Since the outer loop execute for $i = 0, 1, 2, \ldots, 9 < n = 10$, the total number of times the `System.out.printf()` statement execute is $10 \times 20 = 200$. In this example, the sequence

$$(0, 0), (0, 1), (0, 2), \ldots, (0, 19), (1, 0), \ldots, (9, 19)$$

will be printed.

28.5.4. Paying the Piper

Let's adapt the solution for the loan amortization schedule we developed in Section 4.7.3. First, we'll read the principle, terms, and interest as command line inputs. Adapting the formula for the monthly payment and using the math library's `Math.pow()` function, we have the following.

```
1 double monthlyPayment = (monthlyInterestRate * principle) /
2   (1 - pow( (1 + monthlyInterestRate), -n));
```

However, recall that we may have problems due to accuracy. The monthly payment could come out to be a fraction of a cent, say $43.871. For accuracy, we need to ensure that all of the figures for currency are rounded to the nearest cent. The standard math library does have a `Math.round()` function, but it only rounds to the nearest whole number, not the nearest 100th.

However, we can *adapt* the "off-the-shelf" solution to fit our needs. If we take the number, multiply it by 100, we get 4387.1 which we can now round to the nearest whole number, giving us 4387. We can then divide by 100 to get a number that has been rounded to the nearest 100th! In Java, we could simply do the following.

```
monthlyPayment = Math.round(monthlyPayment * 100.0) / 100.0;
```

We can use the same trick to round the monthly interest payment and any other number expected to be whole cents. To output our numbers, we use `System.out.printf()` and take care to align our columns to make make it look nice. To finish our adaptation, we handle the final month separately to account for an over/under payment due to rounding. The full solution can be found in Code Sample 28.10.

```java
public class LoanAmortization {

  public static void main(String args[]) {

    if(args.length != 4) {
      System.err.println("Usage: principle apr terms");
      System.exit(1);
    }

    double principle = Double.parseDouble(args[0]);
    double apr = Double.parseDouble(args[1]);
    int n = Integer.parseInt(args[2]);

    double balance = principle;
    double monthlyInterestRate = apr / 12.0;

    //monthly payment
    double monthlyPayment = (monthlyInterestRate * principle) /
      (1 - Math.pow( (1 + monthlyInterestRate), -n));
    //round to the nearest cent
    monthlyPayment = Math.round(monthlyPayment * 100.0) / 100.0;

    System.out.printf("Principle: $%.2f\n", principle);
    System.out.printf("APR: %.4f%%\n", apr*100.0);
    System.out.printf("Months: %d\n", n);
    System.out.printf("Monthly Payment: $%.2f\n", monthlyPayment);

    //for the first n-1 payments in a loop:
    for(int i=1; i<n; i++) {
      //  compute the monthly interest, rounded:
      double monthlyInterest =
        Math.round( (balance * monthlyInterestRate) * 100.0) / 100.0;
      //  compute the monthly principle payment
      double monthlyPrinciplePayment = monthlyPayment - monthlyInterest;
      //  update the balance
      balance = balance - monthlyPrinciplePayment;
      //  print i, monthly interest, monthly principle, new balance
      System.out.printf("%d\t$%10.2f  $%10.2f  $%10.2f\n", i, monthlyInterest,
        monthlyPrinciplePayment, balance);
    }

    //handle the last month and last payment separately
    double lastInterest = Math.round(
            (balance * monthlyInterestRate) * 100.0) / 100.0;
    double lastPayment = balance + lastInterest;

    System.out.printf("Last payment = $%.2f\n", lastPayment);

  }

}
```

Code Sample 28.10.: Loan Amortization Program in Java

29. Methods

As an object-oriented programming language, functions in Java are usually referred to as *methods* and are essential to writing programs. The distinction is that a function is usually a standalone element while methods are functions that are members of a class. In Java, since everything is a class or belongs to a class, standalone functions cannot be defined.

In Java you can define your own methods, but they need to be placed within a class. Usually methods that act on data in the class (or instances of the class, see Chapter 34) or have common functionality are placed into one class. For example, all the basic math methods are part of the `java.lang.Math` class. It is not uncommon to place similar methods together into one "utility" class.

Java supports method overloading, so within the same class you can define multiple methods with the same name as long as they differ in either the number (also called *arity*) or type of parameters. For example, in the `java.lang.Math` class, there are 3 versions of the absolute value method, `abs()`, one that takes/returns an `int`, one that takes/returns a `double` and one for `float` types. Naming conflicts can easily be solved by ensuring that you place your methods in a class/package that is unique to your application.

In Java, the 8 primitive types (`int`, `double`, `char`, `boolean`, etc.) are always passed by value. All object types, however, such as the wrapper classes `Integer`, `Double` as well as `String`, etc. are passed by reference. That is, the memory address in the JVM is passed to the method. This is done for efficiency. For objects that are "large" it would be inefficient to copy the entire object into the call stack in order to pass it to a method.

Though object types are passed by reference, the method cannot necessarily change them. Recall that the wrapper classes `Integer`, `Double` and the `String` class are all immutable, meaning that once created they cannot be modified. Though they are passed by reference, the method that receives them cannot change them.

There are many *mutable* objects in Java. The `StringBuilder` class for example is a mutable object. If you pass a `StringBuilder` instance to a method, that method is free to invoke mutator methods on the object such as `.append()` that *change* the object's state. Since it is the *same* object as in the calling method, the calling method can "see" those changes.

As of Java 5, you can write and use vararg methods. The `System.out.printf()` method is a prime example of this. However, we will not discuss in detail how to do this. Instead, refer to standard Java documentation. Finally, parameters are *not* optional in Java. This is because Java supports method overloading. You can write multiple versions of the same method that each take a different number of arguments. You can even design them so that the more specific versions (with fewer arguments) invoke the more general versions (with more arguments), passing in sensible "defaults."

29.1. Defining Methods

Defining methods is fairly straightforward. First you create a class to place them in. Then you provide the method signature along with the body of the method. In addition, there are several *modifiers* that you can place in the method signature to specify its *visibility* and whether or not the method "belongs" to the class or to instances of the class. This is a concept we'll explore in Chapter 34. For now, we'll only focus on what is needed to get started.

Typically, the documentation for methods is included with the method definition using "Javadoc" style comments. Consider the following examples.

```
1  /**
2   * Computes the sum of the two arguments.
3   * @param a
4   * @param b
5   * @return the sum, <code>a + b</code>
6   */
7  public static int sum(int a, int b) {
8    return (a + b);
9  }
10
11 /**
12  * Computes the Euclidean distance between the 2-D points,
13  * (x1,y1) and (x2,y2).
14  * @param x1
15  * @param y1
16  * @param x2
17  * @param y2
18  * @return
19  */
20 public static double getDistance(double x1, double y1,
21                           double x2, double y2) {
22   double xDiff = (x1-x2);
```

```
23    double yDiff = (y1-y2);
24    return Math.sqrt( xDiff * xDiff + yDiff * yDiff);
25  }
26
27  /**
28   * Computes a monthly payment for a loan with the given
29   * principle at the given APR (annual percentage rate) which
30   * is to be repaid over the given number of terms.
31   * @param principle - the amount borrowed
32   * @param apr - the annual percentage rate
33   * @param terms - number of terms (usually months)
34   * @return
35   */
36  public static double getMonthlyPayment(double principle,
37                                double apr, int terms) {
38    double rate = (apr / 12.0);
39    double payment = (principle * rate) / (1-Math.pow(1+rate, -terms));
40    return payment;
41  }
```

In each of the examples above, the first modifier keyword we used was `public`. This makes the method visible to all other parts of the code base. Any other piece of code can invoke the method and take advantage of the functionality it provides. Alternatively, we could have used the keywords `private` (which makes it visible only to other methods in the same class) `protected` or "package protected" by omitting the modifier altogether. We will want our methods to be available to other classes, so we'll make most of them `public`. The second modifier is `static` which makes it so that the method belongs to the class itself rather than instances of the class. We discuss visibility keywords and instances in detail in Chapter 34. For now, we will make all of our methods `static`.

After the modifiers, we provide the method signature including the return type, its identifier (name), and its parameter list. Method names must follow the same naming rules as variables: they must begin with an alphabetic character and may contain alphanumeric characters as well as underscores. However, using modern coding conventions we name methods using lower camel casing. Immediately after the signature we provide a method body which contains the code that will be run upon invocation of the method. The method body is enclosed using opening/closing curly brackets.

29.1.1. Void Methods

The keyword `void` can be used in Java to indicate a method does *not* return a value, in which case it is called a "void method." Though it is not necessary, it is still good practice to include a `return` statement.

```
public static void printCopyright()
  System.out.println("(c) Bourke 2015");
  return;
}
```

In the example above, we've also illustrated how to define a method that has no input parameters.

29.1.2. Using Methods

Once a method has been defined in a class, you can make use of the method as follows. First, you may need to import the class itself depending on where it is. For example, suppose that the examples we've presented so far are contained in a class named `Utils` (short for "utilities") which is in a package named `unl.cse`. Then in the class in which we want to call some of these functions we would import it using

```
import unl.cse.Utils;
```

prior to the class declaration. Once the class has been imported, we can invoke a method in the class by first referencing the class and using the *dot operator* to access one of its methods. For example,

```
int a = 10, b = 20;
int c = Utils.sum(a, b); //c contains the value 30

//invoke a method with literal values:
double dist = Utils.getDistance(0.0, 0.0, 10.0, 20.0);

//invoke a method with a combination:
double p = 1500.0;
double r = 0.05;
double monthlyPayment = Utils.getMonthlyPayment(p, r, 60);
```

The `Utils.methodName()` syntax is used because the methods are `static` –they belong to the class and so must be invoked *through* the class using the class's name. We've previously seen this syntax when using `System.` or `Math.` with the standard JDK library functions.

29.1.3. Passing By Reference

Java does not allow you the ability to specify if a variable is passed by reference or by value. Instead, all primitive types are passed by value while all object types are passed by reference. Moreover, most of the built-in types such as `Integer` and `String` are immutable, even though they are passed by reference, any method that receives them cannot change them. Only if the passed object is *mutable* can the method make changes to it (by invoking its methods).

As an example, consider the following piece of code. The `StringBuilder` class is a mutable string object. You can *change* the string contents stored in a `StringBuilder` by calling one of its many methods such as `append()`, which will add whatever string you give it to the end.

In the main method, we create two objects, a `String` and a `StringBuilder` and pass it to a method that makes changes to both by appending `" world!"` to them. Understand what happens here though. The first line in `change()` actually creates a *new* string and then changes what the parameter variable `s` references. The reference to the original string, `"Hello"` is lost and replaced with the new string. In contrast, the `StringBuilder` instance is actually changed via its `append()` method but is *still the same* object.

```
1  public class Mutability {
2
3    public static void change(String s, StringBuilder sb) {
4      s = s + " world!";
5      sb.append(" world!");
6
7      System.out.println("change: s  = " + s);
8      System.out.println("change: sb = " + sb);
9    }
10
11   public static void main(String args[]) {
12     String a = "Hello";
13     StringBuilder b = new StringBuilder("Hello");
14
15     System.out.println("main: s = " + a);
```

```
16      System.out.println("main: b = " + b);
17
18      change(a, b);
19
20      System.out.println("main after: s = " + a);
21      System.out.println("main after: b = " + b);
22
23    }
24  }
```

To see this, observe the following output. When we return to the main method, the original string `s` is unchanged (since it was immutable). However, the `StringBuilder` has been changed by the method.

```
main: s = Hello
main: b = Hello
change: s  = Hello world!
change: sb = Hello world!
main after: s = Hello
main after: b = Hello world!
```

29.2. Examples

29.2.1. Generalized Rounding

Recall that the math library provides a `Math.round()` method that rounds a number to the nearest whole number. We've had need to round to cents as well. We now have the ability to write a method to do this for us. Before we do, however, let's think more generally. What if we wanted to round to the nearest tenth? Or what if we wanted to round to the nearest 10s or 100s place? Let's write a general purpose rounding method that allows us to specify *which* decimal place to round to.

The most natural input would be to specify the place using an integer exponent. That is, if we wanted to round to the nearest tenth, then we would pass it -1 as $0.1 = 10^{-1}$, -2 if we wanted to round to the nearest 100th, etc. In the other direction, passing in 0 would correspond to the usual round function, 1 to the nearest 10s spot, and so

on. We could demonstrate good code reuse (as well as good procedural abstraction) by *scaling* the input value and reusing the functionality already provided in the math library's `Math.round()` method. We could further define a `roundToCents()` method that used our generalized round method. Finally, we could place all of these methods into `RoundUtils` Java class for good organization.

```
1  package unl.cse;
2
3  /**
4   * A collection of rounding utilities
5   *
6   */
7  public class RoundUtils {
8
9    /**
10    * Rounds to the nearest digit specified by the place
11    * argument.  In particular to the (10^place)-th digit
12    *
13    * @param x the number to be rounded
14    * @param place the place to be rounded to
15    * @return
16    */
17   public static double roundToPlace(double x, int place) {
18     double scale = Math.pow(10, -place);
19     double rounded = Math.round(x * scale) / scale;
20     return rounded;
21   }
22
23   /**
24    * Rounds to the nearest cent (100th place)
25    *
26    * @param x
27    * @return
28    */
29   public static double roundToCents(double x) {
30     return RoundUtils.roundToPlace(x, -2);
31   }
32
33 }
```

Observe that this class does not contain a `main()` method. That means that this class

is *not* executable itself. It only provides functionality to other classes in the code base.

30. Error Handling & Exceptions

Java supports error handling through the use of exceptions. Java has many different predefined types of exceptions that you can use in your own code. It also allows you to define your own exception types by creating new classes that *inherit* from the predefined classes. Java uses the standard `try-catch-finally` control structure to handle exceptions and allows you to `throw` your own exceptions.

30.1. Exceptions

Java defines a base class named `Throwable`. There are two majors subtypes of `Throwable`: `Error` and `Exception`. The `Error` class is used primarily for *fatal* errors such as the JVM running out of memory or some other extreme case that your code cannot reasonably be expected to recover from. There are dozens of types of exceptions that are subclasses of the standard Java `Exception` class defined by the JDK including `IOException` (and its subclasses such as `FileNotFoundException`) or `SQLException` (when working with Structured Query Language (SQL) databases). An important subclass of `Exception` is `RuntimeException` which represent *unchecked* exceptions that do *not* need to be explicitly caught (see Section 30.1.4). We'll mostly focus on this type of exception.

30.1.1. Catching Exceptions

To catch an exception in Java you can use the standard `try-catch` control block (and optionally use the `finally` block to clean up any resources). Let's take, for example, the simple task of reading input from a user using `Scanner` and manually parsing its value into an integer. If the user enters a non-numeric value, parsing will fail and result in a `NumberFormatException` that we can then catch and handle. For example,

```
1 Scanner s = new Scanner(System.in);
2 int n = 0;
3
```

```
try {
  String input = s.next();
  n = Integer.parseInt(input);
} catch (NumberFormatException nfe) {
  System.err.println("You entered invalid data!");
  System.exit(1);
}
```

In this example, we've simply displayed an error message to the standard error output and exited the program. That is, we've made the design decision that this error should be fatal. We could have chosen to handle this error differently in the `catch` block.

The code above could have resulted in other exceptions. For example if the `Scanner` had failed to read the next token from the standard input, it would have thrown a `NoSuchElementException`. We can add as many `catch` blocks as we want to handle each exception differently.

```
Scanner s = new Scanner(System.in);
int n = 0;

try {
  String input = s.next();
  n = Integer.parseInt(input);
} catch (NumberFormatException nfe) {
  System.err.println("You entered invalid data!");
  System.exit(1);
} catch (NoSuchElementException nsee) {
  System.err.println("Input reading failed, using default...");
  n = 20; //a default value
} catch(Exception e) {
  System.err.println("A general exception occurred");
  e.printStackTrace();
  System.exit(1);
}
```

Each `catch` block catches a different type of exception. Thus, the name of the variable that holds each exception must be different in the chain of `catch` blocks; `nfe`, `nsee`, `e`.

Note that the last `catch` block was written to catch a generic `Exception`. This last block will essentially catch any other type of exception. Much like an `if-else-if` statement, the first type of exception that is caught is the block that will be executed and they are all mutually exclusive. Thus, a "catch all" block like this should always be the *last* `catch` block. The most specific types of exceptions should be caught first and the most general types should be caught last.

30.1.2. Throwing Exceptions

We can also manually `throw` an exception. For example, we can `throw` a generic `RuntimeException` using the following.

```
throw new RuntimeException("Something went wrong");
```

By using a generic `RuntimeException`, we can attach a message to the exception (which can be printed by code that catches the exception). If we want more fine-grained control over the type of exceptions, we need to define our own exceptions.

30.1.3. Creating Custom Exceptions

To create your own exceptions, you need to create a new class to represent the exception and make it *extend* `RuntimeException` by using the keyword `extends`. This makes your exception a subclass or *subtype* of the `RuntimeException`. This is a concept known as *inheritance* in OOP. Consider the example in the previous chapter of computing the roots of a quadratic polynomial. One possible error situation is when the roots are complex numbers. We could define a new Java class as follows.

```
public class ComplexRootException extends RuntimeException {

  /**
   * Constructor that takes an error message
   */
  public ComplexRootException(String errorMessage) {
    super(errorMessage);
  }
}
```

Now in our code we can throw and catch this new type of exception.

```
//throw this exception:
if( b*b - 4*a*c < 0) {
  throw new ComplexRootException("Cannot Handle complex roots");
}
```

```
try {
  r1 = getComplexRoot01(a, b, c);
} catch(ComplexRootException cre) {
  //handle the exception here...
}
```

30.1.4. Checked Exceptions

A *checked* exception in Java is an exception that *must* be explicitly caught and handled. For example, the generic `Exception` class is a checked exception (others include `IOException`, `SQLException`, etc.). If a checked exception is thrown within a block of code then it must either be caught and handled within that block of code or the method must be specified to explicitly throw the exception. For example, the method

```
public static void processFile() {
  Scanner s = new Scanner(new File("data.csv"));
}
```

would not actually compile because `Scanner` throws a checked `FileNotFoundException`. Either we would have to explicitly `catch` the exception:

```
public static void processFile() {
  Scanner s = null;
  try {
    s = new Scanner(new File("data.csv"));
  } catch (FileNotFoundException e) {
```

```
6       //handle the exception here
7     }
8   }
```

or we would need to specify that the method `processFile()` explicitly `throws` the exception:

```
1   public static void processFile() throws FileNotFoundException {
2     Scanner s = new Scanner(new File("data.csv"));
3   }
```

Doing this, however, would force any code that called the `processFile()` method to surround it in a `try-catch` block and explicitly handle it (or once again, throw it back to the calling method).

The point of a checked exception is to force code to deal with potential issues that can be reasonably anticipated (such as the unavailability of a file). However, from another point of view checked exceptions represent the exact opposite goal of error handling. Namely, that a function or code block can and should *inform* the calling function that an error has occurred, but *not* explicitly make a decision on how to handle the error. A checked exception doesn't make the full decision for the calling function, but it does eliminate *ignoring* the error as an option from the calling function.

Java also supports *unchecked* exceptions which do *not* need to be explicitly caught. For example, `NumberFormatException` or `NullPointerException` are unchecked exceptions. If an unchecked exception is thrown and not caught, it bubbles up through the call stack until some piece of code does catch it. If no code catches it, it results in a fatal error and terminates the execution of the JVM.

The `RuntimeException` class and any of its subclasses are unchecked exceptions. In our `ComplexRootException` example above, because we extended `RuntimeException` we made it an unchecked exception, allowing the calling function to decide not only *how* to handle it, but whether or not to handle it *at all*. If we had instead decided to extend `Exception` we would have made our exception a checked exception.

There is considerable debate as to whether or not checked exceptions are a good thing (and as to whether or not unchecked exceptions are a good thing). Many[1] feel that checked

[1]The author included.

exceptions were a mistake and their usage should be avoided. The rationale behind checked exceptions is summed up in the following quote from the Java documentation [7].

> Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception

The problem is that the JDK's own design violates this principle. For example, `FileNotFoundException` is a checked exception; the reasoning being that a program could *reprompt* the user for a different file. The problem is the assumption that the program we are writing is always interactive. In fact most software is *not* interactive and is instead designed to interact with other software. Reprompting is *not* appropriate or even an option in the vast majority of cases.

As another example, consider Java's SQL library which allows you to programmatically connect to an SQL database. Nearly every method in the API explicitly throws a checked `SQLException`. It stretches the imagination to understand how our code or even a user would be able to recover (programmatically at least) from a lost internet connection or a bad password, etc. In general, you should use unchecked exceptions.

30.2. Enumerated Types

Another way to prevent errors is by restricting the values of a variable that a programmer is allowed to use. Java allows you to define a special class called an enumerated type which allow you to define a fixed list of possible values. For example, the days of the week or months of the year are possible values used in a date. However, they are more-or-less fixed (no one will be adding a new day of the week or month any time soon). An enumerated class allows us to define these values using human-readable keywords.

To create an enumerated type class we use the keyword `enum` (short for enumeration) instead of `class`. Since an enumerated type is a class, it must follow the same rules. It must be in a `.java` source file of the same name. Inside the `enum` we provide a comma-delimited list of keywords to define our enumeration. Consider the following example.

```
1 public enum Day {
2   SUNDAY,
3   MONDAY,
4   TUESDAY,
5   WEDNESDAY,
6   THURSDAY,
7   FRIDAY,
```

```
8     SATURDAY;
9   }
```

In the example, since the name of the enumeration is `Day` this declaration must be in a source file named `Day.java`. We can now declare variables of this type. The possible *values* it can take are restricted to `SUNDAY`, `MONDAY`, etc. and we can use these keywords in our program. However these values belong to the class `Day` and must be accessed statically. For example,

```
1  Day today = Day.MONDAY;
2
3  if(today == Day.SUNDAY || today == Day.SATURDAY) {
4    System.out.println("Its the weekend!");
5  }
```

Note the naming conventions: the name of the enumerated type follows the same upper camel casing that all classes use while the enumerated values follow an uppercase underscore convention. Though our example does not contain a value with multiple words, if it had, we would have used an underscore to separate them.

Using an enumerated type has two advantages. First, it enforces that only a particular set of predefined values can be used. You would not be able to assign a value to a `Day` variable that was not already defined by the `Day` enumeration. Second, without an enumerated type we'd be forced to use a collection of magic numbers to indicate values. Even for something as simple as the days of the week we'd be constantly trying to remember: which day is Wednesday again? Do our weeks start with Monday or Sunday? By using an enumerated type these questions are mostly moot as we can use the more human-readable keywords and eliminate the guess work.

30.2.1. More Tricks

Every `enum` type has some additional built-in methods that you can use. For example, there is a `values()` method that can be called on the `enum` that returns an array of the `enum`'s values. You can use an enhanced for loop to iterate over them, for example,

```
for(Day d : Day.values() {
  System.out.println(d.name());
}
```

In the example above, we used another feature: each `enum` value has a `name()` method that returns the value as a `String`. This example would end up printing the following.

```
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
```

Of course, we may want more human-oriented representations. To do this we could override the class's `toString()` method to return a better string representation. For example:

```
public String toString() {
  if(this == SUNDAY) {
    return "Sunday";
  } else if(this == MONDAY) {
    return "Monday";
  }
  ...
  } else {
    return "Saturday";
  }
}
```

Because `enum` types are full classes in Java, many more tricks can be used that leverage the power of classes including using additional state and constructors. We will cover these topics later.

31. Arrays

Java allows you to declare and use arrays. Since Java is statically typed, arrays must also be typed when they are declared and may only hold that particular type of element. Since Java has automated garbage collection, memory management is greatly simplified. Finally, in Java, only locally scoped primitives and references are allocated on the program call stack. As such, there are no static arrays; all arrays are allocated in the heap space.

31.1. Basic Usage

To declare an array reference, you use syntax similar to declaring a regular variable, but you use square brackets to designate the variable as an array. Arrays for any type of variable can be created including primitives and objects.

```
1 int arr[] = null;
2 double values[] = null;
3 String names[] = null;
```

This example[1] only declares 3 references that can refer to an array, it doesn't actually create them as the references are all set to `null`. To create arrays of a particular size, we need to initialize them using the keyword `new`.

```
1 arr = new int[10];
2 values = new double[20];
3 names = new String[5];
```

[1]You may see code examples that use the alternative notation `int[] arr = null;`. Both are acceptable. Some would argue that this way is more correct because it keeps the brackets with the type, avoiding the type-variable name-type separation in our example. Those preferring the `int arr[] = null;` notation would argue that it is "more natural" because that is how we ultimately index the array. Six of one, half-dozen of the other.

Each of these initializations creates a new array (allocated on the heap) of the specified size (10, 20, and 5 respectively). These arrays can only hold values of the specified type, `int`, `double`, and `String` respectively. The default value for each element in these new arrays will be zero for the numeric types and `null` for object types. Alternatively, you could use a compound declaration/assignment syntax to initialize specific values:

```
int arr[] = { 2, 3, 5, 7, 11 };
```

Using this syntax we do not need to specify the size of the array as the compiler is smart enough to count the number of elements we've provided. The elements themselves are denoted inside curly brackets and delimited with commas. For both types of syntax, the actual array is always allocated on the heap while the reference variables, `arr`, `values`, and `names` are stored in the program call stack.

Indexing

Once an array has been created, its elements can be accessed by indexing them. Java uses the standard 0-indexing scheme so the first element is at index 0, the second at index 1, etc. Indexing an element involves using the square bracket notation and providing an integer index. Once indexed, an array element can be treated as a normal variable and can be used with other operators such as the assignment operator or comparison operators.

```
arr[0] = 42;
if(arr[4] < 0) {
  System.out.println("negative!");
}
System.out.println("arr[1] = " + arr[1]);
```

Recall that an index is actually an offset. The compiler and system know exactly how many bytes each element takes and so an index `i` calculates exactly how many bytes from the first element the i-th element is located at. Consequently it is possible to index elements that are beyond the range of the array. For example, `arr[-1]` or `arr[5]` would attempt to access an element immediately *before* the first element and immediately *after* the last element. Obviously, these elements are not part of the array. If you attempt to access an element outside the bounds of an array, the JVM will throw an `IndexOutOfBoundsException` which is an unchecked `RuntimeException` that you can

catch and handle if you choose. To prevent such an exception you can write code that does not exceed the bounds of the array. Java arrays have a special *length* property that gives you the size of the array. You can access the property using the dot operator, so `arr.length` would give the value 5 for the example above.

Iteration

Using the `.length` property you can design a for loop that increments an index variable to iterate over the elements in an array.

```
1 int arr[] = new int[5];
2 for(int i=0; i<arr.length; i++) {
3   arr[i] = 5 * i;
4 }
```

The for loop above initializes the variable `i` to zero, corresponding to the first element in the array. The continuation condition specifies that the loop continues while `i` is strictly less than the size of the array denoted using the `arr.length` property. This for loop iteration is idiomatic when processing arrays.

In addition, Java (as of version 5) supports foreach loops that allow you to iterate over the elements of an array without using an index variable. Java refers to these loops as "enhanced for loops," but they are essentially foreach loops.

```
1 for(int a : arr) {
2   System.out.println(a);
3 }
```

The syntax still uses the keyword `for`, but instead of an index variable, it specifies the type and a the loop-scoped variable identifier followed by a colon and the array that you want to iterate over. Each iteration of the loop updates the variable `a` to the next value in the array.

31.2. Dynamic Memory

The use of the keyword `new` dynamically allocates memory space (on the heap) for the array. Because Java has automated garbage collection, if the reference to the array goes out of scope, it is automatically cleaned up by the JVM. This process is automated and essentially transparent to us. There are sophisticated algorithms and heuristics at work that determine the "optimal" time to perform garbage collection.

```
int arr[];
//create a new array of size 10:
arr = new arr[10];
//lose the reference by explicitly setting it to null:
arr = null;
//all references to the old memory are now lost and it is
//eligible for garbage collection

//we can also allocate new memory:
arr = new arr[20];
```

31.3. Using Arrays with Methods

We can pass and return arrays to and from methods in Java. We use the same syntax as when we define them by specifying a type and using square brackets. As an example, the following method takes an array of integers and computes its sum.

```
/**
 * This method computes the sum of elements in the
 * given array which contains n elements
 */
public static int computeSum(int arr[]) {
  int sum = 0;
  for(int i=0; i<size; i++) {
    sum += arr[i];
  }
  return sum;
}
```

In Java, arrays are always passed by reference. Though we did not make any changes to the contents of the passed array in the particular example, in general we could have. Any such changes would be realized in the calling method. Unfortunately, there is no mechanism by which we can prevent changes to arrays when passed to methods.[2]

We can also create an array in a method and return it as a value. For example, the following method creates a deep copy of the given integer array. That is, a completely new array that is a distinct copy of the old array. In contrast, a shallow copy would be if we simply made one reference point to another reference.

```
/**
 * This method creates a new copy of the given array
 * and returns it.
 */
public static int [] makeCopy(int a[]) {
  int copy[] = new int[a.length];
  for(int i=0; i<n; i++) {
    copy[i] = a[i];
  }
  return copy;
}
```

The method returns an integer array. In fact, this method is so useful, that it is already provided as part of the Java Software Development Kit (SDK). The class `Arrays` contains dozens of utility methods that process arrays. In particular there is a `copyOf()` method that allows you to create deep copies of arrays and even allows you to expand or shrink their size.

31.4. Multidimensional Arrays

Java supports multidimensional arrays, though we'll only focus on 2-dimensional arrays. To declare and allocate 2-dimensional arrays, we use two square brackets with two specified dimensions.

```
int matrix[][] = new int[10][20];
```

[2]The use of the keyword `final` only prevents us from changing the array reference, not modifying its contents.

This creates a 2-dimensional array of integers with 10 rows and 20 columns. Once created, we can index individual elements by specifying row and column indices.

```
for(int i=0; i<matrix.length; i++) {
  for(int j=0; j<matrix[i].length; j++) {
    matrix[i][j] = 10;
  }
}
```

31.5. Dynamic Data Structures

The Java SDK provides a rich assortment of dynamic data structures as alternatives to arrays including lists, sets and maps. The classes that support and implement these data structures are defined in the Collections library under the `java.util` package. We will cover how to use some of these data structures, but we will not go into the details of how they are implemented nor the OOP concepts that underly them.

The Java `List` is an *interface* that defines a dynamic list data structure. This data structure provides a dynamic collection that can grow and shrink automatically as you add and remove elements from it. It is an interface, so it doesn't actually provide an implementation, just a specification for the publicly available methods that you can use. Two common implementations are `ArrayList`, which uses an array to hold elements, and `LinkedList` which stores elements in linked nodes.

To create an instance of either of these lists, you use the `new` keyword and the following syntax.

```
List<Integer> a = new ArrayList<Integer>();
List<String> b = new LinkedList<String>();
```

The first line creates a new instance of an `ArrayList` that is *parameterized* to hold `Integer` types. The second creates a new instance of a `LinkedList` that has been parameterized to only hold `String` types. The parameterization is specified by placing the parameterized type inside the angle brackets. Because of this parameterization, it

would be a compiler error to attempt to add anything other than `Integer`s to the first list or anything other than `String`s to the second. Once these lists have been created, you can add and remove elements using the `add()` method.

```
a.add(42);
a.add(81);
a.add(17);

b.add("Hello");
b.add("World");
b.add("Computers!");
```

The order that you add elements is preserved, so in the first list, the first element would be 42, the second 81, and the last 17. You can remove elements by specifying an index of the element to remove. Like arrays, lists are 0-indexed.

```
a.remove(0);

b.remove(2);
b.remove(0);
```

As you remove elements, the indices are "shifted" down, so that after removing the first element in the list `a`, 81 becomes the new first element. Removing the last then the first element in the list `b` leaves it with only one element, `"World"` as the first element (at index 0).

You can also retrieve elements from a list using 0-indexing and the `get()` method.

```
List<Double> values = new ArrayList<Double>();
values.add(3.14);
values.add(2.71);
values.add(42.0);

double x = values.get(1); //get the second value, 2.71
double y = values.get(2); //get the third value, 42.0
```

Any attempt to access an element that lies outside the bounds of the `List`, will result in an `IndexOutOfBoundsException` just as with arrays. To stay within bounds you can use the `size()` method to determine how many elements are in the collection. In this example, `values.size()` would return an integer value of 3.

Finally, most collections implement the `Iterable` interface which allows you iterate over the elements using an enhanced for loop just as with arrays.

```
for(Double x : values) {
  System.out.println(x);
}
```

There are dozens of other methods that allow you to insert, remove, and retrieve elements from a Java `List`; refer to the documentation for details.

Another type of collection supported by the Collections library is a `Set`. A set differs from a list in that it is *unordered*. There is no concept of the first element or last element. Moreover, a `Set` does not allow duplicate elements.[3] A commonly used implementation of the `Set` interface is the `HashSet`.

```
Set<String> names = new HashSet<String>();

names.add("Robin");
names.add("Little John");
names.add("Marian");

//this has no effect since Robin is already part of the set:
names.add("Robin");
```

Since the elements in a `Set` are unordered, we cannot use an index-based `get()` method as we did with a `List`. Fortunately, we can still use an enhanced for loop to iterate over the elements.

```
for(String name : names) {
  System.out.println(name);
```

[3]Duplicates are determined by how the `equals()` and possibly the `hashCode()` methods are implemented in your particular objects.

```
}
```

When this code executes we cannot expect any particular order of the three names. Any permutation of the three may be printed. If we executed the loop more than once we may even observe a different enumeration of the names!

Finally, Java also supports a `Map` data structure which allows you to store key-value pairs. The keys and values can be any object type and are specified by two parameters when you create the map. A common implementation is the `HashMap`.

```
//define a map that maps integers (keys) to strings (values):
Map<Integer, String> numbers = new HashMap<Integer, String>();

//add key-value pairs:
numbers.add(10, "ten");
numbers.add(20, "twenty");

//retrieve values given a key:
String name = numbers.get(20); //name equals "twenty"

//invalid mappings result in null:
name = numbers.get(30); //name is null
```

The Collections library is much more extensive than what we've presented here. It includes implementations of stacks, queues, hash tables, balanced binary search trees, and many other dynamic data structure implementations.

32. Strings

As we've previously seen, Java has a `String` class in the standard JDK. Internally, Java strings are stored as arrays of characters. However, because of the `String` class, we never directly interact with this representation. Making using strings much easier than in other languages. Java strings have also supported Unicode since version 1.

Java provides many methods as part of the `String` class that can be used to process and manipulate strings. These methods *do not* change the strings since strings in Java are immutable. Instead, these methods operate by returning a *new* modified string that can then be stored in a variable.

32.1. Basics

We can declare `String` variables and assign them values using the regular assignment operator.

```
String firstName = "Thomas";
String lastName = "Waits";

//we can also reassign values
firstName = "Tom";
```

Note that the reassignment in the last line in the example does *not* change the original string. It just makes the variable `firstName` point to a new string. If there are no other references to the old string, it becomes eligible for garbage collection and the JVM takes care of it. Strings can be *copied* using the `String` class's copy constructor:

```
String copy = new String(firstName);
```

However, since strings are immutable, there is rarely reason to create such a deep copy.

Though you can't change individual characters in a string, you can access them using the

`charAt()` method and providing an index. Characters in a string are 0-indexed just as with elements in arrays.

```
String fullName = "Tom Waits";
//access individual characters:
char firstInitial = fullName.charAt(0); //'T'
char lastInitial = fullName.charAt(4); //'W'

if(fullName.charAt(8) == 's') {
  ...
}
```

32.2. String Methods

The `String` class provides dozens of convenient methods that allow you to process and modify strings. We highlight a few of the more common ones here. A full list of supported functions can be found in standard documentation.

Length

When accessing individual characters in a string, it is necessary to know the length of the string so that we do not access invalid characters. The `length()` method returns an `int` that represents the number of characters in the string.

```
String s = "Hello World!";
char last = s.charAt(s.length()-1);
//last is '!'
```

Using this method we can easily iterate over each character in a string.

```
for(int i=0; i<fullName.length(); i++) {
  System.out.printf("fullName[%d] = %c\n", i, fullName.charAt(i));
}

```

```
//or we can use toCharArray and an enhanced for loop:
for(char c : fullName.toCharArray()) {
  System.out.println(c);
}
```

Concatenation

Java has a concatenation operator built into the language. The familiar plus sign, `+` can be used to combine one or more strings by appending them to each other. Concatenation results in a new string.

```
String firstName = "Tom";
String lastName = "Waits";

String formattedName = lastName + ", " + firstName;
//formattedName now contains "Waits, Tom"
```

Concatenation also works with numeric types and even objects.

```
int x = 10;
double y = 3.14;

String s = "Hello, x is " + x + " and y = " + y;
//s contains "Hello, x is 10 and y = 3.14"
```

When used with objects, the concatenation operator internally invokes the object's `toString()` method. The plus operator as a concatenation operator is actually syntactic sugar. When the code is compiled, it is actually replaced with equivalent code that uses a series of the `StringBuilder` class's (a mutable version of the `String` class) `append()` method. The compiler will actually replace the code using the concatenation operator with code that does not use the concatenation operator. The example above would be replaced with something like the following.

```
String firstName = "Tom";
String lastName = "Waits";

String formattedName = new StringBuilder(lastName)
                        .append(", ")
                        .append(firstName)
                        .toString();
```

You can also use the `StringBuilder` class yourself directly.

Computing a Substring

There are two methods that allow you to compute a substring of a string. The first allows you to specify a beginning index with the entire remainder of the string being included in the returned string. The second allows you to specify a beginning index as well as an ending index. In both cases, the beginning index is *inclusive* (that is the resulting string includes the character at that index), but in the second, the ending index is *exclusive* (the character at the end index is *not* included).

```
String name = "Thomas Alan Waits";

String firstName  = name.substring(0, 6);  //"Thomas"
String middleName = name.substring(7, 11); //"Alan"
String lastName   = name.substring(12);    //"Waits"
```

The result of the two argument `substring()` method will always have length equal to `endIndex - beginIndex`.

32.3. Arrays of Strings

We often need to deal with collections of strings. In Java, we can define arrays of strings. Indeed, we've seen arrays of strings before. In a Java class's `main()` method, command line arguments are passed as an array of strings:

```
public static void main(String args[])
```

We can create our own arrays of strings similar to how we created arrays of `int` and `double` types.

```
//create an array that can hold 5 strings
String names[] = new String[5];

names[0] = "Margaret Hamilton";
names[1] = "Ada Lovelace";
names[2] = "Grace Hopper";
names[3] = "Marie Curie";
names[4] = "Hedy Lamarr";
```

Better yet, we can use dynamic collections such as a `List` or a `Set` of strings.

```
List<String> names = new ArrayList<String>();

names.add("Margaret Hamilton");
names.add("Ada Lovelace");
names.add("Grace Hopper");
names.add("Marie Curie");
names.add("Hedy Lamarr");

System.out.println(names.get(2));  //prints "Grace Hopper"
```

32.4. Comparisons

When comparing strings in Java, we cannot use the numerical comparison operators such as `==`, or `<`. Because strings are objects, using these operators actually compares the variables' *memory addresses*.

```
String a = new String("Hello World!");
String b = new String("Hello World!");

if(a == b) {
  System.out.println("strings match!");
```

```
}
```

The code above will not print anything even though the strings `a` and `b` have the same content. This is because `a == b` is comparing the memory address of the two variables. Since they point to different memory addresses (created by two separate calls to the constructors) they will not be equal.

Instead, the `String` class provides a `compareTo()` method[1] that takes another string and returns something negative, zero, or something positive depending on the relative lexicographic ordering of the strings.

```
int x;
String a = "apple";
String b = "zelda";
String c = "Hello";
x = a.compareTo("banana"); //x is negative
x = b.compareTo("mario");  //x is positive
x = c.compareTo("Hello");  //x is zero

//shorter strings precede longer strings:
x = a.compareTo("apples"); //x is negative

String d = "Apple";
x = d.compareTo("apple");  //x is negative
```

In the last example, `"Apple"` precedes `"apple"` since uppercase letters are ordered before lowercase letters according to the ASCII table. We can also make case-insensitive comparisons using the `compareToIgnoreCase()` method which works the same way. Here, `d.compareToIgnoreCase("apple")` will return zero as the two strings are the same ignoring the cases.

Note that the commonly used `equals()` method only returns `true` or `false` depending on whether or not two strings are the same or different. They cannot be used to provide a relative ordering of two strings.

[1] This method is part of the `String` class due to the fact that strings implement the `Comparable` interface which defines a lexicographic ordering.

```
String a = "apple";
String b = "apple";
String c = "Hello";

boolean result;
result = a.equals(b); //true
result = a.equals(c); //false
```

32.5. Tokenizing

Recall that *tokenizing* is the process of splitting up a string using some *delimiter*. For example, the comma delimited string, `"Smith,Joe,12345678,1985-09-08"` contains four pieces of data delimited by a comma. Our aim is to split this string up into four separate strings so that we can process each one.

Java provides a very simple method to do this called `split()` that takes a string delimiter and returns an array of strings containing the tokens. For example,

```
String data = "Smith,Joe,12345678,1985-09-08";

String tokens[] = data.split(",");
//tokens is { "Smith", "Joe", "12345678", "1985-09-08" }

String dateTokens[] = tokens[3].split("-");
//dateTokens is now { "1985", "09", "08" }
```

The delimiter this method uses is actually a regular expression; a sequence of characters that define a search *pattern* in which special characters can be used to define complex patterns. For example, the complex expression,

`^[+-]?(\d+(\.\d+)?|\.\d+)([eE][+-]?\d+)?$`

will match any valid numerical value including scientific notation. We will not cover regular expressions in depth, but to demonstrate their usefulness, here's an example by which you can split a string along any and all whitespace:

```
String s = "Alpha Beta \t Gamma \n Delta    \t\nEpsilon";
String tokens[] = s.split("[\\s]+");
//tokens is now { "Alpha", "Beta", "Gamma", "Delta", "Epsilon" }
```

33. File I/O

Java provides several different classes and utilities that support manipulating and processing files. In general, most file operations may result in an `IOException`, a checked exception that *must* be caught and handled.

33.1. File Input

Though there are several ways that you can do file input, the easiest is to use the familiar `Scanner` class. We've previously used this class to read from the standard input, `System.in`, but it can also be used to read from a file using the `File` class (in the `java.io` package). The `File` class is just a representation of a file resource and does not represent the actual file (it cannot be opened and closed). To initialize a `Scanner` to read from a file you can use the following.

```
1 Scanner s = null;
2 try {
3   s = new Scanner(new File("/user/apps/data.txt"));
4 } catch (FileNotFoundException e) {
5   //handle the exception here
6 }
```

When initializing a `Scanner` to read from a file, the exception, `FileNotFoundException` *must* be caught and handled as it is a *checked* exception. Otherwise, once created, the `Scanner` class does not throw any checked exceptions.

Once created, you can use the usual methods and functionality provided by the `Scanner` class including reading the `nextInt()`, `nextDouble()`, next string using `next()` or even the entire `nextLine()`. This last one can be used in conjunction with `hasNext()` to read an entire file, line-by-line.

```
String line;
while(s.hasNext()) {
  line = s.nextLine();
  //process the line
}
```

Once we are done reading the file, we can close the `Scanner` to free up resources: `s.close();`. We could have placed all this code within one large `try-catch` block with perhaps a `finally` block to close the `Scanner` once were were done to ensure that it would be closed regardless of any exceptions. However, Java 7 introduced a new construct, the *try-with-resources* statement.

The try-with-resources statement allows us to place the initialization of a "closeable" resource (defined by the `AutoCloseable` interface) into the `try` statement. The JVM will then automatically close this resource upon the conclusion of the try-catch block. We can still provide a `finally` block if we wish, but this relieves us of the need to explicitly close the resource in a `finally` block. A full example:

```
File f = new File("/user/apps/data.txt");
//initialize a Scanner in the try statement
try (Scanner s = new Scanner(f)) {
  String line;
  while(s.hasNext()) {
    line = s.nextLine();
        //process the line
  }
} catch (Exception e) {
  //handle the exception here
}
```

Using the `Scanner` class to do file input offers a more abstract interaction with a file. It also uses a buffered input stream for performance. Binary files can still be read using `nextByte()`. However, when dealing with binary files, the better solution is to use a class that models and abstracts the underlying file. For example, if you are reading or writing an image file, you should use the `java.awt.Image` class to read and write to files. The JDK and other libraries offer a wide variety of classes to model all kinds of data.

33.2. File Output

There are several ways to achieve file output, but we'll look at the simplest and most useful way. Java provides a `PrintWriter` class which offers many convenient methods for writing primitive and `String` types in a formatted manner. It has the usual `print()` and `println()` methods that we are familiar with as well as a `printf()` style method for formatting.

`PrintWriter` is also buffered, depending on how you instantiate it, it might "flush" its buffer after every endline character or you may do it manually by invoking the `flush()` method.

```
int x = 10;
double pi = 3.14;
PrintWriter pw = null;
try {
  pw = new PrintWriter(new File("data.txt"));
} catch(FileNotFoundException fnfe) {
  throw new RuntimeException(fnfe);
}

pw.println("Hello World!");
pw.println(x);
pw.printf("x = %d, pi = %f\n", x, pi);

pw.close();
```

The `close()` method will conveniently close all the underlying resources and flush the buffer for us, writing any buffered data to the file. In addition, `PrintWriter` implements `AutoCloseable` and so it can be used in a try-catch-with resources statement.

Another "convenience" of `PrintWriter` is that is "swallows" exceptions (just as the `Scanner` class did). That means we don't have to deal explicitly with the checked `IOException`s that the underlying class(es) throw as the `PrintWriter` silently catches them (though doesn't handle them). However, this can also be viewed as a disadvantage in that if we want to do error handling, we need to manually check if there was an error (using `checkError()`). The `PrintWriter` class is intended mostly for formatted output. It does not provide a way to write binary data to an output file. Just as with binary input, it is best to use a class that abstracts the file type and data so that we don't have to deal with the low-level details of the binary data.

However, if necessary, binary output can be done using a `FileOutputStream`. Typically, you can load all your data into a byte array and dump it all at once.

```
byte data[] = ...;
try (FileOutputStream fos =
          new FileOutputStream(new File("outfile.bin")) ){
  fos.write(data);
} catch(IOException ioe) {
  throw new RuntimeException(ioe);
}
```

This example uses a try-with-resources statement so the `FileOutputStream` will automatically be closed for us. You *could* wrap a `FileOutputStream` in a `BufferedWriter`, but it will likely not gain you anything in terms of performance. A buffered output stream is better if your writes are frequent and "small." Here small means smaller than the size of the buffer (`BufferedWriter` is typically 8KB). Writing several individual integer values for example would be better done by buffering them and writing them all at once. In our example, we simply wanted to dump a bunch of data, likely more than 8KB in practice, all at once. Moreover, using a `BufferedWriter` would lose us the ability to write raw byte data.

34. Objects

Java is a class-based object-oriented programming language, meaning that it facilitates the creation of objects through the use of classes. Classes are essentially "blueprints" for creating instances of objects. We've been implicitly using classes all along since everything in Java is a class or belongs to a class. Now, however, we will start using classes in more depth rather than simply using static methods.

An *object* is an entity that is characterized by *identity*, *state* and *behavior*. The identity of an object is an aspect that distinguishes it from other objects. The variables and values that a variable takes on within an object is its state. Typically the variables that belong to an object are referred to as *member* variables. Finally, an object may also have functions that operate on the data of an object. In the context of object-oriented programming, a function that belongs to an object is referred to as a *member method*.

As a class-based object-oriented language, Java implements objects using *classes*. A class is essentially a blueprint for creating *instances* of the class. A class simply specifies the member variables and member methods that belong to instances of the class. We can create actual instances through the use of special *constructor* methods. To begin, let's define a class that models a student by defining member variables to support a first name, last name, a unique identifier, and GPA.

To declare a class, we use the `class` keyword. Inside the class (denoted by curly brackets), we place any code that *belongs* to the class. To declare member variables within a class, we use the normal variable declaration syntax, but we do so outside any methods.

```
1  package unl.cse;
2
3  public class Student {
4
5    //member variables:
6    String firstName;
7    String lastName;
8    int id;
9    double gpa;
10
11 }
```

Recall that a package declaration allows you to organize classes and code within a package (directory) hierarchy. Moreover, source code for a class *must* be in a source file with the same name (and is case sensitive) with the `.java` extension. Our `Student` class would need to be in a file named `Student.java` and would be compiled to a class named `Student.class`.

34.1. Data Visibility

Recall that encapsulation involves not only the grouping of data, but the *protection* of data. To provide for the protection of data, Java defines several *visibility* keywords that specify what segments of code can "see" the variables. Visibility in this context determines whether or not a segment of code can *access* and/or *modify* the variable's value. Java defines four levels of visibility using they keywords `public`, `protected` and `private` (a fourth level is defined by the absence of any of these keywords). Each of these keywords can be applied to both member variables and member methods.

- `public` – This is the least restrictive visibility level and makes the member variable visible to every class.
- `protected` – This is a bit more restrictive and makes it so that the member variable is only visible to the code in the same class, same package, or any *subclass* of the class.[1]
- No modifier – the absence of any modifier means that the member variable is visible to any class in the same package. This is also referred to as the *default* or *package protected* visibility level.
- `private` – this is the most restrictive visibility level, `private` member variables are only visible to instances of the class itself. As we'll see later this also means that the member variables are visible *between* instances. That is, one instance can see another instance's variables.

Table 34.1 summarizes these four keywords with respect to their access levels. It is important to understand that *protection* in the context of encapsulation and does not involve protection in the sense of "security." The protection is this context is a design principle. Limiting the access of variables only affects how the rest of the code base interacts with our class and its data. Encapsulation can easily be "broken" by other

[1] Subclasses are involved with *inheritance*, another object-oriented programming concept that we will not discuss here).

code (through reflection[2] or other means) and the values of variables can be accessed or modified.

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
none (default)	Y	Y	N	N
`private`	Y	N	N	N

Table 34.1.: Java Visibility Keywords & Access Levels

We now update our class declaration to incorporate these visibility keywords. In general, it is best practice to make member variables `private` and control access to them via accessor and mutator methods (see below) unless there is a compelling design reason to increase their visibility.

```
1  package unl.cse;
2
3  public class Student {
4
5    //member variables:
6    private String firstName;
7    private String lastName;
8    private int id;
9    private double gpa;
10
11 }
```

34.2. Methods

The third aspect of encapsulation involves the grouping of methods that act on an object's data (its *state*). Within a class, we can declare member methods using the syntax we're already familiar with. We declare a member method by providing a signature and body. We can use the same visibility keywords as with member variables in order to allow or restrict access to the methods. With methods, visibility and access determine whether or not the method may be invoked.

[2]Reflection is a mechanism by which you can write code that modifies other code at runtime. It is more of an *aspect-oriented* programming paradigm.

In contrast to the methods we defined in Chapter 29, when defining a member method, we do *not* use the `static` keyword. Making a variable or a method `static` means that the method belongs to the class and not to instances of the class. We add to our example by providing two `public` methods that compute and return a result on the member variables. We also use Javadoc style comments to document each member method.

```
1  package unl.cse;
2
3  public class Student {
4
5    //member variables:
6    private String firstName;
7    private String lastName;
8    private int id;
9    private double gpa;
10
11   /**
12    * Returns a formatted String of the Student's
13    * name as Last, First.
14    */
15   public String getFormattedName() {
16     return lastName + ", " + firstName;
17   }
18
19   /**
20    * Scales the GPA, which is assumed to be on a
21    * 4.0 scale to a percentage.
22    */
23   public double getGpaAsPercentage() {
24     return gpa / 4.0;
25   }
26
27 }
```

34.2.1. Accessor & Mutator Methods

Since we have made all the member variables `private`, no code outside the class may access or modify their values. It is generally good practice to make member variables private to restrict access. However, if we still want code outside the object to access

or mutate (that is, change) an instance's variables, we can define accessor and mutator methods (or just simply *getter* and *setter* methods) to facilitate this.

Each getter method returns the value of the instance's variable while each setter method takes a value and sets the instance's variable to the new value. It is common to name each getter/setter by prefixing a `get` and `set` to the variable's name using lower camel casing. For example:

```
1 public String getFirstName() {
2   return firstName;
3 }
4
5 public void setFirstName(String firstName) {
6   firstName = firstName;
7 }
```

In the setter example, there is a problem: the code has no effect. There are two variables named `firstName`: the instance's member variable and the variable in the method parameter. The scoping rules of Java mean that the parameter variable name(s) take precedent. This code has no effect because it is setting the parameter variable to itself. It is essentially doing the following.

```
1 int a = 10;
2 a = a;
```

To solve this, we use something called open recursion. When an instance of a class is created, for example,

```
Student s = ...;
```

the reference variable `s` is how we can refer to it. This variable, however, exists *outside* the class. Inside the class, we need a way to refer to the instance itself. In Java we use the keyword `this` to refer to the instance *inside* the class. For example, the member variables of an instance can be accessed using the `this` keyword along with the dot operator (more below). In our example, `this.firstName` would refer to the instance's `firstName` and not to the parameter variable. Even when it is not necessary to use the `this` keyword (as in the getter example above) it is still best practice to do so. Our updated getters and setter methods would thus look like the following.

```java
public String getFirstName() {
  return this.firstName;
}

public void setFirstName(String firstName) {
  this.firstName = firstName;
}
```

One advantage to using getters and setters (as opposed to naively making everything `public`) is that you can have greater control over the values that your variables can take. For example, we may want to do some data validation by rejecting `null` values or invalid values. For example:

```java
public void setFirstName(String firstName) {
  if(firstName == null) {
    throw new IllegalArgumentException("names cannot be null");
  } else {
    this.firstName = firstName;
  }
}

public void setGpa(double gpa) {
  if(gpa < 0.0 || gpa > 4.0) {
    throw new IllegalArgumentException("GPAs must be in [0, 4.0]");
  } else {
    this.gpa = gpa;
  }
}
```

Controlling access of member variables through getters and setters is good encapsulation. Doing so makes your code more predictable and more testable. Making your member variables `public` means that any piece of code can change their values. There is no way to do validation or prevent bad values.

In fact, it is good practice to not even have setter methods. If the value of member variables cannot be changed, it makes the object immutable. We've seen this before with the built-in wrapper classes (`Integer`, `String`, etc.). Immutability is a nice property because it makes instances of the class thread-safe. That is, we can use instances of

the class in a multithreaded program without having to worry about threads changing the state of instances on one another. Immutable classes are also safer to use in certain collections such as a `Set`. Elements in a `Set` are unique; attempting to add a duplicate element will have no effect on the `Set`. However, if the elements we add are mutable, we could end up with duplicates (see Section 34.5).

34.3. Constructors

If we make the (good) design decision to make our class immutable, we still need a way to initialize the object's member variables. This is where *constructors* come in. A constructor is a special method that specifies how an object is constructed. With built-in primitive variables such as an `int`, the Java language (compiler and JVM) "know" how to interpret and assign a value to such a variable. However, with user-defined objects such as our `Student` class, we need to specify how the object is created.

A constructor method has special syntax. Though it may still one of the visibility keywords, it has no return type and its name is the same as the class. A constructor may take any number of parameters. For example, the following constructor allows someone to construct a `Student` instance and specify all four member variables.

```
1 public Student(String firstName, String lastName,
2                int id, double gpa) {
3   this.firstName = firstName;
4   this.lastName = lastName;
5   this.id = id;
6   this.gpa = gpa;
7 }
```

Java allows us to define multiple constructors, or even no constructor at all. If we do not specify a constructor, Java provides a *default*, no argument constructor. This constructor uses default values for all member variables (zero for numerical types, `false` for `boolean` types, and `null` for objects). If we do specify a constructor, this default constructor becomes unavailable. However, we can always "restore" it by explicitly defining it.

```
1 public Student() {
2   this.firstName = null;
3   this.lastName = null;
4   this.id = 0;
```

```
5     this.gpa = 0.0;
6   }
```

Alternatively, we can define constructors that accept a subset of variable values.

```
1   public Student(String firstName, String lastName) {
2     this.firstName = firstName;
3     this.lastName = lastName;
4     this.id = 0;
5     this.gpa = 0.0;
6   }
```

In both of these examples, we repeated a lot of code. One shortcut is to make all your constructors call the most general constructor. To invoke another constructor, we use the `this` keyword as a method call. For example:

```
1   public Student() {
2     this(null, null, 0, 0.0);
3   }
4
5   public Student(String firstName, String lastName) {
6     this(firstName, lastName, 0, 0.0);
7   }
```

Another, very useful type of constructor is the *copy constructor* that allows you to make a *copy* of an instance by passing it to a constructor. The following example copies each of the member variables of `s` into `this`.

```
1   public Student(Student s) {
2     this(s.firstName, s.lastName, s.id, s.gpa);
3   }
```

34.4. Usage

Once we have defined our class and its constructors, we can create and use instances of it. Just as with regular variables, we need to declare instances of a class by providing the type and a variable name. For example:

```
Student s = null;
Student t = null;
```

Both of these declarations are simply just *reference* variables. They may refer to an instance of the class `Student`, but we have initialized them to `null`. To make these variables refer to valid instances, we invoke a constructor by using the `new` keyword and providing arguments to the constructor.

```
Student s = new Student("Alan", "Turing", 1234, 3.4);
Student t = new Student("Margaret", "Hamilton", 4321, 3.9);
```

The process of creating a new instance by invoking a constructor is referred to as *instantiation*. Once instances have been instantiated, they can be used by invoking their methods via the *dot operator*. The dot operator is used to select a member variable (if `public`) or member method.

```
System.out.println(s.getFormattedName());

if(s.getGpa() < t.getGpa()) {
  System.out.println(t.getFirstName() + " has a better GPA");
}
```

34.5. Common Methods

Every class in Java has several methods that they *inherit* from the `java.lang.Object` class. Here, we will highlight three of these important methods.

The `toString()` method returns a `String` representation of the object. However, the default behavior that all classes inherit from the `Object` class is that it returns a string containing the fully qualified class name (package and class name) along with a hexadecimal representation of the JVM memory address at which the instance is stored. An example with our `Student` class may produce something like: `unl.cse.Student@272d7a10`.

We can *override* this functionality and change the behavior of the `toString()` method to return whatever we want. To do so, we simply define the function (with a signature that matches the `toString()` method in the `Object` class) in our class.

```
1 public String toString() {
2   return String.format("%s, %s (ID = %d); %.2f",
3           this.lastName,
4           this.firstName,
5           this.id,
6           this.gpa);
7 }
```

This example would return a `String` containing something similar to

`"Hamilton, Margaret (ID = 4321); 3.90"`

The `toString()` method is a very convenient way to print instances of your class.

Two other methods, the `equals()` and the `hashCode()` method are used extensively by other classes such as the Collections library. The `equals()` method:

`public boolean equals(Object obj)`

takes another object `obj` and returns `true` or `false` depending on whether or not the instance is "equal" to `obj`. Recall that identity is one of the defining characteristics of objects. This method is how Java achieves identity; this method allows you to define what "equality" means. By default, the behavior inherited from the `Object` class simply checks if the object, `this` and the passed object, `obj` are located at the same memory address, essentially `(this == obj)`.

However, *conceptually* we may want different behavior. For example, two `Student` objects may be the same if they have the same `id` value (it is, after all supposed to be a *unique* identifier). Alternatively, we may consider two objects to be the same if *every* member variable holds the same value. Even with only a four member variables, the logic can get quite complicated, especially if we have to account for `null` values. For

this reason, many IDEs provide functionality to automatically generate such methods. The following example was generated by an IDE.

```
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (obj == null) {
    return false;
  }
  if (!(obj instanceof Student)) {
    return false;
  }
  Student other = (Student) obj;
  if (firstName == null) {
    if (other.firstName != null) {
      return false;
    }
  } else if (!firstName.equals(other.firstName)) {
    return false;
  }
  if (Double.doubleToLongBits(gpa) !=
      Double.doubleToLongBits(other.gpa)) {
    return false;
  }
  if (lastName == null) {
    if (other.lastName != null) {
      return false;
    }
  } else if (!lastName.equals(other.lastName)) {
    return false;
  }
  if (nuid != other.nuid) {
    return false;
  }
  return true;
}
```

Related, the `hashCode()` method,

```
public int hashCode()
```

returns an *integer* representation of the object. As with the `equals()` method, the default behavior is to return a value associated with the memory address of the instance. In general, however, the behavior should be overridden to be based on the *entire* state of the object. A *hash* is simply a function that maps data to a "small" set of values. In this context, we are mapping object instances to integers so that they can be used in hash table-based data structures such as `HashSet` or `HashMap`. The `hashCode()` method is used to map an instance to an integer so that it can be used as an *index* in these data structures. Again, most IDEs will provide functionality to generate a good implementation for the `hashCode()` method (as the example below is).

How ever you design the `equals()` and `hashCode()` method, there is a requirement:

if two instances are equal (that is, `equals()` returns `true`) then they *must* have the same `hashCode()` value. This requirement is necessary to ensure that hash table-based data structures operate properly. It is *okay* if unequal objects have equal or unequal hash values. This rule only applies when the objects are equal.

```
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result + ((firstName == null) ? 0 :
                             firstName.hashCode());
  long temp;
  temp = Double.doubleToLongBits(gpa);
  result = prime * result + (int) (temp ^ (temp >>> 32));
  result = prime * result + ((lastName == null) ? 0 :
                             lastName.hashCode());
  result = prime * result + nuid;
  return result;
}
```

34.6. Composition

Another important concept when designing classes is *composition*. Composition is a mechanism by which an object is made up of other objects. One object is said to "own" an instance of another object. We've already seen this with our `Student` example: the `Student` class owns two instances of the `String` class.

To illustrate the importance of composition, we could extend the design of our `Student` class to include a date of birth. However, a date of birth is also made up of multiple pieces of data (a year, a month, a date, and maybe even a time and/or locale). We could design our own date/time class to model this, but its generally best to use what the language already provides. Java 8 introduced the `java.time` package in which there are many updated and improved classes for dealing with dates and times. The class `LocalDate` for example, could be used to model a date of birth:

```
private LocalDate dateOfBirth;
```

We can take this concept further and design our classes to own collections of other classes. For example, we could define a `Course` class and then update our `Student` class to own a collection of `Course` objects representing a student's class schedule (this type of collection ownership is sometimes referred to as *aggregation* rather than strict composition).

```
1 private Set<Course> schedule;
```

Both of these updates beg the question: who is responsible for instantiating the instances of `dateOfBirth` and the `schedule`? Should we force the "outside" user of our `Student` class to build the `LocalDate` instance and pass it to a constructor? Should we allow the outside code to simply provide us a date of birth as a string? Both of these design choices have advantages and disadvantages that have to be considered.

What about the course schedule? We could require that a user provide the constructor with a pre-computed `Set` of courses, but care must be taken. Consider the following (partial) example.

```
1 public Student(..., Set<Course> schedule) {
2   ...
3   this.schedule = schedule;
4 }
```

This is an example of a *shallow* copy: the instance's `schedule` variable is referencing the same collection as the parameter variable. If a change is made, say a course is added, via one of these references, then the change is effectively made for both. If we are going to design it this way, it would be better to make a *deep* copy of the set of courses:

```
1 public Student(..., Set<Course> schedule) {
2   ...
3   this.schedule = new HashSet<Course>(schedule);
4 }
```

This is the same pattern we described above: almost every data structure in the Java Collections library has a (deep) copy constructor.

Alternatively, we could make our design a bit more flexible by allowing the construction of a `Student` instance without having to provide a course schedule. Instead, we could add a method that allowed the outside code to add a course to the `schedule`. Something like the following.

```
1 public void addCourse(Course c) {
2   this.schedule.add(c);
3 }
```

This adds some flexibility to our object, but removes the immutability property. Design is always a balance and compromise between competing considerations.

34.7. Example

We present the full and completed `Student` class in Code Sample 34.1.

```java
package unl.cse;

public class Student {

  private String firstName;
  private String lastName;
  private int id;
  private double gpa;

  public Student(String firstName, String lastName, int id, double gpa) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.id = id;
    this.gpa = gpa;
  }

  public Student() {
    this(null, null, 0, 0.0);
  }

  public Student(String firstName, String lastName) {
    this(firstName, lastName, 0, 0.0);
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public int getId() {
    return id;
  }

  public double getGpa() {
    return gpa;
  }

  /**
   * Returns a formatted String of the Student's
   * name as Last, First.
   */
  public String getFormattedName() {
    return lastName + ", " + firstName;
```

```
47    }
48
49    /**
50     * Scales the GPA, which is assumed to be on a
51     * 4.0 scale to a percentage.
52     */
53    public double getGpaAsPercentage() {
54      return gpa / 4.0;
55    }
56
57    @Override
58    public String toString() {
59      return String.format("%s, %s (ID = %d); %.2f",
60               this.lastName,
61               this.firstName,
62               this.id,
63               this.gpa);
64    }
65
66    @Override
67    public int hashCode() {
68      final int prime = 31;
69      int result = 1;
70      result = prime * result + ((firstName == null) ? 0 : firstName.hashCode());
71      long temp;
72      temp = Double.doubleToLongBits(gpa);
73      result = prime * result + (int) (temp ^ (temp >>> 32));
74      result = prime * result + id;
75      result = prime * result + ((lastName == null) ? 0 : lastName.hashCode());
76      return result;
77    }
78
79
80    @Override
81    public boolean equals(Object obj) {
82      if (this == obj) {
83        return true;
84      }
85      if (obj == null) {
86        return false;
87      }
88      if (!(obj instanceof Student)) {
89        return false;
90      }
91      Student other = (Student) obj;
92      if (firstName == null) {
```

```
93        if (other.firstName != null) {
94          return false;
95        }
96      } else if (!firstName.equals(other.firstName)) {
97        return false;
98      }
99      if (Double.doubleToLongBits(gpa) != Double.doubleToLongBits(other.gpa)) {
100       return false;
101     }
102     if (id != other.id) {
103       return false;
104     }
105     if (lastName == null) {
106       if (other.lastName != null) {
107         return false;
108       }
109     } else if (!lastName.equals(other.lastName)) {
110       return false;
111     }
112     return true;
113   }
114
115
116
117 }
```

Code Sample 34.1.: The completed Java `Student` class.

35. Recursion

Java supports recursion with no special syntax necessary. However, as an object-oriented language, recursion is generally expensive and iterative or other non-recursive solutions are generally preferred. We present a few examples to demonstrate how to write recursive methods in Java. The first example of a recursive method we gave was the toy count down example. In Java it could be implemented as follows.

```
1 public static void countDown(int n) {
2   if(n == 0) {
3     System.out.println("Happy New Year!");
4   } else {
5     System.out.println(n);
6     countDown(n-1);
7   }
8 }
```

As another example that actually does something useful, consider the following recursive summation method that takes an array and an index variable. The recursion works as follows: if the index variable has reached the size of the array, it stops and returns zero (the base case). Otherwise, it makes a recursive call to `recSum()`, incrementing the index variable by 1. When the method returns, it adds its result to the i-th element in the array. To invoke this method we would call it with an initial value of 0 for the index variable: `recSum(arr, 0)`.

```
1 public static int recSum(int arr[], int i) {
2   if(i == arr.length) {
3     return 0;
4   } else {
5     return recSum(arr, i+1) + arr[i];
6   }
7 }
```

This example was not tail-recursive as the recursive call was not the final operation (the sum was the final operation). To make this method tail recursive, we can carry the summation through to each method call ensuring that the summation is done prior to

the recursive method call.

```
public static int recSumTail(int arr[], int i, int sum) {
  if(i == arr.length) {
    return sum;
  } else {
    return recSumTail(arr, i+1, sum + arr[i]);
  }
}
```

As another example, consider the following Java implementation of the naive recursive Fibonacci sequence. An additional condition has been included to check for "invalid" negative values of n for which an exception is thrown.

```
public static int fibonacci(int n) {
  if(n < 0) {
    throw new IllegalArgumentException("Undefined for n < 0");
  } else if(n <= 1) {
    return 1;
  } else {
    return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

Java is not a language that provides implicit memoization. Instead, we need to explicitly keep track of values using a table. In the following example, we use a `Map` data structure to store previously computed values.

```
public static int fibMemoization(int n, Map<Integer, Integer> m) {
  if(n < 0) {
    throw new IllegalArgumentException("Undefined for n < 0");
  } else if(n <= 1) {
    return 1;
  } else {
    Integer result = m.get(n);
    if(result == null) {
      Integer a = fibMemoization(n-1, m);
      Integer b = fibMemoization(n-2, m);
      result = a + b;
      m.put(n, result);
    }
```

```
    return result;
  }
}
```

Java provides an arbitrary precision data type, `BigInteger` that can be used to compute arbitrarily large integer values. Since Fibonacci numbers grow exponentially, using an `int` we could only represent up to to F_{45}. Using `BigInteger` we can support much larger values. An example:

```
public static BigInteger fibMem(int n, Map<Integer, BigInteger> m) {
  if(n < 0) {
    throw new IllegalArgumentException("Undefined for n < 0");
  } else if(n <= 1) {
    return BigInteger.ONE;
  } else {
    BigInteger result = m.get(n);
    if(result == null) {
      BigInteger a = fibMem(n-1, m);
      BigInteger b = fibMem(n-2, m);
      result = a.add(b);
      m.put(n, result);
    }
    return result;
  }
}
```

36. Searching & Sorting

Java provides several methods to search and sort arrays as well as `List`s of elements of *any* type. These methods are able to operate on collections of any type because there are several *overloaded* versions of these functions as well as versions that take a `Comparator` object that specifies *how* the elements should be ordered.

36.1. Comparators

Let's consider a "generic" Quick Sort algorithm as was presented in Algorithm 12.6. The algorithm itself specifies how to sort elements, but it doesn't specify how they are *ordered*. The difference is subtle but important. Quick Sort needs to know when two elements, a, b are in order, out of order, or equivalent in order to decide which partition each element goes in. However, it doesn't "know" anything about the elements a and b themselves. They could be numbers, they could be strings, they could be user-defined objects.
A sorting algorithm still needs to be able to determine the proper ordering in order to sort. In Java this is achieved by using a `Comparator` object, which is responsible for comparing two elements and determining their proper order. A `Comparator<T>` is an interface in Java that specifies a single method:
`public int compare(T a, T b)`

- A `Comparator<T>` is parameterized to operate on any type `T`. The parameterization of `T` is basically a variable for a type. When you create a `Comparator`, you specify an actual type just as we did with with `ArrayList<Integer>`. However, the implementation itself is generic and can operate on any type.
- The method takes two instances, `a` and `b` whose type matches the parameterized type `T`
- The method returns an integer indicating the relative ordering of the two elements:
  - It returns something negative, `< 0` if `a` comes before `b` (that is, $a < b$)
  - It returns zero if `a` and `b` are equal ($a = b$)
  - It returns something positive, `> 0` if `a` comes after `b` (that is, $a > b$)

There is no guarantee on the value's magnitude, it does *not* necessarily return -1 or $+1$; it just returns *something* negative or positive. We've previously seen this pattern when comparing strings and other wrapper classes. Each of the standard types implements

something similar, the `Comparable` interface, that specifies a `compareTo()` method with the same basic contract. Strings for example, are ordered in lexicographic ordering. Numeric types such as `Integer` and `Double` also have `compareTo()` methods that order elements in ascending order. Java refers to these built-in orderings as "natural" orderings.
For user-defined classes, however, we need to specify how to order them. We could use the `Comparable<T>` interface as the built-in wrapper classes have, but using a `Comparator` is more flexible. Making a class `Comparable` locks you into *one* "natural" ordering. If you wanted a different ordering, you would still have to use a `Comparator`. Even for the wrapper classes, if we wanted a descending order, we would need to use a `Comparator`.
As an example, let's write a `Comparator` that orders `Integer` types in ascending order, matching the "natural" ordering already defined.

```
Comparator<Integer> cmpInt = new Comparator<Integer>(){
  public int compare(Integer a, Integer b) {
    if(a < b) {
      return -1;
    } else if(a == b) {
      return 0;
    } else {
      return 1;
    }
  }
};
```

This is new syntax. When we create a `Comparator` in Java we are creating a new instance of a class. However, we didn't define a class. We didn't use `public class...` nor did we place it into a `.java` source file. Instead, this is an anonymous class declaration. The class we've created has no name; the `cmpInt` is the *variable's* name, but the class itself has no name, its "anonymous." This syntax allows us to define and instantiate a class *inline* without having to create a separate class source file. This is an advantage because we generally do not need multiple instances of a `Comparator`; they would all have the same functionality anyway. An anonymous class allows us to create ad-hoc classes with a one-time (or a single purpose) use.
Another issue with this method is that it may not be able to handle `null` values. When `a` and `b` get unboxed for the comparisons, if they are `null`, a `NullPointerException` will be thrown. We discuss how to deal with this below in Section 36.3.2.
Another issue with this `Comparator` is the second case where we use the equality operator `==` to compare values. With the less-than operator, the two integers get unboxed and their values are compared. However, when we use the `==` operator on object instances, it is comparing *memory addresses* in the JVM, not the actual values. We could solve

this issue by rearranging our cases so that the equality is our final case, avoiding the use of the equality operator. Even better, however, we can exploit the built-in natural ordering of the integers by using the `compareTo()` method.

```
1 Comparator<Integer> cmpInt = new Comparator<Integer>(){
2   public int compare(Integer a, Integer b) {
3     return a.compareTo(b);
4   }
5 };
```

What if we wanted to order integers in the opposite, descending order? We could simply write another `Comparator` that reversed the ordering:

```
1 Comparator<Integer> cmpIntDesc = new Comparator<Integer>(){
2   public int compare(Integer a, Integer b) {
3     return b.compareTo(a);
4   }
5 };
```

We might be tempted to instead *reuse* the original comparator and write line 3 above simply as
`return cmpInt(b, a);`
However, Java will not allow you to reference another "outside" variable like this inside a `Comparator` object. An anonymous class in Java is a sort-of closure; a function that has a scope in which variables are *bound*. In this case, the anonymous class has a reference to the `cmpInt` variable, but it is not necessarily "bound" as the variable could be reassigned outside the `Comparator`. If we want to do something like this, Java forces us to make the `cmpInt` variable `final` so that it cannot be changed.
To illustrate some more examples, consider the `Student` object we defined in Code Sample 34.1. The following Code Samples demonstrate various ways of ordering `Student` instances based on one or more of their member variable values.

```
1 /**
2  * This Comparator orders Student objects by
3  * last name/first name
4  */
5 Comparator<Student> byName = new Comparator<Student>() {
6   @Override
7   public int compare(Student a, Student b) {
8     if(a.getLastName().equals(b.getLastName())) {
```

```
      return a.getFirstName().compareTo(b.getFirstName());
    } else {
      return a.getLastName().compareTo(b.getLastName());
    }
  }
};
```

```
/**
 * This Comparator orders Student objects by
 * last name/first name in descending (Z-to-A) order
 */
Comparator<Student> byNameDesc = new Comparator<Student>() {
  @Override
  public int compare(Student a, Student b) {
    if(b.getLastName().equals(a.getLastName())) {
      return b.getFirstName().compareTo(a.getFirstName());
    } else {
      return b.getLastName().compareTo(a.getLastName());
    }
  }
};
```

```
/**
 * This Comparator orders Student objects by
 * id in ascending order
 */
Comparator<Student> byId = new Comparator<Student>() {
  @Override
  public int compare(Student a, Student b) {
          return a.getId().compareTo(b.getId());
  }
};
```

```
/**
 * This Comparator orders Student objects by
 * GPA in descending order
```

```
 */
Comparator<Student> byGpa = new Comparator<Student>() {
  @Override
  public int compare(Student a, Student b) {
          return b.getGpa().compareTo(a.getGpa());
  }
};
```

36.2. Searching & Sorting

We now turn our attention to the search and sorting methods provided by the JDK. Most of these methods are generic implementations that either sort an array or collection using the natural ordering or take a take a parameterized `Comparator<T>` to determine the ordering.

36.2.1. Searching

Linear Search

Java `Set` and `List` collections provide several linear search methods. Both have a `public boolean contains(Object o)` method that returns `true` or `false` depending on whether or not an object matching `o` is in the collection. The `List` collection has an additional `public int indexOf(Object o)` method that returns the index of the first matched element and -1 if no such element is found (`Set`s are unordered and have no indices, so this method would not apply). All of these functions are equality-based and they do not take a `Comparator` object. Instead, the elements are compared using the object's `equals()` (and possibly `hashCode()`) method. The first element `x` such that `x.equals(o)` returns `true` is the element that is determined to match. For this reason, it is important to override both of these methods when designing objects (see Section 36.3.3).

Binary Search

The `Arrays` and `Collections` classes provide many variations on methods that implement binary search. All of these methods assume that the array or `List` being searched are sorted appropriately (you cannot use binary search on a `Set` as it is an unordered collection).
The `Arrays` class provides several `binarySearch()` methods, one for each primitive type as well as variations that allow you to search within a specified range of elements. The most generally useful version is as follows:

```
public static <T> int binarySearch(T[] a, T key, Comparator<T> c)
```

That is, it takes an array of elements as well as key and a `Comparator` all of the same type `T`. It returns an integer representing the index at which it finds the first matching element (there is no guarantee that the first element in the sorted list is returned). If no match is found, then the method returns something negative.[1] Another version has the same behavior but can be called without a `Comparator`, relying instead on the natural ordering of elements. For this variation, the type of elements *must* implement the `Comparable` interface.

The `Collections` class provides a similar method,

```
public static <T> int binarySearch(List<T> list, T key, Comparator<T> c)
```

The only difference is that the method takes a `List` instead of an array. Otherwise, the behavior is the same. We present several examples in Code Sample 36.1.

36.2.2. Sorting

As with searching, the `Arrays` and `Collections` classes provide parameterized sorting methods to sort arrays and `List`s. In Java 6 and prior, the implementation was a hybrid merge/insertion sort. Java 7 switched the implementation to Tim Sort. Here too, there are several variations that rely on the natural ordering or allow you to sort a subpart of the collection.

The `Arrays` provides the following method, which sorts arrays of a particular type with a `Comparator` for that type.

```
public static <T> void sort(T[] a, Comparator<T> c)
```

Likewise, `Collections` provides the following method.

```
public static <T> void sort(List<T> list, Comparator<T> c)
```

Several examples of the usage of these methods are presented in Code Sample 36.2.

36.3. Other Considerations

36.3.1. Sorted Collections

The Java Collections framework also provides several sorted data structures. Though an ordered collection such as a `List` can be sorted calling the `Collections.sort()` method, a sorted data structure *maintains* an ordering. As you add elements to the data structure, they are inserted in the appropriate spot. Such data structures also prevent you from arbitrarily inserting elements in any order.

Java defines the `SortedSet<T>` interface for sorted collections with a `TreeSet<T>` implementation.[2] You can construct a `TreeSet<T>` by providing it a `Comparator<T>`

[1] It actually returns a negative value corresponding to the *insertion point* at which the element would be if it had existed.

[2] As the name implies, the implementation is a balanced binary search tree, in particular a red-black tree.

```
ArrayList<Student> roster = ...

Student castroKey = null;
int castroIndex;

//create a "key" that will match according to the
// Student.equals() method
castroKey = new Student("Starlin", "Castro", 131313, 3.95);
castroIndex = roster.indexOf(castroKey);
System.out.println("at index " + castroIndex + ": " +
  roster.get(castroIndex));

//create a key with only the necessary fields to match
/  the comparator
castroKey = new Student("Starlin", "Castro", 0, 0.0);
//sort the list according to the comparator
Collections.sort(roster, byName);
castroIndex = Collections.binarySearch(roster, castroKey, byName);
System.out.println("at index " + castroIndex + ": " +
  roster.get(castroIndex));

//create a key with only the necessary fields to match
//  the comparator
castroKey = new Student(null, null, 131313, 0.0);
//sort the list according to the comparator
Collections.sort(roster, byId);
castroIndex = Collections.binarySearch(roster, castroKey, byId);
System.out.println("at index " + castroIndex + ": " +
  roster.get(castroIndex));
```

Code Sample 36.1.: Java Search Examples

to use to maintain the ordering (alternatively, you may omit the `Comparator` and the `TreeSet` will use the natural ordering of elements).

```
Comparator<Integer> cmpIntDesc = new Comparator<Integer>() { ... };
SortedSet<Integer> sortedInts = new TreeSet<Integer>(cmpIntDesc);
sortedInts.add(10);
sortedInts.add(30);
sortedInts.add(20);
```

```
List<Student> roster = ...
Student rosterArr[] = ...
Comparator byName = ...
Comparator byGPA = ...

//sort by name:
Collections.sort(roster, byName);
Arrays.sort(rosterArr, byName);

//sort by GPA:
Collections.sort(roster, byGPA);
Arrays.sort(rosterArr, byGPA);
```

Code Sample 36.2.: Using Java Collection's Sort Method

```
//sortedInts has elements 30, 20, 10 in descending order

SortedSet<Student> sortedStudents = new TreeSet<Student>(byName);
//add students, they will be sorted by name
for(Student s : sortedStudents) {
  System.out.println(s);
}
```

When using a `SortedSet` it is important that the `Comparator` properly orders all elements. A `SortedSet` is still a `Set`: it does not allow duplicate elements. If the `Comparator` you use returns zero for any element that you attempt to insert, it will not be inserted. To demonstrate how this might fail, consider our `byName` `Comparator` for `Student` objects. Suppose two students have the same first name and last name, "John Doe" and "John Doe," but have different IDs (as they are different people). The `Comparator` would return 0 for these two objects because they have the same first/last name, even though they are distinct people. Only one of these objects could exist in the `SortedSet`. To solve this problem, it is important to define `Comparator`s that "break ties" appropriately. In this case, we would want to modify the `Comparator` to return a comparison of IDs if the first *and* last name are the same.

36.3.2. Handling `null` values

When sorting collections or arrays of objects, we may need to consider the possibility of uninitialized `null` objects. Some collections allow `null` element(s) while others do

not. How we handle these is a design decision. We could ignore it, in which case such elements would likely result in a `NullPointerException` and expect the user to prevent or handle such instances. This may be the preferable choice in most instances, in fact. Alternatively, we could handle `null` objects in the design of our `Comparator`. Code Sample 36.3 presents a `Comparator` for our `Student` class that orders `null` instances first.

```
Comparator<Student> byNameWithNulls = new Comparator<Student>() {
  @Override
  public int compare(Student a, Student b) {
    if(a == null && b == null) {
      return 0;
    } else if(a == null && b != null) {
      return -1;
    } else if(a != null && b == null) {
      return 1;
    } else {
      if(a.getLastName().equals(b.getLastName())) {
        return a.getFirstName().compareTo(b.getFirstName());
      } else {
        return a.getLastName().compareTo(b.getLastName());
      }
    }
  }
};
```

Code Sample 36.3.: Handling Null Values in Java Comparators

This solution only handles `null` `Student` values not `null` values within the `Student` object. If the getters used in the `Comparator` return `null` values, then we could still have `NullPointerException`s.

36.3.3. Importance of `equals()` and `hashCode()` Methods

Recall that in Chapter 34 we briefly discussed the importance of overloading the `equals()` and `hashCode()` methods of our objects. We reiterate this in the context of searching. Recall that the `Set` and `List` collections provide linear search methods that do not require the use of a `Comparator`. This is because the collections use the object's `equals()` to determine when a particular instance has been found. In the case of hash-based data structures such as `HashSet`, the `hashCode()` method is

also important. To illustrate the importance of these methods, consider the following code.

```
Student a = new Student("Joe", "Smith", 1234, 3.5);
Student b = new Student("Joe", "Smith", 1234, 3.5);

Set<Student> s = new HashSet<Student>();
s.add(a);
s.add(b);
```

If we *do not* override the `equals()` and `hashCode()` methods, at the end of this code snippet, the set will contain *both* of the `Student` instances even though they are *conceptually* the same object (all of their member variables will have the same values). However, if we *do* override the `equals()` and `hashCode()` methods as described in Section 34.5, then the code snippet above would result in the `Set` only having one object (the first one added). This is because during the attempt to add the second instance, the `equals()` method would have returned `true` when compared to the first instance and the duplicate element would not have been added to the `Set` (as `Set`s do not allow duplicates).
Also recall that we saw some of the advantages of immutable objects. Again, we point out another advantage. Suppose that we properly override the `equals()` and the `hashCode()` methods in our `Student` object. Further, suppose that we make our `Student` class mutable: allowing a user to change the ID via a setter. Consider the following code snippet.

```
Student a = new Student("Joe", "Smith", 1234, 3.5);
Student b = new Student("Joe", "Smith", 5679, 3.5);

Set<Student> s = new HashSet<Student>();
s.add(a);
s.add(b);
//s now contains both students
b.setId(1234);
```

When we instantiate the two `Student` instances, they are distinct as their IDs are different. So when we add them to the set, their `equals()` method returns `false` and they are both added to the `Set`. However, when we change the ID using the setter on the last line, both objects are now identical, violating the no-duplicates property of the `Set`. This is not a failing of the `Set` class, just one of the many consequences of designing mutable objects.

36.3.4. Java 8: Lambda Expressions

Java 8 introduced a lot of functional-style syntax, including lambda expressions. Lambda expressions are essentially anonymous functions that can be passed around to other methods or objects. One use for lambda expressions is if we want to sort a `List` with respect to one data field, we need not build a full `Comparator`. Instead we can use the following syntax.

```
List<Student> roster = ...

roster.sort((a, b) -> a.getLastName().compareTo(b.getLastName()));
```

The new syntax is the use of the arrow operator as a lambda expression. In this case, it maps a pair, `(a, b)` to the value of the expression `a.getLastName().compareTo(b.getLastName())` (which takes advantage of the fact that strings implement the `Comparable` interface). With respect to `Comparator`s, we can use the following syntax to build a more complex ordering.

```
Comparator<Student> c =
  (a, b) -> a.getLastName().compareTo(b.getLastName());
c = c.thenComparing( (a, b) ->
                      a.getFirstName().compareTo(b.getFirstName()));
c = c.thenComparing( (a, b) -> a.getGpa().compareTo(b.getGpa()));

//pass the comparator to the sort method:
roster.sort(c);
```

We can make this even more terse using method references, another new feature in Java 8.

```
//using getters as key extractors
myList.sort(
  Comparator.comparing(Student::getLastName)
            .thenComparing(Student::getFirstName)
            .thenComparing(Student::getGpa));
```

There are several other convenient methods provided by the updated `Comparator` interface. For example, the `reversed()` member method returns a new `Comparator` that defines the reversed order. The static methods, `nullsFirst()` and `nullsLast()`

can be used to modify a `Comparator` to order `null` values.

Part III.

The PHP Programming Language

37. Basics

In the mid-1990s the World Wide Web was in its infancy but becoming more and more popular. For the most part, web pages contained static content: articles and text that was "just-there." Web pages were far from the fully interactive and dynamic applications that they've become. Rasmus Lerdorf had a home page containing his resume and he wanted to track how many visitors were coming to his page. With purely static pages, this was not possible. So, in 1994 he developed PHP/FI which stood for Personal Home Page tools and Forms Interpreter. This was a series of binary tools written in C that operated through a web server's Common Gateway Interface. When a user visited a webpage, instead of just retrieving static content, a script was invoked: it could do a number of operations and send back HTML formatted as a response. This made web pages much more dynamic. By serving it through a script, a counter could be maintained that tracked how many people had visited the site. Lerdorf eventually released his source code in 1995 and it became widely used.

Today, PHP is used in a substantial number of pages on the web and is used in many Content Management System (CMS) applications such as Drupal and WordPress. Because of its history, much of the syntax and aspects of PHP (now referred to as "PHP: Hypertext Preprocessor") are influenced or inspired by C. In fact, many of the standard functions available in the language are directly from the C language.

PHP is a scripting language and is not compiled. Instead, PHP code is *interpreted* by a PHP interpreter. Though there are several interpreters available, the de facto PHP interpreter is the Zend Engine, a free and open source project that is widely available on many platforms, operating systems, and web servers. Because it was originally intended to serve web pages, PHP code can be *interleaved* with static HTML tags. This allows PHP code to be embedded in web pages and dynamically interpreted/rendered when a user requests a webpage through a web browser. Though rendering web pages is its primary purpose, PHP can be used as a general scripting language from the command line.

PHP is a dirty, ugly language held together by duct tape, cracked asbestos-filled spackle and sadness (`http://phpsadness.com/`). It is the cockroach of languages: it is invasive and it is a survivor. It is a hydra; rewrite one PHP app and two more pop up. It is much maligned and a source of ridicule. It is the source of hundreds of bad practices, thousands of bad programmers, millions of bugs and an infinite abyss of security holes and vulnerabilities. It is a language that was born in the wild and raised in the darkness by two schizophrenic monkeys. But its got character; and in this world, that's enough.

```php
<?php

  printf("Hello World\n");

?>
```

Code Sample 37.1.: Hello World Program in PHP

```php
<html>
  <head>
    <title>Hello World PHP Page</title>
  </head>
  <body>
    <h1>A Simple PHP Script</h1>

    <?php printf("<p>Hello World</p>");  ?>

  </body>
</html>
```

Code Sample 37.2.: Hello World Program in PHP with HTML

37.1. Getting Started: Hello World

The hallmark of an introduction to a new programming language is the *Hello World!* program. It consists of a simple program whose only purpose is to print the message "Hello World!" to the user. The simplicity of the program allows the focus to be on the basic syntax of the language. It is also typically used to ensure that your development environment, interpretrer, runtime environment, etc. are functioning properly with a minimal example. A basic Hello World! program in PHP can be found in Code Sample 37.1. A version in which the PHP code is interleaved in HTML is presented in Code Sample 37.2.

We will not focus on any particular development environment, code editor, or any particular operating system, interpreter, or ancillary standards in our presentation. However, as a first step, you should be able to write and run the above program on the environment you intend to use for the rest of this book. This may require that you download and install a basic PHP interpreter/development environment (such as a standard LAMP, WAMP or MAMP technology stack) or a full IDE (such as Eclipse, PhpStorm, etc.).

37.2. Basic Elements

Using the Hello World! program as a starting point, we will examine the basic elements of the PHP language.

37.2.1. Basic Syntax Rules

PHP has adopted many aspects of the C programming language (the interpreter itself is written in C). However, there are some major aspects in which it differs from C. The major aspects and syntactic elements of PHP include the following.

- PHP is an interpreted language: it is not compiled. Instead it is interpreted through an interpreter that parses commands in a script file line-by-line. Any syntax errors, therefore, become runtime errors.

- PHP is *dynamically typed*: you do *not* declare variables or specify a variable's type. Instead, when you assign a variable a value, the variable's type dynamically changes to accommodate the assigned value. Variable names *always* begin with a dollar sign, `$`.

- Strings and characters are essentially the same thing (characters are strings of length 1). Strings and characters can be delimited by *either* single quotes *or* double quotes. `"this is a string"`, `'this is also a string'`; `'A'` and `"a"` are both single character strings.

- Executable statements are terminated by a semicolon, `;`

- Code blocks are defined by using opening and closing curly brackets, `{ ... }`. Moreover, code blocks can be *nested*: code blocks can be defined within other code blocks.

- A complete list of reserved and keywords can be found in the PHP Manual: `http://php.net/manual/en/reserved.php` and `http://php.net/manual/en/reserved.keywords.php`

PHP also supports aspects of OOP including classes and inheritance. There is also no memory management in PHP: it has automated garbage collection.
Though not a syntactic requirement, the proper use of whitespace is important for good, readable code. Code inside code blocks is indented at the same indentation. Nested code blocks are indented further. Think of a typical table of contents or the outline of a formal paper or essay. Sections and subsections or points and subpoints all follow proper indentation with elements at the same level at the same indentation. This convention is used to organize code and make it more readable.

37.2.2. PHP Tags

PHP code can be interleaved with static HTML or text. Because of this, we need a way to indicate what should be interpreted as PHP and what should be treated as static text. We can do this using PHP *tags*: the opening tag is `<?php` and the closing tag is `?>`. Anything placed between these tags will be interpreted as PHP code which must adhere to the syntax rules of the language.
A file can contain multiple opening/closing PHP tags to allow you to interleave multiple sections of HTML or text. When an interpreter runs a PHP script, it will start processing the script file. Whenever it sees the opening PHP tag, it begins to execute the commands and stops executing commands when it sees the closing tag. The text outside the PHP tags is treated as raw data. The interpreter includes it as part of its output without modifying or processing it.

37.2.3. Libraries

PHP has many built-in functions that you can use. These standard libraries are loaded and available to use without any special command to import or include them. Full documentation on each of these functions is maintained in the PHP manual, available online at http://php.net/. The manual's web pages also contain many curated comments from PHP developers which contain further explanations, tips & tricks, suggestions, and sample code to provide further assistance.
There are many useful functions that we'll mention as we progress through each topic. One especially useful collection of functions is the math library. This library is directly adapted from C's standard math library so all the function names are the same. It provides many common mathematical functions such as the square root and natural logarithm. Table 37.1 highlights several of these functions; full documentation can be found in the PHP manual (http://php.net/manual/en/ref.math.php). To use these, you simply "call" them by providing input and assigning the output to a variable.

```
1 $x = 1.5;
2 $y = sqrt($x); //y now has the value √x = √1.5
3 $z = sin($x); //z now has the value sin(x) = sin(1.5)
```

In both of the function calls above, the value of the variable `$x` is "passed" to the math function which computes and "returns" the result which then gets assigned to another variable.

37.2.4. Comments

Comments can be written in PHP code either as a single line using two forward slashes, `//comment` or as a multiline comment using a combination of forward slash and asterisk: `/* comment */`. With a single line comment, everything on the line *after* the forward

Function	Description
`abs($x)`	Absolute value, $\lvert x \rvert$
`ceil($x)`	Ceiling function, $\lceil 46.3 \rceil = 47.0$
`floor($x)`	Floor function, $\lfloor 46.3 \rfloor = 46.0$
`cos($x)`	Cosine function[a]
`sin($x)`	Sine function[a]
`tan($x)`	Tangent function[a]
`exp($x)`	Exponential function, e^x, $e = 2.71828\ldots$
`log($x)`	Natural logarithm, $\ln(x)$[b]
`log10($x)`	Logarithm base 10, $\log_{10}(x)$[b]
`pow($x,$y)`	The power function, computes x^y[c]
`sqrt($x)`	Square root function[b]

Table 37.1.: Several functions defined in the PHP math library. [a]All trigonometric functions assume input is in *radians*, **not** degrees. [b]Input is assumed to be positive, $x > 0$. [c]Alternatively, PHP supports exponentiation by using `x ** y`.

slashes is ignored. With a multiline comment, everything in between the forward slash/asterisk is ignored. Comments are ultimately ignored by the interpreter. Consider the following example.

```
//this is a single line comment
$x = 10;  //this is also a single line comment, but after some code

/*
  This is a comment that can
  span multiple lines to format the comment
  message more clearly
*/
$y = 3.14;
```

Most code editors and IDEs will present comments in a special color or font to distinguish them from the rest of the code (just as our example above does). Failure to close a multiline comment will likely result in a fatal interpreter error but with color-coded comments its easy to see the mistake visually.

37.2.5. Entry Point & Command Line Arguments

Every PHP script begins executing at the top of the script file and proceed in a linear, sequential manner top-to-bottom. In addition, you can provide a PHP script with inputs when you run it from the command line. These *command line arguments* are stored in two variables, `$argc` and `$argv`. The first variable, `$argc` is an integer that indicates the *number* of arguments provided *including* the name of the script file being executed. The second, `$argv` is an array (see Section 42) of strings consisting of the command line arguments. To access them, you can *index* them starting at zero, the first being `$argv[0]`, the second `$argv[1]`, the last one would be `$argv[$argc-1]`. The first one is always the name of the script file being run. The remaining are the command line arguments provided by the user when the script is executed. We'll see several examples later.

37.3. Variables

PHP is a dynamically typed language. As a consequence, you do not declare variables before you start using them. If you need to store a value into a variable, you simply name the variable and use an assignment operator to assign the value. Since you do not declare variables, you also do not specify a variable's type. If you assign a string value to a variable, its type becomes a string. If you assign an integer to a variable, its type becomes an integer. If you reassign the value of a variable to a value with a different type, the variable's type also changes.

Internally PHP supports several different types: Booleans, integers, floating point numbers, strings, arrays, and objects. The way that integers are represented may be platform dependent, but are usually 32-bit signed two's complement integers, able to represent integers between $-2,147,483,648$ and 2,147,483,647. Floating point numbers are also platform dependent, but are usually 64-bit double precision numbers defined by the IEEE 754 standard, providing about 16 digits of precision. Strings and single characters are the same thing in PHP. Strings are represented as sequences of characters from the extended ASCII text table (see Table 2.4) which includes all characters in the range 0–255. PHP does not have native Unicode support for international characters.

37.3.1. Using Variables

To use a variable in PHP, you simply need to assign a value to a named variable identifier and the variable is in scope. Variable names *always* begin with a single dollar sign, `$`. The assignment operator is a single equal sign, `=` and is a right-to-left assignment. The variable that we wish to assign the value to appears on the left-hand-side while the value (literal, variable or expression) is on the right-hand-size. For example:

```
1 $numUnits = 42;
2 $costPerUnit = 32.79;
3 $firstInitial = "C";
```

Each assignment also implicitly changes the variable's type. Each of the variables above becomes an integer, floating point number, and string respectively. Assignment statements are terminated by a semicolon like most executable statements in PHP. The identifier rules are fairly standard: a variable's name can consist of lowercase and uppercase alphabetic characters, numbers, and underscores. You can also use the extended ASCII character set in variable names but it is not recommended (umlauts and other diacritics can easily be confused). Variable names are *case sensitive*. As previously mentioned, variable names must *always* begin with a dollar sign, `$`. Stylistically, we adopt the modern lower camel casing naming convention for variables in our code.

If you do not assign a value to a variable, that variable remains undefined or "unset." Undefined variables are treated as `null` in PHP. The concept of "null" refers to uninitialized, undefined, empty, missing, or meaningless values. In PHP the keyword `null` is used which is case insensitive (`null`, `Null` and `NULL` are all the same), but for consistency, we'll use `null`. When `null` values are used in arithmetic expressions, `null` is treated as zero. So, `(10 + null)` is equal to 10. When `null` is used in the context of strings, it is treated as an empty string and ignored. When used in a Boolean expression or conditional, `null` is treated as `false`.

PHP also allows you to define *constants*: values that cannot be changed once set. To define a constant, you invoke a function named `define` and providing a name and value.

```php
define("PI", 3.14159);
define("INSTITUTION", "University of Nebraska-Lincoln");
define("COST_PER_UNIT", 2.50);
```

Constant names are case sensitive. By convention, we use uppercase underscore casing. An attempt to redefine a constant value will raise a script warning, but will ultimately have no effect. When referring to constants later in the script, you use the constant's name. You do not treat it as a string, nor do you use a dollar sign. For example: `$area = $r * $r * PI;`

37.4. Operators

PHP supports the standard arithmetic operators for addition, subtraction, multiplication, and division using `+`, `-`, `*`, and `/` respectively. Each of these operators is a binary operator that acts on two operands which can either be literals, variables or expressions and follow the usual rules of arithmetic when it comes to order of precedence (multiplication and division before addition and subtraction).

```php
$a = 10, $b = 20, $c = 30;
$d = $a + 5;
$d = $a + $b;
$d = $a - $b;
$d = $a + $b * $c;
$d = $a * $b; //d becomes a floating point number with a value .5

$x = 1.5, $y = 3.4, $z = 10.5;
$w = $x + 5.0;
$w = $x + $y;
$w = $x - $y;
$w = $x + $y * $z;
$w = $x * $y;
$w = $x / $y;

//mixing integers and floating point numbers is no problem
$w = $a + $x;
```

PHP also supports the integer remainder operator using the `%` symbol. This operator gives the remainder of the result of dividing two integers. Examples:

```
$x = 10 % 5; //x is 0
$x = 10 % 3; //x is 1
$x = 29 % 5; //x is 4
```

37.4.1. Type Juggling

The expectations of an arithmetic expression involving two variables that are either integers or floating point numbers are straightforward. We expect the sum/product/etc. as a result. However, since PHP is dynamically typed, a variable involved in an arithmetic expression could be anything, including a Boolean, object, array, or string. When a Boolean is involved in an arithmetic expression, `true` is treated as 1 and `false` is treated as zero. If an array is involved in an expression, it is usually a fatal error.[1] Non-null objects are treated as 1 and `null` is treated as 0.
However, when string values are involved in an arithmetic expression, PHP does something called *type juggling*. When juggled, an attempt is made to convert a string variable into a numeric value by parsing it. The parsing goes over each numeric character, converting the value to a numeric type (either an integer or floating point number). The first type a non-numeric character is encountered, the parsing stops and the value parsed *so far* is the value used in the expression.
Consider the examples in Code Sample 37.3. In the first segment, `$a` is type juggled to the value 10, then added to `5`, resulting in 15. In the second example, `$a` represents a floating point number, and is converted to 3.14, the result of adding to 5 is thus 8.14. In the third example, the string does not contain any numerical values. In this case, the parsing stops at the first character and what has been parsed *so far* is zero! Finally, in the last example, the first two characters in `$a` are numeric, so the parsing ends at the third character, and what has been parsed *so far* is 10.
Relying on type juggling to convert values can be ugly and error prone. You can write much more intentional code by using the several conversion functions provided by PHP. For example:

```
$a = intval("10");
$b = floatval("3.14");
$c = intval("ten"); //c has the value zero
```

In all three of the examples above, the strings are converted just as they are when type juggled. However, the variables are *guaranteed* to have the type indicated (integer or floating point number).

[1] PHP does allow you to "add" two arrays together which results in their union.

```php
$a = "10";
$b = 5 + $a; //b = 15

$a = "3.14";
$b = 5 + $a; //b = 8.14

$a = "ten";
$b = 5 + $a; //b = 5

//partial conversions also occur:
$a = "10ten";
$b = 5 + $a; //b = 15
```

Code Sample 37.3.: Type Juggling in PHP

There are several utility functions that can be used to help determine the type of variable. The function `is_numeric($x)` returns `true` if `$x` is a numeric (integer or floating point number) or represents a *pure* numeric string. The functions `is_int($x)` and `is_float($x)` each return `true` or `false` depending on whether or not `$x` is of that type.

```php
$a = 10;
$b = "10";
$c = 3.14;
$d = "3.14";
$e = "hello";
$f = "10foo";

is_numeric($a); //true
is_numeric($b); //true
is_numeric($c); //true
is_numeric($d); //true
is_numeric($e); //false
is_numeric($f); //false

is_int($a); //true
is_int($b); //false
is_int($c); //false
is_int($d); //false
is_int($e); //false
```

Value of `$var`	`isset($var)`	`empty($var)`	`is_null($var)`
`42`	bool(true)	bool(false)	bool(false)
`""` (an empty string)	bool(true)	bool(true)	bool(false)
`" "` (space)	bool(true)	bool(false)	bool(false)
`false`	bool(true)	bool(true)	bool(false)
`true`	bool(true)	bool(false)	bool(false)
`array()` (an empty array)	bool(true)	bool(true)	bool(false)
`null`	bool(false)	bool(true)	bool(true)
`"0"` (0 as a string)	bool(true)	bool(true)	bool(false)
`0` (0 as an integer)	bool(true)	bool(true)	bool(false)
`0.0` (0 as a float)	bool(true)	bool(true)	bool(false)
`var $var;` (declared with no value)	bool(false)	bool(true)	bool(true)
`NULL` byte (`"\0"`)	bool(true)	bool(false)	bool(false)

Table 37.2.: Results for various variable values

```
20 is_int($f); //false
21
22 is_float($a); //false
23 is_float($b); //false
24 is_float($c); //true
25 is_float($d); //false
26 is_float($e); //false
27 is_float($f); //false
```

A more general way to determine the type of a variable is to use the function `gettype($x)` which returns a string representation of the type of the variable `$x`. The string returned by this function is one of the following depending on the type of `$x`: `"boolean"`, `"integer"`, `"double"`, `"string"`, `"array"`, `"object"`, `"resource"`, `"NULL"`, or `"unknown type"`.
Other checker functions allow you to determine if a variable has been set, if its null, "empty" etc. For example, `is_null($x)` returns `true` if `$x` is not set or is set, but has been set to `null`. The function `isset($x)` returns `true` only if `$x` is set and it is not `null`. The function `empty($x)` returns `true` if `$x` represents an empty entity: an empty string, `false`, an empty array, `null`, `"0"`, 0, or an unset variable. Several examples are presented in Table 37.2.

37.4.2. String Concatenation

Strings in PHP can be concatenated (combined) in several different ways. One way you can combine strings is by using the concatenation operator. In PHP the string concatenation operator is a single period.

```
$s = "Hello";
$t = "World!";
$msg = $s . " " . $t; //msg contains "Hello World!"
```

Another way you can combine strings is by placing variables directly inside a string. The variables inside the string are replaced with the variable's values. Example:

```
$x = 13;
$name = "Starlin";
$msg = "Hello, $name, your number is $x";
//msg contains the string "Hello, Starlin, your number is 13"
```

37.5. Basic I/O

Recall the main purpose of PHP is as a scripting language to serve dynamic webpages. However, it does support input and output from the standard input/output. There are several ways to print output to the standard output. The keywords `print` and `echo` (which are aliases of each other) allow you to print any variable or string. PHP also has a `printf()` function to allow formatted output. Some examples:

```
$a = 10;
$b = 3.14;

print $a;
print "The value of a is $a\n";

echo $a;
echo "The value of a is $a\n";

printf("The value of a is %d, and b is %f\n", $a, $b);
```

There are also several ways to perform standard input, but the easiest is to use `fgets`

(short for **f**ile **get** **s**tring) using the keyword `STDIN` (Standard Input). This function will return, as a string, everything the user enters up to *and including* the enter key (interpreted as the endline character, `\n`). To remove the endline character, you can use another function, `trim` which removes leading and trailing whitespace from a string. A full example:

```
1 //prompt the user to enter input
2 printf("Please enter a number: ");
3 $a = fgets(STDIN);
4 $a = trim($a);
```

Alternatively, lines 3–4 could be combined into one: `$a = trim(fgets(STDIN));` The call to `fgets` *waits* (referred to as "blocking") for the user to enter input. The user is free to start typing. When the user is done, they hit the enter key at which point the program resumes and reads the input from the standard input buffer, and returns it as a string value which we assign to the variable `$a`.
The standard input is unstructured. The user is free to type whatever they want. If we prompt the user for a number but they just start mashing the keyboard giving non-numerical input, we may get incorrect results. We can use the conversion functions mentioned above to attempt to properly convert the values. However, this only guarantees that the resulting variable is of the type we want (integer or floating point value for example). The standard input is not a good mechanism for reading input, but it provides a good starting point to develop a few simple programs.

37.6. Examples

37.6.1. Converting Units

Let's write a program that prompts the user to enter a temperature in degrees Fahrenheit and convert it to degrees Celsius using the formula

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

We begin with the basic script shell with the opening and closing PHP tags and some comments documenting the purpose of our script.

```
1 <?php
2
3 /**
4  * This program converts Fahrenheit temperatures to
5  * Celsius
```

```
 */

//TODO: implement this

?>
```

It is common for programmers to use a comment along with a `TODO` note to themselves as a reminder of things that they still need to do with the program. Let's first outline the basic steps that our program will go through:

1. We'll first prompt the user for input, asking them for a temperature in Fahrenheit
2. Next we'll read the user's input and use a conversion function to ensure the input is a floating point number
3. Once we have the input, we can calculate the degrees Celsius by using the formula above
4. Lastly, we will want to print the result to the user to inform them of the value

Sometimes it is helpful to write an outline of such a program directly in the code using comments to provide a step-by-step process. For example:

```
<?php

/**
 * This program converts Fahrenheit temperatures to
 * Celsius
 */

//TODO: implement this

//1. Prompt the user for input in Fahrenheit
//2. Read the Fahrenheit value from the standard input
//3. Compute the degrees Celsius
//4. Print the result to the user

?>
```

As we read each step it becomes apparent that we'll need a couple of variables: one to hold the Fahrenheit (input) value and one for the Celsius (output) value. We'll want to ensure that these are floating point numbers which we can do by making some explicit conversion. We use a `printf()` statement in the first step to prompt the user for input.

`printf("Please enter degrees in Fahrenheit: ");`

In the second step, we'll use the standard input to read the `$fahrenheit` variable value from the user. Recall that we can use `fgets` to read from the standard input, but may have to `trim` the trailing whitespace.

`$fahrenheit = trim(fgets(STDIN));`

If we want to ensure that the variable `$fahrenheit` is a floating point value, we can use `floatval()`:

`$fahrenheit = floatval($fahrenheit);`

We can now compute `$celsius` using the formula provided:

`$celsius = ($fahrenheit - 32) * (5 / 9);`

Finally, we use `printf()` again to output the result to the user:

`printf("%f Fahrenheit is %f Celsius\n", $fahrenheit, $celsius);`

The full program can be found in Code Sample 37.4.

```
1  <?php
2
3  /**
4   * This program converts Fahrenheit temperatures to
5   * Celsius
6   */
7
8  //1. Prompt the user for input in Fahrenheit
9  printf("Please enter degrees in Fahrenheit: ");
10
11 //2. Read the Fahrenheit value from the standard input
12 $fahrenheit = trim(fgets(STDIN));
13 $fahrenheit = floatval($fahrenheit);
14
15 //3. Compute the degrees Celsius
16 $celsius = ($fahrenheit - 32) * (5/9);
17
18 //4. Print the result to the user
19 printf("%f Fahrenheit is %f Celsius\n", $fahrenheit, $celsius);
20
21 ?>
```

Code Sample 37.4.: Fahrenheit-to-Celsius Conversion Program in PHP

37.6.2. Computing Quadratic Roots

Some programs require the user to enter multiple inputs. The prompt-input process can be repeated. In this example, consider asking the user for the coefficients, a, b, c to a quadratic polynomial,

$$ax^2 + bx + c$$

and computing its roots using the quadratic formula,

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

As before, we can create a basic program with PHP tags and start filling in the details. In particular, we'll need to prompt for the input a, then read it in; then prompt for b, read it in and repeat for c. Thus, we have

```
1 printf("Please enter a: ");
2 $a = floatval(trim(fgets(STDIN)));
3 printf("Please enter b: ");
4 $b = floatval(trim(fgets(STDIN)));
5 printf("Please enter c: ");
6 $c = floatval(trim(fgets(STDIN)));
```

We need to take care that we correctly adapt the formula so it accurately reflects the order of operations. We also need to use the math library's square root function.

```
1 $root1 = (-$b + sqrt($b*$b - 4*$a*$c) ) / (2*$a);
2 $root2 = (-$b - sqrt($b*$b - 4*$a*$c) ) / (2*$a);
```

Finally, we print the output using `printf()`. The full program can be found in Code Sample 37.5.
This program was interactive. As an alternative, we could have read all three of the inputs as command line arguments, taking care to convert them to floating point numbers. Lines 8–13 in the program could have been changed to

```
1 $a = floatval($argv[1]);
2 $b = floatval($argv[2]);
3 $c = floatval($argv[3]);
```

Finally, think about the possible input a user could provide that may cause problems for this program. For example:

```php
<?php

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */

printf("Please enter a: ");
$a = floatval(trim(fgets(STDIN)));
printf("Please enter b: ");
$b = floatval(trim(fgets(STDIN)));
printf("Please enter c: ");
$c = floatval(trim(fgets(STDIN)));

$root1 = (-$b + sqrt($b*$b - 4*$a*$c) ) / (2*$a);
$root2 = (-$b - sqrt($b*$b - 4*$a*$c) ) / (2*$a);

printf("The roots of %fx^2 + %fx + %f are: \n", $a, $b, $c);
printf("  root1 = %f\n", $root1);
printf("  root2 = %f\n", $root2);

?>
```

Code Sample 37.5.: Quadratic Roots Program in PHP

- What if the user entered zero for a?
- What if the user entered some combination such that $b^2 < 4ac$?
- What if the user entered non-numeric values?
- For the command line argument version, what if the user provided less than three arguments? Or more?

How might we prevent the consequences of such bad input? How might we handle the event that a user enters bad input and how do we communicate these errors to the user? To begin to resolve these issues, we'll need conditionals.

38. Conditionals

PHP supports the basic if, if-else, and if-else-if conditional structures as well as switch statements. Logical statements are built using the standard logical operators for numeric comparisons as well as logical operators such as negations, AND, and OR.

38.1. Logical Operators

PHP has a built-in Boolean type and supports the keywords `true` and `false`. However, *any* variable can be treated as a Boolean if used in a logical expression. Depending on the variable, it could evaluate to *true* or *false*! For example, an empty string, `""`, `null`, or a numeric value of zero, `0` are all considered *false*. A non-empty string, a non-zero numeric value, or a non-empty array all evaluate to *true*. It is best to avoid these issues by writing clean code that uses clear, explicit statements.
Because PHP is dynamically typed, comparison operators work differently depending on how they are used. First, let's consider the four basic inequality operators, `<`, `<=`, `>`, and `>=`. When used to compare numeric types to numeric types, these operators work as expected and the value of the numbers are compared.

```
1  $a = 10;
2  $b = 20;
3  $c = 20;
4
5  $r = ($a < $b);  //true
6  $r = ($a <= $b); //false
7  $r = ($b <= $c); //true
8  $r = ($a > $b);  //false
9  $r = ($a >= $b); //false
10 $r = ($b >= $c); //true
```

When these operators are used to compare strings to strings, the strings are compared lexicographically according to the standard ASCII text table. Some examples follow, but it is better to use a function (in particular `strcmp` see Chapter 43) to do string comparisons.

```php
$s = "aardvark";
$t = "zebra";

$r = ($s < $t);  //true
$r = ($s <= $t); //true
$r = ($s >= $t); //false
$r = ($s > $t);   //false
```

However, when these operators are used to compare strings to numeric types, the strings are converted to numbers using the same type juggling that happens when strings are mixed with arithmetic operators. In the following example, `$b` gets converted to a numeric type when compared to `$a` which give the results indicated in the comments.

```php
$a = 10;
$b = "10";

$r = ($a <= $b);  //true
$r = ($a < $b);   //false
$r = ($a >= $b);  //true
$r = ($a > $b);   //false
```

With the equality operators, `==` and `!=`, something similar happens. When the types of the two operands match, the expected comparison is made: when numbers are compared to numbers their values are compared; when strings are compared to strings, their content is compared (case sensitively). However, when the types are different they are type juggled and strings are converted to numbers for the purpose of comparison. Thus, a comparison like `(10 == "10")` ends up being *true*! The operators are `==` and `!=` are referred to as *loose* comparison operators because of this.
What if we want to ensure that we're comparing apples to apples? To rectify this, PHP offers another set of comparison operators, *strict* comparison operators, `===` and `!==` (each has an extra equals sign, `=`). These operators will make a comparison *without* type juggling either operand first. Now a similar comparison, `(10 === "10")` ends up evaluating to *false*. The operator `===` will only evaluate to *true* if the both the operands' type *and* value are the same.

```php
$a = 10;
$b = "10";

$r = ($a == $b);  //true
$r = ($a != $b);  //false
```

	Operator(s)	Associativity	Notes
Highest	`++`, `--`	left-to-right	increment operators
	`-`, `!`	right-to-left	unary negation operator, logical not
	`*`, `/`, `%`	left-to-right	
	`+`, `-`	left-to-right	addition, subtraction
	`<`, `<=`, `>`, `>=`	left-to-right	comparison
	`==`, `!=`, `===`, `!==`	left-to-right	equality, inequality
	`&&`	left-to-right	logical AND
	`||`	left-to-right	logical OR
Lowest	`=`, `+=`, `-=`, `*=`, `/=`	right-to-left	assignment and compound assignment operators

Table 38.1.: Operator Order of Precedence in PHP. Operators on the same level have equivalent order and are performed in the associative order specified.

```
6 $r = ($a === $b); //false
7 $r = ($a !== $b); //true
```

The three basic logical operators, not `!`, and `&&`, and or `||` are also supported in PHP.

38.1.1. Order of Precedence

At this point it is worth summarizing the order of precedence of all the operators that we've seen so far including assignment, arithmetic, comparison, and logical. Since all of these operators could be used in one statement, for example,
`($b*$b < 4*$a*$c || $a === 0 || $argc != 4)`
it is important to understand the order in which each one gets evaluated. Table 38.1 summarizes the order of precedence for the operators seen so far. This is not an exhaustive list of PHP operators.

38.2. If, If-Else, If-Else-If Statements

Conditional statements in PHP utilize the keywords `if`, `else`, and `else if`. Conditions are placed inside parentheses immediately after the `if` and `else if` keywords. Examples of all three can be found in Code Sample 38.1.
Observe that the statement, `if($x < 10)` does not have a semicolon at the end. This is because it is a conditional statement that determines the flow of control and *not* an executable statement. Therefore, no semicolon is used. Suppose we made a mistake and *did* include a semicolon:

```php
//example of an if statement:
if($x < 10) {
  printf("x is less than 10\n");
}

//example of an if-else statement:
if($x < 10) {
  printf("x is less than 10\n");
} else {
  printf("x is 10 or more \n");
}

//example of an if-else-if statement:
if($x < 10) {
  printf("x is less than 10\n");
} else if($x === 10) {
  printf("x is equal to ten\n");
} else {
  printf("x is greater than 10\n");
}
```

Code Sample 38.1.: Examples of Conditional Statements in PHP

```
$x = 15;
if($x < 10); {
  printf("x is less than 10\n");
}
```

This PHP code will run without error or warning. However, it will end up printing `x is less than 10`, even though $x = 15$! Recall that a conditional statement *binds* to the executable statement or code block *immediately* following it. In this case, we've provided an *empty* executable statement ended by the semicolon. The code is essentially equivalent to

```
$x = 15;
if($x < 10) {
}
printf("x is less than 10\n");
```

This is obviously not what we wanted. The semicolon was bound to the empty executable statement and the code block containing the print statement immediately followed, but was *not* bound to the conditional statement which is why the print statement executed regardless of the value of x.
Another convention that we've used in our code is where we have placed the curly brackets. If a conditional statement is bound to only one statement, the curly brackets are not necessary. However, it is best practice to include them even if they are not necessary and we'll follow this convention. Second, the opening curly bracket is on the same line as the conditional statement while the closing curly bracket is indented to the same level as the start of the conditional statement. Moreover, the code inside the code block is indented. If there were more statements in the block, they would have all been at the same indentation level.

38.3. Examples

38.3.1. Computing a Logarithm

The logarithm of x is the exponent that some *base* must be raised to get x. The most common logarithm is the natural logarithm, $\ln(x)$ which is base $e = 2.71828\ldots$. But logarithms can be in any base $b > 1$.[1] What if we wanted to compute $\log_2(x)$? Or $\log_\pi(x)$? Let's write a program that will prompt the user for a number x and a base b and computes $\log_b(x)$. Arbitrary bases can be computed using the change of base

[1] Bases can also be $0 < b < 1$, but we'll restrict our attention to increasing functions only.

formula:

$$\log_b(x) = \frac{\log_a(x)}{\log_a(b)}$$

If we can compute *some* base a, then we can compute any base b. Fortunately we have such a solution. Recall that the standard library provides a function to compute the natural logarithm, `log()`). This is one of the fundamentals of problems solving: if a solution already exists, use it. In this case, a solution exists for a different, but similar problem (computing the natural logarithm), but we can *adapt* the solution using the change of base formula. In particular, if we have variables `b` (base) and `x`, we can compute $\log_b(x)$ using
`log(x) / log(b)`
We have a problem similar to the examples in the previous section. The user could enter invalid values such as $b = -10$ or $x = -2.54$ (logarithms are undefined for non-positive values in any base). We want to ensure that $b > 1$ and $x > 0$. With conditionals, we can now do this. Once we have read in the input from the user we can make a check for good input using an `if` statement.

```
1 if($x <= 0 || $b <= 1) {
2   printf("Error: bad input!\n");
3   exit(1);
4 }
```

This code has something new: `exit(1)`. The `exit()` function immediately terminates the script regardless of the rest of the code that may remain. The argument passed to `exit` is an integer that represents an *error code*. The convention is that zero indicates "no error" while non-zero values indicate some error. This is a simple way of performing *error handling*: if the user provides bad input, we inform them and quit the program, forcing them to run it again and provide good input. By prematurely terminating the program we avoid any illegal operation that would give a bad result.
Alternatively, we could have split the conditions into two statements and given a more descriptive error message. We use this design in the full program which can be found in Code Sample 38.2. The program also takes the input as command line arguments. Now that we have conditionals, we can check that the correct number of arguments was provided by the user and quit in the event that they don't provide the correct number.

38.3.2. Life & Taxes

Let's adapt the conditional statements we developed in Section 3.6.4 into a full PHP script. The first thing we need to do is establish the variables we'll need and read them in from the user. At the same time we can check for bad input (negative values) for both the inputs.

```
//prompt for income from the user
printf("Please enter your Adjusted Gross Income: ");

$income = floatval(trim(fgets(STDIN)));

//prompt for children
printf("How many children do you have? ");
$numChildren = intval(trim(fgets(STDIN)));

if($income < 0 || $numChildren < 0) {
  printf("Invalid inputs");
  exit(1);
}
```

Next, we can code a series of if-else-if statements for the income range. By placing the ranges in increasing order, we only need to check the upper bounds just as in the original example.

```
if($income <= 18150) {
  $baseTax = $income * .10;
} else if($income <= 73800) {
  $baseTax = 1815 + ($income  -18150) * .15;
} else if($income <= 148850) {
...
} else {
  $baseTax = 127962.50 + ($income - 457600) * .396;
}
```

Next we compute the child tax credit, taking care that it does not exceed $3,000. A conditional based on the number of children suffices as at this point in the program we already know it is zero or greater.

```
if($numChildren <= 3) {
  $credit = $numChildren * 1000;
} else {
  $credit = 3000;
}
```

Finally, we need to ensure that the credit does not exceed the total tax liability (the credit is non-refundable, so if the credit is greater, the tax should only be zero, not

negative).

```
if($baseTax - $credit >= 0) {
  $totalTax = $baseTax - $credit;
} else {
  $totalTax = 0;
}
```

The full program is presented in Code Sample 38.3.

38.3.3. Quadratic Roots Revisited

Let's return to the quadratic roots program we previously designed that uses the quadratic equation to compute the roots of a quadratic polynomial by reading coefficients a, b, c in from the user. One of the problems we had previously identified is if the user enters "bad" input: if $a = 0$, we would end up dividing by zero; if $b^2 - 4ac < 0$ then we would have complex roots. With conditionals, we can now check for these issues and exit with an error message.

Another potential case we might want to handle differently is when there is only one distinct root ($b^2 - 4ac = 0$). In that case, the quadratic formula simplifies to $\frac{-b}{2a}$ and we can print a different, more specific message to the user. The full program can be found in Code Sample 38.4.

```php
<?php

/**
 * This program computes the logarithm base b (b > 1)
 * of a given number x > 0
 */

if($argc != 3) {
  printf("Usage: %s b x \n", $argv[0]);
  exit(1);
}

$b = floatval($argv[1]);
$x = floatval($argv[2]);

if($x <= 0) {
  printf("Error: x must be greater than zero\n");
  exit(1);
}
if($b <= 1) {
  printf("Error: base must be greater than one\n");
  exit(1);
}

$result = log($x) / log($b);
printf("log_(%f)(%f) = %f\n", $b, $x, $result);

?>
```

Code Sample 38.2.: Logarithm Calculator Program in C

```php
<?php
//prompt for income from the user
printf("Please enter your Adjusted Gross Income: ");

$income = floatval(trim(fgets(STDIN)));

//prompt for children
printf("How many children do you have? ");
$numChildren = intval(trim(fgets(STDIN)));

if($income < 0 || $numChildren < 0) {
  printf("Invalid inputs");
  exit(1);
}

if($income <= 18150) {
  $baseTax = $income * .10;
} else if($income <= 73800) {
  $baseTax = 1815 + ($income  -18150) * .15;
} else if($income <= 148850) {
  $baseTax = 10162.50 + ($income - 73800) * .25;
} else if($income <= 225850) {
  $baseTax = 28925.00 + ($income - 148850) * .28;
} else if($income <= 405100) {
  $baseTax = 50765.00 + ($income - 225850) * .33;
} else if($income <= 457600) {
  $baseTax = 109587.50 + ($income - 405100) * .35;
} else {
  $baseTax = 127962.50 + ($income - 457600) * .396;
}

if($numChildren <= 3) {
  $credit = $numChildren * 1000;
} else {
  $credit = 3000;
}

if($baseTax - $credit >= 0) {
  $totalTax = $baseTax - $credit;
} else {
  $totalTax = 0;
}

printf("AGI:           $%10.2f\n", $income);
printf("Tax:           $%10.2f\n", $baseTax);
printf("Credit:        $%10.2f\n", $credit);
printf("Tax Liability: $%10.2f\n", $totalTax);

?>
```

Code Sample 38.3.: Tax Program in PHP

```php
<?php

/**
 * This program computes the roots to a quadratic equation
 * using the quadratic formula.
 */

if($argc != 4) {
  printf("Usage: %s a b c\n", $argv[0]);
  exit(1);
}

$a = floatval($argv[1]);
$b = floatval($argv[2]);
$c = floatval($argv[3]);

if($a === 0) {
  printf("Error: a cannot be zero\n");
  exit(1);
} else if($b*$b < 4*$a*$c) {
  printf("Error: cannot handle complex roots\n");
  exit(1);
} else if($b*$b === 4*$a*$c) {
  $root1 = -$b / (2*$a);
  printf("Only one distinct root: %f\n", $root1);
} else {
  $root1 = (-$b + sqrt($b*$b - 4*$a*$c) ) / (2*$a);
  $root2 = (-$b - sqrt($b*$b - 4*$a*$c) ) / (2*$a);

  printf("The roots of %fx^2 + %fx + %f are: \n", $a, $b, $c);
  printf("  root1 = %f\n", $root1);
  printf("  root2 = %f\n", $root2);
}

?>
```

Code Sample 38.4.: Quadratic Roots Program in PHP With Error Checking

39. Loops

PHP supports while loops, for loops, and do-while loops using the keywords `while`, `for`, and `do` (along with another `while`). Continuation conditions for loops are enclosed in parentheses, `(...)` and blocks of code associated with the loop are enclosed in curly brackets.

39.1. While Loops

Code Sample 39.1 contains an example of a basic while loop in PHP. As with conditional statements, our code style places the opening curly bracket on the same line as the `while` keyword and continuation condition. The inner block of code is also indented and all lines in the block are indented to the same level.
In addition, the continuation condition does *not* contain a semicolon since it is not an executable statement. Just as with an if-statement, if we *had* placed a semicolon it would have led to unintended results. Consider the following:

```
1 while($i <= 10); {
2   //perform some action
3   $i++; //iteration
4 }
```

A similar problem occurs. The `while` keyword and continuation condition bind to the next executable statement or code block. As a consequence of the semicolon, the executable statement that gets bound to the while loop is *empty*. What happens is

```
1 $i = 1; //Initialization
2 while($i <= 10) {  //continuation condition
3   //perform some action
4   $i++; //iteration
5 }
```

Code Sample 39.1.: While Loop in PHP

```php
$i = 1;
$flag = true;
while($flag) {
  //perform some action
  $i++; //iteration
  if($i>10) {
    $flag = false;
  }
}
```

Code Sample 39.2.: Flag-controlled While Loop in PHP

even worse: the program will enter an infinite loop. To see this, the code is essentially equivalent to the following:

```php
while($i <= 10) {
}
{
  //perform some action
  $i++; //iteration
}
```

In the while loop, we never increment the counter variable `$i`, the loop does nothing, and so the computation will continue on forever! It is valid PHP and will run, but obviously won't work as intended. Avoid this problem by using proper syntax.
Another common use for a while loop is a flag-controlled loop in which we use a Boolean flag rather than an expression to determine if a loop should continue or not. Since PHP has built-in Boolean types, we can use a variable along with the keywords `true` and `false`. An example can be found in Code Sample 39.2.

39.2. For Loops

For loops in PHP use the familiar syntax of placing the initialization, continuation condition, and iteration on the same line as the `for` keyword. An example can be found in Code Sample 39.3.
Semicolons are placed at the end of the initialization and continuation condition, but *not* the iteration statement. With while loops, the opening curly bracket is placed on the same line as the `for` keyword. Code within the loop body is indented, all at the same indentation level.

```
$i;
for($i=1; $i<=10; $i++) {
  //perform some action
}
```

Code Sample 39.3.: For Loop in PHP

```
$i;
do {
  //perform some action
  $i++;
} while($i <= 10);
```

Code Sample 39.4.: Do-While Loop in PHP

39.3. Do-While Loops

PHP also supports do-while loops. Recall that the difference between a while loop and a do-while loop is when the continuation condition is checked. For a while loop it is *prior* to the beginning of the loop body and in a do-while loop it is at the *end* of the loop. This means that a do-while always executes *at least once*. An example can be found in Code Sample 39.4.
The opening curly bracket is again on the same line as the keyword `do`. The `while` keyword and continuation condition are on the same line as the closing curly bracket. In a slight departure from previous syntax, a semicolon *does* appear at the end of the continuation condition even though it is not an executable statement.

39.4. Foreach Loops

Finally, PHP supports foreach loops using the keyword `foreach`. Some of this will be a preview of Section 42 where we discuss arrays in PHP,[1] In short you can iterate over the elements of an array as follows.

```
$arr = array(1.41, 2.71, 3.14);
foreach($arr as $x) {
  //x now holds the "current" element in arr
```

[1]Actually, PHP supports *associative* arrays, which are not the same thing as traditional arrays.

```
}
```

In the `foreach` syntax we specify the array we want to iterate over, `$arr` and use the keyword `as`. The last element in the statement is the variable name that we want to use within the loop. This should be read as "`foreach` element `$x` in the array `$arr`...". Inside the loop, the variable `$x` will be automatically updated on each iteration to the next element in `$arr`.

39.5. Examples

39.5.1. Normalizing a Number

Let's revisit the example from Section 4.1.1 in which we *normalize* a number by continually dividing it by 10 until it is less than 10. The code in Code Sample 39.5 specifically refers to the value 32145.234 but would work equally well with any value of `$x`.

```
$x = 32145.234;
$k = 0;
while($x > 10) {
  $x = $x / 10;
  $k++;
}
```

Code Sample 39.5.: Normalizing a Number with a While Loop in PHP

39.5.2. Summation

Let's revisit the example from Section 4.2.1 in which we computed the sum of integers $1 + 2 + \cdots + 10$. The code is presented in Code Sample 39.6
We could easily generalize this code. Instead of computing a sum up to a particular number, we could have written it to sum up to another variable `$n`, in which case the for loop would instead look like the following.

```
for($i=1; $i<=$n; $i++) {
  $sum += $i;
}
```

```php
$sum = 0;
for($i=1; $i<=10; $i++) {
  $sum += $i;
}
```

Code Sample 39.6.: Summation of Numbers using a For Loop in PHP

39.5.3. Nested Loops

Recall that you can write loops within loops. The inner loop will execute fully *for each* iteration of the outer loop. An example of two nested of loops in PHP can be found in Code Sample 39.7.

```php
$n = 10;
$m = 20;
for($i=0; $i<$n; $i++) {
  for($j=0; $j<$m; $j++) {
    printf("(i, j) = (%d, %d)\n", $i, $j);
  }
}
```

Code Sample 39.7.: Nested For Loops in PHP

The inner loop executes for $j = 0, 1, 2, \ldots, 19 < m = 20$ for a total of 20 iterations. However, it executes 20 times *for each* iteration of the outer loop. Since the outer loop executes for $i = 0, 1, 2, \ldots, 9 < n = 10$, the total number of times the `printf()` statement executes is $10 \times 20 = 200$. In this example, the sequence $(0, 0), (0, 1), (0, 2), \ldots, (0, 19), (1, 0), \ldots, (9, 19)$ will be printed.

39.5.4. Paying the Piper

Let's adapt the solution for the loan amortization schedule we developed in Section 4.7.3. First, we'll read the principle, terms, and interest as command line inputs. Adapting the formula for the monthly payment and using the standard math library's `pow()` function, we get

```php
$monthlyPayment = ($monthlyInterestRate * $principle) /
          (1 - pow( (1 + $monthlyInterestRate), -$n));
```

The monthly payment may come out to be a fraction of a cent, say $43.871. For accuracy, we need to ensure that all of the figures for currency are rounded to the nearest cent. The standard math library does have a `round()` function, but it only rounds to the nearest whole number, not the nearest 100th.
However, we can *adapt* the "off-the-shelf" solution to fit our needs. If we take the number, multiply it by 100, we get 4387.1 which we can now round to the nearest whole number, giving us 4387. We can then divide by 100 to get a number that has been rounded to the nearest 100th!

```
$monthlyPayment = round($monthlyPayment * 100) / 100;
```

We can use the same trick to round the monthly interest payment and any other number expected to be whole cents. To output our numbers, we use `printf()` and take care to align our columns to make make it look nice. To finish our adaptation, we handle the final month separately to account for an over/under payment due to rounding. The full solution can be found in Code Sample 39.8.

```php
<?php
  if($argc != 4) {
    printf("Usage: %s principle apr terms\n", $argv[0]);
    exit(1);
  }

  $principle = floatval($argv[1]);
  $apr = floatval($argv[2]);
  $n = intval($argv[3]);

  $balance = $principle;
  $monthlyInterestRate = $apr / 12;

  //monthly payment
  $monthlyPayment = ($monthlyInterestRate * $principle) /
         (1 - pow( (1 + $monthlyInterestRate), -$n));
  //round to the nearest cent
  $monthlyPayment = round($monthlyPayment * 100) / 100;

  printf("Principle: $%.2f\n", $principle);
  printf("APR: %.4f%%\n", $apr * 100.0);
  printf("Months: %d\n", $n);
  printf("Monthly Payment: $%.2f\n", $monthlyPayment);

  //for the first n-1 payments in a loop:
  for($i=1; $i<$n; $i++) {
    //  compute the monthly interest, rounded:
    $monthlyInterest =
      round( ($balance * $monthlyInterestRate) * 100) / 100;
    //  compute the monthly principle payment
    $monthlyPrinciplePayment = $monthlyPayment - $monthlyInterest;
    //  update the balance
    $balance = $balance - $monthlyPrinciplePayment;
    //  print i, monthly interest, monthly principle, new balance
    printf("%d\t$%10.2f  $%10.2f  $%10.2f\n", $i, $monthlyInterest,
                          $monthlyPrinciplePayment, $balance);
  }

  //handle the last month and last payment separately
  $lastInterest = round( ($balance * $monthlyInterestRate) * 100) / 100;
  $lastPayment = $balance + $lastInterest;

  printf("Last payment = $%.2f\n", $lastPayment);
?>
```

Code Sample 39.8.: Loan Amortization Program in PHP

40. Functions

Functions are essential in PHP programming. PHP provides a large library of standard functions to perform basic input/output, math, and many other functions. PHP also provides the ability to define and use your own functions.
PHP does not support function overloading, so when you define a function and give it a name, that name cannot be in conflict with any other function name in the standard library or any other code that you might use. Careful thought should go into the design and naming of your functions.
PHP supports both call by value and call by reference. As of PHP 5.6, vararg functions are also supported (though earlier versions supported some vararg-like functions such as `printf()`). However, we will not go into detail here. Finally, another feature of PHP is that function parameters are all optional. You may invoke a function with a subset of the parameters. Depending on your PHP setup, the interpreter may issue a warning that a parameter was omitted. However, PHP allows you to define default values for optional parameters.

40.1. Defining & Using Functions

In general, you can define functions anywhere in your PHP script or codebase. They can even appear *after* code that invokes them because PHP *hoists* the function definitions by doing two passes of the script. However, it is good style to include function definitions at the top of your script or in a separate PHP file for organization.

40.1.1. Declaring Functions

In PHP, to declare a function you use the keyword `function`. Because PHP is dynamically typed, a function can return *any* type. Therefore, you do not declare the return type (just as you do not declare a variable's type). After the `function` keyword you provide an identifier and parameters as the function signature. Immediately following, you provide the function body enclosed with opening/closing curly brackets.
Typically, the documentation for functions is included with its declaration. Consider the following examples. In these examples we use a commenting style known as "doc comments." This style was originally developed for Java but has since been adopted by many other languages.

```php
/**
 * Computes the sum of the two arguments.
 */
function sum($a, $b) {
  return ($a + $b);
}

/**
 * Computes the Euclidean distance between the 2-D points,
 * (x1,y1) and (x2,y2).
 */
function getDistance($x1, $y1, $x2, $y2) {
  $xDiff = ($x1-$x2);
  $yDiff = ($y1-$y2);
  return sqrt( $xDiff * $xDiff + $yDiff * $yDiff);
}

/**
 * Computes a monthly payment for a loan with the given
 * principle at the given APR (annual percentage rate) which
 * is to be repaid over the given number of terms (usually
 * months).
 */
function getMonthlyPayment($principle, $apr, $terms) {
  $rate = ($apr / 12.0);
  $payment = ($principle * $rate) / (1-pow(1+$rate, -$terms));
  return $payment;
}
```

Function identifiers (names) follow similar naming rules as variables, however they do not begin with a dollar sign. Function names must begin with an alphabetic character and may contain alphanumeric characters as well as underscores. However, using modern coding conventions we usually name functions using lower camel casing. Another quirk of PHP is that function names are *case insensitive*. Though we declared a function, `getDistance()` above, it could be invoked with either `getdistance()`, `GETDISTANCE` or any other combination of uppercase/lowercase letters. However, good code will use consistent naming and your function calls *should* match their declaration.
The keyword `return` is used to specify the value that is returned to the calling function. Whatever value you end up returning is the return type of the function. Since you do not specify variable or return types, functions are usually referred to as returning a "mixed" type. You could design a function that, given one set of inputs, returns a number while another set of inputs returns a string. You can use the syntax `return;` to return no

value (you do not use the keyword `void`). In practice, however, the function ends up returning `null` when doing this.

40.1.2. Organizing Functions

There are many coding standards that guide how PHP code should be organized. We'll only discuss a simple mechanism here. One way to organize functions is to collect functions with similar functionality into separate PHP source files.
Suppose the functions above are in a PHP source file named `utils.php`. We could include them in another source file (our "main" source file) using an `include_once` function invocation. An example:

```
<?php

include_once("utils.php");

//we can now use the functions in utils.php:
$p = getMonthlyPayment(1000, 0.05, 12);

?>
```

The `include_once` function loads and evaluates the given PHP source file at the point in the code in which it is invoked. The "once" in the function refers to the fact that if the source file was already included in the script/code before, it will not be included a second time. This allows you to include the same source file in multiple source files without a conflict.

40.1.3. Calling Functions

The syntax for calling a function is to simply provide the function name followed by parentheses containing values or variables to pass to the function. Some examples:

```
$a = 10, $b = 20;
$c = sum($a, $b); //c contains the value 30

//invoke a function with literal values:
$dist = getDistance(0.0, 0.0, 10.0, 20.0);

//invoke a function with a combination:
$p = 1500.0;
$r = 0.05;
$monthlyPayment = getMonthlyPayment($p, $r, 60);
```

40.1.4. Passing By Reference

By default, all types (including numbers, strings, etc.) are passed by value. To be able to pass arguments by reference, we need to use slightly different syntax when defining our functions.
To specify that a parameter is to be passed by reference, we place an ampersand, `&` in front of it in the function signature.[1] No other syntax is necessary. When you call the function, PHP automatically takes care of the referencing/dereferencing for you.

```
1  <?php
2
3  function swap($a, $b) {
4    $t = $a;
5    $a = $b;
6    $b = $t;
7  }
8
9  function swapByRef(&$a, &$b) {
10   $t = $a;
11   $a = $b;
12   $b = $t;
13 }
14
15 $x = 10;
16 $y = 20;
17
18 printf("x = %d, y = %d\n", $x, $y);
19 swap($x, $y);
20 printf("x = %d, y = %d\n", $x, $y);
21 swapByRef($x, $y);
22 printf("x = %d, y = %d\n", $x, $y);
23
24 ?>
```

The first function, `swap()` passes both variables by value. Swapping the values only affects the *copies* of the parameters. The original variables `$x` and `$y` will be unaffected. In the second function, `swapByRef()`, both variables are passed by reference as there are ampersands in front of them. Swapping them inside the function swaps the original

[1] Those familiar with pointers in C will note that this is the exact *opposite* of the C operator.

variables. The output to this code is as follows.

```
x = 10, y = 20
x = 10, y = 20
x = 20, y = 10
```

Observe that when we invoked the function, `swapByRef($x, $y);` we used the same syntax as the pass by value version. The only syntax needed to pass by reference is in the function signature itself.

40.1.5. Optional & Default Parameters

Parameters in PHP functions are optional. You can invoke a function without providing a subset of parameters. However, if a parameter is not provided, PHP will treat the parameter as `null`. If we invoked the `getDistance()` function with only two parameters:
`getDistance(10.0, 20.0);`
then inside the function, `$x1` and `$y1` would take on the values 10 and 20, but `$x2` and `$y2` would be `null`. When used in the distance formula calculations, both would be treated as zero. If we had invoked the function with *no* arguments,
`getDistance();`
then all four parameters would be treated as `null` (and thus zero in the calculations). PHP also allows you to define alternative default values for function parameters. Consider the following example.

```
1 function getMonthlyPayment($principle, $apr = 0.05, $terms = 60) {
2   $rate = ($apr / 12);
3   $mp = (($principle * $rate) / (1-pow(1+$rate, -$terms)));
4   return round($mp * 100) / 100;
5 }
```

In this example, the second and third parameter have been given default values of 0.05 and 60 respectively. If a call to this function omits these parameters, they are not treated as `null`, but take on these defaults instead.

```
1 $x = getMonthlyPayment(10000, 0.07, 72);
2 print $x."\n"; //170.49
3
4 //terms will be 60
5 $x = getMonthlyPayment(10000, 0.07);
```

```
6  print $x."\n"; //198.01
7
8  //apr will be 0.05, terms will be 60
9  $x = getMonthlyPayment(10000);
10 print $x."\n"; //188.71
11
12 //balance will be null (0), apr will be 0.05, terms will be 60
13 $x = getMonthlyPayment();
14 print $x."\n"; //0 but also a warning
```

It would not be possible to invoke `getMonthlyPayment()` by omitting only the second argument. Providing n arguments will match the first n parameters. Thus, in your function design you should place any optional/default parameters *last*.

40.1.6. Function Pointers

Functions are just pieces of code that reside somewhere in memory just as variables do. Since we can pass variables by reference, it also makes sense that we would do the same with functions.
In PHP, functions are first-class citizens[2] meaning that you can assign a function to a variable just as you would a numeric value. For example, you can do the following.

```
1 $func = swapByRef;
2
3 $func($x, $y);
```

In the example above, we assigned the function `swapByRef()` to the variable `$func` by using its identifier. The variable essentially holds a reference to the `swapByRef()` function. Since it refers to a function, we can also invoke the function using the variable as in the last line. This allows you to treat functions as callbacks to other functions. We will revisit this concept in Chapter 47.

40.2. Examples

40.2.1. Generalized Rounding

Recall that the standard math library provides a `round()` function that rounds a number to the nearest whole number. We've had need to round to cents as well. We now have

[2]Some would use a much more restrictive definition of first-class and would *not* consider them first-class citizens in this sense

the ability to write a function to do this for us. Before we do, however, let's think more generally. What if we wanted to round to the nearest tenth? Or what if we wanted to round to the nearest 10s or 100s place? Let's write a general purpose rounding function that allows us to specify *which* decimal place to round to.
The most natural input would be to specify the place using an integer exponent. That is, if we wanted to round to the nearest tenth, then we would pass it -1 as $0.1 = 10^{-1}$, -2 if we wanted to round to the nearest 100th, etc. In the other direction, passing in 0 would correspond to the usual round function, 1 to the nearest 10s spot, and so on.
Moreover, we could demonstrate good code reuse (as well as procedural abstraction) by *scaling* the input value and reusing the functionality already provided in the math library's `round()` function. We could further define a `roundToCents()` function that used our generalized round function. Consider the following.

```
1  <?php
2
3  /**
4   * Rounds to the nearest digit specified by the place
5   * argument.  In particular to the (10^place)-th digit
6   */
7  function roundToPlace($x, $place) {
8    $scale = pow(10, -$place);
9    $rounded = round(x * $scale) / $scale;
10   return $rounded;
11 }
12
13 /**
14  * Rounds to the nearest cent
15  */
16 function roundToCents($x) {
17   return roundToPlace($x, -2);
18 }
19
20 ?>
```

We could place these functions into a file named `round.php` and include them in another PHP source file.

40.2.2. Quadratic Roots

Another advantage of passing variables by reference is that we can "return" multiple values with one function call. Functions are limited in that they can only return at most one value. But if we pass multiple parameters by reference, the function can manipulate

the contents of them, thereby communicating (though not strictly returning) multiple values.
Consider again the problem of computing the roots of a quadratic equation,

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

using the quadratic formula,

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

Since there are two roots, we may have to write two functions, one for the "plus" root and one for the "minus" root both of which take the coefficients, a, b, c as arguments. However, if we wrote a single function that took the coefficients as parameters by value as well as two other parameters by reference, we could compute *both* root values, one in each of the by-reference variables.

```
function quadraticRoots($a, $b, $c, &$root1, &$root2) {
  $discriminant = sqrt($b*$b - 4*$a*$c);
  $root1 = (-$b + $discriminant) / (2*$a);
  $root2 = (-$b - $discriminant) / (2*$a);
  return;
}
```

By using pass by reference variables, we avoid multiple functions. Recall that there could be several "bad" inputs to this function. The roots could be complex values, the coefficient a could be zero, etc. In the next chapter, we examine how we can *handle* these errors.

41. Error Handling & Exceptions

Modern versions of PHP support error handling through the use of exceptions. PHP has several different predefined types of exceptions and also allows you to define your own exception types by creating new classes that *inherit* from the generic `Exception` class. PHP uses the standard `try-catch-finally` control structure to handle exceptions and allows you to `throw` your own exceptions.

41.1. Throwing Exceptions

Though PHP defines several different types of exceptions, we'll only cover the generic `Exception` class. We can `throw` an exception in PHP by using the keyword `throw` and creating a new `Exception` with an error message.

```
throw new Exception("Something went wrong");
```

By using a generic `Exception`, we can only attach a message to the exception (which can be printed by code that catches the exception). If we want more fine-grained control over the type of exceptions, we need to define our own exceptions.

41.2. Catching Exceptions

To catch an exception in PHP you can use the standard `try-catch` control block. Optionally (and as of PHP version 5.5.0) you can use the `finally` block to clean up any resources or execute code regardless of whether or not an exception was raised. Consider the simple task of reading input from a user and manually parsing its value into an integer. If the user enters a non-numeric value, parsing will fail and we should instead throw an exception. Consider the following function.

```
1 function readNumber() {
2   $input = readline("Please enter a number: ");
3   if( is_numeric($input) ) {
4     $value = floatval($input);
5   } else {
6     throw new Exception("Invalid input!");
7   }
8 }
```

Elsewhere in the code, we can surround a call to `readNumber()` in a `try-catch` statement.

```
try {
  readNumber();
} catch(Exception $e) {
  printf("Error: exception encountered: " . $e->getMessage());
  exit(1);
}
```

In this example, we've simply displayed an error message to the standard output and exited the program. We've made the design decision that this error should be fatal. We could have chosen to handle this error differently in the `catch` block. The `$e->getMessage()` prints the message that the exception was created with. In this case, `"Invalid input!"`.

41.3. Creating Custom Exceptions

As of PHP 5.3.0 it is possible to define custom exceptions by *extending* the `Exception` class. To do this, you need to declare a new class. We cover classes and objects in Chapter 45. For now, we present a simple example. Consider the example in the previous chapter of computing the roots of a quadratic polynomial. One possible error situation is when the roots are complex numbers. We could define a new PHP exception class as follows.

```
/**
 * Defines a ComplexRoot exception class
 */
class ComplexRootException extends Exception
{
    public function __construct($message = null,
                                $code = 0,
                                Exception $previous = null) {
        // call the parent constructor
        parent::__construct($message, $code, $previous);
    }

    // custom string representation of object
    public function __toString() {
        return __CLASS__ . ": [{$this->code}]: {$this->message}\n";
    }
}
```

Now in our code we can catch and even throw this new type of exception.

```
if( $b*$b - 4*$a*$c < 0) {
  throw new ComplexRootException("Cannot Handle complex roots");
}
```

```
try {
  $r1 = getRoot($a, $b, $c);
} catch(ComplexRootException $e) {
  //handle here
} catch(Exception $e) {
  //handle all other types of exceptions here
}
```

In the code above we had *two* `catch` blocks. Since we can have multiple types of exceptions, we can catch each different type and handle them differently if we choose. Each `catch` block catches a different type of exception. The last `catch` block was written to catch a generic `Exception`. Much like an `if-else-if` statement, the first type of exception that is caught is the block that will be executed and they are all mutually exclusive. Thus, a generic "catch all" block like this should always be the last `catch` block. The most specific types of exceptions should be caught first and the most general types should be caught last.

42. Arrays

PHP allows you to use arrays, but PHP arrays are actually *associative arrays*. Though you can treat them as regular arrays and use contiguous integer indices, they are more flexible than that. Integer indices need to be contiguous or start at zero and you can use strings as indices. In addition, since PHP is dynamically typed, PHP arrays allow mixed types. An array has no fixed type and you can place different mixed types into the same array. PHP arrays are dynamic, so there is no memory management or allocation/deallocation of memory space. Arrays will grow and shrink automatically as you add and remove elements.

42.1. Creating Arrays

Since PHP is dynamically typed, you do not need to declare an array or specify a particular type of variable that it holds. However, there are several ways that you can initialize an array. To create an empty array, you can call the `array()` function. Optionally, you can provide an initial list of elements to insert into the array by providing the list of elements as arguments to the function.

```php
//create an empty array:
$arr = array();

//create an array with elements 10, 20, 30:
$arr = array(10, 20, 30);

//create an array with mixed types:
$arr = array(10, 3.14, "Hello");
```

42.2. Indexing

By default, when inserting elements, 0-indexing is used. In the three examples above, each element would be located at indices 0, 1, and 2 respectively. The usual square bracket syntax can be used to access and assign elements.

```php
//create an array with elements 10, 20, 30:
$arr = array(10, 20, 30);

//get the first element:
$x = $arr[0]; //x has value 10

//change the 3rd element to 5:
$arr[2] = 5;

//print the 2nd element:
printf("$arr[1] = %d\n", $arr[1]);
```

Attempting to access an element at an invalid index does not result in an error or an exception (though a *warning* may be issued depending on how PHP is setup). Instead, if you attempt to access an invalid element, it will be treated as a `null` value.

```php
$arr = array(10, 20, 30);
$x = $arr[10]; //x is null
```

However, an array could have `null` values in it as elements. How do we distinguish whether or not an accessed value was *actually* `null` or if it is not part of the array? PHP provides a function, `array_key_exists()` to distinguish between these two cases. It returns `true` or `false` depending on whether or not a particular index has been set or not.

```php
$arr = array(10, null, 30);
if(array_key_exists(1, $arr)) {
  print "index 1 contains an element\n";
}
if(!array_key_exists(10, $arr)) {
  print "index 10 does not contain an element\n";
}

if($arr[1] === $arr[10]) {
  print "but they are both null\n";
}
```

42.2.1. Strings as Indices

Since arrays in PHP are associative arrays, keys are not limited to integers. You can also use strings as keys to index elements.

```
$arr = array();
$arr[0] = 5;
$arr["foo"] = 10;
$arr["hello"] = "world";

print "value = " . $arr["hello"];
```

Note that strings that contain integer values will be type-juggled into their numeric values. For example, `$arr["10"] = 3;` will be equivalent to `$arr[10] = 3;`. However, strings containing floating point values will not be coerced but will remain as strings, `$arr["3.15"] = 7;` for example.

42.2.2. Non-Contiguous Indices

Another aspect of PHP associative arrays is that integer indices need not be contiguous. For example,

```
$arr = array();
$arr[0] = 10;
$arr[5] = 20;
```

The values at indices 1 through 4 are undefined and the array contains some "holes" in its indices.

42.2.3. Key-Value Initialization

Since associative arrays in PHP can be indexed by either integers or strings and need not be ordered or contiguous, we can use a special key-value initialization syntax to define not only the values, but the keys that map to those values when we initialize an array. The "double arrow," `=>` is used to denote the mapping.

```
$arr = array(
  "foo" => 5,
  4 => "bar",
  0 => 3.14,
```

```
    "baz" => "ten"
);
```

42.3. Useful Functions

There are dozens of useful functions PHP defines that can be used with arrays. We'll only highlight a few of the more useful ones. First, the `count()` function can be used to compute how many elements are stored in the array.

```
$arr = array(10, 20, 30);
$n = count($arr); //n is 3
```

For convenience and debugging, a special function, `print_r()` allows you to print the contents of an array in a human-readable format that resembles the key-value initialization syntax above. For example,

```
$arr = array(
  "foo" => 5,
  4 => "bar",
  0 => 3.14,
  "baz" => "ten"
);
print_r($arr);
```

would end up printing

```
Array
(
    [foo] => 5
    [4] => bar
    [0] => 3.14
    [baz] => ten
)
```

Two other functions, `array_keys()` and `array_values()` return new zero-indexed arrays containing the keys and the values of an array respectively. Reusing the example above,

```php
$keys = array_keys($arr);
$vals = array_values($arr);
print_r($keys);
print_r($vals);
```

would print

```
Array
(
    [0] => foo
    [1] => 4
    [2] => 0
    [3] => baz
)
Array
(
    [0] => 5
    [1] => bar
    [2] => 3.14
    [3] => ten
)
```

Finally, you can use the equality operators, `==` and `===` to compare arrays. The first is the loose equality operator and evaluates to true if the two compared arrays have the same key-value pairs while the second is the strict equality operator and is true only if the arrays have the same key/value pairs in the same order and are of the same type.

42.4. Iteration

If we have an array in PHP that we *know* is 0-indexed and all elements are contiguous, we can use a normal for loop to iterate over its elements by incrementing an index variable.

```php
for($i=0; $i<count($arr); $i++) {
  print $arr[$i] . "\n";
}
```

This fails, however, when we have an associative array that has a mix of integer and string keys or "holes" in the indexing of integer keys. For this reason, it is more reliable to use `foreach` loops. There are several ways that we can use a `foreach` loop. The most general usage is to use the double arrow notation to iterate over each key-value pair.

```
//for each key value pair:
foreach($arr as $key => $val) {
  print "$key maps to $val \n";
}
```

This syntax gives you access to both the key and the value for each element in the array `$arr`. The keyword `as` is used to denote the variable names `$key` and `$val` that will be changed on each iteration of the loop. You need not use the identifiers `$key` and `$val`; you can use any legal variable names for the key/value variables. If you do not need the keys when iterating, you can use the following shorthand syntax.

```
//for each value:
foreach($arr as $val) {
  print "$val \n";
}
```

42.5. Adding Elements

You can easily add elements to an array by providing an index and using the assignment operator as in the previous examples. There are also several functions PHP defines that can insert elements to the beginning (`array_unshift()`), end (`array_push()`) or at an arbitrary position (`array_splice()`).

```
$arr = array(10, 20, 30);

array_unshift($arr, 15);
//arr is now [15, 10, 20, 30]

array_push($arr, 25);
//arr is now [15, 10, 20, 30, 25]

//insert 35 at index 3:
array_splice($arr, 3, 0, 35);
//arr is now [15, 10, 20, 35, 30, 25]
```

Another, simpler way of adding elements is to use the following syntax:

```php
$arr = array(10, 20, 30);
$arr[] = 5;
$arr[] = 15;
$arr[] = 25;
print_r($arr);
```

By using the assignment operator but not specifying the index, the element will be added to the next available integer index. Since there were already 3 elements in the array, each subsequent element is inserted at index 3, 4, and 5 respectively. In general, the element will be inserted at the maximum index value already used plus one. The example above results in the following.

```
Array
(
    [0] => 10
    [1] => 20
    [2] => 30
    [3] => 5
    [4] => 15
    [5] => 25
)
```

42.6. Removing Elements

You can remove elements from an array using the `unset()` function. This function only removes the element from the array, it does not shift other elements down to fill in the unused index.

```php
$arr = array(10, 20, 30);
unset($arr[1]);
print_r($arr);
```

This example would result in the following.

```
Array
(
    [0] => 10
    [2] => 30
)
```

Further, you can use `unset()` to destroy all elements in the array:

`unset($arr);`
destroys the entire array. It does not merely empty the array, but it unsets the variable `$arr` itself.

42.7. Using Arrays in Functions

By default, all arguments to a function in PHP are passed by value; this includes arrays. Thus, if you make any changes to an array passed to a function, the changes are not realized in the calling function. You can explicitly specify that the array parameter is passed by reference so that any changes to the array are realized in the calling function. To illustrate, consider the following example.

```
1 function setFirst($a) {
2   $a[0] = 5;
3 }
4
5 $arr = array(10, 20, 30);
6 print_r($arr);
7 setFirst($arr);
8 print_r($arr);
```

This example results in the following.

```
Array
(
    [0] => 10
    [1] => 20
    [2] => 30
)
Array
(
    [0] => 10
    [1] => 20
    [2] => 30
)
```

That is, the change to the first element does not affect the original array. However, if we specify that the array is passed by reference, then the change is realized. For example,

```
1 function setFirst(&$a) {
2   $a[0] = 5;
3 }
4
5 $arr = array(10, 20, 30);
```

```
6 print_r($arr);
7 setFirst($arr);
8 print_r($arr);
```

This now results in the original array being changed:

```
Array
(
    [0] => 10
    [1] => 20
    [2] => 30
)
Array
(
    [0] => 5
    [1] => 20
    [2] => 30
)
```

42.8. Multidimensional Arrays

PHP supports multidimensional arrays in the sense that elements in an array can be of any type, including other arrays. We can use the same syntax and operations for single dimensional arrays. For example, we can use the double arrow syntax and assign arrays as values to create a 2-dimensional array.

```
1 $mat = array(
2   0 => array(10, 20, 30),
3   1 => array(40, 50, 60),
4   2 => array(70, 80, 90)
5 );
6 print_r($mat);
```

Which results in the following:

```
Array
(
    [0] => Array
        (
            [0] => 10
            [1] => 20
            [2] => 30
        )
```

```
    [1] => Array
        (
            [0] => 40
            [1] => 50
            [2] => 60
        )

    [2] => Array
        (
            [0] => 70
            [1] => 80
            [2] => 90
        )

)
```

Alternatively, you can use two indices to get and set values from a 2-dimensional array.

```
for($i=0; $i<3; $i++) {
  for($j=0; $j<4; $j++) {
    $mat[$i][$j] = ($i+$j)*3;
  }
}
```

which results in:

```
Array
(
    [0] => Array
        (
            [0] => 0
            [1] => 3
            [2] => 6
            [3] => 9
        )

    [1] => Array
        (
            [0] => 3
            [1] => 6
            [2] => 9
            [3] => 12
        )

    [2] => Array
        (
            [0] => 6
            [1] => 9
            [2] => 12
            [3] => 15
        )

)
```

43. Strings

As we've previously seen, PHP has a built-in string type. Internally, PHP strings are simply a sequence of bytes, but for our purposes we can treat it as a 0-indexed character array. PHP strings are mutable and can be changed, but it is considered best practice to treat them as immutable and rely on the many functions PHP provides to manipulate strings.

43.1. Basics

We can create strings by assigning a string literal value to a variable. Strings can be specified by either single quotes or double quotes (there are no individual characters in PHP, only single character strings), but we will use the double quote syntax.

```
$firstName = "Thomas";
$lastName = "Waits";

//we can also reassign values
$firstName = "Tom";
```

The reassignment in the last line in the example effectively destroys the old string. The assignment operator can also be used to make copies of strings.

```
$firstName = "Thomas";
$alias = $firstName;
```

It is important to understand that this assignment makes a deep copy of the string. Changes to the first do not affect the second one. You can make changes to individual characters in a string by treating it like a zero-indexed array.

```
$a = "hello";
$a[0] = "H";
$a[5] = "!";
//a is now "Hello!"
```

The last line extends the string by adding an additional character. You can even remove characters by setting them to the empty string.

```
$a = "Apples!";
$a[5] = "";
//a is now "Apple!"
```

43.2. String Functions

PHP provides dozens of convenient functions that allow you to process and modify strings. We highlight a few of the more common ones here. A full list of supported functions can be found in standard documentation. Because of the history of PHP, many of the same functions defined in the C string library can also be used in PHP.

Length

When accessing individual characters in a string, it is necessary that we know the length of the string so that we do not access invalid characters (though doing so is not an error, it just results in `null`). The `strlen()` function returns an integer that represents the number of characters in the string.

```
$s = "Hello World!";
$x = strlen($s); //x is 12
$s = "";
$x = strlen($s); //x is 0

//careful:
$s = NULL
$x = strlen($s); //x is 0
```

As demonstrated in the last example, `strlen()` will return 0 for `null` strings. Recall that we can distinguish between these two situations by using `is_null()`. Using this function we can iterate over each individual character in a string.

```
$fullName = "Tom Waits";
for($i=0; $i<strlen($fullName); $i++) {
  printf("fullName[%d] = %s\n", $i, $fullName[$i]);
}
```

This would print the following

```
fullName[0] = T
fullName[1] = o
fullName[2] = m
fullName[3] =
fullName[4] = W
fullName[5] = a
fullName[6] = i
fullName[7] = t
fullName[8] = s
```

Concatenation

PHP has a concatenation operator built into the language. To concatenate one or more strings together, you can use a simple period between them as the concatenation operator. Concatenation results in a new string.

```
1 $firstName = "Tom";
2 $lastName = "Waits";
3
4 $formattedName = $lastName . ", " . $firstName;
5 //formattedName now contains "Waits, Tom"
```

Concatenation also works with other variable types.

```
1 $x = 10;
2 $y = 3.14;
3
4 $s = "Hello, x is " . $x . " and y = " . $y;
5 //s contains "Hello, x is 10 and y = 3.14"
```

Computing a Substring

PHP provides a simple function, `substr()` to compute a substring of a string. It takes at at least 2 arguments: the string to operate on and the starting index. There is a third, optional parameter that allows you to specify the length of the resulting substring.

```php
$name = "Thomas Alan Waits";

$firstName  = substr($name, 0, 6); //"Thomas"
$middleName = substr($name, 7, 4); //"Alan"
$lastName   = substr($name, 12);   //"Waits"
```

In the final example, omitting the optional length parameter results in the entire remainder of the string being returned as the substring.

43.3. Arrays of Strings

We often need to deal with collections of strings. In PHP we can define arrays of strings. Indeed, we've seen arrays of strings before. When processing command line arguments, PHP defines an array of strings, `$argv`. Each string can be accessed using an index, `$argv[0]` for example is always the name of the script.
We can create our own arrays of strings using the same syntax as with other arrays.

```php
$names = array(
  "Margaret Hamilton",
  "Ada Lovelace",
  "Grace Hopper",
  "Marie Curie",
  "Hedy Lamarr");
```

43.4. Comparisons

When comparing strings in PHP, we can use the usual comparison operators such as `===`, `<`, or `<=` which will compare the strings lexicographically. However, this is generally discouraged because of type juggling issues and strict vs loose equality/inequality comparisons. Instead, there are several comparator methods that PHP provides to compare strings based on their content. `strcmp($a, $b)` takes two strings and returns an integer based on the lexicographic ordering of `$a` and `$b`. If `$a` precedes `$b`, `strcmp()` returns something negative. It returns zero if `$a` and `$b` have the same content. Otherwise it returns something positive if `$b` precedes `$a`.

```php
$x = strcmp("apple", "banana"); //x is negative
$x = strcmp("zelda", "mario");  //x is positive
```

```
$x = strcmp("Hello", "Hello");  //x is zero

//shorter strings precede longer strings:
$x = strcmp("apple", "apples"); //x is negative

$x = strcmp("Apple", "apple");  //x is negative
```

In the last example, `"Apple"` precedes `"apple"` since uppercase letters are ordered before lowercase letters according to the ASCII table. We can also make case insensitive comparisons if we need to using the alternative, `strcasecmp($a, $b)`. Here, `strcasecmp("Apple", "apple")` will return zero as the two strings are the same ignoring the cases.
The comparison functions also have length-limited versions, `strncmp($a, $b, $n)` and `strncasecmp($a, $b, $n)`. Both will only make comparisons in the first `$n` characters of the strings. Thus, `strncmp("apple", "apples", 5)` will result in zero as the two strings are equal in the first 5 characters.

43.5. Tokenizing

Recall that *tokenizing* is the process of splitting up a string along some *delimiter*. For example, the comma delimited string, `"Smith,Joe,12345678,1985-09-08"` contains four pieces of data delimited by a comma. Our aim is to split this string up into four separate strings so that we can process each one. PHP provides several functions to to this, `explode()` and `preg_split()`.
The simpler one, `explode()` takes two arguments: the first one is a string delimiter and the second is the string to be processed. It then returns an array of strings.

```
$data = "Smith,Joe,12345678,1985-09-08";

$tokens = explode(",", $data);
//tokens is [ "Smith", "Joe", "12345678", "1985-09-08" ]

$dateTokens = explode("-", $tokens[3]);
//dateTokens is now [ "1985", "09", "08" ]
```

The more sophisticated `preg_split()` also takes two arguments,[1] but instead of a simple delimiter, it uses a regular expression; a sequence of characters that define a search *pattern* in which special characters can be used to define complex patterns. For

[1]The "preg" stands for **P**erl Compatible **Reg**ular Expression.

example, the complex expression `^[+-]?(\d+(\.\d+)?|\.\d+)([eE][+-]?\d+)?$` will match any valid numerical value including scientific notation. We will not cover regular expressions in depth, but to demonstrate their usefulness, here's an example by which you can split a string along any and all whitespace:

```
$s = "Alpha Beta \t Gamma \n Delta    \t\nEpsilon";
$tokens = preg_split("/[\s]+/", $s);
//tokens is now [ "Alpha", "Beta", "Gamma", "Delta", "Epsilon" ]
```

44. File I/O

Because of the history of PHP, file functions, just like string functions, were mostly influenced by the C standard library functions and have very similar naming and usage. Writing binary or plaintext data is determined by which functions you use. In general whether or not a file input/output stream is buffered or unbuffered is determined by the system configuration. There are some ways in which this can be changed, but we will not cover them in detail.

44.1. Opening Files

Files are represented in PHP as a "resource" (also referred to as a *handle*) that can be passed around to other functions to read and write to the file. For our purposes, a resource is simply a variable that can be stored and passed to other functions.
To open a file, you use the `fopen()` function (short for **f**ile **open**) which requires two arguments and returns a file handle. The first argument is the file path/name that you want to open for processing. The second argument is a string representing the "mode" that you want to open the file in. There are several supported modes, but the two we will be interested in are reading, in which case you pass it `"r"` and writing in which case you pass it `"w"`. The path can be an absolute path, relative path, or may be omitted if the file is in the current working directory.

```
1  //open a file for reading (input):
2  $input = fopen("/user/apps/data.txt", "r");
3  //open a file for writing (output):
4  $output = fopen("./results.txt", "w");
5
6  if(!$input) {
7    printf("Unable to open input file\n");
8    exit(1);
9  }
10
11 if(!$output) {
12   printf("Unable to open output file\n");
13   exit(1);
14 }
```

```php
$h = fopen("input.data", "r");
while(!feof($h)) {
  //read the next line:
  $line = fgets($h);
  //trim it:
  $line = trim($line);
  //process it, we'll just print it
  print $line;
}
```

Code Sample 44.1.: Processing a file line-by-line in PHP

The two conditionals above check that the file opened successfully. If opening the file failed, `fopen()` returns `false` (and the interpreter issues warning).

44.2. Reading & Writing

When a file is opened, the file handle returned by `fopen()` initially points to the beginning of the file. As you read or write from it, the resource advances through the file content.

Reading

There are a couple of ways to read input from a file. To read a file line by line, we can use `fgets()` to get each line. It takes a single argument: the file handle that was returned by `fopen()` and returns a string containing the entire line *including* the endline character. If necessary, leading and trailing whitespace can be removed using `trim()`. To determine the end of a file you can use `feof()` which returns `true` when the handle has reached the end of the file. Code Sample 44.1 contains a full example of processing a file line-by-line.
Alternatively there is a convenient function, `file_get_contents()` that will retrieve the entire file as a string.

```php
$fileContents = file_get_contents("./inputFile.txt");
```

Writing

There are several ways that we can output to files, but the easiest is to simply use `fwrite()` (short for **f**ile **write**). This is a "binary-safe" file output function that takes two arguments: the file handle to write to and a string of data to be written. It also returns an integer representing the number of bytes written to the file (or `false` on

error).

```php
$x = 10;
$y = 3.14;

//write to a plaintext file
fwrite($output, "Hello World!\n");
fwrite($output, "x = $x, y = $y\n");
```

44.2.1. Using URLs

A nice feature of PHP is that you can use URLs as file names to read and write to a URL. "Reading" from a URL mean connecting to a remote resource, such as a webservice and downloading its contents (which may be HTML, XML or JSON data). Writing to a URL may be used to *post* data to a web page in order to receive a response.
As an example, we can download a webpage in PHP as follows.

```php
$h = fopen("http://cse.unl.edu", "r");
$contents = "";
while(!feof($h)) {
  $contents .= fgets($h);
}

//or just:
$contents = file_get_contents("http://cse.unl.edu");
```

44.2.2. Closing Files

Once you are done processing a file, you should close it using the `fclose()` function.
`fclose($h);`
which takes a single argument, the file handle that you wish to close.

45. Objects

Object-oriented features have been continually added to PHP with each new version. Starting with version 5, PHP has had a full, class-based object-oriented programming support, meaning that it facilitates the creation of objects through the use of classes and class declarations. Classes are essentially "blueprints" for creating instances of objects. An *object* is an entity that is characterized by *identity*, *state* and *behavior*. The identity of an object is an aspect that distinguishes it from other objects. The variables and values that a variable takes on within an object is its state. Typically the variables that belong to an object are referred to as *member variables*. Finally, an object may also have functions that operate on the data of an object. In the context of object-oriented programming, a function that belongs to an object is referred to as a *member method*.
A class declaration specifies the member variables and member methods that belong to instances of the class. We discuss how to create and use instances of a class below. However, to begin, let's define a class that models a student by defining member variables to support a first name, last name, a unique identifier, and GPA.
To declare a class, we use the `class` keyword. Inside the class (denoted by curly brackets), we place any code that *belongs* to the class. To declare member variables in a class, we specify the variable names and their *visibility* inside the class, but outside any methods in the class.

```
1  class Student {
2
3    //member variables:
4    private $firstName;
5    private $lastName;
6    private $id;
7    private $gpa;
8
9  }
```

To organize code, it is common practice to place class declarations in separate files with the same name as the class. For example, this `Student` class declaration would be placed in a file named `Student.php` and included in any other script files that utilized the class.

45.1. Data Visibility

Recall that encapsulation involves not only the grouping of data, but the *protection* of data. The class declaration above achieves the grouping of data. To provide for the protection of data, PHP defines several *visibility* keywords that specify what segments of code can "see" the variables. Visibility in this context determines whether or not a segment of code can *access* and/or *modify* the variable's value. PHP defines three levels of visibility using they keywords `public`, `protected` and `private`. Each of these keywords can be applied to both member variables and member methods.

- `public` – This is the least restrictive visibility level and makes the member variable visible to any code segment.
- `protected` – This is a bit more restrictive and makes it so that the member variable is only visible to the code in the same class, or any *subclass* of the class.[1]
- `private` – this is the most restrictive visibility level, `private` member variables are only visible to instances of the class itself.

Table 45.1 summarizes these four keywords with respect to their access levels. It is important to understand that *protection* is in the context of encapsulation and does not involve protection in the sense of "security." Protection is this context is a design principle. Limiting the access of variables only affects how the rest of the code base interacts with our class and its data.

Modifier	Class	Subclass	World
`public`	Y	Y	Y
`protected`	Y	Y	N
`private`	Y	N	N

Table 45.1.: PHP Visibility Keywords & Access Levels

In general, it is best practice to make member variables `private` and control access to them via accessor and mutator methods (see below) unless there is a compelling design reason to increase their visibility.

45.2. Methods

The third aspect of encapsulation involves the grouping of methods that act on an object's data. Within a class, we can declare member methods using the syntax we're already familiar with. We declare a member method by using the keyword `function` and providing a signature and body. We can use the same visibility keywords as with member

[1] Subclasses are involved with *inheritance*, another object-oriented programming concept that we will not discuss here.

variables in order allow or restrict access to the methods. With methods, visibility and access determine whether or not the method may be invoked.
We add to our example by providing two `public` methods that compute and return a result on the member variables. We also use javadoc-style comments to document each member method.

```
class Student {

  //member variables:
  private $firstName;
  private $lastName;
  private $id;
  private $gpa;

  /**
   * Returns a formatted String of the Student's
   * name as Last, First.
   */
  public function getFormattedName() {
    return $this->lastName . ", " . $this->firstName;
  }

  /**
   * Scales the GPA, which is assumed to be on a
   * 4.0 scale to a percentage.
   */
  public function getGpaAsPercentage() {
    return $this->gpa / 4.0;
  }

}
```

There is some new syntax in the example above. In the member methods, we need a way to refer to the instance's member variables. The keyword `$this` is used to refer to the instance, this is known as open recursion.
When an instance of a class is created, for example,
`$s = new Student();`
the reference variable `$s` is how we can refer to it. This variable, however, exists *outside* the class. Inside the class, we need a way to refer to the instance itself. Since we don't have a variable *inside* the class to reference the instance itself, PHP provides the keyword `$this` in order to do so. Then, to access the member variables we use the *arrow operator*

(more below) and reference the member variable via its identifier *but with no dollar sign.*[2]

45.2.1. Accessor & Mutator Methods

Since we have made all the member variables `private`, no code outside the class may access or modify their values. It is generally good practice to make member variables `private` to restrict access. However, if we still want code outside the object to access or mutate (that is, change) the variable values, we can define accessor and mutator methods (or simply *getter* and *setter* methods) to facilitate this.
Each getter method returns the value of the instance's variable while each setter method takes a value and sets the instance's variable to the new value. It is common to name each getter/setter by prefixing a `get` and `set` to the variable's name using lower camel casing. For example:

```
1  public function getFirstName() {
2    return $this->firstName;
3  }
4
5  public function setFirstName($firstName) {
6    $this->firstName = $firstName;
7  }
```

One advantage to using getters and setters (as opposed to naively making everything `public`) is that you can have greater control over the values that your variables can take. For example, we may want to do some data validation by rejecting `null` values or invalid values. For example:

```
1  public function setFirstName($firstName) {
2    if($firstName === null) {
3       throw new Exception("names cannot be null");
4    } else {
5      $this->firstName = $firstName;
6    }
7  }
8
9  public function setGpa($gpa) {
10   if($gpa < 0.0 || $gpa > 4.0) {
```

[2] You *can* use the syntax `$this->$foo` but it will assume that `$foo` is a string that contains the *name* of another variable, for example, if `$foo = "firstName";` then `$this->$foo` would resolve to the instance's `$firstName` variable. This is useful if your object has been dynamically created by adding variables at runtime that were not part of the original class declaration.

```
11        throw new Exception("GPAs must be in [0, 4.0]");
12    } else {
13      $this->gpa = $gpa;
14    }
15  }
```

Controlling access of member variables through getters and setters is good encapsulation. Doing so makes your code more predictable and more testable. Making your member variables `public` means that any piece of code can change their values. There is no way to do validation or prevent bad values.
In fact, it is good practice to not even have setter methods. If the value of member variables cannot be changed, it makes the object immutable. Immutability is a nice property because it makes instances of the class thread-safe. That is, we can use instances of the class in a multithreaded program without having to worry about threads changing the values of the instance on one another.

45.3. Constructors

If we make the (good) design decision to make our class immutable, we still need a way to initialize the values. This is where a *constructor* comes in. A constructor is a special method that specifies how an object is constructed. With built-in variable types such as an numbers or strings, PHP "knows" how to interpret and assign a value. However, with user-defined objects such as our `Student` class, we need to specify how the object is created.
Just as with functions outside of classes, PHP does not support function overloading inside classes. That is, you can have one and only one function with a given identifier (name). Thus, there is only *one* possible constructor. Moreover, PHP reserves the name `__construct` for the constructor method. The two underscores are a naming convention used by PHP to denote "magic methods" that are reserved and have a special purpose in the language. Further, magic methods *must* be made `public`. Some magic methods provide default behavior while others do not. For example, if you do not define a constructor method, the default behavior will be to create an object whose member variables all have `null` values.
The following constructor allows a user to construct an instance of our `Student` instance and specify all four member variables.

```
1  public function __construct($firstName, $lastName, $id, $gpa) {
2    $this->firstName = $firstName;
3    $this->lastName = $lastName;
4    $this->id = $id;
5    $this->gpa = $gpa;
```

```
}
```

Though we cannot define multiple constructors, we can use the default value feature of PHP functions to allow a user to call our constructor with a different number of parameters. For example,

```
public function __construct($firstName, $lastName,
                            $id = 0, $gpa = 0.0) {
  $this->firstName = $firstName;
  $this->lastName = $lastName;
  $this->id = $id;
  $this->gpa = $gpa;
}
```

45.4. Usage

Once we have defined our class and its constructors, we can create and use instances of it. With regular variables, we simply need to assign them to an instance of an object and their type will dynamically change to match. To create new instances, we invoke a constructor by using the `new` keyword and providing arguments to the constructor.

```
$s = new Student("Alan", "Turing", 1234, 3.4);
$t = new Student("Margaret", "Hamilton", 4321, 3.9);
$u = new Student("John", "Smith");
```

The process of creating a new instance by invoking a constructor is referred to as *instantiation*. Once instances have been instantiated, they can be used by invoking their methods via the same arrow operator we used to access member variables. Outside the class, however this will only work if the member method is `public`.

```
print $t->getFormattedName() . "\n";

if($s->getGpa() < $t->getGpa()) {
  print $t->getFirstName() . " has a better GPA\n";
}
```

45.5. Common Methods

Another useful magic method is the `__toString()` method which returns a string representation of the object. Unlike the constructor method, there is *no* default behavior with the `__toString()` method. If you do not define this function, it cannot be used (and any attempts to do so will result in a fatal error). We can define the method to return the values of all or some of the class's variables in whatever format we want.

```
1 public function __toString() {
2   return sprintf("%s, %s (ID = %d); %.2f",
3           $this->lastName,
4           $this->firstName,
5           $this->id,
6           $this->gpa);
7 }
```

This would return a string containing something similar to
`"Hamilton, Margaret (ID = 4321); 3.90"`

45.6. Composition

Another important concept when designing classes is *composition*. Composition is a mechanism by which an object is made up of other objects. One object is said to "own" an instance of another object.
To illustrate the importance of composition, we could extend the design of our `Student` class to include a date of birth. However, a date of birth is also made up of multiple pieces of data (a year, a month, a date, and maybe even a time and/or locale). We could design our own date/time class to model this, but its generally best to use what the language already provides. PHP 5.2 introduced the `DateTime` object in which there is a lot of functionality supporting the representation and comparison of dates and time.
We can take this concept further and have our own user-defined classes own instances of each other. For example, we could define a `Course` class and then update our `Student` class to own a collection of `Course` objects representing a student's class schedule (this type of collection ownership is sometimes referred to as *aggregation* rather than composition).
Both of these design updates beg the question: who is responsible for instantiating the instances of `$dateOfBirth` and the `$schedule`? Should we force the "outside" user of our `Student` class to build a `DateTime` instance and pass it to a constructor? Should we allow the outside code to simply provide us a date of birth as a string and make the constructor responsible for creating the proper `DateTime` instance? Do we require that a user create a complete array of `Course` instances and provide it to the constructor at instantiation?

A more flexible approach might be to allow the construction of a `Student` instance without having to provide a course schedule. Instead, we could add a method that allowed the outside code to add a course to the student. For example,

```
1 public function addCourse($c) {
2   $this->schedule[] = $c;
3 }
```

This adds some flexibility to our object, but removes the immutability property. Design is always a balance and compromise between competing considerations.

45.7. Example

We present the full and completed `Student` class in Code Sample 45.1.

```php
<?php
class Student {

  private $firstName;
  private $lastName;
  private $id;
  private $gpa;
  private $dateOfBirth;
  private $schedule;

  public function __construct($firstName, $lastName, $id = 0, $gpa = 0.0,
                              $dateOfBirth = null, $schedule = array()) {
    $this->firstName = $firstName;
    $this->lastName  = $lastName;
    $this->id = $id;
    $this->gpa = $gpa;
    $this->dateOfBirth = new DateTime($dateOfBirth);
    $this->schedule = $schedule;
  }

  public function __toString() {
    return $this->getFormattedName() . " born " .
           $this->dateOfBirth->format("Y-m-d");
  }

  /**
   * Returns a formatted String of the Student's
   * name as Last, First.
   */
  public function getFormattedName() {
    return $this->lastName . ", " . $this->firstName;
  }

  /**
   * Scales the GPA, which is assumed to be on a
   * 4.0 scale to a percentage.
   */
  public function getGpaAsPercentage() {
    return $this->gpa / 4.0;
  }

  public function getFirstName() {
    return $this->firstName;
  }

  public function getLastName() {
```

```
47      return $this->lastName;
48    }
49
50    public function getId() {
51      return $this->id;
52    }
53
54    public function getGpa() {
55      return $this->gpa;
56    }
57
58    public function addCourse($c) {
59      $this->schedule[] = $c;
60    }
61
62  }
63
64  ?>
```

Code Sample 45.1.: The completed PHP `Student` class.

46. Recursion

PHP supports recursion with no special syntax necessary. However, recursion is generally expensive and iterative or other non-recursive solutions are generally preferred. We present a few examples to demonstrate how to write recursive functions in PHP.
The first example of a recursive function we gave was the toy count down example. In PHP it could be implemented as follows.

```
function countDown($n) {
  if($n===0) {
    printf("Happy New Year!\n");
  } else {
    printf("%d\n", $n);
    countDown($n-1);
  }
}
```

As another example that actually does something useful, consider the following recursive summation function that takes an array, its size and an index variable. The recursion works as follows: if the index variable has reached the size of the array, it stops and returns zero (the base case). Otherwise, it makes a recursive call to `recSum()`, incrementing the index variable by 1. When the function returns, it adds the result to the i-th element in the array. To invoke this function we would call it with an initial value of 0 for the index variable: `recSum($arr, 0)`.

```
function recSum($arr, $i) {
  if($i === count($arr)) {
    return 0;
  } else {
    return recSum($arr, $i+1) + $arr[$i];
  }
}
```

This example was not tail-recursive as the recursive call was not the final operation (the sum was the final operation). To make this function tail recursive, we can carry the summation through to each function call ensuring that the summation is done prior to

the recursive function call.

```
function recSumTail($arr, $i, $sum) {
  if($i === count($arr)) {
    return $sum;
  } else {
    return recSumTail($arr, $i+1, $sum + $arr[$i]);
  }
}
```

As a final example, consider the following PHP implementation of the naive recursive Fibonacci sequence. An additional condition has been included to check for "invalid" negative values of n for which we throw an exception.

```
function fibonacci($n) {
  if($n < 0) {
    throw new Exception("Undefined for n<0.");
  } else if($n <= 1) {
    return 1;
  } else {
    return fibonacci($n-1) + fibonacci($n-2);
  }
}
```

PHP is not a language that provides implicit memoization. Instead, we need to explicitly keep track of values using a table. In the following example, the table is passed through as an argument.

```
function fibonacciMemoization($n, $table) {
  if($n < 0) {
    return 0;
  } else if($n <= 1) {
    return 1;
  } else if(isset($table[$n])) {
    return $table[$n];
  } else {
    $a = fibonacciMemoization($n-1, $table);
    $b = fibonacciMemoization($n-2, $table);
    $result = ($a + $b);
    $table[$n] = $result;
    return $result;
```

```
    }
}
```

47. Searching & Sorting

PHP provides over a dozen different sorting functions each with different properties and behavior. Recall that PHP has associative arrays which store elements as key-value pairs. Some functions sort by keys, others sort by the value (with some in ascending order, others in descending order). Some functions preserve the key-value mapping and others do not. For simplicity, we will focus on two of the most common sorting functions, `sort()`, which sorts the elements in ascending order and `usort()` which sorts according to a comparator function.

47.1. Comparator Functions

Consider a "generic" Quick Sort algorithm as was presented in Algorithm 12.6. The algorithm itself specifies how to sort elements, but it doesn't specify how they are *ordered*. The difference is subtle but important. Quick Sort needs to know when two elements, a, b are in order, out of order, or equivalent in order to decide which partition each element goes in. However, it doesn't "know" anything about the elements a and b themselves. They could be numbers, they could be strings, they could be user-defined objects.
A sorting algorithm still needs to be able to determine the proper ordering in order to sort. In PHP this is achieved through a *comparator function*, which is a function that is responsible for comparing two elements and determining their proper order. A comparator function has the following signature and behavior.

- The function takes two elements `$a` and `$b` to be compared.
- The function returns an integer indicating the relative ordering of the two elements:
  - It returns something negative if `$a` comes before `$b` (that is, $a < b$)
  - It returns zero if `$a` and `$b` are equal ($a = b$)
  - It returns something positive if `$a` comes after `$b` (that is, $a > b$)

There is no guarantee on the value's magnitude, it does *not* necessarily return -1 or $+1$; it just returns *something* negative or positive. We've previously seen this pattern when comparing strings. The standard PHP library provides a function, `strcmp($a, $b)` that has the same contract: it takes two strings and returns something negative, zero or something positive depending on the lexicographic ordering of the two strings.
The PHP language "knows" how to compare built-in types like numbers and strings. However, to generalize the comparison operation, we can define comparator functions

that encapsulate more complex logic. As a simple first example, let's write a comparator function that orders numbers in ascending order.

```
function cmpInt($a, $b) {
  if($a < $b) {
    return -1;
  } else if($a === $b) {
    return 0;
  } else {
    return -1;
  }
}
```

What if we wanted to order integers in the opposite order? We could write another comparator in which the comparisons or values are reversed. Even simpler, we could reuse the comparator above and "flip" the sign by multiplying by -1 (this is one of the purposes of writing functions: code reuse). Even simpler still, we could just flip the arguments we pass to `cmpInt()` to reverse the order.

```
function cmpIntDesc($a, $b) {
  return cmpInt($b, $a);
}
```

To illustrate some more examples, consider the `Student` class we defined in Code Sample 45.1. The following code samples demonstrate various ways of ordering `Student` instances based on one or more of their components.

```
/**
 * A comparator function to order Student instances by
 * last name/first name in alphabetic order
 */
function studentByNameCmp($a, $b) {
  int result = strcmp($a->getLastName(), $b->getLastName());
  if(result == 0) {
    return strcmp($a->getFirstName(), $b->getFirstName());
  } else {
    return result;
  }
}
```

```php
/**
 * A comparator function to order Student instances by
 * last name/first name in reverse alphabetic order
 */
function studentByNameCmpDesc($a, $b) {
  return studentByNameCmp($b, $a);
}
```

```php
/**
 * A comparator function to order Student instances by
 * id in ascending numerical order
 */
function studentIdCmp($a, $b) {
  if($a->getId() < $b->getId()) {
    return -1;
  } else if($a->getId() === $b->getId()) {
    return 0;
  } else {
    return 1;
  }
}
```

```php
/**
 * A comparator function to order Student instances by
 * GPA in descending order
 */
function studentGpaCmp($a, $b) {
  if($a->getGpa() > $b->getGpa()) {
    return -1;
  } else if($a->getGpa() == $b->getGpa()) {
    return 0;
  } else {
    return 1;
  }
}
```

47.1.1. Searching

PHP provides a linear search function, `array_search()` that can be used to search for an element in an array. The array can be specified to use loose comparisons (default) or strict comparisons. It returns the key (i.e. index) of the first matching element it finds and `false` if the search was unsuccessful. For example:

```
$arr = array(10, 8, 3, 12, 4, 42, 7, 108);
$index = array_search(12, $arr); //index is now 3

$arr = array("hello", 10, "mixed", 12, "20");
//a loose search:
$index = array_search(20, $arr); //index is now 4
//a strict search:
$index = array_search(20, $arr, false); //index is now false.
```

PHP does not provide a standard binary search function. Though you can write your own binary search implementation, likely the reason that that PHP does not provide one is because one is not needed. The purpose of binary search is to search a sorted array efficiently. However, PHP arrays are not usual arrays: they are associative arrays, essentially key-value maps. Retrieving an element via its key is essentially a constant-time operation, even more efficient that binary search. A better solution may be to simply store the elements using a proper key which can be used to retrieve the element later on.

47.1.2. Sorting

PHP's `sort()` function can be used to sort elements in ascending order. This is useful if you have arrays of numbers or strings, but doesn't work very well if you have an array of mixed types or objects.

```
$arr = array("banana", "Apple", "zebra", "apple", "bananas");
sort($arr);
//arr is now ("Apple", "apple", "banana", "bananas", "zebra")

$arr = array(10, 8, 3, 12, 4, 42, 7, 108);
sort($arr);
//arr is now (3, 4, 7, 8, 10, 12, 42, 108)
```

PHP provides a more versatile sorting function, `usort()` (**u**ser defined **sort**) that accepts a comparator function that it uses to define the ordering of elements. To pass a comparator function to the `usort()` function, we pass a string value containing the

name of the comparator function we wish to use. Recall that function names in PHP are case insensitive, though it is still best practice to match the naming. Several examples of the usage of this function are presented in Code Sample 47.1.

```php
$arr = array(10, 8, 3, 12, 4, 42, 7, 108);
usort($arr, "cmpIntDesc");
//arr is now 108, 42, 12, 10, 8, 7, 4, 3

//roster is an array of Student instances
$roster = ...

//sort by name:
usort($roster, "studentByNameCmp");

//sort by ID:
usort($roster, "studentIdCmp");

//sort by GPA:
usort($roster, "studentGpaCmp");
```

Code Sample 47.1.: Using PHP's `usort()` Function

Glossary

abstraction a technique for managing complexity whereby levels of complexity are established so that higher levels do not see or have to worry about details at lower levels.

acceptance testing a phase of software testing that is tested, usually by humans, for acceptability and whether or not it fulfills the business requirements.

ad-hoc testing informal software testing method that is done without much planning usually as a "sanity check" to see if a piece of code works and intended for immediate feedback.

algorithm a process or method that consists of a specified step-by-step set of operations. 17

anonymous class a class that is defined "inline" without declaring a named class; typically created because the instance has a single use and there is no reason to create multiple instances. 484

anonymous function a function that has no identifier or name, typically created so that it can be passed as an argument to another function as a callback. 144

anti-pattern a common software pattern that is used as a solution to recurring problems that is usually ineffective in solving the problem or introduces risks and other problems; a technical term for common "bad-habits" that can be found in software. 152

array an ordered collection of pieces of data, usually of the same type.

assignment operator an operator that allows a user to assign a value to a variable. 33

backward compatible a program, code, library, or standard that is compatible with previous versions so that current and older versions of it can coexist and successfully operate without breaking anything. 29

bar a placeholder name. , *see* foo

baz a placeholder name. , *see* foo

bike shedding a phenomenon in projects where a disproportionate amount of time is spent on trivial matters at the expense of spending adequate resources on more important matters. The term was applied to software development by Kamp [24] but derives from Parkinson's law of triviality [31].

bit the basic unit of information in a digital computer. A bit can be either 1 or 0 (alternatively, *true*/*false*, on/off, high voltage/low voltage, etc.). Originally a portmanteau (mash up) of **b**inary dig**it**. 4, 22, 23

Boolean a data type that represents the truth value of a logical statement. Booleans typically have only two values: *true* or *false*. 30

bug A flaw or mistake in a computer program that results in incorrect behavior that may have unintended such as errors or failure. The term predates modern computer systems but was popularized by Grace Hopper who, when working with the Mark II computer in 1946 traced a system failure to a moth stuck in a relay. 46, 81, 151

byte a unit of information in a digital computer consisting of 8 bits. 4

cache a component or data structure that stores data in an efficiently retrievable manner so that future requests for the data are fast. 208

call by reference when a variable's memory address is passed as a parameter to a function, enabling the function to manipulate the contents of the memory address and change the original variable's value. 142

call by value when a *copy* of a variable's value is passed as a parameter to a function; the function has no reference to the original variable and thus changes to the copy inside the function have no effect on the original variable. 140

callback a function or executable unit of code that is passed as an argument to another function with the intention that the function that it is passed to will execute or "call back" the passed function at some point. 144, 366

cargo cult programming refers to the act of including code, patterns, or processes with no real purpose due to a lack of understanding of a bug or of the problem they were attempting to solve or a lack of understanding of the copied code/pattern itself or an inability to adapt it properly..

case sensitive a language is case sensitive if it recognizes differences between lower and upper case characters in identifier names. A language is case insensitive if it does not.. 19

chomp the operation of removing any endline characters from a string (especially when read from a file); may also refer more generally to removing leading and trailing whitespace from a string or "trimming" it. 337,

closure a function with its own environment in which variables exist. 485

code smell a symptom or common pattern in source code that is usually indicative of a deeper problem or design flaw; smells are usually not bugs and may not cause problems in and of themselves, but instead indicate a pattern of carelessness or low quality of software design or implementation. 152

comparator a function or object that allows you to pass in two elements a, b for comparison and returns an integer indicating their relative order: something negative, zero, or something positive if $a < b$, $a = b$ or $a > b$ respectively. 178, 331, 560

compile the process of translating code in a high-level programming language to a low level language such as assembly or machine code.

computer engineering a discipline integrating electrical engineering and computer science that tends to focus on the development of hardware and its interaction with software.

computer science the mathematical modeling and scientific study of computation.

concatenation the process of combining two (or more) strings to create a new string by appending one of them to the end of the other. 178

constant a variable whose value cannot be changed once set.

continuous integration a software development/engineering process by which developer code is merged and shared on a frequent basis.

contradiction a logical statement that is always *false* regardless of the truth values of the statement's variables. 72

control flow the order in which individual statements in a program are executed or evaluated. 75

Conway's Law an observation that organizations that build software will inevitably design it as modeling the structure of the organization itself. Attributed to Melvin Conway (1967) who originally stated it as "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations".

copy pasta bad coding practice of copy-pasting code and making slight changes when a function or other control structure should be written instead. Copy-pasta practices often lead to poorly designed code, bugs or spaghetti code..

cruft anything that is left over, redundant or getting in the way; in the context of code cruft is code that is no longer needed, legacy or simply poorly written source code.

dangling pointer when a reference to dynamically allocated memory is lost and the memory can no longer be deallocated, resulting in a memory leak. Alternatively, when a reference points to memory that gets deallocated or reallocated but the pointer remains unmodified, still referencing the deallocated memory. 165

dead code a code segment that has no effect on a program either because it is unused or unreachable (the conditions involving the code will never be satisfied). 72

debug the process of analyzing a program to find a fault or error with the code that leads to bad or unexpected results. 152

debugger a software tool that facilitates debugging; usually a debugger simulates the execution of a program allowing a developer to view the contents of a program as it executes and to "walk" through the execution step by step. 152

deep copy in contrast to a shallow copy, a deep copy is a copy of an array or other piece of data that is distinct from the original. Changes to one copy do not affect the other. 166, 319, 443, 449, 557

defensive programming an approach to programming in which error conditions are checked and handled, preventing undefined or erroneous operations from happening in a program. 38, 48, 84

dynamic programming a technique for solving problems that involves iteratively computing values to subproblems, storing them in a table so that they can be used to solve larger versions of the problem. 208

dynamic typing a variable whose type can change during runtime based on the value it is assigned. 31,

edge case a situation or input that represents an extreme value or parameter.

encapsulation the grouping and protection of data together into one logical entity along with the functionality (functions or methods) that act on that data. 30

enumerated type a data type (usually user defined) that consists of a list of named values. 309, 436

exception an event or occurrence of an erroneous or "exceptional" condition that interrupts the normal flow of control in a program, handing control over to exception handler(s). 155

expression a combination of values, constants, literals, variables, operators and possibly function calls such that when evaluated, produce a resulting value. 34

file a resource on a computer stored in memory that holds data. 183

flowchart a diagram that represents an algorithm or process, showing steps as boxes connected by arrows which establish an order or flow. 17

foo along with "bar," "baz," and the full "foobar", foo is a commonly used placeholder name in programming used to denote generic variables, functions, etc. Usually these terms are used as examples when demonstrating a concept. The terms have a long history [21].

foobar a placeholder name. , *see* foo

function a sequence of program instructions that perform a specific task, packaged as a unit, also known as a *subroutine*. 133

function overloading the ability to define multiple functions with the same name but with with a different number of or different types of parameters. 145

fuzzing An automated software testing technique that involves providing invalid or random data as input to a program or piece of code to test that it fails in an expected manner..

garbage collection automated memory management in which a garbage collector attempts to reclaim memory (garbage) that is no longer being used by a program so that it can be reallocated for other purposes. 165, 383

global scope a variable, function, or other element in a program has global scope if it is visible or has effect throughout the entire program. 32,

grok slang, meaning to understand something; originally from *Stranger in a Strange Land* by Robert A. Heinlein..

hardware (computer hardware) the physical components that make up a computer system such as the processor, motherboard, storage devices, input and output devices, etc..

heisenbug a bug in a program that fails to manifest itself under testing or debugging conditions. Derived from the Heisenberg principle which states that you cannot observe something without affecting it, thus tainting your observations..

hexadecimal base-16 number system using the symbols 0, 1, . . . , 9, A, B, C, D, E, F; usually denoted with a prefix `0x` such as `0xff1321ab01`. 470

hoisting usually used in interpreted languages, hoisting involves processing code to find variable or function declarations and processing them before actually executing the code or script. 134

identifier a symbol, token, or label that is used to refer to a variable. Essentially, a variable's name. 18

idiom in the context of programming, an idiom is a commonly used pattern, expression or way of structuring code that is well-understood for users of the language. For example, a for-loop structure that iterates over elements in an array. May also refer to a programming design pattern.. 315,

immutable an object whose internal state cannot be changed once created, alternatively, one whose internal state cannot be *observably* changed once created. 390, 423, 449, 466, 571

inheritance an object oriented programming principle that allows you to derive an object from another object, usually to allow for more specificity.

input data or information that is provided to a computer program for processing. 41

integration testing a type of software testing used to determine if the integration of different components or modules works or not.

interactive a program that is designed to interface with humans by prompting them for input and displaying output directly to them. 41

interactive an informal, abstract, high-level description of a process or algorithm. 41

keyword a word in a programming language with a special meaning in a particular context. In contrast to a reserved word, a keyword *may* be used for an identifier (variable or function name) but it is strongly discouraged to do so as the keyword already has an intended meaning.

kilobyte a unit of information in a digital computer consisting of 1024 bytes (equivalently, 2^{10} bytes), KB for short.

kludge a poorly designed or "thrown-together" solution; a design that is a collection of ill-fitting parts that may be functional, but is fragile and not easily maintained.

lambda expression another term for anonymous functions. 144, 493

lexicographic a generalization of the usual dictionary order as codified with the ASCII character table. 178

linking the process of generating an executable file from (multiple) object files.

lint (or linter) a static code analysis tool that analyzes code for suspicious or error-prone code that is likely to cause problems. 152

literal in a programming language, a literal is notation for specifying a value such as a number of string that can be directly assigned to a variable. 29, 34

load testing a form of performance testing used to determine to what extent a piece of software or a system can handle a high demand or "load".

magic number a value used in a program with unexplained, undocumented, or ambiguous meaning, usually making the code less understandable. 258, 309, 310, 437

mantissa the part of a floating-point number consisting of its significant digits (called a significand in scientific notation). 25

map a data structure that allows you to store elements as key-value pairs with the key mapping to a value.

memoization a technique which uses a table to store previously computed values of a function so that they do not need to be recomputed, essentially the table serves as a cache. 208

memory leak the gradual loss of memory when a program fails to deallocate or free up unused memory, degrading performance and possibly resulting in the termination of the program when memory runs out. 165, 320, 326

naming convention a set of guidelines for choosing identifier names for variables, functions, etc. in a programming language. Conventions may be generally accepted by all developers of a particular language or they may be established for use in a particular library, framework, or organization. 20

network a collection of two or more computer systems linked together through a physical connection over which data can be transmitted using some protocol.

octal base-8 number system using the symbols 0, 1, 2, . . . , 6, 7; usually denoted with a single leading zero such as `0123742`.

open recursion a mechanism by which an object is able to refer to itself usually using a keyword such as `this` or `self`.. 200, 465, 569

operand the arguments that an operator applies to. 33

operator a symbol used to denote some transformation that combines or changes the operands it is applied to to produce a new value. 33

order of precedence the order in which operators are evaluated, multiplication is performed before addition for example. 38

output data or information that is produced as the result of the execution of a program. 41

overflow when an arithmetic operation results in a number that is larger than the specified type can represent overflow occurs resulting in an invalid result. 39

parse to process data to identify its individual components or elements. 179,

persistence the characteristic of data that outlives the process or program that created it; the saving of data across multiple runs of a program. 183

pointer a reference to a particular memory location in a computer. 30, 295

polymorphism an object oriented programming concept that allows you to treat a variable, method, or object as different types.

primitive a basic data type that is defined and provided by a programming language. Typically numeric and character types are primitive types in a language for example. Generally, the user doesn't need to define the operations involving primitive types as they are defined by the language. Primitive data types are used as the basic building blocks in a program and used to *compose* more complex user-defined types. 30, 390

procedural abstraction the concept that a procedure or sequence of operations can be encapsulated into one logical unit (function, subroutine, etc.) so that a user need not concern themselves with the low-level details of how it operates. 134

program stack also referred to as a *call stack*, it is an area of memory where stack frames are stored for each function call containing memory for arguments, local variables and return values/addresses. 137

protocol a set of rules or procedures that define how communication takes place.

pseudocode the act of a program asking a user to enter input and subsequently waiting for the user to enter data. 13

query an operation that retrieves data, typically from a database. 152

queue a data structure that store elements in a FIFO (First-In First-Out) manner; elements can be added to the end of a queue by an *enqueue* operation and removed from the start of a queue by a *dequeue* operation.

radix the base of a number system. Binary, octal, decimal, hexadecimal would be base 2, 8, 10, and 16 respectively.

reentrant a function that can be interrupted during its execution while another thread can successfully invoke the function without the two functions interfering with the data used in either function call. 334

refactor the process of modifying, updating or restructuring code without changing its external behavior; refactoring may be done to make code more efficient, more readable, more reliable, or simply to bring it into compliance with style or coding conventions.

reference a reference in a computer program is a variable that refers to an object or function in memory. 30

regression testing a type of software testing that tests software that previously passed testing but that was changed or refactored in some way. Regression testing tests to see if the software still passes the tests or not in which case it is said to have "regressed".

regular expression a sequence of characters in which special characters and directives can be used to define a complex pattern that can be searched and matched in another string or data. 455, 561

reserved word a word or identifier in a language that has a special meaning to the syntax of the language and therefore cannot be used as an identifier in variables, functions, etc.. 19

scope the *scope* of a variable, method, or other entity in a program is the part of the program in which the name or reference of the entity is bound. That is, the part of the program that "knows" about the variable in which the variable can be accessed, changed, or used. 32, 255, 386

segmentation fault a fault or error that arises when a program attempts to access a segment of memory that it is not allowed access to, usually resulting in the program being terminated by the operating system. 296, 315

shallow copy in contrast to a deep copy, a shallow copy is merely a reference that refers to the original array or piece of data. The two references point to the same data, so if the data is modified, both references will realize it.. 166, 319, 443

short circuiting the process by which the second operand in a logical statement is not evaluated if the value of the expression is determined by the first operand. 74

signature a function signature is how a function is uniquely identified. A signature includes the name (identifier) of the function, its parameter list (and maybe types) and the return type. 134

software any set of machine-readable instructions that can be executed in a computer processor.

software engineering the study and application of engineering principles to the design, development, and maintenance of complex software systems.

spaghetti code a negative term used for code that is overly complex, disorganized or unstructured code.

stack a data structure that stores elements in a LIFO (last-in first-out) manner; elements can be added to a stack via a *push* operation which places the element on the "top" of the stack; elements can be removed from the top of the stack via a *pop* operation. 137, 206

stack overflow when a program runs out of stack space, it may result in a stack overflow and the termination of the program. 205

static analysis the analysis of software that is performed on source (or object) code without actually running or compiling a program usually by using an automated tool that can detect actual or potential problems with the source code (other than syntactic problems that could easily be found by a compiler). 152

static dispatch when function overloading is supported in a language, this is the mechanism by which the compiler determines *which* function should be called based on the number and type of arguments passed to the function when it is called. 145

static typing a variable whose type is specified when it is created (declared) and does not change while the variable remains in scope. 31,

string a data type that consists of a sequence of characters which are encoded under some encoding standard such as ASCII or Unicode. 27, 177

string concatenation an operation by which a string and another data type are combined to form a new string. 37

syntactic sugar syntax in a language or program that is not absolutely necessary (that is, the same thing can be achieved using other syntax), but may be shorter, more convenient, or easier to read/write. In general, such syntax makes the language "sweeter" for the humans reading and writing it. 40, 82, 102, 161, 451

tautology a logical statement that is always *true* regardless of the truth values of the statement's variables. 72

test case an input/output pair that is known to be correct that is to be used to test a unit of software.

test case a collection of tests that are used to test a piece or collection of software.

token when something (usually a string) is parsed, the individual components or elements are referred to as tokens. 179

top-down design an approach to problem solving where a problem is broken down into smaller parts. 3, 133

transpile to (automatically) translate code in one programming language into code in another programming language, usually between two high-level programming languages.

truncation removing the fractional part of a floating-point number to make it an integer. Truncation is *not* a rounding operation. 36, 263, 394,

two's complement A way of representing signed (positive and negative) integers using the first bit as a sign bit (0 for positive, 1 for negative) and where negative numbers are represented as the complement with respect to 2^n (the result of subtracting the number from 2^n) . 24

type a variable's type is the classification of the data it represents which could be numeric, string, boolean, or a user defined type. 22

type casting converting or variable's type into another type, for example, converting an integer into a more general floating-point number, or converting a floating-point number into an integer, truncating and losing the fractional part. 36, 263, 394

underflow when an arithmetic operation involving floating-point numbers results in a number that is smaller than the smallest representable number underflow occurs resulting in an invalid result. 39

Unicode an international character encoding standard used in programming languages and data formats. 449

unit test software testing method that tests an individual "unit" of code; depending on the context, a unit may refer to an individual function, class, or module.

unwinding the process of removing a stack frame when returning from a function. 164

validation the process of verifying that data is correct or conforms to certain expectations including formatting, type, range of values, represents a valid value, etc.. 42

variable a memory location which stores a value that may be set using an assignment operator. Typically a variable is referred to using a name or *identifier*. 18

widget a generic term for a graphical user interface component such as a button or text box.

yak shaving an ostensibly pointless activity that despite its uselessness allows you to overcome a a difficulty and solve a larger problem. Basically, screwing around until you become familiar enough with the problem that the solution presents itself. Attributed to Carlin Vieri who took it from an episode of *Ren and Stimpy*..

Acronyms

ACID Atomicity Consistency Isolation Durability.

ACM Association for Computing Machinery.

ALU Arithmetic and Logic Unit. 4

ANSI American National Standards Institute. 253

API Application Programmer Interface. 15, 184, 436

ASCII American Standard Code for Information Interchange. 27, 66, 183, 187, 260, 325, 332, 385, 391, 454, 561

BYOD Bring Your Own Device.

CE Computer Engineering.

CI Continuous Integration.

CJK Chinese Japanese Korean. 29

CLA Command Line Arguments.

CLI Command Line Interface. 46

CMS Content Management System. 497

CMYK Cyan-Magenta-Yellow-Key.

CPU Central Processing Unit. 4, 74

CRUD Create Retrieve Update Destroy.

CS Computer Science.

CSS Cascading Style Sheets.

CSV Comma Separated Values. 179, 337

CWD Current Working Directory. 184

CYA Cover Your Ass.

DRY Don't Repeat Yourself. 292

EB Exabyte.

ECMA European Computer Manufacturers Association.

EDI Electronic Data Interchange.

EOF End Of File. 184

FIFO First-In First-Out.

FOSS Free and Open Source Software.

FUD Fear Uncertainty Doubt.

GB Gigabyte.

GCC GNU Compiler Collection.

GDB GNU Debugger.

GIF Graphics Interchange Format.

GIGO Garbage-In Garbage-Out.

GIMP GNU Image Manipulation Program.

GIS Geographic Information System.

GNU GNU's Not Unix!. 49, 324

GUI Graphical User Interface. 42, 144, 366

HTML HyperText Markup Language. 181, 184, 497, 565

IDE Integrated Development Environment. 7, 21, 46, 49, 258, 385, 390, 471, 502

IEC International Electrotechnical Commission. 26, 253

IEEE Institute of Electrical and Electronics Engineers. 26, 260, 306, 503

IP Internet Protocol. 201

ISO International Organization for Standardization. 253, 343

JDBC Java Database Connectivity.

JDK Java Development Kit. 383, 385, 388, 427, 431, 436, 449, 458, 487

JEE Java Enterprise Edition.

JIT Just In Time. 12

JPEG Joint Photographic Experts Group. 183

JRE Java Runtime Environment.

JSON JavaScript Object Notation. 185, 565

JVM Java Virtual Machine. 12, 383, 384, 389, 423, 431, 435, 440, 442, 449, 467, 470, 484

KB Kilobyte. 162, 460

KISS Keep It Simple, Stupid.

LIFO Last-In First-Out. 137

LOC Lines Of Code.

MAC Media Access Control. 201

MB Megabyte. 162

MP3 MPEG-2 Audio Layer III.

MPEG Moving Picture Experts Group.

MVP Minimum Viable Product.

MWE Minimum Working Example. 254

NIST National Institute of Standards and Technology.

ODBC Open Database Connectivity.

OEM Original Equipment Manufacturer.

OOP Object-Oriented Programming. 177, 384, 386, 433, 444, 499

PB Petabyte.

PDF Portable Document Format. 15

PHP PHP: Hypertext Preprocessor (a recursive backronym; used to stand for Personal Home Page).

PNG Portable Network Graphics.

POJO Plain Old Java Object.

POSIX Portable Operating System Interface. 259, 306, 336, 343

RAM Random Access Memory. 5

REPL Read-Eval-Print Loop.

RGB Red-Green-Blue.

ROM Read-Only Memory.

RTFM Read The "Freaking" Manual.

RTM Read The Manual.

SD Software Development.

SDK Software Development Kit. 443, 444

SE Software Engineering.

SEO Search Engine Optimization.

SQL Structured Query Language. 431, 436

SSL Secure Sockets Layer. 296

STEAM Science, Technology, Engineering, Art, and Math.

STEM Science, Technology, Engineering, and Math.

TB Terabyte.

TCP Transmission Control Protocol.

TDD Test-Driven Development.

TSV Tab Separated Values. 179

TUI Text User Interface.

UI User Interface.

UML Unified Modeling Language.

URL Uniform Resource Locator.

UTC Coordinated Universal Time. 343

UTF-8 Universal (Character Set) Transformation Format–8-bit.

UX User Experience.

VLSI Very Large Scale Integration. 4

W3C World Wide Web Consortium.

WWW World Wide Web. 181, 383, 497

XML Extensible Markup Language. 21, 185, 565

YAGNI You Ain't Gonna Need It.

Index

aggregation, 199
algorithm, 215
anonymous class, 484
arrays, 159
    in C, 313
    in Java, 439
    in PHP, 547
    indexing, 160
    iteration, 161
    multidimensional, 166
    static, see static array160
arrow operator, 347
assignment operator, 33
associative arrays, 168

bandwidth, 5
basic input, 41
basic output, 41
binary, 4, 23
    counting, 23
binary search, 213
    analysis, 218
    C, 373
bit, 4
block, *see* code block
Boolean, 30
bottom-up design, 3, 200
buffered, 187
bug, 151
byte, 4

C
    arrays, 313
    comparators, 361
    conditionals, 271
    error handling, 305
    file I/O, 335
    functions, 291
    loops, 283
    recursion, 357
    strings, 325
cache, 208
call by reference, 140
call by value, 140
call stack, 7
callback, 144, 366
calling functions, *see* invoking functions
case sensitive, 19
class, 198, 386
class-based, 198
code block, 13
command line arguments, 44
comment, 14
comparator, 178, 238, 239, 361
    in Java, 483
comparison operators, 66
compiled language, 7
composition, 199, 344, 472
compound logic, 71
computer, 4
conditionals, 65
    if statement, 75
    if-else statement, 76
    if-else-if statement, 78
    in C, 271
    in Java, 403
    in PHP, 515
conjunction, *see* logical operators–and
constants, 33

constructor, 199, 467
contradiction, 72
control flow, 17
copy constructor, 468

De Morgan's laws, 72
debugger, 152
debugging, 46, 49
deep copy, 166, 353
defensive programming, 48, 84, 153
disjunction, *see* logical operators–or
do-while loop, 100
dot operator, 346
dynamic memory, 164
dynamic programming, 208
dynamic typing, 31

elementary operation, 215
encapsulation, 30, 197
    C, 341
enumerated type, 30
enumerated types
    in Java, 436
error code, 154
error handling, 151
    exception, 155
    in C, 305
    in Java, 431
    in PHP, 543
exception, 155

fatal error, 153
file, 183
file I/O, 183
    in C, 335
    in Java, 457
    in PHP, 563
first-class citizen, 142
floating point number, 25
flowchart, 17
for loop, 99
foreach loop, 102, 161
function hoisting, 535
function pointers, 365
function prototype, 291
functional programming, 142
functions, 133
    in C, 291
    in Java, 423
    in PHP, 535

garbage collection, 165
Gauss's Formula, 222
generic programming, 212
global scope, 32
graphical user interface, 42

hardware, 4
heap, 160
Heap Sort, 237
Hello World
    C, 253
hello world
    Java, 384
hexadecimal, 5
hoisting, 134, 535

identifier, 18
if statement, 75
if-else statement, 76
if-else-if statement, 78
immutable, 423
index, 160
infinite loop, 97, 103
Insertion Sort, 224
Integrated Development Environment, 7
interpreted language, 7
invoking functions, 136

Java
    arrays, 439
    Comparators, 483
    conditionals, 403
    enumerated types, 436
    error handling, 431
    file I/O, 457
    loops, 415
    methods, 423
    objects, 461
    recursion, 479

strings, 449

kilobyte, 4

lambda expression, 493
linear search, 212
C, 372
linked list, 167
linter, 152
list
in Java, 444
lists, 167
literal, 34
logic errors, 48
logical operators, 65
and, 69
negation, 68
or, 70
loops, 95
do-while loop, 100
for loop, 99
foreach loop, 102
in C, 283
in Java, 415
in PHP, 527
infinite loop, 103
while loop, 97
lower camel casing, 20

main memory, 5
mantissa, 25
maps, 168
member methods, 198
member variables, 198
memoization, 208, 358
memory, 5
memory leak, 165, 326
memory management, 165
Merge Sort, 232
method overloading, 423
multidimensional arrays, 166, 443

natural order, 242
nested loops, 103
nesting, 84
null, 30

object, 30, 197
objects, 197
defining, 198
in Java, 461
using, 200
open recursion, 200, 465
in PHP, 569
operator, 33
logical, 65
operators
addition, 35
arithmetic, 35
assignment, *see* asignment operator33
C, 262
compound assignment, 40
division, 35
increment, 40
integer division, 37
multiplication, 35
negation, 35
string concatenation, 37
subtraction, 35
optional parameters, 145
order of precedence, 38
logic, 73
overflow, 39
overloading, 483

parse, 179
pass by reference, 297, 423
pass by value, 423
paths, 184
absolute, 184
relative, 184
persistence, 183
PHP
arrays, 547
conditionals, 515
error handling, 543
file I/O, 563
functions, 535

loops, 527
magic method, 571
recursion, 577
strings, 557
pointers, 295
pollute the namespace, 33
polymorphism, 374
preprocessor directive, 255
primitive types, 30
printf, 43
problem solving, 2
procedural abstraction, 134, 301
program stack, 137
pseudocode, 13, 50

Quick Sort, 227

recursion, 203
C, 357
in Java, 479
in PHP, 577
reentrant, 334
regression testing, 48
runtime errors, 47

scope, 32, 134
global, 137
local, 136
search
binary search, 213
linear search, 212
search algorithm
analysis, 215
searching, 211
segmentation fault, 264, 296
Selection Sort, 221
analysis, 222
sequential control flow, 13
set, 167
in Java, 446
sets, 167
shallow copy, 166, 353
short circuiting, 74
software, 4
software testing, 48
Sorting
C, 374
sorting, 211, 220
Heap Sort, 237
Insertion Sort, 224
Merge Sort, 232
Quick Sort, 227
Selection Sort, 221
stability, 243
Tim Sort, 237
stack overflow, 162, 345
standard input, 42
standard output, 42
static array, 160
static typing, 31
static vs dynamic memory, 162
strings, 27, 177
comparison, 178
in C, 325
in Java, 449
in PHP, 557
tokenizing, 179
structures, 341
arrays, 348
defining, 341
style guide, 20, 22
syntactic sugar, 40
syntax error, 13, 47

tail recursion, 205
tautology, 72
ternary if-else operator, 82
test case, 49
test cases, 48
testing, 3
Tim Sort, 237
top-down design, 3, 133
transpiler, 12
truncation, 36
try-with-resources, 458
two's complement, 24
type casting, 36, 263
type juggling, 505

unbuffered, 187

underflow, 39
underscore casing, 20
unicode, 29
upper camel casing, 20
use case, 3

vararg function, 145, 424
variable, 18
    identifier, 18
    naming conventions, 19
    naming rules, 19
    scope, 32
    types, 22
variable argument function, *see* vararg function
variables
    C, 260
visibility, 197, 424, 462

while loop, 97

Bibliography

[1] Mars climate orbiter. http://mars.jpl.nasa.gov/msp98/orbiter/, 1999. [Online; accessed 17-March-2015].

[2] Moth in the machine: Debugging the origins of 'bug'. Computer World Magazine, September 2011.

[3] errno.h: system error numbers - base definitions reference. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/errno.h.html, 2013. [Online; accessed 13-September-2015].

[4] ISO/IEC 9899 - Programming Languages - C. http://www.open-std.org/JTC1/SC22/WG14/www/standards, 2013.

[5] Java platform standard edition 7. http://docs.oracle.com/javase/7/docs/api/, 2015. [Online; accessed 10-February-2015].

[6] List of software bugs. https://en.wikipedia.org/wiki/List_of_software_bugs, 2015. [Online; accessed 12-September-2015].

[7] Unchecked exceptions — the controversy. https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html, 2015. [Online; accessed 15-September-2015].

[8] Douglas Adams. *Dirk Gently's Holistic Detective Agency*. Pocket Books, 1987.

[9] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. *Softw. Pract. Exper.*, 23(11):1249–1265, November 1993.

[10] Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html, 2006.

[11] Michael Braukus. NASA honors apollo engineer. NASA News, September 2003.

[12] IEEE Computer Society. Portable Applications Standards Committee, England) Open Group (Reading, Institute of Electrical, Electronics Engineers, and IEEE-SA Standards Board. *Standard for Information Technology: Portable Operating System Interface (POSIX) : Base Specifications*. Number Issue 7 in IEEE Std. 2008.

[13] Edsger W. Dijkstra. Why numbering should start at zero. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html, 1982. [Online; accessed September 25, 2015].

[14] Bruce Eckel. *Thinking in Java*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 2005.

[15] Internet Goons. Do i cast the result of malloc? http://stackoverflow.com/questions/605845/do-i-cast-the-result-of-malloc. [Online; accessed September 27, 2015].

[16] Arend Heyting. *Die formalen Regeln der intuitionistischen Logik*. Berlin, 1930. First use of the notation ¬ as a negation operator.

[17] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[18] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[19] IEC. *IEC 60559 (1989-01): Binary floating-point arithmetic for microprocessor systems*. 1989. This Standard was formerly known as IEEE 754.

[20] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. August 2008.

[21] Donald E. Eastlake III, Carl-Uno Manros, and Eric S. Raymond. Etymology of "foo". *RFC*, 3092:1–14, 2001.

[22] ISO. *ISO 8601:1988. Data elements and interchange formats — Information interchange — Representation of dates and times*. 1988. See also 1-page correction, ISO 8601:1988/Cor 1:1991.

[23] John McCarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[24] Poul-Henning Kamp. The bikeshed email. http://phk.freebsd.dk/sagas/bikeshed.html. [Online; accessed September 12, 2016].

[25] Brian W. Kernighan. *Programming in C – A Tutorial*. Bell Laboratories, Murray Hill, New Jersey, 1974.

[26] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[27] Donald E. Knuth. Von neumann's first computer program. *ACM Comput. Surv.*, 2(4):247–260, December 1970.

[28] Tony Long. The man who saved the world by doing ... nothing. Wired, September 2007.

[29] M. V. Wilkes, D. J. Wheeler and S. Gill. *The preparation of programs for an electronic digital computer, with special reference to the EDSAC and the use of a library of subroutines*. Addison-Wesley Press, Cambridge, Mass., 1951.

[30] United States Government Accountability Office. COLLEGE TEXTBOOKS: Students have greater access to textbook information (gao-13-368), 2013.

[31] Cyril Northcote Parkinson. *Parkinson's Law, or the Pursuit of Progress*. John Murray, London, 1958.

[32] Richard E. Pattis. Textbook errors in binary searching. In *Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '88, pages 190–194, New York, NY, USA, 1988. ACM.

[33] Giuseppe Peano. Studii de Logica Matematica. In *Atti della Reale Accademia delle scienze di Torino*, volume 32 of *Classe di Scienze Fisiche Matematiche e Naturali*, pages 565–583. Accademia delle Scienze di Torino, Torino, April 1897.

[34] Tim Peters. [Python-Dev] Sorting. Python-Dev mailing list, https://mail.python.org/pipermail/python-dev/2002-July/026837.html, July 2002.

[35] Bertrand Russell. The theory of implication. *American Journal of Mathematics*, 28:159–202, 1906.

[36] Bruce A. Tate. *Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages*. Pragmatic Bookshelf, 1st edition, 2010.

[37] Alfred North Whitehead and Bertrand Arthur William Russell. *Principia mathematica; vol. 1*. Cambridge Univ. Press, Cambridge, 1910.

[38] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.