Teaching the Tapestry

An Instructor's Manual to Accompany A Computer Science Tapestry

David W. Reed

Department of Mathematics and Computer Science
Dickinson College
Carlisle, Pennsylvania USA
# Teaching the Tapestry

## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. INTRODUCTION</td>
<td>7</td>
</tr>
<tr>
<td>CONTENTS OF THIS MANUAL</td>
<td>9</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>9</td>
</tr>
<tr>
<td>2. TEACHING TIPS</td>
<td>10</td>
</tr>
<tr>
<td>FEATURES OF THE TAPESTRY</td>
<td>10</td>
</tr>
<tr>
<td>CHAPTER BY CHAPTER TIPS</td>
<td>13</td>
</tr>
<tr>
<td>Chapter 1 Tips</td>
<td>14</td>
</tr>
<tr>
<td>Chapter 2 Tips</td>
<td>15</td>
</tr>
<tr>
<td>Chapter 3 Tips</td>
<td>18</td>
</tr>
<tr>
<td>Chapter 4 Tips</td>
<td>20</td>
</tr>
<tr>
<td>Chapter 5 Tips</td>
<td>23</td>
</tr>
<tr>
<td>Chapter 6 Tips</td>
<td>26</td>
</tr>
<tr>
<td>Chapter 7 Tips</td>
<td>31</td>
</tr>
<tr>
<td>Chapter 8 Tips</td>
<td>32</td>
</tr>
<tr>
<td>Chapter 9 Tips</td>
<td>36</td>
</tr>
<tr>
<td>Chapter 10 Tips</td>
<td>38</td>
</tr>
<tr>
<td>Chapter 11 Tips</td>
<td>40</td>
</tr>
<tr>
<td>Chapter 12 Tips</td>
<td>42</td>
</tr>
<tr>
<td>3. QUIZ/REVIEW QUESTIONS</td>
<td>47</td>
</tr>
<tr>
<td>Chapter 1 Questions</td>
<td>48</td>
</tr>
<tr>
<td>Chapter 2 Questions</td>
<td>56</td>
</tr>
<tr>
<td>Chapter 3 Questions</td>
<td>69</td>
</tr>
<tr>
<td>Chapter 4 Questions</td>
<td>77</td>
</tr>
<tr>
<td>Chapter 5 Questions</td>
<td>87</td>
</tr>
<tr>
<td>Chapter 6 Questions</td>
<td>104</td>
</tr>
<tr>
<td>Chapter 7 Questions</td>
<td>117</td>
</tr>
<tr>
<td>Chapter 8 Questions</td>
<td>120</td>
</tr>
<tr>
<td>Chapter 9 Questions</td>
<td>131</td>
</tr>
<tr>
<td>Chapter 10 Questions</td>
<td>141</td>
</tr>
<tr>
<td>Chapter 11 Questions</td>
<td>147</td>
</tr>
<tr>
<td>Chapter 12 Questions</td>
<td>154</td>
</tr>
</tbody>
</table>
4. SAMPLE ASSIGNMENTS

ASSIGNMENT 1: SIMPLE PROGRAMS AND CLASS USE 169
ASSIGNMENT 2: RANDOM SENTENCE GENERATION 174
ASSIGNMENT 3: BINARY REDUCTION GUESSING GAME 186
ASSIGNMENT 4: MARBLE JAR PUZZLES 194
ASSIGNMENT 5: RANDOM WALL WALKS 205
ASSIGNMENT 6: SPELL-CHECKING 211
ASSIGNMENT 7: HUNT THE WUMPUS 218
ASSIGNMENT 8: SORTING, RECURSION, AND EFFICIENCY 232
ASSIGNMENT 9: PIXEL-MAPPED GRAPHICS 242
ASSIGNMENT 10: LINKED LISTS AND QUEUEING SIMULATIONS 256

5. SAMPLE TESTS

TEST 1 (CHAPTERS 1 - 5) 269
TEST 2 (CHAPTERS 6 - 8) 278
TEST 3 (CHAPTERS 9 - 12) 286
1. Introduction

This manual is designed to accompany A Computer Science Tapestry, by Owen L. Astrachan (McGraw-Hill, Inc., 1996, ISBN 0-07-002036-1), and to provide resources to assist the instructor using the text. The Tapestry was developed at Duke University and used in the introductory courses there, and has since been adopted at numerous colleges and universities. The text assumes no prior programming experience on the part of the student, and so is applicable in a wide variety of settings.

The Tapestry has several attractive features that distinguish it from other CS 1 texts. First of all, it is an introduction to computer science that uses C++, not a C++ text. This emphasis is significant, since the full breadth and depth of C++ is too much to present in an introductory course. Instead, the text focuses on problem solving and programming, with C++ as the language of expression. Advanced features of the language which are not instrumental to problem solving at the introductory level are avoided. Certainly, a course using the Tapestry should produce students that are reasonably adept C++ programmers, but learning every nuance of the language is not the focus.

There are three main goals behind the Tapestry. The first is to provide the student with a strong grounding in design and analysis. Probably the most difficult thing about programming for beginners is just knowing how to get started. Being able to understand a problem and see the structure of its solution is a considerable hurdle for novice programmers. The Tapestry takes an apprenticeship approach to programming which addresses this difficulty. Students first learn by studying and enhancing existing programs and problem solving approaches. Then, after picking up the patterns of problem solving, they are better prepared to design programs of their own. Once a student writes a program, however, it is important to enforce the idea that a program is a means to an end, not an end unto itself. The text stresses the use of programs as tools for analyzing phenomena, such as the expected distance of a random walk or the letter frequencies in works of literature. Hopefully, a student will develop stronger analytical skills after taking this course, and will recognize the computer as a powerful tool in simulating and analyzing complex phenomena.

The second goal of the Tapestry is to develop and hone the problem solving skills of the student. Unlike some texts which introduce applications to illustrate features of the language, the Tapestry takes the opposite approach. Applications are presented which motivate the introduction of problem solving approaches and language features. This application-driven approach allows for interesting and challenging problems to be studied, and has the added benefit of introducing language features in a specific context. Good examples of this are the Computer Science Excursions scattered throughout the text, which take an in-depth look at complex problems such as the development of a Calendar class and the efficient manipulation of very large integers. When needed, new language features are introduced to assist in tackling these applications. By focusing on the problems and not the language, the intention is that the student will develop general problem solving skills, not techniques tied to a specific formalism.
Finally, the third goal of the Tapestry is to expose the student to a broad view of the field of computer science. Too often, students leave their first course with the impression that computer science equals programming (in much the same way that first-year math students equate mathematics with calculus). Through its use of applications and Biographical sketches of eminent computer scientists, the Tapestry attempt to give the student a flavor of the diversity of study within computer science, from scientific computing to theoretical computer science to encryption.
Contents of This Manual

This manual is intended to supply you, the instructor, with materials and resources that complement the text. In addition to standard materials such as sample assignments and tests, there are ideas and advice as to how you can integrate the course materials into your course, as well as an extensive library of short quiz/review questions and sample tests. The manual is divided into the following chapters:

Chapter 1 (which you are currently reading) gives an overview of the philosophy of the Tapestry, and outlines the contents of the manual.

Chapter 2 discusses some of the features of the Tapestry, and provides tips and suggestions concerning the use of the text and accompanying materials.

Chapter 3 contains an extensive library of short questions which can be used on quizzes and tests, or as review problems.

Chapter 4 contains ten sample programming assignments, which can be used in both open lab and closed lab settings.

Chapter 5 contains three sample tests, corresponding to the three main sections of the text.

Acknowledgments

The material in this manual was developed while teaching the introductory sequence of courses at Duke University and, more recently, at Dickinson College. While at Duke, I had the opportunity to work with Owen Astrachan as a colleague, and we spent endless hours arguing (amicably) CS 1 issues and computer science in general. His commitment to teaching and his ability to both entertain and challenge his students is remarkable. I certainly want to thank Owen for helping to shape my approach to teaching.

At Dickinson College, a small liberal arts college in Carlisle, Pennsylvania, I have been pleased to find an academic atmosphere that both expects and encourages innovation in teaching. Throughout the curriculum here, I have been able to build on materials originally developed at Duke and experiment with new approaches as well. I would especially like to acknowledge two people here, Craig Miller and Allan Rossman, who have made teaching and being a part of the department at Dickinson rewarding and enjoyable.

Finally, I would like to acknowledge the people who served as subjects in the testing of these materials, the students I have taught both at Duke and at Dickinson. Student feedback has been instrumental in shaping these materials, and in the way I approach teaching in general. And, of course, I would like to thank my wife Laura who has suffered the most, having to share me
with my classes and having to listen to endless stories about lectures that worked well or didn't. Sorry.

2. Teaching Tips

This chapter contains advice on how to fully utilize the features of the Tapestry in your class. Like all advice, it can be ignored and your course can be taught exactly as you see fit. Hopefully, though, you will find some of the experiences that I have had (as well as ideas I have borrowed from others) useful in integrating the text in your course. Teaching tips range from a discussion of the different features of the Tapestry and how they can be utilized, to specific lecture approaches and exercises that can be used to reinforce course concepts.

Features of the Tapestry

In addition to the explanatory text and multitudes of sample programs, the Tapestry includes several recurring features which can be used to enhance the understanding of programming concepts and an overall picture of computer science. By stressing some of these features (and perhaps de-emphasizing others), the focus of the course can be tailored to your particular needs. For example, a broader view of computer science can be obtained by stressing the Biographical Sketches and the Computer Science Excursions in the text. Likewise, more depth in C++ programming can be obtained by assigning more Pause and Reflect Exercises and Chapter Exercises. The recurring features in the text are discussed below, with a brief description on how they can be integrated into the course.

Biographical Sketches

As was discussed in the introduction, one of the goals behind the Tapestry was to provide students with an introductory programming course which showed them more about computer science than just programming. One way of exposing students to the breadth of computer science is by introducing them to some of the influential figures in the field. Scattered throughout the text are brief biographies of well-known and respected computer scientists. By reading these biographies, students are exposed to some of the history and underlying methodology that links computer science researchers together. Perhaps more importantly, however, the students are shown the great breadth of research that comprises the field, from theoretical foundations to programming to computer ethics, and everywhere in between.

Stumbling Blocks
When learning to program, there are common mistakes that students tend to make, and certain concepts that tend to be confusing. For example, confusion between the assignment operator \(=\) and the equality operator \(==\) can lead to hard-to-locate bugs in a program. Similarly, variable aliasing via reference parameters can be unclear to beginning programmers. The Stumbling Block icon is used to highlight points of potential confusion, and to provide additional information to help the students. Advice for avoiding common mistakes is given, as well as examples which further demonstrate important concepts. Paying close attention to these stumbling blocks can help solidify the students' understanding of the material, and often avoid confusion down the road.
Pause and Reflect Exercises

Learning to program is not a passive activity. Students must be engaged so that they reason about what they are reading, and try out ideas as they go. As such, each section in the text is followed by a collection of Pause and Reflect exercises. As their name suggests, these exercises are designed to make the student pause and reflect on the material before moving on to the next section. The exercises may be conceptual, e.g. describe why procedures/functions are useful, or concrete, e.g. write or modify small code segments. Some questions focus on the programming environment, directing the student to explore their C++ implementation.

The Pause and Reflect exercises are much more than optional review questions, and their importance cannot be over-emphasized. Many important concepts and language details are first presented as interactive Pause and Reflect exercises. For example, the idea that formal and actual parameters must have matching types is first introduced in a Chapter 2 Pause and Reflect exercise. Students should be strongly encouraged to attempt these exercises as they read the text. One way to do this is to include Pause and Reflect questions on quizzes and tests.

Case/Class Studies

For the most part, code that is presented and discussed in the text is in the context of complete, working programs. In this way, students can take the working programs (which are publicly accessible at http://www.cs.duke.edu/~ola/book.html) and experiment, testing their understanding on the actual code. This is in contrast to many texts which provide small code segments that must be embedded in larger programs before being executed. In addition, Case Studies and Class Studies are scattered throughout the text, providing in-depth studies of the step-by-step development of programs and classes. The use of case studies to aid students in understanding the process of problem solving has been stressed by many influential educators in recent years (with Mike Clancy at the forefront), and has been adopted by the Educational Testing Service on its Advanced Placement exams for computer science.

Chapter Reviews

At the end of each chapter, the main concepts and structures presented in the chapter are summarized. The short, bullet-point format of these reviews serves as a useful reference for students when programming or studying the material. In addition, the chapter reviews can be used by students to help gauge their understanding of the material. After reading the chapter, each student can compare his or her understanding with the highlighted points. Any bullet-point that is not fully understood after reading the chapter should be studied further before continuing in the text.

Exercises
Each chapter (and computer science excursion) in the text concludes with exercises which reinforce the material presented. These problems are resources that can be utilized in numerous ways.

- Students may solve these problems on their own to review the material and test their understanding.

- The instructor may use them as examples to be discussed in class. Because students learn more from examples that they have worked on themselves, it can be beneficial to notify the students as to which exercises will be discussed and have them attempt to solve them on their own first.

- The problems may be used for group work in class. Periodically, I like to divide the class into small groups (2 - 4 students each) and have them solve problems as a group. A group of students is generally able to solve more challenging problems than individuals since the students help each other and discuss different approaches. Working in a group of their peers also tends to draw out less confident students who would be reluctant to speak up in class. For the instructor, it can be very informative to observe the thought processes of the students as they attempt to apply their knowledge to the problems.

- Some of the exercises may be used as homework assignments or test questions.

Excursions

Many of the chapters in the text end with a Computer Science Excursion which delves into some topic in computer science. Like the case and class studies, these excursions allow for the study of deeper topics, with time spent understanding and analyzing the motivation and development of the code. The subject matter of the excursions tends to be much broader than those of the case and class studies, focusing on general topics in computer science such as algorithm complexity, embedded real-time systems, and number representation. The excursion topics are self-contained, so omitting them will not deprive your students of C++ and programming skills that are used later in the text. However, excursions do provide familiarity with interesting topics and reinforce programming concepts that are first presented elsewhere.

Chapter by Chapter Tips

The Tapestry is divided into three main sections (after a brief introductory chapter). Chapters 2 through 5 present the basic concepts of programming, including functions, variables, conditional, and loops. As stated earlier, the focus is on using C++ as a language for problem solving, so programming constructs are generally motivated by examples first. Classes are used throughout these chapters, but mainly as abstract data types to be used but not studied in great detail. Chapters 6 through 8 focus on class design and implementation, and the structuring of
data using Vectors. Depending on your particular goals, you may find some sections of these chapters optional. For example, if you are comfortable with having your students work exclusively with the provided Vector class, you can skip the discussion of arrays altogether. Chapters 9 through 11 build on the basics to study more advanced topics in problem solving, programming, and analysis. You may wish to pick and choose from these chapters as time allows. For example, you may wish to put off recursion or sorting for a later course and focus more time on linked lists.

The following sections contain tips and advice for integrating the Tapestry into your course. Advice can range from highlighting important topics, to pointing out potential difficulties, to suggesting approaches and examples for your lectures. The advice and ideas described come from my experiences teaching this material both at Duke University and at Dickinson College, as well as from other instructors (most notably, Owen Astrachan at Duke and David Levine at Gettysburg College). Additional tips and resources can be found at Owen's web page: http://www.cs.duke.edu/~ola/book.html.

**Chapter 1 Tips**

The main goal of chapter 1 is to provide a quick overview of computer science and a context for programming. As has been stated numerous times, one of the themes of the Tapestry is that computer science is more than just programming. The tapestry metaphor, that computer science weaves together different subfields into a broad discipline of study is stressed, along with an appreciation of the role that algorithms play in computer science.

Learning how to program is first and foremost about learning how to organize your thoughts and specify step-by-step solutions to problems. Whether that solution is written in English as an algorithm or coded into C++ for execution on a computer, the process is pretty much the same. Stressing this process before the students start looking at code can start them out with the right attitude towards problem solving. I joke with my students that "the earlier they start coding a program, the later they will finish." Sitting down at a computer and trying to write code from scratch is not the way for beginners to learn to program. Taking the time to first think about the problem and sketch out an algorithm before coding will usually be more productive. Several of the exercises at the end of the chapter ask the students to think about specific problems and to sketch out algorithms. These exercises can be assigned as homework or solved as part of in-class discussions to get students into an algorithmic state of mind early.

Another important concept that is discussed at the end of this chapter is that of abstraction. One of the goals of object-oriented programming is to make the development of programs easier by encouraging code reuse. A library of code, once written and tested, can be used over and over with virtually no cost. Throughout the text, students will use predefined classes, sometimes without any deep understanding of the actual implementations of these classes. By dealing with these classes abstractly, ignoring their implementation details, the students will be able to solve more interesting and complex problems than if they were just working with the bare C++ primitives.
As with any course, it is important that the students understand the course objectives and expectations right away so that they can determine if this course is really for them. Assigning this chapter as reading at the first class, and spending a day (or, at most, two) on the main ideas, should provide a fair picture of the conceptual content of the course. The course expectations, the number and types of programming assignments, and the testing criteria used in the course, clearly need to be presented by the individual instructor.

Chapter 2 Tips

The main goal of chapter 2 is to present the basics of C++ programming, and to stress the importance of functions as units of abstraction. The Tapestry takes a different approach from many introductory texts in that it introduces functions and parameters early, even before it discusses variables and assignments. This ordering emphasizes the importance of abstraction, and also makes it possible for students to start writing programs that produce tangible results right away.

Programs like the one for drawing heads out of interchangeable parts demonstrate clearly how the use of functions as units of abstraction makes it easier to reuse pieces of code. In addition, this program is fun, allowing for some creativity early in the course. A good extension to this example is to consider how functions simplify modifications to the code. Suppose you have written several functions for drawing different heads (as in the Pause & Reflect exercises). If you then wished to change the width of heads, the use of functions would make such a modification easier. You would only need to modify the functions which draw the head parts, e.g., Hair, Sides, Eyes, Ears. The higher-level functions which draw the actual heads would remain unchanged.

The head-drawing functions do present a style inconsistency in the text. You may have noticed that the headers for these functions are split across two lines, with the return type (void) on one line and the function name with parentheses on the next line. This style is not continued through the rest of the text, where the more standard convention of listing the return type on the same line with the name is followed. This is not a very serious issue, and you may use these as examples of how whitespace is for the most part irrelevant in C++. Or, you may choose to present these examples with the return type on the same line for consistency's sake.

On another style issue, some people refer to void functions, i.e., functions that do not return a value, as procedures. This no doubt goes back to Pascal and similar languages which actually had both types of module. If you wish, you can make this distinction and refer to the void functions in this chapter as procedures. However, this distinction is somewhat artificial, and may lead to some confusion on the part of the students. I have found that students do not have trouble with the idea of a function returning no value (or else they think of it as returning some default, "undefined" value). Not introducing the term "procedure" means that there is one less term the students need to remember and relate to the rest of the language.

Another advantage of studying functions before variables is that it provides simple examples for tracing program execution. Since the additional complexity of expressions and assignments
is not added to the mix, I have found that students have very little trouble with the process of a function call. The idea that control transfers to the function body when that function is called, and then returns back to the calling point, is demonstrated nicely in the various versions of the Hello World program and the head-drawing program. In particular, the fact that Hello World is first developed using only main, and then an equivalent program using functions is developed, helps the student to visualize the control process.

The use of parameters is motivated very nicely by the Old MacDonald program. Here, you have a program that can first be written using only main, but the duplication of code is horrendous. If you wished to change the order of verse in the song, you would have to move large sections of code inside main. Similarly, if you wished to add new verses, you would have to add new sections of code, very similar to existing verses. Separating code into functions can help to simplify main and make changing the order of verses easier. If you defined a separate function for printing each verse, e.g., CowVerse, PigVerse, DuckVerse, then main could just contain a sequence of calls to these functions. Changing the order of verses would simply require changing the order of the function calls. Using parameters, however, the program can be simplified even further. A single parameterized Verse function can be used in place of the separate functions (CowVerse, PigVerse, etc.). By going through these development steps, students see how a parameterized function represents a class of functions, a different function for each set of arguments.

Many important details about parameters are first presented in the discussion of the Old MacDonald program. In particular, the importance of order in matching arguments with parameters is demonstrated with an example. The fact that Verse("pig","oink") and Verse("oink","pig") produce very different verses is important to understand. Since parameters are covered before variables in the text, it is worthwhile to step through several examples, stressing how the value of the argument is passed in and stored in the parameter. If the students are comfortable thinking of a parameter as a memory cell that stores a copy of the argument, then they will have a good basis for understanding variables and even reference parameters in later chapters. You might also consider discussing or assigning the Pause & Reflect exercises referring to Old MacDonald (2.25 - 2.31). These exercises address important questions such as what happens if you specify the wrong number of arguments in a function call, or the wrong type of argument.

The discussion on style at the end of this chapter is as much for the benefit of the instructor as the students. If you want your students to develop good habits with respect to program design and presentation, you need to set a good example. Students learn good style by studying code that you present to them. If you present examples of code with bad variable names or inadequate documentation, then don't be surprised when students hand in assignments at that same standard. I give my students style guidelines early and augment them throughout the course, making sure that they understand that a consistent and readable style is important (and will be graded as such). My general motto is that they should code for "readability first, efficiency second." As a result, they should do everything possible to make code understandable, even to someone who may not know the purpose of the code ahead of time. In particular, they should:

• use meaningful identifiers
I do not generally give strict rules such as the number of spaces to indent inside a function, but I do stress that they find a readable style that they are comfortable with, and be consistent.

At some point in this chapter, you will probably want to take the time to introduce your students to their particular programming environment. Since all of the programs discussed in the Tapestry can be downloaded from an ftp site, it is good idea to make these programs available to the students for their experimentation. If you are using the string class defined in the text (as opposed to the standard string class if available), then you will be forced to talk about multiple-file compilation and linking right away. For example, to compile and run the Old MacDonald program, you will need to also compile and link in the file CPString.cc. If you are using Turbo or Borland C++ on a PC, or Symantec or CodeWarrior on a Mac, then you will need to build a project. In each of these environments, you create a project and add each of the required .cc files (e.g., oldmac2.cc and CPString.cc). Then, when you run the program, each of the .cc files is compiled and the resulting object files are linked into an executable. The students can be insulated from much of this detail by precompiling the code from the text into a library, which can then be linked in to every program that they write. In this way, students will not need to link specific files from the text, but can instead link in this library each time. Steps for setting up a library differ for each environment. See Owen's web page for tips: http://www.cs.duke.edu/~ola/book.html.

In a UNIX environment, there are several options for compiling and linking multiple-file projects. One is to have the students manually compile and link all of the files. For example, using the g++ compiler, students can create an executable called oldmac using the following command:

```bash
g++ -g -o oldmac oldmac2.cc CPString.cc
```

Here, the -g option includes debugging information (in case you wish to have them use a debugger such as gdb), the -o option specifies the name of the executable file to be created (here, oldmac), and then each of the .cc files to be compiled is listed. Alternatively, you may want to introduce the idea of a Makefile, which lists all of the dependencies between files. In addition to making compilation easier, a Makefile also makes compilation smarter - if a source file has been compiled since last modified, then it will not be recompiled again. A Makefile with entries corresponding to the Old MacDonald program would be:

```bash
# set up compiler and options
CC = g++
CFLAGS = -g

# set up c++ suffixes & relationship between .cc and .o
.SUFFIXES: .cc
.cc.o:
 $(CC) $(CFLAGS) -c $<
```

17
# list file dependencies & actual compilation command
CPString.o: CPString.h
oldmac2.o: oldmac2.h CPString.h

oldmac: oldmac2.o CPString.o
$(CC) $(CFLAGS) -o oldmac oldmac2.o CPString.o

If this text is stored in a file named `Makefile`, you can compile and link the Old MacDonald program simply by typing

```
make oldmac
```

Once again, it is possible to insulate the students from some of these details by precompiling the code from the text into a library. Then the students can use a generic `Makefile` that will compile any single-file program, automatically looking in the right place for header files and the compiled implementations. See Owen's web page for sample `Makefiles` which can be used for this approach: [http://www.cs.duke.edu/~ola/book.html](http://www.cs.duke.edu/~ola/book.html).

As is the case with all of the Computer Science Excursions, the excursion at the end of this chapter may be skipped. This particular excursion, however, has some very nice features which call for its inclusion in your course. Using only functions and `cout` statements, this excursion demonstrates that it is possible to produce complex results and study complex phenomena using a computer. This may be surprising and enlightening to students, in that they will be able to study an interesting and nontrivial problem right away. The excursion previews the topic of efficiency that is studied in greater depth later, and gives the impression from the start that computer science is more than just programming.

### Chapter 3 Tips

The main goal of chapter 3 is to introduce variables and expressions, and to present examples of the `Input/Process/Output` pattern of behavior found in many programs. This is a relatively short chapter, but there is a real danger of bogging down in the material. If you are not careful, you may find yourself explaining esoteric details such as the length of `ints` versus `long ints`, or the associativity of the `mod` operator. You do not want to get too detail-oriented at this point in the course, the students just don't have the context for it yet. As such, you should be careful to control the level of detail on topics such as these.

Concerning the different number types, you may want to describe why there are different integer and real number types (i.e., the tradeoff between storage size and range of representation). But, you should terminate any discussion with an unambiguous rule: if you need to store an integer value, use a variable of type `int`; if you need to store a real value, use a variable of type `double`.

Concerning the evaluation of expressions, students need to know about precedence and associativity rules since they will need to be able to understand why certain expressions are
evaluated the way that they are. For example, they should know that $1+2*3$ will evaluate to 7, not 9. However, spending too much time on an issue such as precedence encourages the students to make frequent use of these rules, which usually leads to errors or at least code that is difficult to read. I stress that the computer has no problem evaluating complex expressions using precedence and associativity rules, but people do. Thus, they should use parentheses to clearly specify the order of evaluations in any complex expressions that they write.

In many texts, variables are first introduced along with assignment statements. It is interesting to note that assignment statements are not even covered until the following chapter. Instead, variables are used here solely for storing values read in from the user. Once these variables have values read in, they are used in expressions that are immediately written as output. In this way, variables are similar to the parameters introduced in the previous chapter: they are names which refer to values (not changeable entities). This connection provides a context for understanding variables, which can then be extended in the next chapter.

The declaration of variables leaves open a style question that you should consider. C++ allows for variables to be declared anywhere within the code, as opposed to having to declare them at the top of the function (as was the case in Pascal, and to some degree in C). In the text, all variables inside a function are declared together at the top of the function. The advantage of this style is that it is uniform - if you ever want to look at the declaration for a particular variable, you can go right to the top of the function and find it. I personally prefer the style where you declare variables as close to their actual use as possible. The advantage of this style is that it is more flexible and allows for better "paragraphing" of the code. That is, you can divide your function into code segments, separated by blank lines, so that each segment performs a specific subtask. By declaring variables within the segment in which they are used, the "paragraphs" of code are relatively independent. Whichever style you choose, strive to be consistent.

No matter what style of declaration you use, you should certainly dissuade students from using global variables. In fact, the concept of declaring variables outside of functions is not even mentioned until much later in the text (under very specific circumstances). Students with some programming experience may know about global variables and be tempted to use them, however. I have a general rule concerning variables: every variable that appear in a function should be either (1) a parameter to that function, or (2) declared inside that function (but not both). By applying this rule to every function that they write, the students will never be tempted to use global variables. Ensuring that the same name is not declared as both a parameter and a variable also avoids problems that can occur because of this subtle error.

The last part of this chapter concerns familiarizing the students with classes. Having the students at least understand the ideas behind a class early has several advantages. The most important one is that it allows for more interesting exercises and assignments. At this point, the students only know about functions, variables, and input/output. There is a limit to the number of interesting and challenging things that they can do with these simple tools. By providing them with predefined classes, however, the options are unlimited. The Balloon class is a good example of this: a simple program which creates a Balloon object and makes several member function calls can produce a large amount of complex and unpredictable output.
The intuition that I push with my students is that a class is nothing more than a user-defined type. I start by asking them to describe what an integer is. With some prodding, they usually come to the realization that an integer is more than just a whole number, it is also an entity that you can apply operations to (e.g., addition, multiplication, comparison). Similarly, they are comfortable with the idea that a string is a sequence of letters along with some operations (e.g., find the length, access a character). In general, a type is a collection of data along with operations that can be applied to that data. When you define a class, you are simply defining a new type that extends the C++ language. Once that type is defined, you can declare variables of that type and apply the appropriate operations, just like any other type.

Studying the details of the Balloon class is not really necessary at this point, what is important is specifying the interface. As long as the students know that Balloon is a new type, and they know how to call its member functions (i.e., apply the operations), then they should be able to write programs that manipulate Balloons. From there, you can introduce other classes in your lectures and in assignments that hide unnecessary details and yet allow the students to solve interesting problems right away. It should also be noted that Susan Rodger at Duke has developed a nice graphical implementation of the Balloon class, where the student can actually see a balloon rise and descend on the screen. Running under X-windows, this and other similar extensions to classes can make the introduction of classes even more interesting to students.

Chapter 4 Tips

The main goal of chapter 4 is to begin focusing on more complex problems that require state (i.e., assignments) and conditional execution (i.e., if-else statements). Since the students have already seen variables and expressions in earlier contexts, the concept of an assignment should not be that difficult. Conditional execution, on the other hand, can sometimes be difficult to understand for beginning students. The fact that you can interrupt the sequential execution of code often requires numerous examples to make students truly comfortable.

The most important feature to stress about assignments is that they involve a right-to-left directionality. The expression on the right-hand side of the assignment is evaluated first, and then the resulting value is assigned to the variable on the left. If this point is not made clear, then assignments in which the same variable appears on both sides may be confusing. For example, the assignment \( x = x + 1 \) makes no sense mathematically if you interpret the equals sign as equality. However, there is no contradiction if you first evaluate the right-hand side (add one to the current value of \( x \)), and then assign that value to the left-hand side (make it the new value of \( x \)).

You should also stress the distinction between assignments (\( = \)) and equality (\( == \)). When you read an assignment statement out loud, never use the word equals (such as "\( x \) equals \( x + 1 \)"). Instead, say "\( x \) gets \( x + 1 \)" or "\( x \) is assigned \( x + 1 \). This will help to avoid confusion between the two.

Before introducing any new control statement, it is important to motivate the need for that statement. In this way, the students have a context in which to place the new tool. When
motivating if statements, have the students think of algorithms that involve choices. In the text, modifying the change program change.cc so that it only prints out coins that are used involves choices: for each coin, only print out its count if it is not zero. To assign letter grades based on student averages involves choices: if the average is 90 or greater, then they earn an 'A', 80 or better earns a 'B', and so on. Most algorithms involve choices, sometimes between only two alternatives (as with the coins) and sometimes between many (as with the grades). Given this context, the students are then ready to study how these choices can be implemented in C++.

When writing if statements, there are many presentation styles that have been proposed. By definition, an if statement is of the form

```cpp
if (test) statement;
```

A block statement (one or more statements inside braces) is only necessary when more than one statement is to be executed as a result of the if test. However, many parsing errors (e.g. dangling else) can be avoided if you always use braces as part of an if statement. In my courses, I never show the students an if statement that does not use braces. In the text, the opening and closing braces are each placed on separate lines, left justified to line up under the if. Another common style (which I tend to use) is to place left brace on the same line as the if test. These two styles are illustrated below:

```cpp
if (test) {  
  statement list;
}  

if (test) {  
  statement list;
  statement list;
}
```

Note that in both of these styles, the right brace lines up under the if and the body of the if statement is indented. This makes it easy to see where the statement begins and ends. The advantage of the first style is that it is symmetric and consistent with the presentation of functions. The advantage of the second style is that it is one line shorter, without sacrificing readability. Either of these styles is acceptable, but you should strive to be consistent.

When constructing and studying if statements, you should stress the true/false abstraction concerning the tests. Conceptually, the relational operators return a true/false value. If the test in an if statement evaluates to true, then the statements inside the brackets are executed. In reality, there are some flaws in the implementation of the abstraction, such as the fact that boolean values are displayed as integers, and integer values can be used in if tests (with zero representing false, nonzero representing true). While you can mention these details, stress the true/false abstraction. It is much more natural, and you don't really want students making use of integers in tests anyway.

Students generally do not have a hard time with tests involving logical operators. They are used to using "and" and "or" in everyday sentences. The concept of short-circuit evaluation is new to them, however, and so requires some discussion. Stress both the efficiency and safety advantages of short-circuit evaluation. In terms of efficiency, evaluating from left-to-right and stopping whenever the final value can be determined may save unnecessary work. For example, if you are evaluating the test `(A && B)` and `A` evaluates to false, then there is no point in evaluating `B`. Since you know that the final value must be false ("false and anything" is false),

21
you can return this value without ever evaluating B. In terms of safety, the left-to-right order of evaluation can be taken advantage of to avoid run-time errors. For example, consider the following test: (numScores > 0 && scoreTotal/numScores > 0.90). If numScores is zero, then the first part of the test will evaluate to false, resulting in a short-circuit. If the order of the two tests were reversed, however, a run-time error (division by zero) would ensue. By placing the tests in this particular order, you know that the second test will only get evaluated if the number of scores is nonzero.

Once students are comfortable with the idea of a function as a unit of abstraction, they generally do not have much trouble with functions that return values. The text first discusses mathematical functions that are defined in the math.h library, since these are the most natural examples. It then proceeds to develop simple functions such as a boolean function that determines if a particular year is a leap year. There are numerous other examples that you can use at this point:

- **bool IsEven(int num)**
  // postcondition: returns true if num is divisible by two

- **bool IsIncreasing(int num1, int num2, int num3)**
  // postcondition: returns true if num1 <= num2 <= num3

- **string DayOfWeek(int num)**
  // postcondition: returns day of week where
  1 --> "Sunday", 2 --> "Monday", ...

Some of the examples in this chapter have a property that may be disturbing to some of you. Functions such as IsLeapYear and NumToString contain multiple return statements, which is perfectly legal in C++. This was not possible in Pascal, and as such many people hold the view that a function should have only one exit point. It has been my experience that students are not confused by multiple exit points, and multiple returns often simplify the code considerably. For example, the in the NumToString function each case in the if-else statement has its own return statement. To avoid multiple returns, you would need to have a variable which was assigned a value in each of the if-else cases, and then return the value of that variable at the end of the function. This is more work (and not any more intuitive) than just returning the appropriate value as soon as it is determined.

At the very end of this chapter, the C++ string class is described in some detail, focusing on the member functions (operations) defined for this class. It is good to return to the idea of abstract classes after so much low-level detail about the C++ language. The string class will be used extensively throughout the text, so taking time to understanding and experiment with the operations on strings is worthwhile here.

The excursion for this chapter involves the use of recursion to perform exponentiation efficiently. Because this application involves so many new topics (induction and recursion, complexity, and experimentation using a Timer class), it represents a serious investment in time. For inexperienced students, you will probably want to skip this excursion altogether. If you have the time and relatively mature students, however, this excursion does provide a good preview of many important concepts that will be reemphasized later in the text.
**Chapter 5 Tips**

The main goal of chapter 5 is to attack more complex problems, in particular, those requiring repetition. For the first time, the actual implementation of a class is presented, and that class is then used in larger programs. The while loop is introduced as a control statement for conditional repetition, and other auxiliary control statements are introduced as well.

After using the *Balloon* class and *string* class, the students should be prepared to understand the design and implementation of a simple class. The class chosen in the text models dice, partly because they are natural and intuitive and partly because a *Dice* class is useful in many subsequent problems. When studying (or even designing) a class, always start with the .h header file. The header file describes the class abstractly, specifying the data fields and listing prototypes for all of the member functions (operations). By studying and understanding the header file, you know everything necessary to use the class. In particular, you know which member functions are public, and so can be called from a client program. You also know which functions and data fields are private, and so are invisible outside of the class. Clearly, it makes sense that someone rolling a die should not be able to change the number of sides on that die. By making this data field private, the integrity of that value is ensured. If you want to understand how the class is implemented (or to implement it yourself), you then need to delve into the .cc implementation file.

Since the *Dice* class is the only class used in the rest of the chapter (excluding for the excursion), you may choose to skim over the details of its implementation for now and focus on using *Dice* objects. Chapters 6 and 7 work through more examples of class design and implementation. If you would like for your students to start learning about how classes are implemented, there are numerous examples similar to the *Dice* class that can also be presented. Probably my favorite example for demonstrating classes is the *MarbleJar* class, which is used in Sample Assignment 4. Numerous statistical puzzles involve drawing marbles randomly out of a jar. For example, suppose you had a jar containing the same number of black and white marbles. If you drew two random marbles from the jar, would they more likely be the same color or different colors? A class which models a marble jar can be defined for simulating puzzles such as these.

```cpp
class MarbleJar
{
    public:
        MarbleJar(int black, int white); // initializes jar
        string DrawMarble();  // draws and returns color
        void AddMarble(string color);  // adds marble to jar
        bool IsEmpty();        // returns true if empty
    private:
        int myNumBlack, myNumWhite; // marbles in the jar
        RandGen myGen;            // random number generator
};
```

Here, the *MarbleJar* class provides the basic operations needed for these puzzle: drawing a random marble, placing a marble in the jar, and seeing if the jar is empty. Since these are the
only operations allowed, it is imperative that the number of marbles of each color be stored in a private data field. A person should not be able to draw or add more than one marble at a time, and certainly should not be able to peek into the jar.

As a disclaimer, I should note that this class is designed for simplicity and not robustness. For example, the `IsEmpty` member function really ought to be declared as a const function since it does not alter any fields of the `MarbleJar` object. As is, you cannot pass a `MarbleJar` object by const reference and then call the `IsEmpty` member function on that object. Similarly, it would be better to define an enumerated type for the color of marbles. As is, it is possible to pass a string other than "black" or "white" to the `AddMarble` member function. While these improvements in the code can be introduced later, their additional complexity is not warranted at this point in the course.

Classes like `Dice` and `MarbleJar` are especially useful for motivating while loops. Using dice, it is very natural to propose repetitive tasks, such as rolling two dice until you get a 7, or rolling the dice until you get consecutive rolls that are the same. Similarly, you can propose tasks involving a marble jar if you have taken the time to introduce this class. These examples have the advantage of not only motivating repetition, but also being fun. Another example that I use with while loops is the paper folding puzzle.

If you start with a piece of paper and fold it in half, it is twice as thick as before. If you fold that in half, it is then four times as thick as a single sheet. If you continue folding, how many times will you have to fold before the thickness of the paper reaches from the earth to the sun?

Starting with an initial estimate of the thickness of a piece of paper, this puzzle can be solved using a simple while loop which repeatedly doubles the thickness and continues until the thickness reaches 92,960,000 miles. Students are often amazed that the answer to this puzzle is approximately 51 folds (give or take a fold or two based on your initial thickness estimate). This example allows you to discuss just how fast an exponential rate of growth really is. This idea, and the converse idea that repeatedly halving a number decreases its size very quickly, are key ideas in searching and sorting. Presenting this idea under many different guises (the folding puzzle, the guessing game in Sample Assignment 3, the recursive exponentiation in Excursion 2) goes a long way towards impressing its importance upon the students.

The example in the text using Newton's method to approximate the square root of a number can have many uses in the lectures. While some students immediately roll up their eyes at the sight of mathematics, the fact that you can obtain arbitrarily precise approximations to the square root using this simple refinement formula is interesting to many students. There are also opportunities to talk about modifications to the algorithm. In the text, the refinement of the approximation stops when the relative error between the desired number and the square of the approximation is small enough. This is not the only stopping criteria possible, however. You might propose a stopping condition where the refinement ends whenever successive approximations are sufficiently close.

The square root example also is the application where constants are first introduced. Students should be encouraged to use constants whenever there is some value that will remain
unchanged throughout program execution, but which might change between executions. For example, the EPSILON distance might change if you wished to get greater precision in your square root approximations. Since constants cannot be accidentally changed during execution (the compiler makes sure that no assignments are made to a constant), it is safe to place all constants at the top of the program (i.e., make them global). This means that you do not need to pass these values to functions, and they are easy to find and change at the top of the file.

Concerning the alternate control statements described in this chapter, I am a firm believer in the idea that fewer is better. I do not stress switches since they are not as flexible as if statements, and a switch can be replaced by a series of cascading if-else statements without much trouble. Similarly, do-while loops are can usually be replaced quite easily with while loops, so I do not stress them (although I do use them occasionally). I do find for loops to be useful since they tend to be easier to read and less prone to error than their while loop equivalents. Since the initialization, test, and update code is all placed inside the parentheses, it is less likely that you will forget one of these parts when you implement a for loop. To avoid confusion as to which loop to use when, I make a simple rule: if you know how many repetitions ahead of time, use a for loop; if not, use a while loop.

The idea of nested loops can be difficult to beginners as they try to cope with the wheels-within-wheels effect. As a general rule, I warn my students to stay away from nesting while loops. Trying to keep track of when the inner loop ends and the outer one takes over can be overwhelming. Besides, applications that call for nested while loops are few and far between. Nested for loops, on the other hand, are relatively common. Tracing examples is really the only way to get across the behavior of nested loops. For example:

```c
for (i = 0; i < 5; i++) {
    for (j = 0; i < 5; j++) {
        cout << i <<", " << j << endl;
    }
}

for (i = 0; i < 5; i++) {
    for (j = 0; j < i; j++) {
        cout << i <<", " << j << endl;
    }
}
```

If you allow for variables to be declared anywhere in code, then you will need to be prepared to talk about scope. The idea that a variable declared inside a function is local to that function is not too difficult. What must be stressed, however, is that variable declarations are local to whatever block they are defined in. So, if you declare a variable inside a for loop, then that variable exists only inside that loop. For example, in the loop below

```c
for (i = 0; i < 100; i++) {
    ...
    int count = 0;
    ...
}
```
the variable count will be created and initialized during each loop pass. A related example is the following:

```c++
for (int i = 0; i < 100; i++) {
    ...
}
```

The C++ standard states that the scope of the variable i is the loop itself, so i will not exist outside of the for loop. This implies that you can have multiple for loops of this form, each with its own version of i. Older implementations of C++ may not support this feature and instead will consider the scope of the variable to extend beyond the loop. For this reason, code provided in the text does not declare variables in the for loop header.

The craps.cc program (Program 5.13) has some code which you may wish to avoid, or at least explain to your students. The default case in the switch involves rolling the dice until either a 7 or the player's point is obtained. To do this, an infinite while loop is used, with an if statement that allows for breaking out of the while loop. You may find that you prefer an alternate way of coding this loop. For example,

```c++
sum = RollTwo();
while (sum != point && sum != 7) {
    sum = RollTwo();
}
if (sum == point) {
    cout << "got your point! a winner" << endl;
} else if (sum == 7) {
    cout << "loser, lost going for " << point << endl;
}
```

The excursion for this chapter involves the development and use of a complex class for representing dates. This example shows the real power of classes to encapsulate very complex data and operations into a type that is simple to use. A full study of this class introduces many new concepts, including multiple constructors and operator overloading. Still, if you have the time it can be a worthwhile exercise in class use.

**Chapter 6 Tips**

The main goal of chapter 6 is to tackle problems involving large amounts of input and/or output. At this point in the course, the students have learned (or at least been exposed to) a wide variety of programming tools and techniques. In fact, I push the metaphor of a "programming toolbox", and as new language features and programming patterns are introduced, they are added to the toolbox. At this point, the students know about variables and constants, functions as units of abstraction, numerous control structures (if statements, if-else statements, while loops, for loops, etc.), and several programming patterns (fencepost problems, using variables as counters, sums, and flags, etc.). Of course, knowing when to use the right
tool for the right job is of the utmost importance, so I find that this is a good point to review the contents of the toolbox.

For example, I might provide the following toolbox organizer:

<table>
<thead>
<tr>
<th>TOOL</th>
<th>ADVICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td>each variable corresponds to a memory cell that can store a value use meaningful identifiers, no global variables</td>
</tr>
<tr>
<td>constants</td>
<td>use whenever you have a special value that may change from run to run global constants are OK since they can't be changed during execution</td>
</tr>
<tr>
<td>functions</td>
<td>use functions to break a program into meaningful and manageable pieces a function should perform one conceptual task only</td>
</tr>
<tr>
<td>parameters</td>
<td>use parameters to create general functions a parameter is like a local variable that stores a copy of the argument</td>
</tr>
<tr>
<td>if-else</td>
<td>use whenever execution is conditional, based on some test if more than two alternatives, use cascading if-else's</td>
</tr>
<tr>
<td>while loop</td>
<td>use when repetition is called for, based on some condition a do-while loop can be used, the loop test is evaluated at the end</td>
</tr>
<tr>
<td>for loop</td>
<td>use when repetition is called for, and the number of reps is known note that a for loop is really just new notation for a while loop</td>
</tr>
<tr>
<td>counter</td>
<td>a variable which keeps count of some event need to initialize to zero, then increment each time the event occurs</td>
</tr>
<tr>
<td>sum</td>
<td>a variable which keeps a running total of some values need to initialize to zero, then add each successive value to the sum</td>
</tr>
<tr>
<td>flag</td>
<td>a variable (usually a bool) which keeps track of some state need to initialize one way, then flip if the state changes</td>
</tr>
<tr>
<td>class</td>
<td>implements a new type encapsulates data and the operations on that data</td>
</tr>
</tbody>
</table>

After reviewing the contents of the toolbox, I like to stress how problem solving can be seen as knowing how to apply the appropriate tools to the problem at hand. To enforce this idea, I sometimes split the class into small groups (3-4 students per group), and present them with several problems to solve. These problems generally involve designing and encoding a function or code segment from scratch, so they must recognize which tools are appropriate for the given problem and integrate them into a solution. Since they are free to discuss ideas among their group, students are able to learn from each other and reinforce their own understanding of the concepts. When different groups take different approaches to solving a problem, this can also be instructive to discuss as a class. From the instructor's perspective, being able to wander from group to group and listen in on their thought processes can also be very informative.

Examples of the types of problems that I might propose are:
1. Complete the `multiples` function below, which takes a positive integer as an argument and writes out all integers between 1 and 100 that are multiples of that argument.

```c++
void multiples(int num)
// precondition : num > 0
// postcondition: writes all multiples of num between 1 and 100
```

2. Complete the `multimultiples` function below, which takes two positive integers as arguments and writes out all integers between 1 and 100 that are multiples of either argument (or both).

```c++
void multimultiples(int num1, int num2)
// precondition : num1 > 0, num2 > 0
// postcondition: writes all multiples of num1 and/or num2 between 1 and 100
```

3. Complete the `hailstone` function below, which takes a positive integer as an argument and writes out the hailstone sequence starting at that number. The hailstone sequence is a sequence of positive numbers defined as follows: if the number X is odd, then the next number in the sequence is 3X+1; if X is even, the next number is X/2. For example, the hailstone sequence starting at 5 is

```
5 16 8 4 2 1 4 2 1 4 2 1 4 2 1...
```

Once the number 1 is reached, the sequence gets stuck in a 4-2-1 loop. It is an unsolved problem in mathematics whether every hailstone sequence ends up stuck in the 4-2-1 loop, regardless of the starting number.

```c++
void hailstone(int num)
// precondition : num > 0
// postcondition: writes hailstone sequence starting at num
```

The text uses a nice progression of code to demonstrate how a sequence of input values can be read and processed. Starting from pseudocode, the idea of an input driven loop is first discussed abstractly. Then, a C++ loop which implements this idea is developed. While an alternative coding style is presented using the stream operator `fail`, students do not seem to have much trouble with a loop of the following form:

```c++
while (cin >> num) {
    // process num
}
```

I stress the abstraction that a cin statement returns a true/false value, depending on whether the input was successful. While this is not entirely accurate (the `>>` operator actually returns a reference to the stream, which is coerced into a boolean value), this gives the correct intuition.

After describing how to read input using a while loop, this idea is generalized to reading input from a file. The progression highlights the similarities between reading from standard input
and reading from files. Once an input file stream is declared and opened, you can read from that file stream in the same way that you read from `cin`. This is a very nice feature of C++.

Finally, all of the details of reading from a file are encapsulated into a class. In addition to providing another example of class design and implementation, this example teaches a good lesson about abstraction and code reuse. By placing the details of file manipulation inside a class, you spare yourself having to worry about these details every time you want to read from a file. Instead, you can simply use the `WordStreamIterator` class that you have just defined. For example, the author fingerprint program developed later in the chapter is greatly simplified by its use of the `WordStreamIterator` class. Instead of cluttering up this program with all of the details of file manipulation, these details are abstracted away.

The author fingerprint program introduces reference parameters, which can lead to some confusion among the students. I try to get my students to understand reference parameters (and their relation to value parameters) at two levels. Abstractly, they should think of the parameter passing methods as having directionality. For a value parameter, a value is passed into the function (from argument to parameter), but no value is passed back. A reference parameter has two-way directionality, however: a value is passed into the function, and any changes to the parameter are passed back to the argument as well. For the most part, this intuitive understanding suffices when the student need to decide which type of passing mode to use in a specific instance. As you know, however, this view of parameter passing is oversimplified and can lead to some misunderstandings if the students do not also get a more detailed understanding of parameters. So I also discuss the different ways in which these parameter passing modes are implemented. For value parameters, I stress the fact that what you are passing is a value (hence the name). Whatever value the argument represents is passed to the function, which is then stored in the parameter. Thus, the parameter is essentially a local variable that stores a copy of the argument value. On the other hand, a reference parameter does not result in a copy being made. Instead, a reference to the argument is passed (in reality, a pointer, but they don’t know about pointers yet). Perhaps a better word than reference is alias, since passing an argument by reference results in the parameter name becoming an alias for the argument.

To differentiate between the behaviors of the two parameter passing modes, I have the students trace through numerous toy examples. For example, I define a function with two arguments, one passed by value and the other by reference, and have the students trace calls to that function. When a value parameter is passed, they draw a box inside the function, labeled with the parameter name, and copy the argument value into that box. When a reference parameter is passed, they simply write an additional name (an alias) next to the box corresponding to the argument variable.

When I feel that they have a pretty good understanding of parameter passing, there are two questions I present to them. If they are able to answer these questions, then I feel confident that they really do understand the difference between value and reference parameters. The first question concerns the types of values that can be passed as arguments.
• When using pass by-value, an argument can be any expression (e.g., a constant, a variable, the sum of two numbers, ...), but using pass by-reference, an argument must be a variable. Explain why this is the case.

They should be able to recognize that since passing by-value results in a copy of the argument value being stored in a new memory location, any type of value can be used. On the other hand, a reference parameter does not allocate any new storage. Instead, the parameter name becomes an alias for the original argument. Thus, the original argument must be a variable that can be aliased and accessed via the parameter.

The second question involves the somewhat devious example:

```c++
#include <iostream.h>

void foo(int & x, int & y)
{
  x++;
  y++;
  cout << x << " , " << y << endl;
}

int main()
{
  int a = 1;
  foo(a, a);
  cout << a << endl;
  return 0;
}
```

In order to predict the output of this program, the student really need to understand the aliasing nature of reference parameters. Since the variable `a` is passed twice as reference parameters, both parameter names `x` and `y` become aliases for the same variable. Thus, when you increment `x`, you are also incrementing `y` (and `a`). Similarly, incrementing `y` affects the other variables. If the students can understand this example and predict the correct output, then they truly understand reference parameters.

Once the students understand parameter passing, I stress simple guidelines for determining which type of passing mode to use in specific situations. If the function needs to change the argument, then you have no choice but to pass that argument as a reference parameter. If not, then you shouldn't even give the function the opportunity to change it; so use a value parameter for safety. A third parameter passing mode can be introduced at this point for passing large objects. Recall that passing an argument as a value parameter results in a copy of that argument being stored in the parameter. For large objects such as strings (or any user-defined class with numerous data fields), the cost of physically making a copy of the object can be significant. As an alternative, the object can be passed by-reference (so that no copy is made) but preceded by the keyword `const`. Since the compiler will not allow const-reference...
parameters to be altered, this passing mode combines the efficiency of reference parameters with the safety of value parameters. (It is interesting to note that, unlike reference parameters, it is possible to pass an expression by const-reference. In such a case, the compiler treats the expression as in pass by-value, making a temporary copy of the expression for the parameter.)

The excursion for this chapter involves the use of a List class for storing and manipulating words. The author fingerprint program developed in this chapter has various interesting extensions, such as counting the number of occurrences of individual words, or looking for word duplications. Unfortunately, these kinds of applications require being able to store the words, which the students are not prepared to do at this time. This excursion provides an abstract List class which can be used to store a list of items of any type, including strings. This is a very instructive excursion since it flows naturally from the examples in the text. It also demonstrates how a class can be used to encapsulate details that should (at least temporarily) be hidden but nevertheless may be useful in solving interesting problems.

Chapter 7 Tips

The main goal of chapter 7 is to provide more in-depth examples of class design and implementation. In particular, classes are developed which are then used in various simulations. Simulations of events such as random walks and coin flips can be engaging to the students, and also provide ample opportunities for experimentation and analysis.

When it comes to class design, it is important to stress the interface first. You should first focus on the behavior of the class, i.e., the operations that will be allowed, then you can fill in the implementation details. As such, the first code you should consider when studying or designing a class is the .h header file. When implementing a class, write the header file first, then implement and test each of the member functions incrementally.

Before jumping into the random walk class developed in this chapter, you may wish to return to classes that the students have been using. The students should now be ready to understand the details of the classes such as Balloon, Dice, and MarbleJar that they have been using throughout the course. Working from these familiar examples, you can then develop new classes.

The idea of a private member function (as in the RandomWalk class) may require some explanation to your students. Up to this point, the class examples that they have seen have declared all member functions to be public and all data fields to be private. A private member function at first seems pointless since it cannot be called from outside the class. However, the use of a private member function as a helper for the other member functions is common, and so should be explained.

The idea of classes which coordinate to solve some problem is an important one. Unfortunately, the use of an observer class (as in section 7.2) can be very confusing. The fact that the random walk simulation is divided up among two classes which become intertwined is difficult to
grasp. This is further exacerbated by the use of reference variables. You may find that this example is not worth the confusion that it may bring, and choose to skip it.

A more useful example which demonstrates class coordination is the Coin class developed in section 7.3. Here, the Coin class utilizes the Dice class by having a Dice object as a data field. Since a coin may be viewed as a two-sided die, the Coin operations can be implemented using Dice operations. Two new language features are introduced in the Coin class: enumerated types and initializer lists. Enumerated types are useful in many examples (in fact, you could return to classes such as MarbleJar and integrate enumerated types). Initializer lists are much tougher to motivate.

Students often have a tough time understanding the distinction between a variable declaration and definition. A declaration tells the compiler what type of value is to be stored in a variable; a definition actually allocates memory for storing such a value. Data fields inside a class are merely declarations - they do not result in any storage being allocated. These data fields are not defined until an object of this class is created (i.e., the class constructor is called). At that point, storage is finally allocated for these data fields. For built-in data types and class objects with default constructors (no argument), this distinction is transparent. When the data field belongs to a class whose constructor requires an argument, however, the distinction between declaration and definition is paramount. Since the data field is not defined until the constructor for the class is called, it is there that the constructor for that data field must be called as well. This is done using the initializer list notation.

At the end of this chapter, structs are introduced for storing heterogeneous collections of data. The use of structs in C++ is somewhat of a throwback to C, where the struct was the only data structure for combining different data fields. As in C, structs can store data fields, but they can also have member functions as well. In fact, the only difference between a struct and a class in C++ is the default protection. By default, data fields and member functions of a struct are public, whereas they are private by default in classes. Since it is good practice to explicitly specify the protection of public and private sections in a class, this distinction is somewhat meaningless. In my classes, I now avoid the discussion of structs altogether, and simply use classes with all public fields if needed.

The excursion for this chapter is an application which extends some of the ideas found in the RandomWalk and Coin classes. It uses an enumerated type and presents another example of collaborating classes, where one class contains a reference to another class. If you skipped this part of the RandomWalk example, then you will probably want to skip this excursion as well. If not, then the excursion further illustrates these difficult concepts. The heart monitor application motivates the use of collaborating classes nicely, and presents a believable real-world application for the students consideration. At Duke, Susan Rodger has developed a graphical interface to the Heart class which can also add to the appeal of the example.

**Chapter 8 Tips**
The main goal of chapter 8 is to change the focus of programs from being control-driven to being data-driven. The students have already seen how to read in large amounts of data from files using a WordStreamIterator. They have also modeled complex phenomena using class objects such as dice and random walks. Without the capability of storing data and accessing it randomly, the range of applications is rather limited. By using Vectors as primitive list structures, however, complex data can be stored and manipulated as needed.

At the beginning of this chapter, the terms array and Vector are introduced. The distinction between these two terms can be confusing, especially to students who may have had some previous experience with arrays in some other programming language. The array type is built-in to C++, and can be used to store a homogenous collection of items, with each element of the array accessible via an index. Unfortunately, arrays are rather primitive data structures, with many idiosyncrasies due to their underlying pointer implementation. The Vector class is provided to insulate the students from some of these confusing features. Thus, you may think of an array as a low-level data structure, while a Vector is a full-blown data type (with operations such as length and SetSize defined). Of course, you probably don't want to burden your students with all of these details now. Instead, I would recommend sticking with the term Vector when referring to a homogeneous list that supports random access.

Perhaps the most important feature of the Vector class is that it is much safer to use than built-in arrays. When you access an array using an out-of-bounds index, the results are unpredictable. Sometimes you might get an error, sometimes not. Sometimes your program might behave as expected, sometimes not. This can be very confusing to beginning students. The Vector class, on the other hand, performs bounds checking. If you attempt to access a Vector using an out-of-bounds index, an error message will be printed and the program will terminate. For this and many other reasons (discussed later), starting your students out with Vectors as opposed to low-level arrays is a good thing.

An excellent example for motivating Vectors is the dice count example presented in the text. This example demonstrates how an arbitrary number of related items can be grouped under one name and manipulated uniformly. I first present the program dieroll.cc, which uses separate variables for each possible dice total. Students have no trouble understanding this code since they have dealt with dice and counters previously. And since the program is still relatively short, they do not seem overly concerned with the similarities between different sections of code. When I ask them to generalize this code from 4-sided to 6-sided dice, the extension is obvious: add four more counter variables, four more cases to the switch, and four more cout statements at the end. However, they begin to become uncomfortable with the obvious duplication of code that is occurring in the program. If I suggest 10-sided dice, then they appreciate the need for a different approach. And finally, if I require that the program work for dice with an arbitrary number of sides, they realize that the code will not generalize at all.

The idea that a Vector could be used to store a collection of counters can then be introduced. Instead of having 12 different variables, one for each dice total, a single Vector containing 12 values could be used (call it diceStats). Instead of referring to each count under a separate variable name, then, each count will be referred to by its index in the Vector (the number of twos rolled will be diceStats[2], the number of threes will be diceStats[3], etc.).
initialize all of the counts or to print them out at the end, a for loop can be used to step through
the indices and handle all of the counts systematically. Not only does this approach lead to
shorter, cleaner code, it generalizes easily. To adjust the number of dice sides, you do not need
to add variable declarations and switch cases, you simply need to change the constant which
specifies the number of sides. This constant then controls the size of the Vector and the number
of loops through the initialization and output loops.

The idea of using a Vector as a means of grouping together counters that can be manipulated
uniformly is further demonstrated using the letter frequency example. This example returns to
the familiar context of text manipulation, and uses a Vector of counters to count the number of
occurrences of letters in a file. You might also consider an extension to the author fingerprint
program developed in chapter 6. There, you were able to read in words and count how many
of them were short (length < 4), medium (4 <= length <= 7), and long (length > 7). Using a
Vector of counters, you can generalize this to keep track of the actual lengths of words, so that
you could print out the number of 1-letter words, 2-letters words, and so on.

The details of declaring Vectors should not be too difficult for students, since a Vector is just
another class. The size of the Vector is specified when the Vector is declared, and this value is
then passed as an argument to the constructor for the Vector class (which performs the
necessary initialization). You can also specify the initial value for items in the Vector. This can
simplify examples such as the dice count and letter frequency since the counters can be
automatically initialized to zero. As was the case with all user-defined classes, you should
stress that Vectors should be passed by const-reference instead of by-value whenever possible.
Since a Vector may hold large amounts of data, you want to avoid the time and space costs of
copying this data into a value parameter.

As a word of warning, if you are using a PC or Mac you may find yourself inundated with
warning messages whenever you include Vector.h. For example, Turbo C++ produces
numerous messages warning that inline functions containing for loops are not expanded. This
is unfortunate, since the messages tend to intimidate students and make locating real error
messages more difficult. The messages are a consequence of the way that C++ handles
templated classes (such as the Vector class). A templated class cannot be separated into a
header file and implementation file, and compiled separately. To avoid potential problems due
to multiple inclusion, the member functions of the Vector class are defined inline (inside the
class definition). This usually means that the functions are treated as macros, with function
calls in the code being expanded as opposed to a normal function call. Not all C++
implementations perform inline function expansion in all cases, and so warning messages may
appear. Alternative approaches to defining templated classes are described at Owen's web

By combining Vectors and structs, complex data can be stored and accessed efficiently. An
example of this is the cdshuffl.cc program in the text. Here, a struct type is defined for
storing information about a CD track. A Vector of these structs is then used to store
information on all of the tracks on a CD. I have found that drawing pictures of data structures
such as the one below helps the students to visualize the relationships between the data. For
example,
Using a picture such as this, I can stress the hierarchical nature of the data structure. The entire structure is a Vector, each element of the Vector is a struct (of type TrackInfo), and each struct has two fields (info and trackNumber). When attempting to access the data stored in this structure, the students need to work down through this hierarchy. For example,

<table>
<thead>
<tr>
<th>expression</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>allTracks</td>
<td>Vector&lt;TrackInfo&gt;</td>
</tr>
<tr>
<td>allTracks[0]</td>
<td>TrackInfo</td>
</tr>
<tr>
<td>allTracks[0].info</td>
<td>string</td>
</tr>
<tr>
<td>allTracks[0].trackNumber</td>
<td>int</td>
</tr>
</tbody>
</table>

Drawing a picture such as the one above on the board, I can point to a particular memory cell and ask the students to write the expression which refers to that particular cell. Or, conversely, I can give them an expression and ask them to identify the corresponding memory cell. In addition to the CD data structure, there are numerous examples that have this hierarchical structure combining structs and Vectors. For example, a database of student records might be implemented as a Vector, with each element being a struct that contains information on a particular student (e.g., name, address, birth date, transcripts). Or, a used car lot might maintain its inventory as a Vector of structs, each containing information on a particular car (e.g., make, model, year, price). More complex data structures can be developed, perhaps having Vectors as data fields in a struct.

Following the pattern used throughout the text, the lessons learned in the dice and letter frequencies examples are generalized into an abstract class. The WordList class contains a Vector of strings as a data field, as well as state variables specifying the size of the list and the number of entries currently stored. By encapsulating the Vector manipulation details, this abstract class can be used in numerous applications, such as in the extended fingerprint case study. Sample Assignment 6 presents another application that utilizes both the WordList class and the WordStreamIterator class from chapter 6.

The WordList class also introduces the tradeoffs between sequential search and binary search. The original version of the Search member function implements sequential search, but a binary search version is subsequently developed. I personally feel that the idea of binary reduction (or conversely, exponential growth) is an important idea in computer science that can be appreciated by introductory level students. This is why I introduce the idea under several different guises throughout the course. The paper folding example discussed in the chapter 5 Teaching Tips, the guessing game in Sample Assignment 3, the exponentiation algorithm in the chapter 4 Excursion -- these are all examples where a value or range is repeatedly halved or doubled. If your students have already seen and understood these intuitive examples, then the idea of binary search should be pretty natural. Table 8.2 in the text compares sequential search
and binary search on specific list sizes. When numbers like these are put in a real-world context, students find them even more revealing. For example, I ask the students to consider looking up a name in a phone book (where the names are alphabetized). For each of the following groups, we then determine how many inspections in the phone book would be needed to find an arbitrary person. Starting from just the students in their section all the way up to the entire world population, they are impressed at how few inspections are needed to find one name out of so many.

<table>
<thead>
<tr>
<th>Group</th>
<th>Inspections</th>
</tr>
</thead>
<tbody>
<tr>
<td>small section (25)</td>
<td>5</td>
</tr>
<tr>
<td>medium section (50)</td>
<td>6</td>
</tr>
<tr>
<td>large section (100)</td>
<td>7</td>
</tr>
<tr>
<td>small college (1,000)</td>
<td>10</td>
</tr>
<tr>
<td>medium college (10,000)</td>
<td>14</td>
</tr>
<tr>
<td>large college (40,000)</td>
<td>16</td>
</tr>
<tr>
<td>USA (260,000,000)</td>
<td>28</td>
</tr>
<tr>
<td>world (6,000,000,000)</td>
<td>33</td>
</tr>
</tbody>
</table>

Depending on the amount of time you have and the level of your students, you may want to skip section 8.5 which covers built-in arrays. By using Vectors throughout, there is no specific need for the students to understand arrays. However, if you are concerned about insulating your students too much from the details of the language, you may take some time to cover these details. Whatever you choose, it is important to stress that the array type is built-in to the C++ language, in contrast with the user-defined Vector type. Understanding some of the details of arrays can lead to a better understanding (and appreciation for) Vectors.

The excursion for chapter 8 involves designing, testing, and using a class for representing large integers. The `BigInt` class uses a Vector to store the individual digits in an integer that can be arbitrarily long. This application provides another example of embedding a data structure within a class, and so can be seen as a nice extension of the other examples in this chapter. The topic of operator overloading may be more than you want to get into at this point, however.

**Chapter 9 Tips**

The main goal of chapter 9 is to gain a better understanding of data structures, in particular looking at strings (as sequences of characters) and streams (as sequences of lines). The fact that characters are treated as integers in C++ has both advantages and disadvantages. An advantage is that numeric operations such as addition and comparison are already defined for characters. The disadvantage is the confusion that can occur between a character's abstract value (say, 'A') and its actual representation (ASCII value 65). While I describe the purpose of ASCII codes, I stress that the students should never make use of the actual codes. They can assume that lower case letters are all consecutive (and similarly, upper case letters), but they should never mix character values and integer codes. For example, to test if a character is a lower case letter, the second example is preferable to the first. The functions provided in the `ctype.h` library are helpful in maintaining the character abstraction, as shown by the third version below.
if (97 <= ch && ch <= 122) { // uses ASCII codes
    cout << "lower" << endl;
}

if ('a' <= ch && ch <= 'z') { // uses char ordering
    cout << "lower" << endl;
}

if (islower(ch)) { // uses ctype.h library
    cout << "lower" << endl;
}

After studying Vectors in the previous chapter, students do not have a hard time predicting how strings work. A string is a class that encapsulates a Vector of characters, along with operations on those characters. Since string access and manipulation is so similar to Vector access and manipulation, this is probably a good time to assign some class exercises, where groups of students will design and write functions for manipulating strings. Writing functions such as the following will reinforce the ideas of strings and Vectors as composite data structures, with components accessible via indexing.

```cpp
void PrintReverse(const string & str)
// postcondition: prints str in reverse (back to front)

bool IsNumber(const string & str)
// postcondition: returns true if str contains only digits

bool IsPalindrome(const string & str)
// postcondition: returns true if str is the same backwards

bool NumOccurrences(char ch, const string & str)
// postcondition: returns number of times ch is in str
```

The term abstract data type is finally introduced in chapter 9, although it could have been used throughout the course. In earlier chapters, classes were supplied which the students understood only in an abstract sense. These classes (e.g., `string`, `Vector`, `Balloon`, `WordList`) are essentially new data types that extend the C++ language. Whether you stress the term abstract data type is up to you. In a course where you wanted to push the object-oriented features of C++, this is a pretty standard term. If you are not overly concerned with OOP (or are content to let that wait until a later course), the important issue is that the students understand the ideas behind classes and abstraction.

Another example of an abstract data type is an input stream. When streams were used in earlier chapters, it was stressed that reading from an input stream is the same, regardless of whether that stream is connected to the keyboard (standard input) or a file (input file streams). At an abstract level, we can think of an input stream as an abstract object, consisting of lines of input. Using the extraction operator `>>`, we can access those lines of input. There are topics in section 9.2 that can be skipped to save time (or to avoid too many obscure details). String streams are very handy for parsing input that is line-oriented. Unless there is a specific
example for which this is needed, however, you can safely skip this topic. Similarly, the stream member functions, while useful in certain situations, can most likely be skipped. As to the stream member functions and manipulators for formatting output, your students should be made aware of these routines, but you should try to avoid spending much time at all on such details.

The compact disk case study at the end of this chapter returns to an application that was first introduced in the previous chapter. Here, we wish to be able to store CD information and perform calculations such as determining the total length of the CD (given the lengths of all of its tracks). In order to do this, we will need to be able to represent and add times together. This could be done on an ad hoc basis, or better yet, we could develop a class to represent and manipulate times. This is done in the **ClockTime** class, which stores a time (some number of hours, minutes, and seconds) and provides operations such as addition. Operator overloading is introduced here (although it was mentioned in a previous excursion).

The excursion for chapter 9 involves the study of how numbers are represented in C++. This excursion is in keeping with the goal that students learn more than just programming. Here, they are introduced to the differences between the different numerical types in C++ and binary number representation. As a lead in to the next chapter on recursion, recursive routines for converting decimal numbers to binary (and vice versa) are developed. Depending on your goals (and your time constraints), you may choose to cover this material or not.

**Chapter 10 Tips**

Chapter 10 has two main goals. One is to introduce recursion as a powerful approach to problem solving. While recursion is often relegated to the second course in computer science, it can be extremely useful for solving complex problems in a clear, compact way. The second main goal of this chapter is to return to the ideas of scope and lifetime that were superficially described earlier in the text, and make these concepts clear.

If you have taught recursion to beginning programmers before, you no doubt know that it can be very difficult for them. For the instructor, finding the right way to introduce and motivate recursion is equally hard. Most texts begin their coverage of recursion using toy problems. While the factorial function is easy to trace, it seems pointless to students since they find the iterative version more natural. Fibonacci is better in that the recursive solution is much shorter and clearer than an iterative solution. Of course, immediately after presenting this example we then turn around and show the students how bad the recursive solution is in terms of its inefficiency. If we cannot come up with examples that call for recursion that the students can understand, the battle is lost.

In previous excursions, the text has already introduced two good applications for recursion. The recursive exponentiation function in the chapter 4 excursion is both natural and efficient. Similarly, the recursive routines for converting decimal and binary numbers in the chapter 10 excursion are good examples of how recursion can lead to simpler and shorter code. A third, more abstract, example is found in the directory structure of computer file systems. All of these
examples provide applications where recursion is a useful alternative to iteration. Understanding this motivation is paramount before the students can begin to follow the mechanics of recursion.

While tracing recursion on toy problems such as fibonacci is useful, I have found that it sometimes leads to more trouble than it is worth. If there is one problem that students tend to have more than anything with recursion, it is that they try to think too much. For toy problems, the only way to determine the final outcome is to trace the entire recursive calling sequence. For more complex routines, this level of detail is just not possible. I try to focus on the inductive nature of recursion. Every recursive routine needs to have at least one base case, a scenario that is so simple to solve that recursion is not necessary. As long as the base case is handled correctly, you should be able to assume that a recursive call will produce the correct result (the induction hypothesis). Then, you need only combine the results produced by the recursive call to produce the final answer. For example, consider the recursive exponentiation function:

```c
double Power(double base, double expo)
// precondition : expo >= 0
// postcondition: returns base^expo
{
    if (0 == expo) {
        return 1.0;
    }
    else {
        double semi = Power(base, expo/2);
        if (expo % 2 == 0) {
            return semi*semi;
        } else {
            return base*semi*semi;
        }
    }
}
```

In order to know that this function behaves as desired, it is not necessary to trace through entire examples. It is enough to know that it handles the base case correctly (returns 1.0 when the exponent is 0), and correctly utilizes the answer provided by a recursive call to obtain the overall answer. If we assume that the recursive call returns the correct value $\text{base}^{\exp/2}$, then it is not difficult to reason that $\text{base}^{\exp/2} \times \text{base}^{\exp/2}$ will result in the correct answer when the exponent is even, and $\text{base}^{\exp/2} \times \text{base}^{\exp/2}$ will result in the correct answer when the exponent is odd. If the students can think inductively, recursion is much simpler to understand since you only need to worry about one level of computation. The Towers of Hanoi puzzle described in Exercise 10.5 is another one which illustrates the power of recursion. By focusing on one level of computation only, the solution is fairly simple:

```c
void Hanoi(int from, int to, int aux, int numDisks)
// precondition : top numDisks-1 disks on 'from' peg are all smaller than top disk on 'aux' peg
// postcondition: top numDisks disks moved from 'from' peg to 'to' peg
```
{ if (1 == numDisks) {
    cout << "move from " << from << " to " << to << endl;
} else {
    Hanoi(from, aux, to, numDisks-1);
    Hanoi(from, to, aux, 1);
    Hanoi(aux, to, from, numDisks-1);
}

In chapter 5, the concept of scope was first described, referring to that section of the program in which a variable name can be used. Variables that are declared inside a function are local to that function, and so their names are not visible outside the function. The same holds for variables that are declared inside block statements (e.g., inside a for loop). A variable declared outside of all functions is known as a global variable. The scope of a global variable is the entire program. In general, assigning and accessing global variables is a bad idea. The independence of functions is compromised since two calls to the same function might produce different results, depending on the value of the global variable. And since a global variable is accessible everywhere, it is difficult to know exactly when and where it might be changed. As was stated earlier, global constants are acceptable since they cannot be changed during execution.

Lifetime is a separate but often related property. Lifetime refers to the time during which storage is associated with a variable. For the most part scope and lifetime correspond to each other: the lifetime of a variable begins when its scope is entered and terminates when its scope is exited. There are exceptions to this, however. Declaring a variable to be static affects its lifetime without affecting its scope. For example, a static variable declared inside a function still has that function as its scope. Its lifetime, however, is unlimited. Even when the function terminates, the storage for that variable will remain. If that function is called again, the variable will still have the value from the previous call. For applications such as keeping a count of the number of function calls, static variables are preferable to global variables since their scope still limits all changes to occur inside the function.

Chapter 11 Tips

Chapter 11 has two main goals. One is to introduce and analyze several algorithms for sorting a list. Sorting is an important task in computer science, as demonstrated by the previous analysis of binary search in chapter 8. A sorted list of items can be searched much more efficiently than an unsorted list. Since there is a wide variety of algorithms for sorting, it also provides many opportunities for the comparison and analysis of algorithms. The second main goal of this chapter is to introduce multi-dimensional data structures using matrices, i.e., vectors of vectors. Matrices are useful in many applications, including graph algorithms and computer graphics.
When it is time to talk about sorting algorithms, I try to get my students to propose algorithms orally. I use some visual aid, such as a handful of playing cards, and ask them to describe a process for sorting the cards into increasing order. It is often interesting to see the thought processes they go through. They find it easy to sort a specific hand of cards, but much more difficult to generalize their description so that it works for arbitrary hands. When they do finally formalize a working algorithm, more times than not it is selection sort. Selection sort has a nice high-level description: find the card with lowest rank and swap it into the front. Then find the next lowest card and swap it into the second position. Continue doing this until you have gone through all of the cards and swapped each into its correct place. The implementation of this algorithm, especially as shown in sort1.cc using helper functions, is about as straightforward as you can get.

The implementation of a sorting algorithm is good motivation for function templates. If you wrote a sorting routine to sort integers and another one to sort doubles, these routines would look identical except for the types of the parameters and temporary variables. It would be a serious nuisance to have to implement a different version of the sorting routine for every type of value to be sorted. Templates allow for a single function which sorts a Vector of some unspecified type. In the same way that templated classes (such as Vector) must be instantiated at compile time, the specific instances of a templated function must be instantiated as well. This is done implicitly however, whenever you make a call to the templated function with a specific type. For example, the following function could be used to swap values of any type:

```cpp
template <class Type>
void Swap(Type & a, Type & b)
// generic swap routine
{
    Type temp = a;
    a = b;
    b = temp;
}
```

If this function is called with two ints as arguments, then an instance of this function with int substituted for the Type is compiled. If the function is also called with strings as arguments, a different version with string substituted for the Type is compiled.

A careful analysis of selection sort shows that to sort a list of N items, roughly N^2 comparisons will be made. There are N-1 passes through the loop, with the ith pass requiring N-i comparisons to find the smallest item. Using Gauss' formula, this adds up to N(N-1)/2 comparisons. If the number of comparisons is graphed relative to the size of the list, you will find that the resulting curve is quadratic in shape, i.e., similar to the curve y = x^2. Since we are interested in the general shape of curves (corresponding to the general cost of a sort), the exact number of comparisons is not important, just the fact that it is on the order of N^2, or O(N^2). The big-Oh notation is useful for describing the rate of growth of an algorithms cost. If an algorithm is O(N^2), it means that its cost (e.g. number of comparisons) grows proportional to the square of the input size. That is, if you double the size of the input, the cost goes up by a factor of 4. Other examples of O(N^2) sorting algorithms are described in the chapter exercises.
A more efficient but also more complex sorting algorithm is quick sort. Quick sort uses recursion to sort the list in only $O(N \log N)$ comparisons. Recall from previous examples with binary reduction (paper folding, guessing game, binary search, recursive exponentiation), repeatedly halving a number or range reduces it to 1 in logarithmic time. This behavior accounts for the logarithmic term in the complexity of quick sort. The original list is first partitioned into sublists, which are then sorted recursively. If the partitioning produces sublists that are roughly equal in size, then the number of recursions will be logarithmic. As Table 11.1 shows, the rate of growth for quick sort is much slower than for selection sort. Another $O(N \log N)$ sort, merge sort is described in exercise 11.5. Also see Sample Assignment 8 for an experimental analysis of two different sorting algorithms.

Multi-dimensional data is explored at the end of this chapter, where the `Matrix` class is defined. A `Matrix` is a two-dimensional Vector, storing data that can be accessed via two indices (a row and a column). Internally, a `Matrix` is defined as a Vector of Vectors. The fact that the Vector class is already defined makes implementing the `Matrix` class much easier (although the details of the implementation are only shown in an appendix). The class study at the end of this chapter provides one application where two-dimensional data needs to be stored and manipulated. Another, perhaps more engaging, example is found in Exercise 11.7, where a graphics image is represented as a two-dimensional `Matrix` of pixels. A slightly different version of this exercise can be found in Sample Assignment 9.

The final section of this chapter describes two-dimensional arrays. This optional section corresponds to the section at the end of chapter 8 where built-in C++ arrays are described. If you were concerned about students becoming reliant on the Vector class and covered built-in arrays, then you probably want to cover the details of two-dimensional arrays here as well. If not, you should probably skip this section and simply use the `Matrix` class when applicable.

**Chapter 12 Tips**

The main goal of chapter 12 is to introduce the idea of dynamic memory allocation and the use of linked lists for storing data. Without a doubt, pointers and linked lists are the hardest conceptual topics in this course. In fact, one of the real advantages of C++ over C as an instructional language is that it is much easier in C++ to insulate the programmer from pointers than in C. Unfortunately, the students' conceptual difficulties with pointers are usually magnified by time constraints, since this is typically the last topic covered in the course. The main problems that student tend to have with pointers are

- They have a hard time distinguishing between pointers and things they point to.
- They must understand when to allocate and deallocate dynamic memory.
- They must avoid dereferencing bad pointers, especially since the result of such an operation is not always predictable.

I have found that just jumping into pointers with the idea that the students will eventually appreciate them only adds to the confusion. Pointers and linked lists need to be sufficiently
motivated so that the students see where all of the details are headed. The advantages of understanding pointers that I stress to them are:

- They will be able to understand how reference parameters work.
- They will be able to understand the dynamic behavior of Vector and string objects.
- They will be able to construct lists that use memory proportional to the number of items stored.

As a first step (but with the above goals in mind), the students need to thoroughly understand what a pointer is and how it is used. I stress the concept of a pointer as being an address. Each memory cell inside the computer has an address (a number) by which it is referenced. A pointer is just a variable that stores the address of another memory cell, and so can be used to indirectly access the contents of that cell. The dereferencing operator * takes a pointer (an address) and returns the value that is stored at that address. Conversely, the address-of operator & takes a variable and returns the address of that variable. Continuing the analogy between pointers and addresses, dereferencing is like going to a house at a particular address and seeing who lives there. Using the address-of operator is the opposite operation, like asking a particular person for their address.

Code segments such as the following can be used to help understand these operators:

```cpp
int num = 6; // creates variable, stores 6
int * ptr = &num; // assigns ptr to point to num,  // (address of num stored in ptr)
cout << *ptr << endl; // outputs 6  // (value stored at ptr address)
*ptr = 3; // change value stored at address
cout << num << endl; // outputs 3
```

The first tangible use of pointer that you can get across to your students is in understanding reference parameters. Remind them of how reference parameters work - the parameter is given a reference to the argument, so that changes to the parameter simultaneously affect the argument. Once they know about pointers, it can be understood that what you are really doing is passing a pointer to the argument. The parameter stores and implicitly dereferences that pointer to access the original argument. To make this connection explicit, I show them a function using a reference parameter, and an equivalent (C-style) function using pointers:

```cpp
void foo(int & x)
{
    x = x + 5;
}

void foo(int * x)
{
    *x = *x + 5;
}

int a = 0;
foo(a);
```

The C-style code demonstrates that what is really happening with reference parameters is that the address of the argument is being passed, and then that address is being implicitly dereferenced inside the function.
Before jumping into linked lists, a good example to familiarize the students with pointers and dynamic memory allocation is a Vector of pointers. In order to store items using a Vector of pointers, you will need to allocate space each time a new item is added to the list, and deallocate that space when the item is deleted. The advantages of this approach over a standard Vector of items is that copying Vector entries is much quicker using pointers. Instead of swapping entire Vector elements (which may be complex data aggregates), you need only swap their pointers. In addition, a Vector that is only partially full will waste less space when the entries are only pointers. Working through this example, it is important to stress the lifetime rules for dynamically allocated memory. If you have a pointer to dynamically allocated memory and the lifetime of that pointer ends, this does not mean that the lifetime of the memory it points to ends. To the contrary, the only way to get rid of dynamically allocated memory is for you to explicitly deallocate it yourself (using delete). If you accidentally lose the pointer to dynamically allocated storage, then that storage is lost forever (I use the term "memory leakage" for instances such as this).

When it is finally time to consider linked lists, I always start with a visual demonstration. I propose that we use members of the class to store a list of numbers. Each person is capable of storing a number in their left hand (I limit number the number size to 5 so that they can use the fingers on one hand). With their right hands, they can point to a fellow student. We then discuss how it would be possible to link together a group of students, with each storing a number in their left hand and a pointer to the next student in line with their right hand. Using this "hands-on" approach, we are able to visually demonstrate the structure of linked lists and the process of adding and removing nodes from the list. The analogy is really quite striking. The class of students is the memory heap, where I can allocate a new student from the heap simply by saying the magic word "new". When I allocate a new student (node), I need to have some arm to point to it. The problem of memory linkage is illustrated whenever we lose the arm pointing to a student. For example, when I add a student to the linked list, I stress that they are our only link to the next student in the list. If they ever let their arm stray, then an entire portion of the list is lost. Those students who were connected but lost still must store their numbers and point to their neighbors, but they can never be accessed or reclaimed. As a transition form the human linked list to C++ linked lists, I define the following struct:

```cpp
class PersonNode {
    int leftHand;
    PersonNode * rightHand;
};
```

Beyond the human linked list, there is no substitute for drawing lots of pictures. For any sequence of operations that must be performed on linked lists, I stress with students that they first go through the steps pictorially, then encode it. As an exercise, I give the students a sequence of assignments and have them draw the contents of the list after each assignment. For example:

```cpp
class IntNode {
    int data;
    IntNode * next;
};
```
IntNode * list = new IntNode;
list->data = 4;
list->next = new IntNode;
list->next->data = 2;
IntNode * temp = list->next;
temp->next = new IntNode;
temp->next->data = 9;
temp->next->next = NULL;

The tricky nature of linked list manipulation makes this topic ideal for group work where the students can help each other work through the code details. For example, I might divide them into groups and have them write small functions such as the following:

```cpp
void PrintContents(IntNode * list)
// postcondition: prints contents of nodes in linked list

bool IsMember(IntNode * list, int num)
// postcondition: returns true if num stored in linked list

int NumOccur(IntNode * list, int num)
// postcondition: returns true # occurrences of num in list

void AddAtFront(IntNode * & list, int num)
// postcondition: adds num at front of list

void AddAtEnd(IntNode * & list, int num)
// postcondition: adds num at end of list

void DeleteFromFront(IntNode * & list)
// postcondition: removes node at front of list

void DeleteFromEnd(IntNode * & list)
// postcondition: removes node at end of list
```

Once they have worked through the laborious details of the linked list operations, students will see the real advantage of encapsulating these operations in a `List` class. The use of a dummy node (or header node) at the front of the linked list can simplify some of the list operations. For example, adding to a list that has a dummy node does not require a special case to worry about adding at the front - you can never add a node before the dummy node. It is instructive to contrast these operations which assume a dummy node with earlier code that does not.

Now that the students are familiar with pointers and dynamic memory, they are in a position to understand how strings and Vectors are implemented. Since the lengths of strings and Vectors can change during execution, it is not surprising that they use dynamically allocated storage to store their data. For a string, the data is a dynamically allocated array of characters. For the Vector, the array can store any type (depending on the supplied template class). If you have time, it is interesting to contrast Vectors with arrays, and then study how the Vector class uses dynamically allocated arrays to store its data. In particular, the `SetSize` member function allocates a new array and copies the contents of the old array over. The effect is that the Vector is resized, but in fact a completely different block of storage is being used.
The excursion at the end of this chapter provides a peek at some of the more powerful data structures that are used in computer science. These data structures are relatively easy to understand (at least in principle), and could in fact be implemented using the current knowledge of the students. Since hash tables and binary search trees will no doubt be covered in a subsequent computer science course, these examples provide a nice lead in to the next course for students who are interested in continuing.
The use of short, in-class quizzes has several advantages, especially in an introductory-level course. Quizzes can be used to reinforce concepts and materials covered in previous classes. A quiz at the beginning of a class period can remind the students of the main points from the previous lecture, and also encourage them to review their notes before class. Quizzes provide instant feedback to students, allowing them to gauge their understanding of the concepts. Similarly, the quizzes provide feedback to the instructor, allowing for misconceptions or points of confusion to be addressed early. On a less pedagogical note, quizzes seem to instill the students with a greater sense of responsibility in the course. Certainly, the threat of a quiz at the beginning of the class period encourages students to arrive on time and in a focused state. Beyond this, however, quizzes tell the students that they are responsible for understanding the material covered in the previous class. This encourages them to ask more questions during lectures since they know that the material may appear on the next quiz.

Since frequent quizzes provide the students with valuable feedback, the ideal is to have a short quiz at the beginning of each lecture. However, the amount of work involved in writing, administering, and grading daily quizzes can be formidable. Building a collection of potential questions (as provided in this chapter) helps in reducing the development time for quizzes. To reduce the time spent administering and grading the quizzes, I have found "random daily quizzes" to be very effective. Using some random event, like flipping a coin or picking a card, quizzes are made conditional. For example, if the coin comes up heads, then the students take the quiz that day and turn it in for a grade. If it comes up tails, the quiz is not administered, although the students are still given the quiz questions for review on their own. In this way, the students receive the quiz questions as a study guide each day, but the time commitment for the instructor is reduced.

My personal feeling is that quizzes should not count as a significant portion of the overall grade. In my classes, quiz grades only affect students whose overall grades are borderline. Again, the point is to provide feedback, not grades. As such, the quiz questions tend to be short and direct, highlighting the most important concepts from the previous lecture. A typical quiz would consist of two or three questions, and would only take five minutes of class time.
Chapter 1 Questions

QUESTION 1:
TRUE or FALSE? The halting problem, as formulated and proven by Alan Turing, implies that there are problems that cannot be solved with a computer.

TRUE

QUESTION 2:
What is an algorithm? Give a real-life example of an algorithm.

An algorithm is a step-by-step plan used in a process to achieve some end. For example, a recipe is an algorithm used in the cooking process to create a specific dish.

QUESTION 3:
Describe an algorithm for finding the tallest person in a room.

1. Line up all of the people
2. Compare the first and second people in line
3. If the first is taller, swap them in the line.
4. Continue working down the line, comparing and possibly swapping the 2nd and 3rd, 3rd and 4th, etc.

When done, the tallest person will be the last one in line.

QUESTION 4:
What are the three essential themes which form the core of computer science?

Theory
| Architecture | Language |
QUESTION 5:
In mathematics, there are an infinite number of real numbers between any two real numbers. On a computer, only finitely many of these real numbers can be represented exactly.

This is an example of which recurring concept?

- levels of abstraction
- conceptual and formal models
- efficiency and complexity

conceptual and formal models

QUESTION 6:
To the casual user, a VCR is a box with knobs and/or buttons that are used to control the recording and playing of video cassettes. To an electrician, a VCR might be viewed as a collection of electronic components which convert the magnetic patterns on a tape into electrical signals (and vice versa). A physicist, on the other hand, might view a VCR as a device for controlling the flow of electrons. Most people (at least those under the age of 30) have little difficulty programming a VCR, and yet they know very little about electronics and physics.

This is an example of which recurring concept?

- levels of abstraction
- conceptual and formal models
- efficiency and complexity

levels of abstraction

QUESTION 7:
TRUE or FALSE? The term architecture refers to the individual components which make up a computer, as well as their organization.

TRUE
QUESTION 8:
TRUE or FALSE? A binary number is an integer that is divisible by two.
FALSE

QUESTION 9:
TRUE or FALSE? The language C++ is an extension of the language C, with the added capability of supporting object-oriented programming.
TRUE

QUESTION 10:
What is the term for the process of finding and correcting errors in a computer program?
debugging

QUESTION 11:
Which of the following computer scientists is recognized as the inventor of Quicksort?

- Alan Turing
- C.A.R. Hoare
- Donald Knuth
- Alonzo Quick

C.A.R. Hoare
QUESTION 12:
TRUE or FALSE? Because the design and coding of programs is generally harder than the
design and construction of physical components, programs are referred to as hardware.

FALSE
QUESTION 13:
As was discussed in the text, different computers may have different underlying architectures, requiring different machine languages for controlling the hardware components. One advantage of high-level languages (like C++) is that they can be used on different kinds of computers, even those whose machine languages differ. How is this accomplished?

Each type of computer has its own compiler, which translates the high-level language into the low-level machine language of that specific computer. In this way, the high-level language is machine independent, and any computer can execute programs in that language (as long as a compiler for that machine exists).

QUESTION 14:
TRUE or FALSE? Grace Murray Hopper, one of the first programmers in the U.S. and co-designer of COBOL, attained the rank of Admiral in the Navy.

TRUE

QUESTION 15:
What does OOP stand for (in computer science)?

Object-Oriented Programming

QUESTION 16:
TRUE or FALSE? In C++, the word class is used to refer to a family of components sharing common characteristics, such as clocks or cars.

TRUE

QUESTION 17:
Which of the following computer scientists designed the C++ language?

- Bjarne Stroustrup
• Dennis Ritchie
• Linus Tarnlund
• Grace Murray Hopper

Bjarne Stroustrup
Chapter 2 Questions

QUESTION 18:
To whom is the following quote attributed?

The success of the UNIX system stems from its tasteful selection of a few key ideas and their elegant implementation. The model of the UNIX systems has led a generation of software designers to new ways of thinking about programming.

- Alan Turing
- Grace Murray Hopper
- Dennis Ritchie

Dennis Ritchie

QUESTION 19:
TRUE or FALSE? Every C++ program must have exactly one function named `main`.

TRUE

QUESTION 20:
Libraries of predefined programming components can be accessed in a C++ program using what kind of statement?

#include
QUESTION 21:
The C++ input/output "devices" `cin` and `cout` are examples of

- streams
- pipes
- rivers
- bytes

QUESTION 22:
What is output by the following program?

```c++
#include <iostream.h>

void Func1()
{
    cout << "Print 1" << endl;
}

void Func2()
{
    cout << "Print 2" << endl;
}

int main()
{
    cout << "Print 3" << endl;
    Func2();
    Func1();
    cout << "Print 4" << endl;
    return 0;
}
```

Print 3
Print 2
Print 1
Print 4
QUESTION 23:
What is output by the following program?

```cpp
#include <iostream.h>

void Func1()
{
    cout << "Print 1" << endl;
}

void Func2()
{
    Func1();
    cout << "Print 2" << endl;
}

int main()
{
    Func2();
    cout << "Print 3" << endl;
    Func1();
    cout << "Print 4" << endl;
    return 0;
}
```

Print 1
Print 2
Print 3
Print 1
Print 4

QUESTION 24:
What is output by the following statements? Be precise.

```cpp
cout << "The sum of " << 5 << " and ";
cout << 7 << " is " << endl << endl;
cout << 5+7 << endl;
```

The sum of 5 and 7 is
12
QUESTION 25:
How many lines of output are produced by the program below?

```cpp
#include <iostream.h>

void foo()
{
    cout << "foo" << endl;
}

void bar()
{
    foo();
    cout << "bar" << endl;
}

void biz()
{
    foo();
    bar();
    cout << "biz" << endl;
}

int main()
{
    foo();
    bar();
    biz();
    return 0;
}
```

7

QUESTION 26:
Is the following a legal C++ statement? If so, what is output? If not, why not?

```cpp
cout << "Harry shouted "Cubs win!" again." << endl;
```

It will not compile, since the double-quote before the word Cubs ends the string. In order to include double-quotes inside of strings, an escape character (backslash) must be used. The correct statement is:

```cpp
cout << "Harry shouted \"Cubs win!\" again." << endl;
```
QUESTION 27:
TRUE or FALSE? Ada Lovelace was instrumental in popularizing the work of Charles Babbage, who designed mechanical computing devices in the 19th century.

TRUE

QUESTION 28:
TRUE or FALSE? The type string is a fundamental type, i.e., it is built into the language C++ in the same way that types int and double are.

FALSE

QUESTION 29:
TRUE or FALSE? Iterative enhancement is a design process by which a program is developed in stages, with each stage refining the version from the previous stage.

TRUE

QUESTION 30:
TRUE or FALSE? Given a function whose header is as follows:

```c++
void AddToPayroll(string person, double salary)
```

the call `AddToPayroll(27499.99, "Pat Kelly")` will be caught as illegal by the compiler.
TRUE
QUESTION 31:
Consider the following program:

```cpp
#include <iostream.h>
#include "CPstring.h"

void Greet(string name)
{
    cout << "Hello " << name << "!" << endl;
}

int main()
{
    Greet("Laura");
    return 0;
}
```

A. What is output by the program?

Hello Laura!

B. What is the formal parameter in the above program? What is the actual parameter (i.e., the argument)?

<table>
<thead>
<tr>
<th>formal parameter:</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>actual parameter (argument):</td>
<td>&quot;Laura&quot;</td>
</tr>
</tbody>
</table>

QUESTION 32:
Consider the following function:

```cpp
void WelcomeWagon(string state, string name)
{
    cout << "Hello " << name << endl;
    cout << "Welcome to " << state << endl;
}
```

What is output by the call WelcomeWagon("Laura", "Maryland")?

Hello Maryland
Welcome to Laura
QUESTION 33:
What is output by the following program?

```cpp
#include <iostream.h>
#include "CPstring.h"

void Twice(string word)
{
    cout << word << " " << word << " ";
}

int main()
{
    Twice("Happy");
    Twice("Joy");
    cout << endl;
    return 0;
}
```

Happy Happy Joy Joy

QUESTION 34:
Write a C++ void function called Faves which has two parameters, a person's favorite color and ice cream flavor, and produces output customized to those favorites. For example, the call Faves("green", "pistachio") should produce the following output:

```
So, your favorite color is green and your favorite ice cream is pistachio.

Those are my favorites too!
```

```cpp
void Faves(string color, string flavor)
{
    cout << "So, your favorite color is " << color << " and " << endl;
    cout << "your favorite ice cream is " << flavor << endl;
    cout << "Those are my favorites too!" << endl;
}
```
**QUESTION 35:**
Write a C++ void function called \texttt{PrintDice} which has two parameters, the value of two dice rolls, and which outputs the dice sum. For example, the call \texttt{PrintDice(4,3)} should produce the following output:

\begin{verbatim}
Dice rolled: 4 and 3
Total = 7
\end{verbatim}

\begin{verbatim}
void PrintDice(int die1, int die2)
{
    cout << "Dice rolled: " << die1 << " and " << die2 << endl;
    cout << "total = " << die1 + die2 << endl;
}
\end{verbatim}

**QUESTION 36:**
TRUE or FALSE? In C++, a function can have at most one parameter.

FALSE

**QUESTION 37:**
In general, a function must be defined in a program before it can be called. Thus, it would seem that \texttt{main} must be the last function in every program. How can this be avoided?

As long as a prototype of the function appears before the calling point, the actual function definition may appear anywhere in the program. Therefore, main can appear anywhere in the program as long as the necessary function prototypes precede it.

**QUESTION 38:**
TRUE or FALSE? In C++, both \texttt{Hello_World} and \texttt{HELLO_world} are legal and distinct identifiers.

TRUE
QUESTION 39:
Which of the following are legal C++ identifiers? For those that are not legal, explain why.

- aa123
- 2Cool4U
- A_B_Cs
- MyFavoriteThings
- two words

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Legal Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa123</td>
<td>legal</td>
</tr>
<tr>
<td>A_B_Cs</td>
<td>legal</td>
</tr>
<tr>
<td>MyFavoriteThings</td>
<td>legal</td>
</tr>
<tr>
<td>2Cool4U</td>
<td>not legal because it begins with a digit.</td>
</tr>
<tr>
<td>two words</td>
<td>not legal because it contains a space.</td>
</tr>
</tbody>
</table>

QUESTION 40:
Consider the following functions:

```cpp
void Print1()
{
    cout << "foo" << endl << "foo" << endl;
}

void Print2()
{
    Print1();
    Print1();
}

void Print3()
{
    Print2();
    Print2();
    Print2();
}
```

How many lines of output are produced by the call `Print1()`?
- 2 lines of output

How many lines of output are produced by the call `Print2()`?
- 4 lines of output

How many lines of output are produced by the call `Print3()`?
- 8 lines of output
Print3() --> 12 lines of output
**Chapter 3 Questions**

**QUESTION 41:**
What is a variable?

A variable is a named memory location capable of storing a specific type of value.

**QUESTION 42:**
What is the name of a C++ type for representing integers (e.g., ..., -2, -1, 0, 1, 2, ...)?

int (also char, short int, short, long int, long)

**QUESTION 43:**
What is the name of a C++ type for representing real numbers (e.g., -9.3, 0.0, 3.1415)?

double (also float, long double)

**QUESTION 44:**
Write variable declarations to store an individual's first and last names, age (an integer), and salary (a real).

```cpp
string firstName, lastName;
int age;
double salary;
```
QUESTION 45:
Immediately after the declaration

```c
double salary;
```

what is the value stored in the variable `salary`? Explain.

undefined

Until a variable is explicitly assigned a value, its value is undefined.

QUESTION 46:
Write a code segment which reads in a dog's age, and prints out the equivalent age in human years. (Note: a dog year is equivalent to 7 human years.)

The following is a sample execution (with the user's input in bold):

```
How old is your dog? 11
That's 77 to you and me.
```

```c
cout << "How old is your dog?";
int dogYears;
cin >> dogYears;
cout << "That's " << 7*dogYears << " to you and me." << endl;
```

QUESTION 47:
In C++, the following statements produce different output. Why? What is output by each statement?

```c
cout << 9/5 << endl;
cout << 9.0/5 << endl;
```

Dividing two integers yields another integer (any fractional remainder is discarded). Any division involving a double yields a double. Thus,

```c
cout << 9/5 << endl;  --> 1
```
cout << 9.0/5 << endl;  -- 18
QUESTION 48:
Let $a$ and $b$ represent two arbitrary integers. What value is represented by the following expression? Explain.

$$ (a \div b) \times b + (a \mod b) $$

The value of this expression is $a$.

Since integer division discards the remainder, multiplying $a/b$ times $b$ and then adding back the remainder $a\%b$ produces the original value $a$.

QUESTION 49:
What values are represented by each of the following expressions?

$$ 2 + 3 \times 5 $$
$$ 3 + 2 - 1 $$
$$ 8 \div 2 \div 2 $$

$$ 2 + 3 \times 5 \rightarrow 2 + (3 \times 5) \rightarrow 2 + 15 \rightarrow 17 $$
$$ 3 + 2 - 1 \rightarrow (3 + 2) - 1 \rightarrow 5 - 1 \rightarrow 4 $$
$$ 8 \div 2 \div 2 \rightarrow (8 \div 2) \div 2 \rightarrow 4 \div 2 \rightarrow 2 $$

QUESTION 50:
Which of the following accomplishments are not attributed to John Kemeny?

- invented the programming language BASIC
- developed the first “time-sharing” computer
- invented the transistor
invented the transistor
QUESTION 51:
TRUE or FALSE? In C++, a class is a user-defined type, containing both data and member functions for manipulating that data.

TRUE

QUESTION 52:
The process of creating an executable program from source code is generally performed in two stages. Describe these two stages.

1. Compilation: source code is compiled into object code
2. Linking: all of the object files are linked together into one executable program

QUESTION 53:
Consider a simplified version of the fly.cc program from the text (pp. 89-90).

```cpp
#include <iostream.h>
#include "balloon.h"

int main()
{
    Balloon montgolfier;
    montgolfier.Ascend(50);
    montgolfier.Cruise(5);
    montgolfier.Descend(0);
    return 0;
}
```

Without the statement `#include "balloon.h"` the program will not compile. Why?
The file balloon.h contains the interface to the **Balloon** class. If this file is not included, then the type **Balloon** is not defined.
QUESTION 54:
Consider the Balloon class from the text (pp. 92-93). The member function AdjustAltitude is in the private section of the Balloon class. This means that (circle one)

- it cannot be called from a client program such as fly.cc
- it can be called from a client program such as fly.cc

it cannot be called from a client program such as fly.cc

QUESTION 55:
Consider the Balloon class from the text (pp. 92-93). A variable montgolfier of type Balloon has three methods (member functions) accessible to the programmer: Ascend, Cruise, and Vent. Which of the following statements correctly invokes the Ascend method?

- Ascend(montgolfier, 100);
- Ascend.montgolfier(100);
- montgolfier.Ascend(100);

montgolfier.Ascend(100);

QUESTION 56:
TRUE or FALSE? Donald Knuth, perhaps the foremost scholar in the field of computer science, is author of the classic three volume set The Art of Computer Programming.

TRUE
Chapter 4 Questions

QUESTION 57:
What is output by the following (somewhat tricky) code segment?

```cpp
int x;
cout << (x = 7) << endl;
```

7

QUESTION 58:
What is output by the following (somewhat tricky) code segment? Justify your answer in one or two sentences.

```cpp
int num = 5;
int prod = 8;
if (prod = num) {
    cout << "equal" << endl;
} else {
    cout << "different" << endl;
}
```

equal

This is an example of a common error -- using "=" instead of "==" for comparison. An assignment does return a value, the value being assigned. In this case, prod is being assigned the value 5, and since this is nonzero, the if guard evaluates to true.

QUESTION 59:
Write an if/else statement which outputs the word "yes" if variables x and y have the same value, otherwise it outputs the word "no".

```cpp
if (x == y) {
    cout << "yes" << endl;
} else {
    cout << "no" << endl;
}
```
QUESTION 60:
The following code segment causes a compiler error. Why? How can the error be fixed?

```cpp
if (language == "C++")
    cout << "Welcome to the world of C++" << endl;
cout << " the language of the future!" << endl;
else
    cout << "Oh well, programming is programming." << endl;
```

Although the indentation makes this code appear to be a single if/else statement, it is not interpreted this way by the compiler. Since two cout statements follow the if, the first statement is interpreted as closing a simple if statement. Then, the else keyword does not fit.
The two cout statements can be turned into a single, compound statement by enclosing them in braces.

QUESTION 61:
Write a code segment which reads in two strings, and outputs the word "different" if the input values are different.

```cpp
string word1, word2;
cout << "Enter two words: ";
cin >> word1 >> word2;
if (word1 != word2) {
    cout << "different" << endl;
}
```

QUESTION 62:
The arithmetic assignment `count += 1` is equivalent to which of the following expressions?

- `count = +1`
- `count = count + 1`
- `count == +1`

`count = count + 1`
QUESTION 63:
Write a code segment which reads in an integer (call it \texttt{num}) and outputs the word "positive" if \texttt{num} is greater than zero, "zero" if \texttt{num} is equal to zero, or "negative" if \texttt{num} is less than zero.

```cpp
int num;
cout << "Enter an integer: ";
cin >> num;

if (num > 0) {
    cout << "positive" << endl;
}
else if (num == 0) {
    cout << "zero" << endl;
}
else {
    cout << "negative" << endl;
}
```

QUESTION 64:
What is output by the following (somewhat tricky) code segment?

```cpp
int foo = -1;

if (foo < 0) {
    cout << "negative" << endl;
    foo = -foo;
}
if (foo == 0) {
    cout << "zero" << endl;
}
if (foo > 0) {
    cout << "positive" << endl;
}
```

```
negative
positive
```
QUESTION 65:
TRUE or FALSE? The following statement prints a "proper" message identifying whether the value of num is even.

```cpp
if (num % 2 == 0){
    cout << "number is even" << endl;
} else {
    cout << "number is odd" << endl;
}
```

TRUE

QUESTION 66:
The following expression was intended to determine whether num is between low and high (inclusively). Unfortunately, it doesn't work. Write a corresponding expression that does work as intended.

```cpp
(low <= num <= high)
```

```
(low <= num && num <= high)
```

QUESTION 67:
What is output by the following code segment?

```cpp
int x = 1, y = 0;
if (x >= y && x != 0) {
    cout << "foo" << endl;
} else {
    cout << "bar" << endl;
}
```

foo
QUESTION 68:
If x = 3, y = 7, and z = 10, which of the following boolean expressions evaluate to true?

- (x <= y && y <= z)
- !(x+y) == z)
- (x != y || y > z)
- !(x > z && y > z)
- (x == y || (x != z && y == z))

\[
\begin{aligned}
(x <= y && y <= z) \\
(x != y || y > z) \\
!(x > z && y > z)
\end{aligned}
\]

QUESTION 69:
The following if statement relies on short-circuited evaluation to avoid a division-by-zero error in the test. Rewrite this statement so that it does not rely on short-circuited evaluation. (Hint: think nested if's.)

```cpp
if (numNums > 0  &&  (sumNums/numNums) > 90) {
    cout << "Excellent grade!" << endl;
}
```

```cpp
if (numNums > 0) {
    if (sumNums/numNums > 90) {
        cout << "Excellent grade!" << endl;
    }
}
```
QUESTION 70:
TRUE or FALSE? Claude Shannon founded the field of information theory, a subfield of computer science that is used today in developing methods for encrypting information.

TRUE

QUESTION 71:
Define a function called TriangleArea which computes the area in a given triangle. The function should have two parameters, the height and base of the triangle (both doubles), and should return the area of that triangle. Recall that the area of a triangle is the base times half the height.

```cpp
double TriangleArea(double height, double base)
// precondition : height > 0, base > 0
// postcondition: returns the area of the triangle
{
    return 0.5 * height * base;
}
```

QUESTION 72:
Complete the function ValidGrade whose header is given below:

```cpp
bool ValidGrade(int grade)
// postcondition: returns true if 0<=grade<=100, else false
```

```cpp
bool ValidGrade(int grade)
// postcondition: returns true if 0<=grade<=100
{
    return (0 <= grade && grade <= 100);
}
```
**QUESTION 73:**

Complete the function `NonDecreasing` whose header is given below:

```c
bool NonDecreasing(int num1, int num2, int num3)
// postcondition: returns true if num1<=num2<=num3, else false
```

```c
bool NonDecreasing(int num1, int num2, int num3)
// postcondition: returns true if num1<=num2<=num3
{
    return (num1 <= num2 && num2 <= num3);
}
```

**QUESTION 74:**

Complete the function `Graduate` whose header is given below:

```c
string Graduate(string name)
// precondition: name is a person's name
// postcondition: returns name with "Dr. " prepended
```

For example, the call `Graduate("Laura Hansen")` should return "Dr. Laura Hansen".

```c
string Graduate(string name)
// precondition: name is a person's name
// postcondition: returns name with "Dr. " prepended
{
    return "Dr. " + name;
}
```

**QUESTION 75:**

TRUE or FALSE? Common mathematical functions such as `sqrt`, `pow`, and `floor` are defined in the `iostream.h` library and are automatically loaded when you include that library.

**FALSE**
QUESTION 76:
Complete the function Dist whose header is given below. Recall that the distance between two points \((x_1, y_1)\) and \((x_2, y_2)\) is given by: \(\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}\).

```cpp
def double Dist(double x1, double y1, double x2, double y2) 
    // precondition: coordinates of (x1,y1) and (x2,y2)
    // postcondition: returns distance between these points
    return sqrt( pow((x1-x2), 2) + pow((y1-y2), 2) );
```

QUESTION 77:
What is output by the following code segment?

```cpp
string str = "foobar";
if (str.length() > 5) {
    cout << str.Upcase() << endl;
}

cout << str << endl;
```

FOOBAR
foobar

QUESTION 78:
Which of the following computer scientists founded the GNU software project, and is hailed by many as “the world's best programmer?”

- Donald Knuth
- Dennis Ritchie
- Richard Stallman
- David Gries
Chapter 5 Questions

QUESTION 79:
Consider the Dice class discussed in the text (pp. 165-166):

```cpp
class Dice {
    public:
        Dice(int sides); // constructor
        int Roll(); // return the random roll
        int NumSides(); // how many sides this die has
        int NumRolls(); // # times this die rolled
    private:
        RandGen myGenerator; // random number generator
        int myRollCount; // # times die rolled
        int myNumSides; // # sides on die
};
```

Write a statement which defines a 5-sided die called `quint`.

```cpp
Dice quint(5);
```

The following statement causes a compiler error. Why? Write an equivalent cout statement which correctly prints the number of die sides.

```cpp
cout << "Number of sides = " << quint.numSides << endl;
```

Since there are no parentheses after the member function name `numSides`, this expression is not interpreted as a function call (instead, it refers to the actual function itself). Even when no arguments are required, parentheses must be specified in a function call.

```cpp
cout << "Number of sides = " << quint.numSides() << endl;
```

Write a code segment which outputs the results of 10 rolls of the quint die.

```cpp
int rep = 1;
while (rep <= 10) {
    cout << quint.Roll() << endl;
    rep++;
}```
)
QUESTION 80:
What is a constructor (in the context of C++ classes)? What are constructors used for?

A constructor is a member function (method) of a class that is automatically called whenever an object of that class is created. A constructor is used to automatically perform any initialization needed for an object of the class (e.g., initializing data fields to zero).

QUESTION 81:
Consider the following code segment?

```cpp
cin >> num;
while (num <= 50) {
  <lots of fun stuff here>
  cin >> num;
}
// assertion:
```

At the point in the code labeled as an assertion, what must be true about the value of `num`?

```
num > 50
```

QUESTION 82:
What is the last number printed by the loop below?

```cpp
int num = 0;
while (num < 100) {
  num = num + 2;
  cout << num << endl;
}
```

```
100
```
QUESTION 83:
Give an example of an infinite loop.

```
while (true) {
    // anything
    OR
    // anything ...
}
```

QUESTION 84:
Complete the function PrintReps whose header is given below:

```
void PrintReps(string name, int numReps)
// postcondition: prints name numReps times
```

```
void PrintReps(string name, int numReps)
// postcondition: prints name numReps times
{
    while (numReps > 0) {
        cout << name << endl;
        numReps--;
    }
}
```

QUESTION 85:
What is output by the following code segment?

```
int x = 16;
while (x > 1) {
    cout << x << endl;
    x = x / 2;
}
```

```
16
8
4
```
QUESTION 86:
What is output by the following code segment?

```c
int n = 4, p = 1;
while (n > 0) {
p *= n;
n--;
cout << p << endl;
}
```

4
12
24
24

QUESTION 87:
Consider the following code segment:

```c
int prod = 1;
int num = 5;
int k = 1;
while (k <= num){
prod = prod * k;
k += 1;
}
cout << prod << endl;
```

What value is output as a result of executing the above code segment?

120

TRUE or FALSE? Changing the assignment `prod = prod * k` to `prod *= k` results in the same value being printed.

TRUE

TRUE or FALSE? Changing the assignment `k += 1` to `k++` results in the same value being printed.

TRUE
QUESTION 88:
Consider the function `SomeSum` given below:

```c
int SomeSum(int number)
// precondition: number >= 0
// postcondition: ???
{
    int sum = 0;
    while (number > 0) {
        sum += number % 10;
        number = number / 10;
    }
    return sum;
}
```

What value is returned by the call `SomeSum(0)`?

0

What value is returned by the call `SomeSum(7)`?

7

What value is returned by the call `SomeSum(512)`?

8

Given an arbitrary positive integer \( n \), what value is returned by the call `SomeSum(n)`?

The sum of all of the individual digits in \( n \).
QUESTION 89:
Describe one advantage of using named constants in a program.

Named constants make changing constant values easier since only the constant definition must be changed (as opposed to each occurrence of the constant).

Since constants cannot be changed, it is safe to place all constants at the top of the file (i.e., make them global). This makes it easy to locate and change constant values.

Mnemonic constant names provide the reader with more meaning, and thus make code easier to read, debug, and modify.

QUESTION 90:
What is output by the following piece of code?

```cpp
int cnt, num = 10;
for (cnt = num; cnt < 5; cnt++) {
    cout << "foo" << endl;
    num += 1;
}
cout << num << endl;
```

10

QUESTION 91:
Fill in the for loop shown below so that all the even numbers between 2 and 246 are printed.

```cpp
int num;
for (num = 2; num <= 246; num += 2) {
    cout << num << endl;
}
```
cout << num << endl;
}
QUESTION 92:
Write a for loop which prints out the numbers 1 through 50 in decreasing order.

```cpp
int k;
for (k = 50; k >= 1; k--) {
    cout << k << endl;
}
```

QUESTION 93:
Rewrite the following while loop as a for loop.

```cpp
int value = 10;
while (value < 1000) {
    cout << value << endl;
    value *= 2;
}
```

```cpp
int value;
for (value = 10; value < 1000; value *= 2) {
    cout << value << endl;
}
```

QUESTION 94:
Rewrite the following for loop as a while loop.

```cpp
int k, number, sum = 0;
for(k = 0; k < numValues; k++){
    cin >> number;
    sum += number;
}
```

```cpp
int k, number, sum = 0;
k = 0;
while (k < numValues) {
    cin >> number;
    sum += number;
    k++;
}
QUESTION 95:
What is output by the following piece of code?

```c
int counter = 0, number = 30;
do {
    counter += 1;
    number /= 2;
} while (number != 1);
cout << counter << endl;
```

4

QUESTION 96:
TRUE or FALSE? The body of a do-while loop is always executed at least once, while the body of a while loop might not be executed at all.

TRUE

QUESTION 97:
How many stars are printed by the following piece of code?

```c
int j, k;
for (j = 1; j <= 3; j++) {
    for (k = 1; k <= 4; k++) {
        cout << "*";
    }
}
```

12
QUESTION 98:
What is output by the following piece of code?

```cpp
int outer, inner;
for (outer = 0; outer < 2; outer++) {
    for (inner = 0; inner < 3; inner++) {
        cout << outer << " " << inner << endl;
    }
}
```

0 0
0 1
0 2
1 0
1 1
1 2

QUESTION 99:
Rewrite the following switch as a cascading if-else statement.

```cpp
switch (numCorrect) {
    case 0:
        cout << "Pretty pathetic!" << endl;
        break;
    case 1:
        cout << "You did OK." << endl;
        break;
    default:
        cout << "Great job!" << endl;
}
```

```cpp
if (0 == numCorrect) {
    cout << "Pretty pathetic!" << endl;
}
else if (1 == numCorrect) {
    cout << "You did OK." << endl;
}
else {
    cout << "Great job!" << endl;
}
```
QUESTION 100:
The following code segment results in an infinite loop when executed, regardless of what strings are input by the user. Why is this so?

    string name = "Bjarne";
    while (name != "Donald") {
        string name;
        cin >> name;
        cout << name << endl;
    }

There are two different name variables in this code segment. Inside the loop, a new value is read into the local variable. However, the name variable in the loop test is in the global scope, and so is unaffected by the code in the loop.

QUESTION 101:
Consider the following code segment:

    int count = 0, num = 4;
    if (num > 0) {
        int count = 1;
        int k;
        for (k = 0; k < num; k++) {
            count *= 2;
        }
        cout << count << endl;
    }
    cout << count << endl;

What is output by this code segment:

<table>
<thead>
<tr>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

Now suppose that the declaration inside the if-body (int count = 1) was replaced by the simple assignment count = 1. What would be output by this modified code?

| 16 |
QUESTION 102:
According to DeMorgan's Law, which of the following boolean expressions is equivalent to the expression: \((\text{not } (A \text{ and } B))\)?

- \((\text{not } A) \text{ and } B\)
- \(((\text{not } A) \text{ and } (\text{not } B))\)
- \(((\text{not } A) \text{ or } (\text{not } B))\)
- \((A \text{ or } (\text{not } B))\)

\(((\text{not } A) \text{ or } (\text{not } B))\)

QUESTION 103:
Using DeMorgan's Law, simplify the following boolean expression so that the not operator (!) does not appear: \(! (a \neq b \text{ || } b > c)\)

\((a == b \&\& b <= c)\)
Chapter 6 Questions

QUESTION 104:
Write a code segment which reads in a sequence of integers, terminated with a negative value, and outputs their sum. For example, if the user enters

4 6 13 0 3 -1

then your code should output 26. Note: the negative sentinel value should not be included in the sum.

```cpp
int sum = 0;
int num;
cin >> num;
while (num >= 0) {
    sum += num;
    cin >> num;
}
cout << sum << endl;
```

QUESTION 105:
TRUE or FALSE? Before founding Microsoft and becoming the richest man in America, Bill Gates began his career by writing the first BASIC compiler for microcomputers.

TRUE

QUESTION 106:
Consider the following code segment. If the user enters `foo bar biz` at the prompt, what is output?

```cpp
string s1, s2;
cout << "Please enter two strings: ";
cin >> s1 >> s2;
cout << s2 << endl << s1 << endl;
```

bar
foo
QUESTION 107:
Complete the following code segment so that it reads and echoes integers until the input is exhausted (or a non-integer is read).

```cpp
int num;
while (                    ) {
    cout << num << endl;
}

int num;
while (cin >> num) {
    cout << num << endl;
}
```

QUESTION 108:
The code segment below is intended to read in two integers and print their average. However, executing this code with input values 4 and 5 produces the incorrect value 4. Explain why this is the case. Without changing the types of any of the variables, modify the code segment so that the correct average is printed.

```cpp
int sum = 0;               // accumulate a sum
int numNums = 0;           // count number of numbers
int num;
while (cin >> num){        // as long as input succeeds
    sum += num;
    numNums++;
}
double average = sum/numNums;
cout << average << endl;
```

Since both `sum` and `numNums` are integer values, the division in the assignment will be integer division, resulting in the loss of any fractional part:

double average = sum/numNums;

In order to perform floating-point division, one of the operands must be a floating-point value. This can be accomplished by casting one of the operands explicitly, e.g.,

double average = double(sum)/numNums;
QUESTION 109:
Consider the following code segment:

```cpp
int num;
while (cin >> num && num > 0) {
    cout << num;
}
```

If the user enters the following as input, what is output by this code segment?

```
1 2 3 A 4 5 -1 6
^Z
```

A) 1 2 3

B) 1 2 3 4 5

C) 1 2 3 A 4 5

D) 1 2 3 4 5 6

E) 1 2 3 4 5 -1 6

A) 1 2 3

QUESTION 110:
The following code segment is intended to read integers from standard input until a non-integer is entered, and then print out the smallest integer value read in.

```cpp
int num, smallest = 0;
while (cin >> num) {
    if (num < smallest) {
        smallest = num;
    }
}
cout << smallest << endl;
```

What would be printed if the first value entered by the user was a non-integer? Explain.

The first input value would cause the while test to fail, going directly to the cout statement. Since smallest was initialized to 0, this is the value that will be printed.

Assuming there are some input integers, is there any other way in which this code could produce the wrong answer? Explain.
If all of the input integers are negative, then the smallest variable will never be updated. Thus, 0 will be printed even though that is not the smallest value.
QUESTION 111:
Consider the `WordStreamIterator` class defined in the text (pp. 250-252).

```cpp
class WordStreamIterator
{
    public:
        WordStreamIterator(); // constructor
        void Open(string name); // bind stream to text file
        void First(); // initialize iterator
        string Current(); // returns current word
        bool IsDone(); // true if iterator is done
        void Next(); // advances to next word
    private:
        string myWord; // the current word
        bool myDone; // true if no more words
        ifstream myInput; // stream to read from
};
```

A. Write a code segment which creates a `WordStreamIterator` called `text` and initializes it to be able to read from a file called "words.dat".

```cpp
WordStreamIterator text;
text.Open("words.dat");
```

B. The following code segment is intended to read in all of the words from the text file, printing out each words as it is read in. However, this code segment does not behave as intended. Why not? Modify the code so that it does work.

```cpp
text.First();
while (!text.IsDone()) {
    cout << text.Current() << endl;
}
```

This code will print the first word in the file over and over and over. Since the `Next` method is never called, the iterator never advances. The following call needs to be placed inside the loop after the `cout` statement:

```cpp
text.Next();
```
QUESTION 112:
An industrious C++ student was working on her home computer when she found that the following code segment behaved strangely.

```
  double salary = 16400.49;
  cout << int(salary) << endl;
  cout << int(2*salary) << endl;
```

The first cout statement printed 16400 as expected, but the second cout printed -32736. Explain why this happened. Would this code necessarily produce the same output on a different machine?

Presumably, the student is working on a machine where `INT_MAX` is 32767. If a value is larger than the maximum integer, then casting it to an int may produce unexpected results.

It is possible that on another machine, if the range of type `int` is larger, the cast will behave as expected.

QUESTION 113:
What is output by the following program?

```
#include <iostream.h>

void Func(int x, int & y)
{
  x += 5;
  y += x;
  cout << "Inside: " << x << " " << y << endl;
}

int main()
{
  int a = 10, b = 20;
  Func(a, b);
  cout << "Outside: " << a << " " << b << endl;

  return 0;
}
```

Inside: 15 35
Outside: 10 35
QUESTION 114:
TRUE or FALSE? By default, parameters in C++ are reference parameters.

FALSE

QUESTION 115:
Consider the program below which prints the number 1.

```cpp
#include <iostream.h>

void Change(int num) {
    num = 22;
}

int main() {
    int small = 1;
    Change(small);
    cout << small << endl;
    return 0;
}
```

Why is the number 1 printed when the program runs (as opposed to the number 22)?

Since small is passed by value, changes to the formal parameter num do not effect the argument's value.

Describe a modification to the function Change that would cause the number 22 to be printed in the program above. (Only Change can be altered).

If small is passed by reference, then the formal parameter num becomes an alias for the argument, and so assignments to num also affect small. The header to Change must be altered:

```cpp
void Change(int & num);
```
QUESTION 116:
What is output by the following program?

```cpp
#include <iostream.h>

void Func(int & x, int & y)
{
    x += 1;
    y += 2;
    cout << "Inside: " << x << " " << y << endl;
}

int main()
{
    int x = 5, y = 10;
    Func(y, x);
    cout << "Outside: " << x << " " << y << endl;
    return 0;
}
```

```
Inside: 11 7
Outside: 7 11
```

QUESTION 117:
TRUE or FALSE? In the function whose header is given below, any attempt to alter the parameter `emp` inside this function will result in a compiler error.

```cpp
void PrintEmployee(const Employee & emp)
```

**TRUE**

QUESTION 118:
TRUE or FALSE? In C++, generic data types (e.g. a list of items of some type X) can be implemented using templates.

**TRUE**
QUESTION 119:
What is output by the following program?

```cpp
#include <iostream.h>

void Func1(int x, int y)
{
    x += 1;
    y += 2;
    cout << "Inside1: " << x << " " << y << endl;
}

void Func2(int & x, int & y)
{
    x += 1;
    y += 2;
    cout << "Inside2: " << x << " " << y << endl;
}

int main()
{
    int a = 3;
    Func1(a, a);
    cout << "Outside1: " << a << endl;

    int b = 9;
    Func2(b, b);
    cout << "Outside2: " << b << endl;

    return 0;
}
```

```
Inside1: 4 5
Outside1: 3
Inside2: 12 12
Outside2: 12
```
Chapter 7 Questions

QUESTION 121:
Most of the classes that we have studied in this course have data fields which are private and member functions which are public. Does it make sense to have private member functions? If so, give an example where a private member function would be useful. If not, explain.

Private member functions do make sense as help functions for the other member functions. For example, in the RandomWalk class, several of the member functions needed to be able to simulate taking a single step. Instead of having each of these functions contain the same code, the code for taking a step could be abstracted away into a separate function. Since that function is intended for the use of the other member functions only (not for a client program), the member function is declared private.

QUESTION 122:
Consider the following code segment:

```cpp
enum Color {red, yellow, green, blue};

Color fave = green;
if (fave == blue || fave == green) {
    fave = red;
}
cout << fave << endl;
```

What is output by this code segment? Justify your answer.

- blue
- green
- red
- none of the above

none of the above -- an enumeration is simply a name given to some integer constant. By default, the constants are assigned starting at zero, so red==0, yellow==1, etc. In this case, fave ends up as red, so the integer 0 is output.
QUESTION 123:
TRUE or FALSE? According to Software Engineering researcher Mary Shaw, "Software now accounts for the lion’s share of the cost of developing and using computer systems."

TRUE

QUESTION 124:
Consider the Coin class as discussed in the text (pp. 310-312).

```
#include "dice.h"

eenum FlipValue {heads, tails}; // possible coin flips

class Coin
{
  public:
    Coin();              // constructor
    FlipValue Flip();    // return the random flip
    int NumFlips();      // # times this coin flipped
  private:
    Dice myDie;          // will be a two-sided die
};
```

Why must the dice.h file be included here?

Since the Coin class has a data field of type Dice, the header file which defines the Dice class must be included.

Complete the function DecisionMaker whose header is given below:

```
string DecisionMaker()
// postcondition: returns either "yes" or "no", depending on a coin flip
{
  Coin penny;
  if (heads == penny.Flip()) {
    return "yes";
  }
  return "no";
}
```
QUESTION 125:
Consider the `Person` struct type, which can be used to store information about a person. Given this type, complete the function `Birthday` whose header is given below:

```cpp
struct Person {
    string firstName, lastName;
    char middleInitial;
    int age;
}

void Birthday(Person & whomever)
// postcondition: whomever is altered so that the
// person's age is incremented by 1
```

```cpp
void Birthday(Person & whomever)
// postcondition: whomever is altered so that the
// person's age is incremented by 1
{
    whomever.age++;
}
```

QUESTION 126:
TRUE or FALSE? The only difference between a struct and a class is that by default all data and functions in a struct are public, whereas the default in a class is that everything is private.

**TRUE**

QUESTION 127:
TRUE or FALSE? Input and output streams should always be passed by-value.

**FALSE**
Chapter 8 Questions

QUESTION 128:
Which of the following properties is not true of an array data type?

- each item in the array is of the same type
- items in the array are numbered by an index
- accessing the first item in the array is faster than accessing the last item

accessing the first item in the array is faster than accessing the last item

QUESTION 129:
Write a declaration that creates a Vector of 50 strings called words.

Vector<string> words(50);

QUESTION 130:
TRUE or FALSE? The following declaration creates a Vector of 100 integers, all initialized to zero.

Vector<int> nums(100,0);

TRUE
QUESTION 131:

Consider the following incomplete function. When completed, this function will repeatedly generate random numbers in some range (1 to n) and keep a frequency count of each random number generated. The, it will print out all of the counts.

```cpp
void CountRandoms(int n)
// precondition : n > 0
// postcondition: picks 1000 random numbers in range 1..n,
//                and displays the frequency of each number
{
    RandGen rando; // random number generator
    Vector<int> counts(n+1, 0); // create and init counters

    int rep;
    for (rep = 0; rep < 1000; rep++) {
        int num = rando.RandInt(1, n);

        // insert code to increment corresponding counter
    }

    int k;
    for (k = 1; k <= n; k++) {
        cout " # of " << k << "'s = " << counts[k] << endl;
    }
}
```

A. If you are only generating random numbers in the range 1 to n, why is the `counts` Vector declared to be of size n+1?

Each element of the `counts` Vector is a count corresponding to a random number. Since the random numbers start at 1 but the first index is 0, you have a choice. Either create a Vector of size n and then map each random number to an index one smaller, or else create a Vector of size n+1 (indexed from 0 to n) and essentially waste the 0 index.

B. Add code at the specified location to complete this function.

```cpp
Insert the following line: counts[num]++;
```
QUESTION 132:
TRUE or FALSE? Suppose that nums is a Vector of 100 integers. The following code segment will correctly display the contents of nums.

```cpp
int k;
for (k = 1; k <= 100; k++) {
cout << nums[k] << endl;
}
```

FALSE (the array is indexed from 0 to 99)

QUESTION 133:
All of the entries in the array A are to be shifted one position so that a new entry can be added at the front. Assuming that there are numEntries entries in the array, which of the following statements correctly shifts the entries?

A) for (int index = 0; index <= numEntries; index++) {
   A[index+1] = A[index];
}

B) for (int index = numEntries; index >= 1; index--) {
   A[index+1] = A[index];
}

C) for (int index = numEntries; index >= 0; index--) {
   A[index+1] = A[index];
}

D) for (int index = 0; index < numEntries; index++) {
   A[index+1] = A[index];
}

E) for (int index = numEntries-1; index >= 0; index--) {
   A[index+1] = A[index];
}

E) for (int index = numEntries-1; index >= 0; index--)
   A[index+1] = A[index];
QUESTION 134:
Complete the following function called \texttt{Max}, which returns the maximum value stored in a Vector.

```c
int Max(Vector<int> nums, int size)
// precondition: nums contains size integers (size > 0)
// postcondition: returns maximum value in nums
{
    int maxSoFar = nums[0];
    int k;
    for (k = 1; k < size; k++) {
        if (nums[k] > maxSoFar) {
            maxSoFar = nums[k];
        }
    }
    return maxSoFar;
}
```

QUESTION 135:
Complete the function whose header is given below:

```c
void DoubleEach(Vector<int> & nums, int numNums)
// precondition: nums contains numNums integers
// postcondition: each number in nums is doubled
{
    int k;
    for (k = 0; k < numNums; k++) {
        nums[k] *= 2;
    }
}
```
QUESTION 136:
Write the function $\text{Sum}$ whose header is given below.

```cpp
double Sum(Vector<double> nums, int numNums)  
// precondition : nums contains numNums doubles  
// postcondition: returns nums[0]+...+nums[numNums-1]
{
    double sumNums = 0.0; 
    int k;  
    for (k = 0; k < numNums; k++) {  
        sumNums += nums[k];  
    }  
    return sumNums;  
}
```

QUESTION 137:
Write the function $\text{OddProd}$ whose header is given below (pay attention to the postcondition).

```cpp
double OddProd(Vector<double> nums, int numNums)  
// precondition: nums contains numNums doubles  
// postcondition: returns nums[1]*nums[3]*...*nums[k]  
// where k is largest odd # < numNums  
// (if numNums < 2, returns 1.0)
{
    double product = 1.0; 
    int k;  
    for (k = 1; k < numNums; k += 2) {  
        product *= nums[k];  
    }  
    return product;  
}
```
QUESTION 138:

Complete the following function called Scale, which adds bonus points to each grade in a Vector. Note that for this problem, grades are not allowed to exceed 100 -- any grade which would exceed 100 with the bonus added should simply be set to 100.

```cpp
void Scale(Vector<int> grade, int numGrades, int bonus)
// precondition: grade contains numGrades integers
// postcondition: each grade is increased by the bonus amount (but will never exceed 100)
{
    int k;
    for (k = 0; k < numGrades; k++) {
        grade[k] += bonus;
        if grade[k] > 100 {
            grade[k] = 100;
        }
    }
}
```

QUESTION 139:

Consider a list of 1000 names, sorted in alphabetical order. If you are searching for a particular name using sequential search, how many names will you have to inspect (WORST CASE)? How about using binary search (WORST CASE)?

Sequential search: 1000 (may have to inspect every name)
Binary search: 10 (repeated halving reduces 100 down to 1 in only 10 steps)
QUESTION 140:
Assume the existence of a function `MaxIndex` that returns the index (location) of the largest value in a Vector:

```cpp
int MaxIndex(Vector<double> A, int numElements)
// precondition: A contains numElements doubles
// postcondition: returns index of largest value in A
```

Complete the definition of the `MaxOut` function below that sets all of the elements of the Vector to the largest of its elements. For example, if `list` represents (1,2,5,3,4,2), then the call `MaxOut(list, 6)` should change list so that it represents (5,5,5,5,5,5).

```cpp
void MaxOut(Vector<double> & A, int numElts)
// precondition: A contains numElts integer values
// postcondition: all elements of A are assigned the max
//                of the original values in list A
{
    double maxValue = A[MaxIndex(A,numElts)];

    int k;
    for (k = 0; k < numElts; k += 1) {
        A[k] = maxValue;
    }
}
```
QUESTION 141:
Assume that a Vector of integers is to be kept in sorted order. The function `FindIndex` whose header is given below should return the location of where a specific integer appears (or should appear) in a Vector. For example, given the Vector `A` diagrammed below:

```
13 | 21 | 33 | 52 | 61 | 72
```

the call `FindIndex(A, 6, 52)` should return 3 since 52 is located in `A` with index 3. The call `FindIndex(A, 6, 19)` should return 1 since if 19 was inserted into `A` it would go in the location with index 1. Note that the second parameter to `FindIndex` is the number of elements in the Vector. Complete the definition of `FindIndex`:

```cpp
int FindIndex(Vector<int> A, int numElts, int key)
// precondition: A is sorted and contains numElts ints
// postcondition: return index of location of key in A
//                if not found, return index of where
//                it would be if inserted into list
{  
    int index = 0;
    while (index < numElts && key > A[index]) {
        index++;
    }
    return index;
}
```

QUESTION 142:
Write a code segment to shift all `n` elements of a Vector `A` one place to the right leaving an "empty" spot in the zero-th location. After shifting, store the value 0 in `A[0].`

```cpp
int k;
for (k = n; k > 0; k--) {
    A[k] = A[k-1];
}
A[0] = 0;
```
QUESTION 143:
The function below is intended to store value in the Vector A so that the elements of A remain in sorted order.

```cpp
void Insert(Vector<int> & A, int & numElts, int value)
// postcondition: value is stored in A, still sorted
{   
    int k = numElts;
    while (k > 0  &&  value <= A[k-1]){
        A[k] = A[k-1];
        k--;
    }
    // store value in the proper location here
    numElts++;
}
```

Why is numElts passed as a reference parameter?

Since numElts will change, the parameter must be a reference parameter or else the change will not affect the argument.

What is the purpose of the  \( k > 0 \)  part of the loop guard?

This test guards against an index out-of-bounds error in the case where the new value is added at the beginning of the Vector.

What single statement should be added after the loop to store value in its proper location?

```cpp
A[k] = value;
```

What would happen if this function were called on a Vector that was already full, i.e., numElts == A.length()? How could you avoid this problem?

An index out-of-bounds error would occur when you tried to shift into the numElts index. You would need to check the size of the Vector before shifting, and resize it to be larger if more space was needed.
QUESTION 144:
Which of the following computer scientists is best known for his advocation of the use of formal methods in designing and implementing software and in the training of undergraduates in computer science?

- Alan Turing
- John Kemeny
- David Gries

David Gries

QUESTION 145:
TRUE or FALSE? Unlike Vectors, attempting to access an array element with an out-of-bounds index will not necessarily cause a run-time error.

TRUE

QUESTION 146:
Unlike other C++ types, an array that is passed by-value to a function is not copied. Changes made to the parameter inside a function can affect the original array argument. Explain why this is the case.

An array is like a handle that can be used to access the individual elements. When an array is passed by-value, the handle is copied as with any other type, but the copy of the handle allows access to the original array elements, allowing them to be changed.
Chapter 9 Questions

QUESTION 147:
Which of the following best summarizes the Church-Turing thesis?

- Given enough time, a computer can solve any problem.
- From a theoretical standpoint, all computers have the same power.
- The speed of computers doubles roughly every two years.

From a theoretical standpoint, all computers have the same power.

QUESTION 148:
What is output by the following code segment? Which if any of the assignments are illegal?

```cpp
char ch1 = 'A';
cout << ch1 << endl;
char ch2 = ch1 + 1;
cout << ch2 << endl;
int diff = 'd' - 'b';
cout << diff << endl;
```

A
B
2

QUESTION 149:
Write a for loop which prints out the characters 'A' through 'Z' in reverse order (i.e. starting with 'Z' and ending with 'A').

```cpp
char ch;
for (ch = 'Z'; ch >= 'A'; ch--) {
    cout << ch << endl;
}
```
QUESTION 150:
Recall that the string member function `length` can be used to determine the number of characters in a string. Since the `>>` operator reads strings delimited by whitespace, the strings extracted from a text file are liable to contain punctuation marks. Complete the function whose header is given below for counting non-punctuation characters in a string. For example, `NoPuncLength("foo!")` should return 3, while `NoPuncLength("that's,"`) should return 5. Recall: the `ctype.h` library file includes the `ispunct` function, which returns true if its char argument is a punctuation mark, else false.

```cpp
#include <ctype.h>

int NoPuncLength(string str)
// postcondition: returns # of non-punct chars in str
```

```cpp
int NoPuncLength(string str)
// postcondition: returns # of non-punct chars in str
{
    int k, count = 0;
    for (k = 0; k < str.length(); k++) {
        if (!ispunct(str[k])) {
            count++;
        }
    }
    return count;
}
```

QUESTION 151:
TRUE or FALSE? As was the case with Vectors, attempting to access a character in a string with an out-of-bounds index will cause program termination.

TRUE

QUESTION 152:
TRUE or FALSE? In addition to his great contributions during the early years of computers, John von Neumann is also considered to be the founder of Game Theory.

TRUE
QUESTION 153:
What is an Abstract Data Type (ADT)? How are ADT’s implemented in C++?

An abstract data type is a collection of data elements and the associated operations that are applied to that data. For example, a list is an ADT in which the data is a sequence of items on which you can perform operations such as Insert, Select, and Find. In C++, ADT’s are implemented as classes: the data elements are implemented as data fields of the class, and the operations are member functions.

QUESTION 154:
The following code segment reads lines of text from standard input and prints the total number of lines read in. Augment this code so that it also prints out the average number of characters per line.

```cpp
int num_lines = 0;

string str;
while ( getline(cin, str) ) {
    num_lines++;
}

cout << "Number of lines = " << num_lines << endl;

int num_lines = 0, num_chars = 0;
string str;
while ( getline(cin, str) ) {
    num_lines++;
    num_chars += str.length();
}

if (num_lines > 0) {
    cout << "Avg number of chars per line = "
    << double(num_chars)/num_lines << endl;
}
```
QUESTION 155:

Consider the following code segment which reads integers from a file called "nums.in" and echoes them to a file called "nums.out":

```c++
ifstream input;
input.open("nums.in");

ofstream output;
output.open("nums.out");

int num;
while (input >> num) {
    output << num << endl;
}
```

A. Assuming the file "nums.in" exists and contains only integers, will the file "nums.out" be identical to "nums.in" after executing this code? Why or why not?

No. Since each number is printed followed by an endl, the output file will have one integer per line. This need not be the case in the input file, since the >> operator recognizes any whitespace as separating numbers.

B. What would happen if there was no input file named "nums.in"? Would there be a run-time error? What would happen to "nums.out"?

There would be no error, but the contents of "nums.out" would be wiped out (if that file already existed). This is due to the fact that even though the open operation failed, the code would still proceed to open the output stream (erasing any existing "nums.out"). The while loop test would fail immediately since the input stream is not opened properly, so the output file would remain empty.

C. Add to the above code segment so that if the input file does not exist, an error message is displayed and no further work is attempted.

```c++
ifstream input;
input.open("nums.in");

if (input.fail()) {
    cout << "No such file!" << endl;
} else {
    ofstream output;
    output.open("nums.out");

    while (input >> num) {
        output << num << endl;
    }
}
```
QUESTION 156:
Consider the following simple function.

```c++
void DoIt(int num = 0)
{
    cout << num << endl;
}
```

What (if anything) is output as a result of call `DoIt(5)`?

5

What (if anything) is output as a result of call `DoIt()`?

0

QUESTION 157:
What is output by the following code segment?

```c++
istringstream istr("11 43   9  800");
int num;
while (istr >> num) {
    cout << num << endl;
}
```

11
43
9
800

QUESTION 158:
Which of the following computer scientists invented the programming language Pascal?

- John Kemeny
- Mary Shaw
- Blaise Pascal
- Niklaus Wirth
Niklaus Wirth
**QUESTION 159:**

TRUE or FALSE? A flag is a variable or quantity that can have one of two values (e.g., true/false, on/off, yes/no).

**TRUE**

**QUESTION 160:**

What is output by the following code segment? Be specific.

```cpp
int k, PI = 3.14159;
for (k = 1; k <= 3; k++) {
    cout.precision(k);
    cout << PI << endl;
}
```

3.1
3.14
3.142

**QUESTION 161:**

Consider the following struct definition:

```cpp
struct Date {
    string day;
    string month;
    int year;
};
```

Complete the operator definition below which tests two dates for equality:

```cpp
bool operator==(const Date & d1, const Date & d2)
// postcondition: returns true if d1 and d2 are same date
//                otherwise, returns false
{
    return (d1.day == d2.day &&
            d1.month == d2.month &&
            d1.year == d2.year);
}
```
QUESTION 162:
Consider the following code segment involving double variables $x$ and $y$:

```java
double sum = 0;
int k;
for (k = 1; k <= y; k++) {
    sum += x/y;
}
```

After completing the loop, the variable `sum` should be equal to $x$. However, depending on the values of $x$ and $y$, this may not be the case. The `sum` may be very close but not equal to $x$. Explain why this is the case.

There may be some roundoff error that causes the sum of all of the fractions not to add up precisely. This is due to the fact that floating-point values can only be stored with so many digits of precision, after that point values are rounded.

QUESTION 163:
TRUE or FALSE? The binary number 111 represents the (decimal) value 7.

**TRUE**

QUESTION 164:
Using a single bit, only two different values can be represented (0 or 1). Using four bits, how many different values can be represented?

16
Chapter 10 Questions

QUESTION 165:
Consider the following recursive function:

```cpp
void Something(int n)
{
    if (n == 0) {
        cout << "base case" << endl;
    } else {
        cout << "recursive case (" << n << ") " << endl;
        Something(n-1);
    }
}
```

A. What is returned by the call `Something(0)`?
B. What is returned by the call `Something(1)`?
C. What is returned by the call `Something(3)`?

A. base case
B. recursive case (1)
   base case
C. recursive case (3)
   recursive case (2)
   recursive case (1)
   base case

QUESTION 166:
Is the following recursive function guaranteed to terminate on all string inputs? Explain.

```cpp
int stopper(string str)
{
    if (str.length() < 10) {
        cout << str;
        stopper(str + "?");
    }
}
```

Yes, this function will terminate. There is a base case (when the length of the string is \( \geq 10 \)), and every recursive call gets closer to the base case (the string is getting longer).
QUESTION 167:
Consider the following recursive function:

```c
int foo(int num) {
    if (num == 1) {
        return 0;
    } else {
        return (1 + foo(num/2));
    }
}
```

A. What is returned by the call `foo(1)`?

B. What is returned by the call `foo(4)`?

C. What is returned by the call `foo(1024)`?

A. `foo(1)` --> 0
B. `foo(4)` --> 2
C. `foo(1024)` --> 10

QUESTION 168:
What is output by the following program?

```c
#include <iostream.h>
int x = 1, y = 2;

void Func(int y) {
    x = 33;
    cout << x << " , " << y << endl;
}

int main() {
    int y = 9;
    Func(x);
    cout << x << " , " << y << endl;
    return 0;
}
```

33 1
33 9
QUESTION 169:
Consider the recursive Fibonacci function discussed in the text (with an additional `cout` statement added).

```cpp
int RecFib(int n)
// precondition: n >= 0
// postcondition: returns the nth Fibonacci number
{
    cout << "Recursive call" << endl;
    if (0 == n || 1 == n) {
        return 1;
    }
    else {
        return RecFib(n-1) + RecFib(n-2);
    }
}
```

For each of the following calls, how many times is the message "Recursive call" printed?

- RecFib(0) --> 1
- RecFib(1) --> 1
- RecFib(2) --> 3
- RecFib(3) --> 5
- RecFib(4) --> 9

QUESTION 170:
TRUE or FALSE? Declaring a variable to be `static` extends its scope to the entire program.

FALSE
QUESTION 171:
What is output by the following program?

```cpp
#include <iostream.h>

void Func()
{
    int count = 0;
    count += 1;
    cout << count << endl;
}

int main()
{
    Func();
    Func();
    Func();
    Func();
    return 0;
}
```

If the `count` variable in `Func` were made static, i.e. the declaration was changed as below, what would be the output of the program?

```cpp
static int count = 0;
```

QUESTION 172:
To whom is the following quote attributed?

The truth may be that computer science does not by itself constitute a sufficiently broad education, and that it is better studied in combination with one of the physical sciences or with one of the older branches of engineering.

• Alan Turing
• Richard Stallman
• Maurice Wilkes
Maurice Wilkes
Chapter 11 Questions

QUESTION 173:
TRUE or FALSE? Bubble sort is a dog!

TRUE

QUESTION 174:
Consider the selection sort algorithm discussed in the text.

```
template <class Type>
void Sort(Vector<Type> & a, int numElts)
// precondition : a contains numElts items
// postcondition: a is sorted in non-decreasing order
{
    int k, index;
    for (k = 0; k < numElts-1; k++) {
        index = MinIndex(a, k, numElts-1);
        Swap(a[k], a[index]);
    }
}
```

A. In order to sort a Vector with 100 elements, how many times will the `Swap` function be called?

99

B. Write the missing templated `Swap` function which swaps the values of its two arguments.

```
template <class Type>
void Swap(Type & val1, Type & val2)
// postcondition: values of val1 and val2 are swapped
{
    Type temp = val1;
    val1 = val2;
    val2 = temp;
}
```
QUESTION 175:
In C++, it is possible to overload function names by having different functions with the same name. These functions must have different parameter lists, however. Why is it illegal to have two functions with the same name that differ only in their return type?

Because the compiler would have no way of differentiating between the two -- calls to the functions would look the same.

QUESTION 176:
TRUE or FALSE? Shafi Goldwasser received the Godel prize in Theoretical Computer Science for her work with randomized algorithms, where random events such as coin flips are instrumental in making decisions.

TRUE

QUESTION 177:
TRUE or FALSE? The sum from 1 to \( N \) can be expressed by the formula \( \frac{N \times (N+1)}{2} \).

TRUE

QUESTION 178:
Quick sort is, on average, an \( O(n \log n) \) sorting algorithm. Worst case, however, it is \( O(n^2) \). Describe the factor which contributes to this difference in complexity.
The difference comes down to the choice of the pivot element. If the pivot yields partitions that are roughly equal in size, then only a logarithmic number of recursions are necessary to reach the base case (a list with 0 or 1 element). If the pivot is not roughly in the middle, however, the number of recursions can be linear on the size of the list, yielding a quadratic complexity.
QUESTION 179:
What is the order of complexity of each of the following sorting algorithms:

- quick sort
- selection sort
- insertion sort

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick sort</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>selection sort</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>insertion sort</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

QUESTION 180:
Consider the following declaration:

```cpp
Matrix<int> mat(10, 5);
```

A. How many integers can be stored in this matrix?

50

B. What type of value is returned by the expression `mat[0]`?

- int
- Vector of ints
- Matrix of ints

Vector of ints

C. Write a C++ statement which assigns the top-left element (1st row, 1st column) of `mat` to be zero.

```cpp
mat[0][0] = 0;
```
QUESTION 181:
The code below correctly sets all elements of a "square" matrix $M$ to 0. Alter the code so that diagonal elements are set to 1, but other elements are set to 0. For example, a 5x5 matrix should be set as in the diagram:

\[
\begin{array}{ccccc}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

```cpp
int r, c;
for (r = 0; r < M.Rows(); r++) {
    for (c = 0; c < M.Cols(); c++) {
        if (r == c) {
            M[r][c] = 1;
        } else {
            M[r][c] = 0;
        }
    }
}
```

```cpp
int r, c;
for (r = 0; r < M.Rows(); r++) {
    for (c = 0; c < M.Cols(); c++) {
        if (r == c) {
            M[r][c] = 1;
        } else {
            M[r][c] = 0;
        }
    }
}
```
**QUESTION 182:**

Complete the function whose header is given below:

```cpp
int MatrixSum(const Matrix<int> & mat)
// postcondition: returns the sum of all numbers in mat
{
    int r, c, sum = 0;
    for(r = 0; r < M.Rows(); r++){
        for(c = 0; c < M.Cols(); c++){
            sum += mat[r][c];
        }
    }
    return sum;
}
```

**QUESTION 183:**

TRUE or FALSE? When passing a multi-dimensional array as a parameter, all dimensions must be given as constants in the function header/prototype.

FALSE
Chapter 12 Questions

QUESTION 184:
TRUE or FALSE? The following declaration creates two pointers to integers.

```c
int * p, q;
```

FALSE

QUESTION 185:
What is output by the following code segment?

```c
int num = 20;
int * ptr = &num;
*ptr = 5;
cout << *ptr << " " << num << endl;
```

5 5

QUESTION 186:
TRUE or FALSE? Attempting to dereference a pointer whose value is 0 (or NULL) results in a run-time error.

TRUE
QUESTION 187:
The C++ function shown below sets both of its parameters to zero, so that the call `MakeZero(a, b)` sets variables `a` and `b` to 0.

```cpp
void MakeZero(int & first, int & sec)
{
    first = 0;
    sec = 0;
}
```

The header for a similar C function is shown below. Fill in the function body so that the call `MakeZero(&a, &b)` would set both `a` and `b` to 0.

```cpp
void MakeZero(int * first, int * sec)
{

}
```

```cpp
void MakeZero(int * first, int * sec)
{
    *first = 0;
    *sec = 0;
}
```

QUESTION 188:
Which of the following expressions is equivalent to the notation: `ptr->data`?

- `*ptr.data`
- `(*(ptr).data)`
- `*(ptr.data)`
- `(*ptr).data`
QUESTION 189:
Assume that a program contains the following function.

```cpp
void Dummy(char & z, char * ptr)
{
    *ptr = 'o';
    cout << z << *ptr;
}
```

What is written by the following code segment?

```cpp
char c = 'f';
char * p = &c;
Dummy(c, p);
cout << c << *p << endl;
```

A) ffff  
B) fofo  
C) ooff  
D) foff  
E) oooo

E) oooo

QUESTION 190:
TRUE or FALSE? Alan Perlis was the first recipient of the Turing Award, which represents the highest achievement in the field of computer science.

TRUE

QUESTION 191:
What is a destructor? When is it called and what is its purpose?

A destructor is a special member function that is automatically called when an object of that class is destroyed (its lifetime ends). A destructor is used to perform any "clean-up operations", primarily deallocating any dynamically allocated memory from within the class object.
QUESTION 192:
The following statement creates an array of 100 integers called *nums*:

```c
int * nums = new int[100];
```

Write a statement which assigns the first element of this array to be 0.

```c
nums[0] = 0;
```

Write a statement which subsequently deletes the allocated memory for reuse.

```c
delete [] nums;
```

QUESTION 193:
Consider the following code segment which defines the *pair* struct and dynamically allocates such a struct:

```c
struct pair {
    int x, y;
};
pair * p = new pair;
```

Write a statement which sets the *x* field of the struct pointed to by *p* to be 0.

```c
p->x = 0;
```

QUESTION 194:
TRUE or FALSE? Unlike a Vector, a linked list only provides sequential access to its elements.
TRUE
QUESTION 195:
What is the purpose of the following code segment (assume that list points to the front of a linked list of IntNodes)? Feel free to draw pictures to illustrate your explanation.

```c
IntNode * step = list;
while (step != NULL) {
    IntNode * temp = new IntNode;
    temp->info = step->info;
    temp->next = step->next;
    step->next = temp;
    step = step->next->next;
}
```

Every node in the linked list is duplicated. For example, if the list contained the numbers 1, 2, and 3 (in that order), then the resulting list would contain 1, 1, 2, 2, 3, 3.

QUESTION 196:
Add code to the loop below so that all nodes of the linked list of integers contain zero.

```c
struct IntNode{
    int info;
    IntNode * next;
};

while (list != NULL){
    while (list != NULL){
        list->info = 0;
        list = list->next;
    }
}
```

while (list != NULL){
    list->info = 0;
    list = list->next;
}
QUESTION 197:
Write the function LastNode whose header is given below. LastNode returns a pointer to the last node of list.

```c
struct IntNode{
    int info;
    IntNode * next;
};

IntNode * LastNode(IntNode * list)
// postcondition: returns pointer to last node in list
//                (returns NULL if list is empty)
{
    if (list == NULL) {
        return NULL;
    }
    while (list->next != NULL) {
        list = list->next;
    }
    return list;
}
```

QUESTION 198:
Complete the definition of the NodeCount function whose header is given below.

```c
int NodeCount(Node * list)
// postcondition: returns number of nodes in linked-list
{
    int count = 0;
    while (list != NULL) {
        count++;
        list = list->next;
    }
    return count;
}
```
QUESTION 199:
Complete the function below that determines the total number of characters stored in all nodes of a list.

```c
struct StringNode{
    string word;
    StringNode * next;
};

int CountChars(StringNode * list)
// precondition : list points to a linked-list
// postcondition: returns total number of
//                characters in all strings
{
    int count = 0;
    while (list != NULL){
        count += list->word.length();
        list = list->next;
    }
    return count;
}
```
QUESTION 200:
Consider a function for deleting the last node in a linked list.

```c
struct IntNode {
    int info;
    IntNode * next;
};

void DeleteLast(IntNode * & list)
// precondition : list points to a linked-list of ints
// postcondition: deletes last node (if nonempty)
```

A. Looking at the header above, why is the pointer to the linked list passed by-reference?

| If there is only one node in the list, then the list pointer must be changed to NULL. If the pointer is passed by-value, it is impossible affect the argument. |

B. Complete the definition of this function. Note: no memory leakage should occur.

```c
void DeleteLast(IntNode * & list)
// precondition : list points to a linked-list of ints
// postcondition: deletes last node (if nonempty)
{
    if (list != NULL) {
        if (list->next == NULL) {
            delete list;
            list = NULL;
        } else {
            IntNode * step = list;
            while (step->next->next != NULL) {
                step = step->next;
            }
            delete step->next;
            step->next = NULL;
        }
    }
}
```
QUESTION 201:
Consider a simplified List class which only allows for adding, printing, and determining the length of the list.

```cpp
template <class Type>
class SimpleList
{
    public:
        SimpleList();
        void Add(const Type & t);
        void Print() const;
        int Length() const;
    private:
        // assorted data fields
};
```

A. Write a declaration which creates a SimpleList called nums which can be used to store integers.

```cpp
SimpleList<int> nums;
```

B. What is the purpose of the keyword const at the end of the Print and Length function prototypes? Why doesn't Add have const at the end?

The const specifies that these member functions are guaranteed not to change any of the data fields of the class object. This is necessary if they are to be called on a SimpleList that has been passed as a const-reference parameter at some point. Since Add does need to change the list, it cannot be declared as const.

QUESTION 202:
What is output by the following code segment?

```cpp
char letter[3] = {'a', 'b', 'c'};
char * ptr = letter;
*(ptr+1) = 'x';
```

```
axc
```
QUESTION 203:

Given an array of integers called grade, each of the following statements assigns the first element to be a zero. Explain why the seemingly nonsensical expression $0[\text{grade}]$ is equivalent to $\text{grade}[0]$.

$$
\text{grade}[0] = 0;
0[\text{grade}] = 0;
$$

Underlying array indexing is pointer arithmetic and dereferencing. The expression $\text{grade}[0]$ is equivalent to $(\text{grade} + 0)$, i.e., add 0 to the grade pointer and then dereference. Since arithmetic is reflexive, this is equivalent to $*(0 + \text{grade})$, or $0[\text{grade}]$.

QUESTION 204:

TRUE or FALSE? Suppose words has been declared as an array of strings. Then the expressions words[2] and *(words+2) refer to the same string.

**TRUE**

QUESTION 205:

Suppose nums is a dynamically allocated array storing 100 integers. What is the result of the following code segment?

```c
int * temp = new int[200];

int k;
for (k = 0; k < 100; k++) {
    temp[k] = nums[k];
}

delete [] nums;
nums = temp;
```

The final effect is that nums has been increased in capacity to 200 elements, with the original elements intact. In reality, nums is pointing to a new sequence of memory cells,
twice as big as before, with values copied over from the original memory cells. After copying, the original memory cells were deallocated.
4. Sample Assignments

This chapter contains ten sample programming assignments which can be used in conjunction with the text. These assignments stress two of the themes of the text: (1) the importance of abstraction and code reuse, and (2) the use of programs as tools for solving problems and analyzing phenomena. Many of the assignments involve the use of classes developed and defined in the text. For example, Assignment 8 involves writing a spell-checker that uses the `WordStreamIterator` class to read in words, and a variant of the `WordList` class to store and look up words. Forcing students to use existing classes sends the right message about code reuse, and also allows for more complex and interesting problems to be solved. Most of these assignments also have an analytical component as well. In addition to code that they must write, the students are asked use their program to perform simulations and analyze the results. For example, Assignment 6 involves implementing a variant of the `RandomWalk` class, experimenting with the two variants, and analyzing their relationship. Forcing students to think about and explain the results produced by their programs sends the message that programs are not ends unto themselves, but tools for solving (often complex) problems.

The text follows a use-read-modify-write approach to classes, where students first use classes without worrying about their implementation, then study existing classes, make small modifications to classes, and finally write their own classes. This step-by-step approach is also followed in the assignments. Assignments 1 through 4 involve writing programs that use existing classes, but don't require an understanding of the internal workings of the classes. Assignments 5 and 6 involve taking classes defined in the text and understanding them well enough to make slight modifications. Assignments 7 through 10 involve writing major portions of classes (although the framework for each class is described) and integrating those classes into larger programs.

As a precursor to these ten assignments, it is often useful to begin the course with a mini-assignment to familiarize the students with their particular C++ environment and the procedure for submitting programs. This assignment, call it Assignment 0, can be as simple as typing in the "hello world" program, compiling and running it, and submitting a program listing. If you are using some system of electronic submission, this can be tested here as well. In this way, the details of how to create, compile, and submit a program are handled early, before you start focusing on C++ features.

Special attention has been made to assure that these assignments are applicable in both open lab and closed lab environments. The assignments are reasonably self-contained (with liberal references to the text, of course), and so could be given to students for their completion outside of class. The experimental and analytical nature of many of the assignments also lends itself well to closed labs, where the students are expected to work on and possibly complete the assignment in a specific lab period. As always, you may need to adjust the length and/or difficulty of assignments to suit your particular situation.
Assignment 1

Simple Programs and Class Use

For this assignment, you will write several small programs in order to become more comfortable with the basic structure of C++ programs and the compilation process. These programs involve output (via cout statements), input (via cin statements), arithmetic computation, and the use of a previously defined class.

PART 1: Write a C++ program called metaname.cc that asks for a name (first and last name) and prints out a C++ program that when compiled and run, prints out the name. The program produced by your program should be broken up onto separate lines and properly indented. For example, below is a sample run (with user input in bold):

> metaname
Enter first and last name: Bjarne Stroustrup
#include <iostream.h>

int main()
{
    cout << "Bjarne Stroustrup" << endl;
    return 0;
}

PART 2: Write a C++ program called ages.cc that asks for a person's age (a long integer) and prints out that age using different units of time. In particular, you should print out the person's age in months, in days, in hours, and in seconds. To simplify matters, you may ignore leap years and assume that every year is 365 days long. For example:

> ages
How many years old are you? 20
20 years is equivalent to:
    240 months
or 7300 days
or 175200 hours
or 10512000 minutes
or 630720000 seconds
PART 3: Consider the Balloon class described in section 3.4 of the text. Write a program called fly.cc that uses a variable of type Balloon and performs the following sequence of actions:

1. Prompt the user for desired altitude and a number of time steps.
2. Cause the balloon to ascend to one-third of the specified altitude, then cruise for one-quarter of the time steps.
3. Repeat step 2 twice more (ascend one-third, cruise one-quarter, ascend one-third, cruise one-quarter).
4. Descend halfway to the ground, cruise for the remaining time steps, and then descend back to the ground.

For example, below is part of a sample run (with user input in bold):

> fly
Welcome to Balloons-R-Us.
How high (in meters) would you like to go today? 30
How long (in minutes) do you want your flight to be? 12

***** (Height = 0) Ascending to 10 meters *****
0 meters  Burn! ...

***** Cruising at 10 meters  with margin +/- 5 for 3 time-steps *****
10 meters (time step 0)
10 meters (time step 1)  wind-shear drop 5 meters
5 meters (time step 2)  wind-shear bump up 1 meters

***** (Height = 6) Ascending to 20 meters *****
6 meters  Burn! ...
16 meters  Burn! ...

***** Cruising at 26 meters  with margin +/- 5 for 3 time-steps *****
26 meters (time step 0)  wind-shear drop 4 meters
22 meters (time step 1)
22 meters (time step 2)

***** (Height = 22) Ascending to 30 meters *****
22 meters  Burn! ...

•
•
•
Assignment 1 Solution

#include <iostream.h>
#include "CPstring.h"

int main()
{
    string firstName, lastName;
    cout << "Enter first and last name: ";
    cin >> firstName >> lastName;

    cout << "#include <iostream.h>" << endl;
    cout << endl;
    cout << "int main()" << endl;
    cout << "{" << endl;
    cout << "    cout << \"" << firstName << \" " << lastName << \"\" << endl;" << endl;
    cout << endl;
    cout << "    return 0;" << endl;
    cout << "}" << endl;

    return 0;
}

#include <iostream.h>

int main()
{
    long int age;
    cout << "How many years old are you? ";
    cin >> age;

    cout << age << " years is equivalent to:" << endl;
    cout << " " << age*12 << " months" << endl;
    cout << " or " << age*365 << " days" << endl;
    cout << " or " << age*365*24 << " hours" << endl;
    cout << " or " << age*365*24*60 << " minutes" << endl;
    cout << " or " << age*365*24*60*60 << " seconds" << endl;

    return 0;
}

172
// File : fly.cc
// Author: Dave Reed

#include <iostream.h>
#include "balloon.h"

int main()
{
    Balloon montgolfier;
    int height;  // how high to fly (meters)
    int duration;  // how long to fly (minutes)

    cout << "Welcome to Balloons-R-Us." << endl;
    cout << "How high (in meters) would you like to go today? ";
    cin >> height;
    cout << "How long (in minutes) do you want your flight "
        << "to be? ";
    cin >> duration;

    montgolfier.Ascend(height/3);  // ascend to 1/3 height
    montgolfier.Cruise(duration/4);  // cruise for 1/4 time

    montgolfier.Ascend(2*height/3);  // ascend to 2/3 height
    montgolfier.Cruise(duration/4);  // cruise for 1/4 time

    montgolfier.Ascend(height);  // ascend to full height
    montgolfier.Cruise(duration/4);  // cruise for 1/4 time

    montgolfier.Ascend(height/2);  // ascend to 1/2 height
    montgolfier.Cruise(duration/4);  // cruise for 1/4 time

    montgolfier.Descend(0);  // come to earth

    return 0;
}
Assignment 2

Random Sentence Generation

For this assignment, you will write a program that generates random sentences based on simple grammar rules. In addition to working with if-else statements and functions that return values, the assignment is designed to walk you through the iterative process of coding and testing a C++ program. The assignment is divided into several parts, each of which builds upon the previous one. When you complete one part, you are to save it and make a copy which can then be augmented for the next part. In this way, you will have a record of each step in the development of the final program.

PART 1: Consider the program below which calls a function called noun to generate a random noun. The noun function rolls a die and uses the result of that roll to select a noun out of four possibilities. The Dice class is defined in the text (pp. 165-166), but a full understanding of that class is not required here. The important features to note are that the declaration of a Dice object creates a die with the specified number of sides, and a call to the Roll member function returns a random roll of that die.

```cpp
#include <iostream.h>
#include "CPstring.h"
#include "dice.h"

string noun() // postcondition: returns a random noun
{
    Dice die(4); // creates a die with 4 sides
    int roll = die.Roll(); // rolls die, saves result
    if (roll == 1) { // if die roll was 1
        return "man";
    }
    else if (roll == 2) { // if die roll was 2
        return "woman";
    }
    else if (roll == 3) { // if die roll was 3
        return "mouse";
    }
    else { // otherwise (must be 4)
        return "computer";
    }
}

int main()
{
    cout << noun() << endl;
    return 0;
}
```
Write a program called grammar1.cc which augments the above program by also defining a function that returns a random article, and another function that returns a random verb. Have the main program then print out one word of each type.

You may use whatever words you like in defining these functions. When you declare a Dice object in each of your functions, be sure to specify the correct number of sides. That is, if you have 5 verbs to choose from, you will need to create and roll a 5-sided die in order to choose among them.

**PART 2:** Sentences are sequences of words that are put together in specific ways. One way to describe the valid structure of sentences is through grammar rules. For example, consider the following grammar rule:

\[
\text{sentence} \leftarrow \text{article} + \text{noun} + \text{verb}
\]

This rule states that a sentence can be constructed by concatenating an article, noun, and verb (in that order). For example, "the mouse ran" and "a man loved" are simple sentences (we will ignore capitalization and punctuation for now). Assuming the existence of the functions article, noun, and verb, the following C++ function can be used to generate simple sentences of this form (note the spaces between the words in the sentence):

```cpp
string sentence()
// postcondition: returns a simple sentence
{
    return article() + " " + noun() + " " + verb();
}
```

More complex sentences can be generated using more complex grammar rules. For example, if we first define the structure of verb phrases and noun phrases, a sentence can be constructed using these components.

\[
\begin{align*}
\text{nounPhrase} & \leftarrow \text{article} + \text{adjective} + \text{noun} \\
\text{verbPhrase} & \leftarrow \text{verb} + \text{nounPhrase} \\
\text{sentence} & \leftarrow \text{nounPhrase} + \text{verbPhrase}
\end{align*}
\]
Copy your program to a file called `grammar2.cc` and add functions which return random noun phrases, verb phrases, and sentences based on these grammar rules. The main program should simply call the `sentence` function and print the string that it returns.
PART 3: Using the existing grammar rules, all sentences will have the same number of words in them. This is certainly not the case in reality, where sentences may have many optional parts. In grammar rules, placing a part-of-speech in brackets specifies that it is optional. For example, consider the following grammar rules:

\[
\begin{align*}
\text{prepPhrase} & \leftarrow \text{preposition} + \text{article} + \text{noun} \\
\text{nounPhrase} & \leftarrow \text{article} + [\text{adjective}] + \text{noun} + [\text{prepPhrase}] \\
\text{verbPhrase} & \leftarrow \text{verb} + \text{nounPhrase} \\
\text{sentence} & \leftarrow \text{nounPhrase} + \text{verbPhrase}
\end{align*}
\]

Here, the noun phrase has two optional parts, an adjective and a prepositional phrase. Thus, a noun phrase can be as simple as "the woman" or as complex as "the happy woman beside the computer". In the first sentence, both the adjective and the prepositional phrase are omitted; in the second, both are included. It may certainly be the case that one is included while the other is omitted, generating noun phrases such as "the little mouse" and "the computer with the mouse".

Copy your program to a file called `grammar3.cc` and augment/modify the code so that it matches the above grammar rules. To handle optional parts-of-speech, define a function called `optional` that has one parameter, a string, and which either returns that same string or else the empty string (with equal probability). This function should roll a 2-sided die to decide which to return. Then, in the noun phrase function, you can call the optional function with an adjective or prepositional phrase to see if they are included.

Note: the omission of optional parts of speech should not result in consecutive spaces within the sentence.

ANALYSIS: Once you have the program in `grammar3.cc` working, use it to generate 20 different random sentences. Then answer the following questions:

- What is the shortest possible sentence that can be generated from these grammar rules? Explain.

- What was the shortest sentence you obtained?

- What is the longest possible sentence that can be generated from these grammar rules? Explain.

- What was the longest sentence you obtained?
Out of the twenty calls, how many times did an adjective appear? Is this close to what you would expect? Explain.

Out of the twenty calls, how many times did a prepositional phrase appear? Is this close to what you would expect? Explain.
Assignment 2 Solution

// File : grammar1.cc
// Author: Dave Reed
//
// This program generates and prints out a random article, noun, and verb.
/////////////////////////////////////////////////////////////////

#include <iostream.h>
#include "CPstring.h"
#include "dice.h"

string article();                  // function prototypes
string noun();
string verb();

int main()
{
    cout << article() << endl;
    cout << noun() << endl;
    cout << verb() << endl;

    return 0;
}

/////////////////////////////////////////////////////////////////

string article()
// postcondition: returns a random article
{
    Dice die(3);

    int roll = die.Roll();

    if (roll == 1) {
        return "the";
    } else if (roll == 2) {
        return "a";
    } else {
        return "some";
    }
}

string noun()
// postcondition: returns a random noun
{
    Dice die(4);

    int roll = die.Roll();

    if (roll == 1) {
        // function prototypes
return "man";
} else if (roll == 2) {
    return "woman";
} else if (roll == 3) {
    return "mouse";
} else {
    return "computer";
}

string verb()
// postcondition: returns a random verb
{
    Dice die(5);
    int roll = die.Roll();

    if (roll == 1) {
        return "bit";
    } else if (roll == 2) {
        return "kissed";
    } else if (roll == 3) {
        return "liked";
    } else if (roll == 4) {
        return "envied";
    } else {
        return "ignored";
    }
}
// File: grammar2.cc
// Author: Dave Reed

// This program generates a simple random sentence comprised
// of a noun phrase (article, adjective, then noun) followed
// by a verb phrase (verb then noun phrase).

#include <iostream.h>
#include "CPstring.h"
#include "dice.h"

string article(); // function prototypes
string noun();
string verb();
string adjective();
string nounPhrase();
string verbPhrase();
string sentence();

int main()
{
    cout << sentence() << endl;
    return 0;
}

#include <iostream.h>
#include "CPstring.h"
#include "dice.h"

string adjective()
// postcondition: returns a random adjective
{
    Dice die(5);
    int roll = die.Roll();
    if (roll == 1) {
        return "big";
    } else if (roll == 2) {
        return "little";
    } else if (roll == 3) {
        return "happy";
    } else if (roll == 4) {
        return "ugly";
    } else {
        return "courteous";
    }
}

string nounPhrase();
// postcondition: returns a random noun phrase
{
    return article() + " " + adjective() + " " + noun();
}

string verbPhrase()
// postcondition: returns a random verb phrase
{
    return verb() + " " + nounPhrase();
}

string sentence()
// postcondition: returns a random sentence
{
    return nounPhrase() + " " + verbPhrase();
}
// File: grammar3.cc
// Author: Dave Reed

// This program generates a complex random sentence which may include optional adjectives and prepositional phrases.

#include <iostream.h>
#include "CPstring.h"
#include "dice.h"

string article();                  // function prototypes
string noun();
string verb();
string adjective();
string preposition();
string optional(string str);
string nounPhrase();
string verbPhrase();
string prepPhrase();
string sentence();

int main()
{
    cout << sentence() << endl;
    return 0;
}

#include <iostream.h>
#include "CPstring.h"  // Functions article(), noun(), verb(), adjective(), verbPhrase(), and sentence() as in grammar2.cc

string preposition()
// postcondition: returns a random preposition
{
    Dice die(3);

    int roll = die.Roll();

    if (roll == 1) {
        return "with";
    }
    else if (roll == 2) {
        return "by";
    }
    else {
        return "beside";
    }
}

string optional(string str)
// postcondition: randomly returns either str or ""
{  
    Dice die(2);
if (die.Roll() == 1) {
    return str;
} else {
    return "";
}

string nounPhrase()
// postcondition: returns a random noun phrase
{
    return article() + optional(" " + adjective()) + " " + noun() + optional(" " + prepPhrase());
}

string prepPhrase()
// postcondition: returns a random prepositional phrase
{
    return preposition() + " " + article() + " " + noun();
}

**ANALYSIS:**

Shortest possible sentence = 5 words (no adjectives or prepositional phrases)
Longest possible sentence = 13 words (2 adjectives and 2 prepositional phrases)

On average, you would expect to have 20 adjectives, one per sentence. This is due to the fact that there are two possible adjectives per sentence, each with a 50% chance. Similarly, you would expect 20 prepositional phrases, one per sentence.
Assignment 3

Binary Reduction Guessing Game

For this assignment, you will write a program that plays a guessing game with the user. In designing your program, you will need to be able to understand the process of the game, translate that process into pseudocode, and then formalize the pseudocode into a working program. Breaking the problem down into subtasks is often helpful in getting a handle on a solution. Similarly, you should break the design of your program into pieces, identifying the subtasks that must be accomplished and tackling each one separately. In addition, it is often useful to identify the types of control structures (e.g., if-else statement, while loop) that will be needed before you start programming. In this way, you have a framework for your program before you start getting overwhelmed by programming details.

PART 1: Write a program called game.cc that plays a guessing game with the user. The user is asked to think of a number between 1 and some constant (say 100). Then, the program tries to guess the number, with the user responding as to whether each guess is correct, too high, or too low. For example, below is a sample run (with user input in bold):

```
> game
Think of a number between 1 and 100.
Type 'ok' to continue: ok

The computer guesses 50.
Is that guess...
(1) correct?
(2) too high?
(3) too low?
Enter your response: 3

The computer guesses 75.
Is that guess...
(1) correct?
(2) too high?
(3) too low?
Enter your response: 4
Your response must be between 1 and 3 - try again: 2

The computer guesses 62.
Is that guess...
(1) correct?
(2) too high?
(3) too low?
Enter your response: 1

The superior computer mind triumphs!
And it only took 3 guesses.
```
In order to minimize the number of guesses, your program should utilize a "halving" strategy. That is, it should keep track of the possible range of numbers, and always guess the number in the middle of that range. In this way, each guess reduces the range of possible numbers by one half. For example, if the maximum number is 100, then the original range of possible numbers is 1 to 100. The program should guess 50, the midpoint in the range 1 to 100. If that guess is too high, then the number must lie in the range 1 to 49. Similarly, if that guess is too low, then the number must lie in the range 51 to 100. Each subsequent guess will reduce the range of possible numbers even further, by lowering the upper bound on the range (if the guess is too high) or raising the lower bound (if the guess is too low). Eventually, the lower and upper bounds on the range will converge and the user's number will be guessed.

The following pseudocode may help in organizing your program:

```plaintext
initialize lower and upper bounds of range

while user has not guessed the number, repeatedly:
  pick the middle number between lower and upper bounds
  ask the user if that is correct, too high, or too low
  update upper or lower bound based on response

print the total number of guesses
```

Note that if the user enters an invalid response (an integer not in the range 1 to 3), then an error message should appear and they should be reprompted until they finally do give a valid response. Also, if the program happens to guess the number on the first try, the output should be grammatically correct ("1 guess" instead of "1 guesses"). Use a constant to specify the upper bound on the range, so that this value can be changed easily if desired.

**PART 2:** Consider the possibility of a user who cheats by changing his or her number in the middle of the game. For example, suppose the user thinks of a number between 1 and 100, say 20. The program guesses 50, which is too high, so it next guesses 25. If the user then changes their number to 79, the program will never guess it (it has already reduced its focus to numbers less than 50). Instead, the program will hone in on the wrong number, and the lower and upper bounds will eventually cross (lower bound > upper bound). Augment your program so that it recognizes when the user has changed his or her number, and denounces that person for the cheater that they are!
ANALYSIS: Once you have your game program working, answer the following questions:

1. Play the game 5 times and list the number of guesses required for each game:

2. What is the maximum number of guesses that may be required to guess a number in the range 1 to 100? How did you determine this? Give a specific number in the range 1 to 100 that requires this maximum number of guesses.

3. What is the maximum number of guesses required to guess a number in the range 1 to 200? How about the range 1 to 400? What is the pattern here?
Assignment 3 Solution

// File : game.cc
// Author: Dave Reed

// This program plays a guessing game with the user. The user
// is asked to think of a number between 1 and MAX_RANGE. The
// program tries to guess the number, keeping track of the
// range in which it must lie and always guessing the middle.
// At the end, the total number of guesses is displayed, or
// else the user is chastised if it turns out they cheated.
>Password: 100; // guess between 1 & MAX_RANGE
#include <iostream.h>
#include "CPstring.h"

const int MAX_RANGE = 100; // guess between 1 & MAX_RANGE

void GiveIntro(); // function prototypes
int GetResponse(int guess);
void GiveResults(int response, int numGuesses);

int main()
{
    int lowerBound = 1;
    int upperBound = MAX_RANGE;
    int numGuesses = 0;

    GiveIntro();

    int response = 0;
    while (response != 1 && lowerBound <= upperBound) {
        int middle = (lowerBound + upperBound) / 2;

        response = GetResponse(middle);
        if (response == 2) {
            upperBound = middle - 1;
        } else if (response == 3) {
            lowerBound = middle + 1;
        }
        numGuesses++;
    }

    GiveResults(response, numGuesses);

    return 0;
}

void GiveIntro()
// postcondition: gives introduction to the game

// File : game.cc
// Author: Dave Reed

// This program plays a guessing game with the user. The user
// is asked to think of a number between 1 and MAX_RANGE. The
// program tries to guess the number, keeping track of the
// range in which it must lie and always guessing the middle.
// At the end, the total number of guesses is displayed, or
// else the user is chastised if it turns out they cheated.
>Password: 100; // guess between 1 & MAX_RANGE
#include <iostream.h>
#include "CPstring.h"

const int MAX_RANGE = 100; // guess between 1 & MAX_RANGE

void GiveIntro(); // function prototypes
int GetResponse(int guess);
void GiveResults(int response, int numGuesses);

int main()
{
{  
cout << "Think of a number between 1 and 100." << endl;
cout << "Type 'ok' when you are ready: ";

string ok;
cin >> ok;
}

int GetResponse(int guess)  
// precondition: guess is the program's current guess  
// postcondition: displays prompt and reads user's response  
{
cout << endl;
cout << "The computer guesses " << guess << "." << endl;
cout << "Is that guess..." << endl;
cout << "  (1) correct?" << endl;
cout << "  (2) too high?" << endl;
cout << "  (3) too low?" << endl;
cout << "Enter your response: ";

int response;
cin >> response;
while (response < 1 || response > 3) {
    cout << "Your response must be between 1 and 3 "
        << "- try again: ";
    cin >> response;
}
return response;
}

void GiveResults(int response, int numGuesses)  
// precondition: response in range [1..3], numGuesses > 0  
// postcondition: if response is 1, then displays number of  
// guesses; if not, then scolds the cheater  
{
if (response == 1) {
    cout << endl << "The superior computer mind triumphs!"
         << endl;
cout << "And it only took " << numGuesses;
if (numGuesses == 1) {
    cout << " guess." << endl;
}
else {
    cout << " guesses." << endl;
}
}
else {
    cout << endl << "You cheated, you varmint!" << endl;
}
}
ANALYSIS:

The maximum number of guesses with a range 1 to 100 is 7. This is due to the fact that each guess reduces the range by one half (or by one more than that if the range is odd). Therefore, the range of possible numbers is reduced from 100 to 50 to 25 to 12 to 6 to 3 to 1 to 0, in seven guesses. This worst-case scenario will occur when the number to be guessed is the maximum number in the range, here 100.

Because each guess reduces the range by one half, doubling the range only increases the maximum number of guesses by one. Thus, a range of 200 requires at most 8 guesses, while 400 requires at most 9 guesses.
Assignment 4
Marble Jar Puzzles

In this assignment, you will make use of a previously defined C++ class in order to perform two different simulations. The simulations involve repetition, counting, and the use of abstraction in accessing the member function of the provided class. The point of the simulations is to help you solve two different brain-teaser puzzles involving jars containing black and white marbles. Puzzles such as these are common, especially among statisticians and discrete mathematicians.

1. Suppose a jar contains the same number of black and white marbles. If you reach in and pull out two random marbles, are they more likely to be the same color or different colors?

2. Suppose a jar has an arbitrary mixture of black and white marbles. As long as there are at least two marbles in the jar, you are to draw two random marbles and
   • if they are the same color, throw them both away and add a brand new black marble to the jar.
   • if they are different colors, throw away the black one and return the white marble to the jar.

Can the marble drawing process go on forever? Furthermore, if you are told the initial contents of the jar, can you predict the final outcome?

To help in solving these kinds of puzzles, it will be useful to have a class which simulates the operations allowed on a jar of marbles. Such a class is defined in the files jar.h and jar.cc.

```cpp
class MarbleJar
{
public:
    MarbleJar(int black, int white); // initializes jar contents
    string DrawMarble();            // draws and returns color
    void AddMarble(string color);   // adds marble to jar
    bool IsEmpty();                 // returns true if empty
private:
    int myNumBlack, myNumWhite;    // marbles in the jar
    RandGen myGen;                 // random number generator
};
```

It is important to note that this class provides the necessary operations for manipulating a jar and only those operations. For many puzzles, allowing the person to peek inside the jar would tarnish the result. As such, the data fields which store the number of marbles are hidden in the
private section of the class. The person is only allowed to remove a random marble, add a marble of a specific color, and know when the jar is empty (equivalent to picking up the jar and shaking it to see if there is anything inside).
PART 1: Using the MarbleJar class, write a C++ program called puzzle1.cc that will help you solve the first marble puzzle described above. Your program should prompt the user for the initial number of marbles of each color. Then it should repeatedly draw two marbles, noting when they are the same color and then returning them to the jar. It should do this some constant number of times (say 1000) and then output the number and percentage of each type of result. For example, below is a sample run (with user input in bold):

    > puzzle1
    How many marbles of each color in the jar? 10

    Out of 1000 trials,
    same colors drawn 479 times
    different colors drawn 521 times

    That means they were the same 47.9% of the time.

ANALYSIS: Once you have your program working, answer the following questions:

- Perhaps using repeated simulations as a guide, what is the answer to this puzzle? Are the two marbles more likely to be same color or different colors. Can you explain why?

- Does increasing the total number of marbles in the jar affect the answer in any way?
PART 2: Similarly, write a C++ program called `puzzle2.cc` that will help you solve the second puzzle. Your program should prompt the user for the initial contents of the jar and create a corresponding `MarbleJar` object. It should then repeatedly draw two marbles from the jar and perform the appropriate action based on the colors of the drawn marbles. If the process ends (with less than two marbles in the jar), then the final contents of the jar should be displayed.

For example,

```
> puzzle2
How many black marbles in the jar?   4
How many white marbles in the jar?   5
1: I drew black and white -- replacing a white marble.
2: I drew white and white -- replacing a black marble.
3: I drew white and black -- replacing a white marble.
4: I drew white and black -- replacing a white marble.
5: I drew white and black -- replacing a white marble.
6: I drew white and white -- replacing a black marble.
7: I drew black and black -- replacing a black marble.
8: I drew black and white -- replacing a white marble.
9: Only one marble left... it is white.
```

ANALYSIS: Once you have your program working, answer the following questions:

- Perhaps using repeated simulations as a guide, what is the answer to this puzzle? Can the process of drawing and replacing marbles go on forever? Why or why not?

- If you know the initial contents of the jar, can you predict the final outcome? Explain.
Assignment 4 Solution

// Marble Jar class: for probability simulations
//
// CONSTRUCTION: with two ints (# of black & white marbles, resp.)
//
// ******************PUBLIC OPERATIONS******************
// String DrawMarble()      --> Return color of random marble
//                           (either "BLACK" or "WHITE")
//                           Note: marble is removed from jar
// void AddMarble(String marble) --> Add marble (specify color)
// Bool IsEmpty()           --> Return true if no marbles in jar
#
#ifndef _JAR_H_
#define _JAR_H_

#include "CPstring.h"
#include "rando.h"

class MarbleJar
{
  public:
    MarbleJar(int black, int white);
    string DrawMarble();
    void AddMarble(string marble);
    bool IsEmpty();
  private:
    int myNumBlack, myNumWhite; // number of marbles in the jar
    RandGen myGen;             // random number generator
};

#endif
// Marble Jar class implementation

#include <iostream.h>               // needed for cin & cout
#include "CPstring.h"
#include "jar.h"                    // needed for MarbleJar class

MarbleJar::MarbleJar(int black, int white)
// precondition : black >= 0, white >= 0
// postcondition: myNumBlack = black, myNumWhite = white
{
    myNumBlack = black;
    myNumWhite = white;
}

string MarbleJar::DrawMarble()
// precondition : jar is not empty (myNumBlack+myNumWhite > 0)
// postcondition: draws a random marble and returns its color
// (either "black" OR "white"). As a side effect,
// the corresponding color count is decremented.
{
    if (myGen.RandInt(myNumBlack+myNumWhite) < myNumBlack) {
        myNumBlack--;
        return "black";
    }
    else {
        myNumWhite--;
        return "white";
    }
}

void MarbleJar::AddMarble(string marble)
// precondition : marble = "black" or "white"
// postcondition: adds a marble of the specified color to the jar,
//                incrementing the color count.
{
    if (marble == "black") {
        myNumBlack++;
    } else if (marble == "white") {
        myNumWhite++;
    }
}

bool MarbleJar::IsEmpty()
// precondition : jar contains myNumBlack and myNumWhite marbles
// postcondition: returns true if jar is empty (no marbles)
{
    return (myNumBlack + myNumWhite == 0);
}
// File: puzzle1.cc
// Author: Dave Reed
//
// This program creates a MarbleJar, repeatedly draws two
// marbles and counts how often they are the same.

#include <iostream.h>
#include "jar.h"

const int NUM_REPS = 1000; // number of repetitions for
// drawing marble pairs

int main()
{
    int numOfEach;
    cout << "How many marbles of each color? ";
    cin >> numOfEach;
    MarbleJar jar(numOfEach, numOfEach); // create jar

    int numSame = 0; // init. counter

    int cnt;
    for (cnt = 1; cnt <= NUM_REPS; cnt++) { // repeatedly,
        string marble1 = jar.DrawMarble(); // draw marbles
        string marble2 = jar.DrawMarble();

        if (marble1 == marble2) { // if same,
            numSame++;
            // add to count
        }

        jar.AddMarble(marble1); // return to jar
        jar.AddMarble(marble2);
    }

    cout << "Out of " << NUM_REPS << " trials: " << endl;
    cout << "\t same colors drawn " << numSame << " times " << endl;
    cout << "\t diff colors drawn " << NUM_REPS - numSame
    << " times " << endl;
    cout << "That means they were the same "
    << 100*(double(numSame)/NUM_REPS)
    << "% of the time" << endl;

    return 0;
}
// File : puzzle2.cc
// Author: Dave Reed

// This program creates a jar of marbles (using MarbleJar class),
// and repeatedly draws two marbles, replacing one as specified
// by the puzzle. It stops when < 2 marbles left in the jar.

#include <iostream.h>
#include "jar.h"

int main()
{
    int numBlack;
    cout << "How many black marbles in the jar? ";
    cin >> numBlack;

    int numWhite;
    cout << "How many white marbles in the jar? ";
    cin >> numWhite;

    MarbleJar jar(numBlack, numWhite);  // create & initialize jar
    int rep = 1;                        // counter for # of reps
    string marble1, marble2, replace;   // marbles to be used

    while ( !jar.IsEmpty() ) {          // repeat until empty
        marble1 = jar.DrawMarble();     // draw first marble

        if ( !jar.IsEmpty() ) {         // if not empty now,
            marble2 = jar.DrawMarble(); // draw second marble &
            if (marble1 == marble2) {  // determine type
                replace = "black";      // to be returned
            } else {
                replace = "white";
            }
        }

        jar.AddMarble(replace);        // return marble

        cout << rep << ":  I drew " << marble1 << " and "
            << marble2 << " -- replacing a " << replace
            << " marble." << endl;
    } else {
        cout << rep << ":  Only one marble left... it is "
            << marble1 << "." << endl;
    }

    return 0;
}
ANALYSIS:

For puzzle 1, it is more likely that the marbles will be different colors. This is due to the fact that after drawing the first marble, there is one fewer of that color in the jar. If you started with N marbles of each color, then there are only N-1 remaining of that same color (as opposed to N of the other color). Thus you are more likely to draw a marble of the other color. In fact, the chances of drawing the same color twice can be given by the formula (N-1)/(2N - 1). When the initial number of marbles is small, the difference of one marble between the two colors can be significant (consider when N is 1). As the number of marbles increases, the chance of getting the same color approaches 50% (as can be seen by plugging numbers into the above formula).

For puzzle 2, it is clear that the drawing process cannot go on forever. At each step, two marbles are drawn and one is replaced. Thus, each step reduces the number of marbles in the jar by one. If there were N marbles in the jar to start with, after N-1 draws there will be only one marble left. In order to know what color the final marble will be, you only need to know whether the initial number of white marbles was odd or even. The only way to remove white marbles from the jar is in pairs: 2 whites drawn results in a black being returned, a black and a white results in the white being returned. Thus, if there was an even number of whites to start with, all of them will be removed and the remaining marble will be black. If the initial number of whites is odd, then the final marble will be white.
Assignment 5

Random Wall Walks

For this assignment, you will perform and analyze simulations using the `RandomWalk` class defined in the text (pp. 294-296). This class can be used to simulate the meanderings of a frog, that is able to jump in one of two directions at any given time point. You will need to be able to copy and modify the `RandomWalk` class so that it performs a slightly different simulation. You will then be able to compare the two different variants of the random walk.

As before, consider a frog that is sitting on its lily pad (which we may think of as 0 on the x-axis). Only now suppose that the lily pad is sitting up against a wall, so that it is impossible for the frog to jump to the left of the lily pad. If the frog is at the lily pad and jumps to the left, it just bounces off the wall and lands back on the lily pad. It may certainly jump to the right as freely as before, and may jump to the left as before as long as it is not at the wall.

We would like to know how this constrained random walk compares to the unconstrained random walk described in the text. In particular, if we have a goal distance in mind, which type of random walk will cause the frog to reach that goal distance faster? To determine this, define a new class called `RandomWallWalk` which can be used to simulate our new type of random walk. Since this class will be very similar to the `RandomWalk` class, you should start by copying this class and then making the necessary modifications. When defining the `RandomWallWalk` class, be sure to note that a jump against the wall, even though it doesn't result in any movement, still counts as a jump.

Then, write a program called `compare.cc` that will allow you to compare the two walks. Your program should prompt the user for the number of walks to be simulated and the goal distance for each walk. It should then simulate that many random walks of each type. Each walk should continue until the frog reaches the specified distance from the lily pad (for the original unconstrained random walk, the frog could reach that distance in either the positive or negative direction). The program should keep track of the number of steps for each type of walk, and print the average number of steps for each.

For example, below is a sample run (with user input in **bold**):

```
> compare
How many walks do you want? 100
What is the goal distance for each walk? 10

random walk (no wall) : 104.08 steps per walk
random walk (with wall) : 117.41 steps per walk
```
ANALYSIS: Using your program, answer the following questions.

- Run your program five times specifying 100 random walks with goal distance 10, and list the results below. How consistent are the results that you obtained from your simulations?

- Does consistency increase or decrease if you increase the number of walks? Why is this the case? Give actual numbers to back up your claim.

- Does consistency increase or decrease if you increase the goal distance? Why is this the case? Give actual numbers to back up your claim.

- Does one type of walk consistently require more steps than the other? Why or why not?
Assignment 5 Solution

The RandomWallWalk class should be identical to the RandomWalk class except for the TakeStep member function, which must check if the frog is up against the wall before moving to the left:

```cpp
void RandomWallWalk::TakeStep()
// postcondition: one step of random walk taken
{
    Dice coin(2);
    switch (coin.Roll())
    {
        case 1:
            if (myPosition > 0) {      // extra test here
                myPosition--;
            }
            break;
        case 2:
            myPosition++;
            break;
    }
    mySteps++;
}
```
double AvgWallWalks(int numReps, int goal);

int main()
{
    int numWalks;
    cout << "How many walks do you want? ";
    cin >> numWalks;

    int goalDist;
    cout << "What is the goal distance for each walk? ";
    cin >> goalDist;

    cout << endl;
    cout << "random walk (no wall) : "
    << AvgWalks(numWalks, goalDist)
    << " steps per walk" << endl;
    cout << "random walk (with wall): "
    << AvgWallWalks(numWalks, goalDist)
    << " steps per walk" << endl;

    return 0;
}

double AvgWalks(int numReps, int goal)
// precondition : numReps > 0, goal > 0
// postcondition: returns avg number of steps needed to reach
// goal in RandomWalk (numReps experiments)
{
    RandomWalk frog(INT_MAX);
    int totalSteps = 0;

    int rep;
    for (rep = 0; rep < numReps; rep++) {
        frog.First();
        while (frog.GetPosition() > -goal &&
            frog.GetPosition() < goal) {
            frog.Next();
        }
        totalSteps += frog.GetSteps();
    }
    return totalSteps / (double)numReps;
}
double AvgWallWalks(int numReps, int goal)
// precondition : numReps > 0, goal > 0
// postcondition: returns avg number of steps needed to reach
//                goal in RandomWallWalk (numReps experiments)
{
    RandomWallWalk frog(INT_MAX);
    int totalSteps = 0;

    int rep;
    for (rep = 0; rep < numReps; rep++) {
        frog.First();
        while (frog.GetPosition() < goal) {
            frog.Next();
        }
        totalSteps += frog.GetSteps();
    }

    return double(totalSteps)/numReps;
}

ANALYSIS:

Since you are averaging 100 different random walks, the results should be relatively consistent. Increasing the number of walks should increase consistency, since the few aberrant walks that occur will have many more "normal" walks to balance them out. In general, the more random experiments you average, the more likely you are to obtain the expected value.

If you increase the goal distance for the walks, the consistency of the results is likely to decrease. This is because a larger goal implies a larger space for the frog to jump back and forth in. When the goal distance is short, there is less space to jump around in without reaching a goal, so the number of steps will be smaller and more consistent. At the limit of this argument is a goal distance of 1. For the unconstrained random walk, the goal distance is always reached in a single step.

When comparing the two variants, there is a tradeoff introduced by the wall. Although the wall implies that there is now only one direction in which the frog can achieve the goal distance, the range in which the frog can wander is smaller. For example, suppose the goal distance is 10. Ten jumps to the left achieves the desired distance when there is not a wall, but get the frog nowhere if there is one. On the other hand, three jumps to the left followed by ten to the right do achieve the goal distance when there is a wall, but do not when there isn't one. On the whole, these factors seem to balance out, with average numbers of steps being quite similar. If anything, the number of steps with the wall seems a bit higher.
Assignment 6

Spell-checking

Most word processors have the ability to scan a document and compare each word with a stored dictionary. When a word is encountered that does not appear in the dictionary, the user is notified and has the option of changing the spelling of the word, ignoring the word, or adding that word to the dictionary for future reference. This assignment involves implementing a simplified spell-checker for text files. In order to accomplish this, you will need to be able to read and write words from files, store the words in a list, and look up words in the list.

Your program, call it spell.cc, should first prompt the user for the name of the dictionary file, read in the words from that file, and store them for future reference. It should then prompt the user for the name of the text file which is to be spell-checked. Each word in that file should be read in and compared with the dictionary of known words. If the word read in does not appear in the dictionary, then the user should be notified and given the option of (1) skipping that occurrence of the word, (2) ignoring all occurrences of the word, or (3) adding the word to the dictionary. Choosing to ignore all occurrences or add the word to the dictionary should mean that any future occurrences of the word will not be flagged as misspellings. At the end, the possibly updated dictionary should be written back out, overwriting the initial dictionary.

To keep this assignment relatively straightforward, we will make some simplifying assumptions:

- Most spell-checkers suggest possible corrections for misspelled words and even replace the words in the original file. For this assignment, you only need to notify the user of misspelled words. You do not have to find close matches or alter the original text file at all.

- Most spell-checkers consider the significance of capitalization when checking words, ignoring capitalization at the beginning of a sentence but allowing it otherwise. For this assignment, your spell-checker should simply be case-insensitive. You may assume that the words in the dictionary do not contain upper-case letters (and, of course, any additions that you make to the dictionary must ensure this).

- Most spell-checkers ignore punctuation symbols at the beginning and end of words, but allow them internally. For this assignment, your spell-checker should remove all punctuation symbols. The ispunct function defined in the library ctype.h should prove useful here.

Two classes defined in the text should prove useful in completing this assignment. The WordStreamIterator class (pp. 250-252) can be used to read words from a file. And with some
modifications, the WordList class (pp. 359-361) can be used to store and access words. Two modifications must be made to this class however. First, a member function must be added that allows you to determine if a particular word is stored in the list. Second, another Print function must be added that allows for the words in the list to be output to a file. This function can have the same name (Print), but should have a string as parameter, the name of the output file. Also, remove the statement in the Print function which prints the total number of words in the file (if you ever need to print this, you can determine the count by calling GetUnique.)
ANALYSIS: Once you have completed your spell-checker, consider some of the more advanced features that were not implemented as a part of this assignment. You do not have to implement these features, but take a moment to think about how they might be incorporated into your program. Describe the approaches you would take towards adding these features, and discuss any difficulties that you may foresee.

- Suggesting possible replacement words when an unknown word is found:

- Actually modifying the original text file to replace misspelled words:

- Handling capitalization at the beginning of sentences:

- Handling punctuation at the beginning and end of words:
Assignment 6 Solution

Modifications to WordList class:

1. Add the IsStored member function:

   ```cpp
   bool WordList::IsStored(const string & word)
   // postcondition: returns true if word stored in myList
   { return (Search(word) != -1); }
   ```

2. Modify the Print member function:

   ```cpp
   void WordList::Print()
   // postcondition: all elements of myList printed to cout
   { int k;
     for (k = 0; k < myCount; k++) {
       cout << myList[k] << endl;
     }
   }
   ```

3. Add a Print member function that writes to a file:

   ```cpp
   void WordList::Print(string filename)
   // postcondition: all elements of myList printed to file
   { ofstream ostr;
     ostr.open(filename);

     int k;
     for (k = 0; k < myCount; k++) {
       ostr << myList[k] << endl;
     }
   }
   ```
This program is a simple spell-checker. It prompts the user for the dictionary file, and reads in the words. Then prompts for the file to be checked, reading each word and checking against the stored dictionary. For each unknown word, the user is given the option of skipping the word, ignoring all occurrences, or adding it to the dictionary.

#include <iostream.h>
#include <ctype.h>
#include "CPstring.h"
#include "worditer.h"
#include "wordlist.h"

void ReadDictionary(string & filename, WordList & dict);
void ProcessTextFile(WordList & dict);
string StripPunct(const string & str);

int main()
{
  string dictFile;
  WordList dictionary(100);
  
  ReadDictionary(dictFile, dictionary);

  ProcessTextFile(dictionary);

  dictionary.Print(dictFile);

  return 0;
}

void ReadDictionary(string & filename, WordList & dict)
// precondition : filename is name of an existing text file
// postcondition: reads words from file and stores in dict
{
  cout << "Enter the name of the dictionary file: ";
  cin >> filename;

  WordStreamIterator dictIter;
  dictIter.Open(filename);

  for (dictIter.First(); !dictIter.IsDone(); dictIter.Next()) {
    dict.Update(dictIter.Current());
  }
}

void ProcessTextFile(WordList & dict)
// precondition : dict stores a dictionary of words
// postcondition: reads words from user specified text file, comparing with the words in the dictionary.
Any unknown word is either skipped, ignored, or added to the dictionary (user's choice).

```cpp
string textFile;
cout << "Enter the name of the text file: ";
cin >> textFile;

WordStreamIterator textIter;
textIter.Open(textFile);

WordList ignored(100);

for (textIter.First(); !textIter.IsDone(); textIter.Next()) {
    string word = StripPunct(textIter.Current());
    word = word.Downcase();
    if (!dict.IsStored(word) && !ignored.IsStored(word)) {
        cout << "Unknown word:  " << word << endl;
        cout << "Do you want to ... " << endl;
        cout << "(S)kip this occurrence of the word" << endl;
        cout << "(I)gnore all occurrences of the word" << endl;
        cout << "(A)dd the word to the dictionary" << endl;

        string response;
cin >> response;
        if (response[0] == 'S' || response[0] == 's') {
            dict.Update(word);
        } else if (response[0] == 'I' || response[0] == 'i') {
            ignored.Update(word);
        }
    }
}

string StripPunct(const string & str)
// postcondition: returns str with all punct chars removed
{
    string stripped = "";
    int k;
    for (k = 0; k < str.length(); k++) {
        if (!ispunct(str[k])) {
            stripped += str[k];
        }
    }
    return stripped;
}
```
ANALYSIS:

Suggesting possible replacement words: You would need to have some algorithm for searching through the dictionary and recognizing words that are close. For example, you could look for words with missing or extra letters, or perhaps two letters interchanged.

Replacing misspelled words: This would require more care in reading in the text so that formatting information is not lost. You would need to read in the text using getline, and then extract words one at a time for testing. Misspelled words could be replaced in that string, and then the updated string written back out to the original file.

Handling capitalization: This would require keeping track of where sentences begin and end. As you processed words, you would need to recognize when a word ends with a period or question mark, and then treat the next word differently.

Handling punctuation: This would be relatively simple, involving traversing the word from both ends and removing punctuation marks. Once non-punctuation characters are encountered, the removal process ends (leaving punctuation marks inside words).
Assignment 7

Hunt the Wumpus

One of the first computer games in the 1970's was an adventure game called "Hunt the Wumpus". In this game, you are forced to wander around a maze of caves, hunting the dreaded wumpus (or possible more than one wumpi). You are armed only with hand grenades, which you can throw at a wumpus to kill it. The problem is that wumpi are both fierce and fast. If you ever wander into a cave that contains a wumpus, then that wumpus will attack and kill you before you even have a chance to throw a grenade. Your only hope is to throw the grenade into the cave from an adjacent cave. Fortunately, wumpi are rather odorous creatures, so you will be able to smell a wumpus when you are in an adjacent cave. Of course, you won't know which cave the wumpus is in (every cave is connected to three other caves), so you will have to guess when you throw your grenade.

The object of the game is to kill all of the wumpi before you run out of grenades (and without getting killed). As if this weren't hard enough, you don't know exactly how many wumpi there are, or how many grenades you have (you will have 4 grenades for every wumpus). Plus, there are other things to watch out for. Somewhere in the maze is a group of giant bats, which will pick you up and fly you to some random cave (but at least they will only drop you in an empty cave). There is also a bottomless pit which you must avoid. Luckily, there is some warning about these hazards: you will be able to hear the flapping wings of the bats and feel a draft coming from the pit when you are in an adjacent cave. Oh, and there are the Lost Caverns of the Wyrm, which are very difficult to get out of. Below is a sample execution of the game.

```
> wumpus
HUNT THE WUMPUS: Your mission is to explore the maze of caves and destroy all of the wumpi (without getting yourself killed). To move to an adjacent cave, enter 'M' and the tunnel number. To toss a grenade into a cave, enter 'T' and the tunnel number.

You are currently in The Fountainhead
   (1) unknown
   (2) unknown
   (3) unknown

What do you want to do? m 2

You are currently in The Silver Mirror
   (1) The Fountainhead
   (2) unknown
   (3) unknown

What do you want to do? m 3

You are currently in Shelob's Lair
   (1) The Silver Mirror
   (2) unknown
   (3) unknown

What do you want to do? m 3

You are currently in The Lost Caverns of the Wyrm
   (1) unknown
   (2) unknown
```
You smell a WUMPUS!

What do you want to do? t l
Missed, dagnabit!
DANGER: Any nearby wumpi are on the move.
A wumpus is coming toward you with big, gnarly teeth... CHOMP CHOMP CHOMP
GAME OVER
Note: This game in no way intends to promote or condone violence. If you object to the violent nature of this game, alter the rules accordingly. Perhaps a wumpus is a poor soul who has been placed under a spell, and instead of grenades you are tossing bottles of potion that release the person from that spell. Use your imagination.

PART 1: In order to write a program for playing Hunt the Wumpus, you will first need to complete the implementation of the Maze class, whose definition is attached to the end of this assignment. This class stores all of the information about a maze of caves, as well as state information about the game (location of the player and wumpi, number of grenades left, etc.). It also provides member functions needed in order to write a game playing program.

Your program should assume the existence of a data file called "caves.dat", which stores the configuration of the cave maze. The first line of this file should specify the number of caves in the maze, and then each subsequent line will contain the following information:

\[\text{<cave number> <adjacent1> <adjacent2> <adjacent3> <cave name>}\]

where the cave number is a unique number between 0 and the number of caves minus 1, the next three numbers are the numbers of the caves adjacent to this one (all caves are adjacent to three other caves), and the rest of the line is the name of the cave. For example, the standard data file for this game is given below:

\[
\begin{align*}
20 \\
0 & 1 & 4 & 9 \text{The Fountainhead} \\
1 & 0 & 2 & 5 \text{The Rumpus Room} \\
2 & 1 & 3 & 6 \text{Buford's Folly} \\
3 & 2 & 4 & 7 \text{The Hall of Kings} \\
4 & 0 & 3 & 14 \text{The Silver Mirror} \\
5 & 1 & 9 & 11 \text{The Gallimaufry} \\
6 & 2 & 7 & 12 \text{The Den of Iniquity} \\
7 & 3 & 6 & 8 \text{The Findledelve} \\
8 & 7 & 3 & 13 \text{The Page of the Deniers} \\
9 & 0 & 5 & 10 \text{The Final Tally} \\
10 & 9 & 11 & 14 \text{Ess four} \\
11 & 5 & 10 & 12 \text{The Trillion} \\
12 & 6 & 11 & 13 \text{The Scrofula} \\
13 & 8 & 12 & 18 \text{Ephemeron} \\
14 & 4 & 10 & 15 \text{Shelob's Lair} \\
15 & 15 & 16 & 19 \text{The Lost Caverns of the Wyrm} \\
16 & 15 & 17 & 19 \text{The Lost Caverns of the Wyrm} \\
17 & 16 & 17 & 18 \text{The Lost Caverns of the Wyrm} \\
18 & 13 & 17 & 19 \text{The Lost Caverns of the Wyrm} \\
19 & 15 & 16 & 17 \text{The Lost Caverns of the Wyrm}
\end{align*}
\]

However, it should be possible to customize the game simply by replacing this file with a different cave configuration. The data structure used to store the maze information is a Vector of structs, with each struct containing the name of the cave, the numbers of adjacent caves, the contents of the cave (e.g., wumpus, bats, ...), and whether or not that cave has been previously visited. The constructor for the Maze class, which is provided for you, reads in the information
from the "caves.dat" file and initializes the Vector. For example, the contents of the data file above would be stored as shown below:
The constructor determines how many wumpi there are going to be (a random number between 1 and the number of caves divided by four), initializes the number of grenades (four grenades per wumpus), and randomly places the wumpi, bats and pit in the maze.

Complete the definition of the Maze class by implementing the following member functions:

- **Toss:** This function is called with a tunnel number (1 - 3) as argument. It simulates throwing a grenade from the current location into the cave corresponding to that tunnel. Note that given the current location, the corresponding cave number can be determined by the expression

  \[
  \text{myCaves[myCurrentLoc].adj[tunnel-1]}
  \]

  If the player has no grenades left, then the function should inform him/her of that fact. Otherwise it should decrement the number of grenades and display the result of the explosion. If a wumpus is in the specified cave, then that wumpus is killed and the wumpus count is decremented. Any wumpi in adjacent caves, however, are alerted by the explosion. For each adjacent wumpus, the function should pick an empty cave adjacent to the wumpus’ location (if one exists), and move the wumpus there. If it so happens that the wumpus moves into the cave containing the player, then it devours the player.

- **Move:** This function is called with a tunnel number (1 - 3) as argument. It simulates moving from the current location into the cave corresponding to that tunnel. If that cave contains a wumpus, then the player is devoured. If the cave contains a pit, then the player falls into the pit and is killed. If the cave contains bats, then the player is moved to a random, empty cave in the maze. Note: the `FindEmpty` private member function should prove handy here.

- **Feel:** This function returns true if a cave adjacent to the current location contains a pit.

- **Hear:** This function returns true if a cave adjacent to the current location contains bats.

- **Smell:** This function returns true if a cave adjacent to the current location contains a wumpus.
• **CurrentName**: This function returns the name of the cave where currently located.

• **AdjacentName**: This function is called with a tunnel number (1 - 3) as argument. If the cave corresponding to that tunnel has been previously visited, then the function returns the cave name. If not, it returns "unknown". In this way, caves that have been previously explored will be recognized when they are next encountered.

• **StillAlive**: This function returns true if the player is still alive.

• **StillWumpi**: This function returns true if at least one wumpus is still alive in the maze.

You should write a driver program to test your code. For easier testing, consider modifying the constructor temporarily so that you know exactly where things are. As always, interleave the coding and testing phases -- test each function independently as you complete it.

**PART 2**: Using your Maze class, write a program which allows the user to play Hunt the Wumpus. You may find that your program will be quite short, since most of the work is done in the Maze member functions.

When grading this program, special attention will be paid to style and robustness. As always, your program should be modular and should follow the commenting, indentation, and naming conventions presented earlier in the semester. Robustness refers to your program's ability to behave correctly in all cases, even those that may be unexpected or rare. As you test your program, you should be very careful to think about any such cases that might arise, and test for them. For example, a wumpus that is startled by a nearby explosion is supposed to run to an adjacent empty cave. What if there are no empty caves adjacent to a wumpus? Although this is unlikely, especially with a small number of wumpi, it is possible. In such a case, the wumpus should stay where it is. Special cases such as these may affect your design of the wumpus program, and may force some modifications to code in the Maze member functions.
// Maze class: for representing and traversing an "adventure" maze
//
// CONSTRUCTION: with no initializer
//
// ***********************PUBLIC OPERATIONS***********************
//   void Toss(int tunnel) --> toss grenade (if any) into tunnel
//                             if wumpus there, then it is killed
//                             wumpi in adj. caves move randomly
//   void Move(int tunnel) --> move down tunnel to adj. cave
//                             if wumpus there, then player killed
//                             if pit, then player killed
//                             if bats, then flown to random cave
//   bool Feel()           --> return true if adjacent to a pit
//   bool Hear()           --> return true if adjacent to bats
//   bool Feel()           --> return true if adjacent to wumpus
//   string CurrentName()  --> return name of current cave
//   string AdjacentName(int tunnel) --> return name of adj. cave
//   bool StillAlive()     --> return true if player is alive
//   bool StillWumpi()     --> return true if any wumpi alive

class Maze
{
public:
    Maze();
    void Toss(int tunnel);
    void Move(int tunnel);
    bool Feel() const;
    bool Hear() const;
    bool Smell() const;
    string CurrentName() const;
    string AdjacentName(int tunnel) const;
    bool StillAlive() const;
    bool StillWumpi() const;

private:
    enum CaveContents {EMPTY, WUMPUS, PIT, BATS};
    struct Cave    // struct for storing data on a cave
    {
        string name;    // name of cave
        int adj[3];    // indices of adjacent caves
        CaveContents contents; // contents of cave
        bool visited;   // flags whether already visited
    };
    Vector<Cave> myCaves;    // list of all caves
    int myNumCaves;    // number of caves
    int myNumGrenades; // current number of grenades held
    int myNumWumpi;    // current number of wumpi alive
    int myCurrentLoc;  // current location (cave number)
    bool myIsAlive;    // status of player (alive or dead)
    int FindEmpty();   // returns # of a random empty cave
};
Maze::Maze()
// constructor
// postcondition: initializes the maze of caves as defined in
//                "caves.dat", randomly places objects in caves
{
    ifstream istr("caves.dat");     // open caves data file,
    assert(istr);                   // verifying that it exists

    istr >> myNumCaves;             // read in number of caves,
    assert(myNumCaves > 3);         // making sure enough space
    myCaves.SetSize(myNumCaves);    // if ok, resize Vector

    int a, caveNum;                 // read in & store cave info
    while (istr >> caveNum) {
        for (a = 0; a < 3; a++) {
            istr >> myCaves[caveNum].adj[a];
        }
        getline(istr, myCaves[caveNum].name);
        myCaves[caveNum].contents = EMPTY;
        myCaves[caveNum].visited = false;
    }

    RandGen rando;                  // initialize state variables
    myNumWumpi = rando.RandInt(1, myNumCaves/4);
    myNumGrenades = 4*myNumWumpi;
    myIsAlive = true;
    myCurrentLoc = 0;

    myCaves[myCurrentLoc].contents = WUMPUS;   // temporarily mark
    // initial location

    for (a = 1; a <= myNumWumpi; a++) {        // initial location
        myCaves[FindEmpty()].contents = WUMPUS; // place wumpi,
        myCaves[FindEmpty()].contents = PIT;    // bats and pit
        myCaves[FindEmpty()].contents = BATS;    // when done, mark
        myCaves[myCurrentLoc].contents = EMPTY;  // as empty again
    }
}
Assignment 7 Solution

// Implementation of Maze class
@PostMapping{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

#include <iostream.h>
#include <fstream.h>
#include "CPstring.h"
#include "rando.h"
#include "maze.h"

Maze::Maze()
// constructor
// postcondition: initializes the maze of caves as defined in
//                "caves.dat", randomly places objects in caves
{
    ifstream istr("caves.dat");     // open caves data file,
    assert(istr);                   // verifying that it exists
    istr >> myNumCaves;             // read in number of caves,
    assert(myNumCaves > 3);         // making sure enough space
    myCaves.SetSize(myNumCaves);    // if ok, resize Vector

    int a, caveNum;                 // read in & store cave info
    while (istr >> caveNum) {
        for (a = 0; a < 3; a++) {
            istr >> myCaves[caveNum].adj[a];
        }
        getline(istr, myCaves[caveNum].name);
        myCaves[caveNum].contents = EMPTY;
        myCaves[caveNum].visited = false;
    }

    RandGen rando;                  // initialize state variables
    myNumWumpi = rando.RandInt(1, myNumCaves/4);
    myNumGrenades = 4*myNumWumpi;
    myIsAlive = true;
    myCurrentLoc = 0;

    myCaves[myCurrentLoc].contents = WUMPUS; // temporarily mark
    // initial location

    for (a = 1; a <= myNumWumpi; a++) {
        myCaves[FindEmpty()].contents = WUMPUS; // place wumpi,
    }

    myCaves[FindEmpty()].contents = PIT;   // bats and pit
    myCaves[FindEmpty()].contents = BATS;   // when done, mark
    myCaves[myCurrentLoc].contents = EMPTY;  // initial location
}

void Maze::Toss(int tunnel)
// precondition : 1 <= tunnel <= 3
// postcondition: if no grenades, an error message is displayed
if (myNumGrenades == 0) {
    cout << "You have no more grenades to throw!" << endl;
} else {
    myNumGrenades--;

    int target = myCaves[myCurrentLoc].adj[tunnel-1];
    if (myCaves[target].contents == WUMPUS) {
        cout << "Got that sucker!" << endl;
        myNumWumpi--;
        myCaves[target].contents = EMPTY;
        if (myNumWumpi == 0) {
            cout << "And that was the last one!" << endl;
        }
    } else {
        cout << "Missed, dagnabit!" << endl;
    }
}

if (myNumWumpi > 0) {
    cout << "DANGER: Any nearby wumpi are on the move."
    << endl;

    RandGen rando;
    int f;
    for (f = 0; f < 3; f++) {
        int found = myCaves[myCurrentLoc].adj[f];
        if (myCaves[found].contents == WUMPUS &&
            (myCaves[myCaves[found].adj[0]].contents == EMPTY ||
             myCaves[myCaves[found].adj[1]].contents == EMPTY ||
             myCaves[myCaves[found].adj[2]].contents == EMPTY)) {
            int newWumpus = myCaves[found].adj[rando.RandInt(0,2)];
            while (myCaves[newWumpus].contents != EMPTY) {
                newWumpus = myCaves[found].adj[rando.RandInt(0,2)];
            }
        }
    }

    myCaves[found].contents = EMPTY;
    myCaves[newWumpus].contents = WUMPUS;
    if (newWumpus == myCurrentLoc) {
        cout << "A wumpus is coming toward you with big, "
        << "gnarly teeth... CHOMP CHOMP CHOMP";
        myIsAlive = false;
    }
}

void Maze::Move(int tunnel)
// precondition : 1 <= tunnel <= 3
// postcondition: move into specified cave tunnel (altering
// myCurrentLoc). If new location contains a wumpus
// or pit, then player is killed; if it contains
// bats, then move to random empty cave.
{
    myCaves[myCurrentLoc].visited = true;
    myCurrentLoc = myCaves[myCurrentLoc].adj[tunnel-1];
    if (myCaves[myCurrentLoc].contents == WUMPUS) {
        cout << "You have entered the Wumpus' lair. "
             << "Here he comes... CHOMP CHOMP CHOMP!" << endl;
        myIsAlive = false;
    }
    else if (myCaves[myCurrentLoc].contents == BATS) {
        cout << "FLAP FLAP FLAP..."
             << " A horde of giant bats have swept you away"
             << endl;
        myCaves[myCurrentLoc].visited = true;
        myCurrentLoc = FindEmpty();
    }
    else if (myCaves[myCurrentLoc].contents == PIT) {
        cout << "EEEEEEEEEeeeeeeeeee..."
             << " You have fallen into a bottomless pit."
             << endl;
        myIsAlive = false;
    }
}

string Maze::CurrentName() const
// postcondition: returns name of cave where currently located
{
    return myCaves[myCurrentLoc].name;
}

string Maze::AdjacentName(int tunnel) const
// precondition: 1 <= tunnel <= 3
// postcondition: returns name of cave adj. to the current one,
// if not previously visited, name is "unknown"
{
    if (myCaves[myCaves[myCurrentLoc].adj[tunnel-1]].visited) {
        return myCaves[myCaves[myCurrentLoc].adj[tunnel-1]].name;
    } else {
        return "unknown";
    }
}

bool Maze::Feel() const
// postcondition: returns true if an adj. cave contains a pit
{
    return
    (myCaves[myCaves[myCurrentLoc].adj[0]].contents == PIT ||
    myCaves[myCaves[myCurrentLoc].adj[1]].contents == PIT ||
    myCaves[myCaves[myCurrentLoc].adj[2]].contents == PIT);
}

bool Maze::Hear() const
// postcondition: returns true if an adj. cave contains bats
{
bool Maze::Smell() const
// postcondition: returns true if an adj. cave contains a wumpus
{
    return
    (myCaves[myCaves[myCurrentLoc].adj[0]].contents == WUMPUS ||
     myCaves[myCaves[myCurrentLoc].adj[1]].contents == WUMPUS ||
     myCaves[myCaves[myCurrentLoc].adj[2]].contents == WUMPUS);
}

bool Maze::StillAlive() const
// postcondition: returns true if player is still alive
{
    return myIsAlive;
}

bool Maze::StillWumpi() const
// postcondition: returns true if at least one wumpus is alive
{
    return (myNumWumpi > 0);
}

int Maze::FindEmpty()
// postcondition: returns the number of a random empty cave
{
    RandGen rando;
    int room = rando.RandInt(0,myNumCaves-1);
    while (myCaves[room].contents != EMPTY) {
        room = rando.RandInt(0,myNumCaves-1);
    }
    return room;
}
void DisplayIntro();
void DisplayLocation(const Maze & m);
void GetCommand(char & action, int & target);

int main()
{
    Maze m;

    DisplayIntro();

    while (m.StillAlive() && m.StillWumpi()) {
        DisplayLocation(m);

        char action;
        int target;
        GetCommand(action, target);

        if (action == 'T') {
            m.Toss(target);
        } else {
            m.Move(target);
        }
    }

    cout << endl << "GAME OVER" << endl;

    return 0;
}

void DisplayIntro()
// postcondition: displays introduction to the game
{
    cout << "HUNT THE WUMPUS: Your mission is to explore the "
    << "maze of caves" << endl;
    cout << "and destroy all of the wumpi (without getting "
    << "yourself killed)." << endl;
    cout << "To move to an adjacent cave, enter 'M' and the "
    << "tunnel number." << endl;
    cout << "To toss a grenade into a cave, enter 'T' and the "
    << "tunnel number." << endl;
}
void DisplayLocation(const Maze & m)
   // precondition: none
   // postcondition: displays location and names of adjacent caves,
   //                as well as warnings about nearby dangers
   {
      cout << endl;
      cout << "You are currently in " << m.CurrentName() << endl;
      cout << "    (1) " << m.AdjacentName(1) << endl;
      cout << "    (2) " << m.AdjacentName(2) << endl;
      cout << "    (3) " << m.AdjacentName(3) << endl;

      if (m.Feel()) {
         cout << "You feel a draft." << endl;
      }
      if (m.Hear()) {
         cout << "You hear the flapping of wings." << endl;
      }
      if (m.Smell()) {
         cout << "You smell a WUMPUS!" << endl;
      }
   }

void GetCommand(char & action, int & target)
   // precondition: none
   // postcondition: prompts and reads commands from the user:
   //                a char ('T' or 'M') followed by an int (1..3)
   {
      cout << endl << "What do you want to do? ";

      target = 0;
      cin >> action;
      action = toupper(action);
      if (action == 'M' || action == 'T') {
         cin >> target;
      }

      if (target < 1 || target > 3) {
         string dummy;
         getline(cin,dummy);
         cout << "Enter either 'M' for move or 'T' for toss a "
              << "grenade," << endl;
         cout << "followed by a number (1-3) specifying the "
              << "cave opening." << endl;
         GetCommand(action, target);
      }
   }
Assignment 8

Sorting, Recursion, and Efficiency

Since sorting is such a central task in computer science, numerous algorithms have been
developed for sorting a list of items into nondecreasing order. As such, these algorithms
provide ample opportunity for comparison and for careful analysis. For this assignment, you
will implement two different sorting algorithms that are described in the text, and compare
their efficiency.

PART 1: Implement the Merge sort algorithm described in Exercise 11.5 (pp. 527-528). Similar
to Quick sort, Merge sort breaks a list in two and recursively sorts each half. Unlike Quick sort,
however, Merge sort is $O(N \log N)$ in the worst case, where $N$ is the number of items to be
sorted. In the exercise, part of the code for a templated $\text{MergeSort}$ function is provided for you.
The only remaining code that must be written is for the $\text{Merge}$ function, which merges two
sorted sections of a Vector together. That is, the header for the function should appear as
follows:

```cpp
template <class Type>
void Merge(Vector<Type> & a, int left, int mid, int right)
// precondition : 0 <= left <= mid <= right <= a.length()
//                a[left] <= ... <= a[mid], and
//                a[mid+1] <= ... <= a[right]
// postcondition: a[left] <= ... <= a[right]
```

Merging the sorted two sections of the Vector into a single sorted section can be done efficiently.
In particular, if each sorted section contains $N$ items, then the two sections can be merged in
$O(N)$ steps. To do so, you need an auxiliary Vector of size $2N$. The two sorted sections are
scanned from left to right, and the smaller element is copied to the auxiliary Vector. The
scanning index is incremented only when an element is copied. When all of the elements from
both section have been copied to the auxiliary Vector, then the contents of that Vector (now
completely sorted) can be copied back into the original Vector.

Write this missing $\text{Merge}$ function and test your code thoroughly. To do so, write a driver
program which reads in integers and stores them in a Vector. Then, sort the Vector and display
the contents. Similarly, test your code on a Vector of strings. Since the relational operators (==,<,<=,...) are defined for strings, your templated code should work on a Vector of strings as well
as a Vector of numbers.

PART 2: Similarly, implement the Insertion sort algorithm described in Exercise 11.1 (pp. 525-526). Like Selection sort, Insertion sort is an $O(N^2)$ sort, where $N$ is the number of items to be
sorted. A function for performing an Insertion sort on a Vector of ints is given in the Exercise.
Generalize this function using a template so that the resulting function can be used to sort a
Vector containing any type for which the relational operators are defined. Test your function thoroughly.
PART 3: In order to compare the two sorting algorithms head-to-head, write a program called `sorter.cc` which will generate a Vector of random words and use each of the algorithms to sort the Vector. By timing their performances on the same Vector, you can compare the two algorithms directly.

Your program should prompt the user for the number of words to be sorted. It should then generate that many random words and store them in a Vector. There should be a constant specifying the maximum length of a random word. Each word should consist of a random sequence of letters, with a random length between 1 and the constant. For example, if the constant is 5, then your program should generate random words whose lengths range from 1 to 5. You should then sort this Vector using each of the sorting algorithms, Merge sort and Insertion sort. (Note: you will need to make a copy of the Vector so that each algorithm sorts the exact same sequence of words.) Using the `CTimer` class, time each of the algorithms and display the amount of time required. For example,

```
> sorter
How many words (length <= 5) do you want to sort? 400

Time to sort using Merge sort = 0.1 sec
Time to sort using Insertion sort = 0.216667 sec
```

ANALYSIS: Using your program, sort Vectors of the following sizes with word length <= 5, and list the timings for each of the algorithms.

<table>
<thead>
<tr>
<th></th>
<th>Insertion Sort</th>
<th>MergeSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>800 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1600 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3200 words</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Since Insertion sort is an $O(N^2)$ sort, its rate of growth should be quadratic. That is, if you double the size of the list, then the amount of time needed to sort it should increase by a factor of 4. Do your timings support this claim?
• Is Merge sort faster than Insertion sort in all cases? If not, is there a point after which Merge sort is always faster? Explain.
**ANALYSIS:** Now, increase the maximum word length to 10 and time the algorithms again.

<table>
<thead>
<tr>
<th></th>
<th>Insertion Sort</th>
<th>MergeSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>400 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>800 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1600 words</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3200 words</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Does changing the lengths of the words affect the rate of growth of the Insertion sort algorithm. That is, do the timings still increase by a factor of 4 as the list size doubles? Should this be the case? Explain.

- How does changing the word lengths affect the comparison between the two algorithms? Is one algorithm affected more by the change in word lengths? Explain any changes that in the occur in the timings.
Assignment 8 Solution

// File : compare.cc
// Author: Dave Reed

// This program sorts a Vector of random words using both Merge
// sort and Insertion sort, timing each sort on the same Vector.

#include <iostream.h>
#include "CPstring.h"
#include "vector.h"
#include "rando.h"
#include "ctimer.h"

const int MAX_LENGTH = 5;

void GetRandomWords(Vector<string> & words, int numWords);
template <class Type> void MergeSort(Vector<Type> & a, int n);
template <class Type> void InsertionSort(Vector<Type> & a, int n);

int main()
{
    int numWords;
    cout << "How many words (length <= " << MAX_LENGTH
    << ") do you want to sort? ";
    cin >> numWords;

    CTimer stopwatch1, stopwatch2;
    Vector<string> words(numWords);
    GetRandomWords(words, numWords);
    Vector<string> copy = words;

    stopwatch1.Start();
    MergeSort(words, numWords);
    stopwatch1.Stop();

    stopwatch2.Start();
    InsertionSort(copy, numWords);
    stopwatch2.Stop();

    cout << "Time to sort using Merge sort = "
    << stopwatch1.CumulativeTime() << " sec" << endl;
    cout << "Time to sort using Insertion sort = "
    << stopwatch2.CumulativeTime() << " sec" << endl;

    return 0;
}

void GetRandomWords(Vector<string> & words, int numWords)
// postcondition: words contains numWords random words, each with length <= MAX_LENGTH
{
  RandGen gen;

  int i;
  for (i = 0; i < numWords; i++) {
    words[i] = "";
    int k, len = gen.RandInt(1, MAX_LENGTH);
    for (k = 0; k < len; k++) {
      words[i] += gen.RandInt('a', 'z');
    }
  }
}

template <class Type>
void Merge(Vector<Type> & a, int left, int mid, int right)
// precondition : 0 <= left <= mid <= right <= a.length()
//                a[left] <= ... <= a[mid], and
//                a[mid+1] <= ... <= a[right]
// postcondition: a[left] <= ... <= a[right]
{
  int size = right-left+1, midSize = mid-left;
  Vector<Type> temp(size);

  int i;
  for (i = 0; i < size; i++) {
    temp[i] = a[left+i];
  }

  int leftFront = 0, rightFront = midSize+1;
  for (i = left; i <= right; i++) {
    if (leftFront > midSize) {
      a[i] = temp[rightFront];
      rightFront++;
    } else if (rightFront > size-1) {
      a[i] = temp[leftFront];
      leftFront++;
    } else if (temp[leftFront] < temp[rightFront]) {
      a[i] = temp[leftFront];
      leftFront++;
    } else {
      a[i] = temp[rightFront];
      rightFront++;
    }
  }
}

template <class Type>
void MergeSort(Vector<Type> & a, int left, int right)
// precondition : 0 <= left <= mid <= right <= a.length()
// postcondition: a[left] <= ... <= a[right]
if (left < right) {
    int mid = (left+right)/2;
    MergeSort(a, left, mid);
    MergeSort(a, mid+1, right);
    Merge(a, left, mid, right);
}

template <class Type>
void MergeSort(Vector<Type> & a, int n)
// precondition : a contains n items (n >= 0)
// postcondition: a[0] <= ... <= a[n-1]
{
    MergeSort(a, 0, n-1);
}

template <class Type>
void InsertionSort(Vector<Type> & a, int n)
// precondition : a contains n items (n >= 0)
// postcondition: a[0] <= ... <= a[n-1]
{
    int k, loc;
    Type hold;
    for (k = 1; k < n; k++) {
        hold = a[k];
        loc = k;
        while (loc > 0 && a[loc-1] > hold) {
            a[loc] = a[loc-1];
            loc--;
        }
        a[loc] = hold;
    }
}
ANALYSIS:

Running on a PowerMac 6100/66: to sort words with lengths <= 5:

<table>
<thead>
<tr>
<th></th>
<th>Insertion Sort</th>
<th>MergeSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 words</td>
<td>0.0 sec</td>
<td>0.0166667 sec</td>
</tr>
<tr>
<td>200 words</td>
<td>0.0333333 sec</td>
<td>0.0333333 sec</td>
</tr>
<tr>
<td>400 words</td>
<td>0.1666667 sec</td>
<td>0.0666667 sec</td>
</tr>
<tr>
<td>800 words</td>
<td>0.6666667 sec</td>
<td>0.3666667 sec</td>
</tr>
<tr>
<td>1600 words</td>
<td>2.7166667 sec</td>
<td>0.9333333 sec</td>
</tr>
<tr>
<td>3200 words</td>
<td>10.416667 sec</td>
<td>2.5666667 sec</td>
</tr>
</tbody>
</table>

According to the numbers, Insertion Sort does exhibit quadratic behavior, since doubling the number of words results in a sort that is roughly 4 times as long.

Because of the overhead of the merging process, MergeSort can be slower when the number of words to be sorted is small. Here, Insertion Sort is quicker on 100 words, and they are the same on 200 words. After that point, however, the $O(N \log N)$ sort will always be faster.

Running on a PowerMac 6100/66: to sort words with lengths <= 10:

<table>
<thead>
<tr>
<th></th>
<th>Insertion Sort</th>
<th>MergeSort</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 words</td>
<td>0.0 sec</td>
<td>0.0166667 sec</td>
</tr>
<tr>
<td>200 words</td>
<td>0.05 sec</td>
<td>0.0333333 sec</td>
</tr>
<tr>
<td>400 words</td>
<td>0.1666667 sec</td>
<td>0.0833333 sec</td>
</tr>
<tr>
<td>800 words</td>
<td>0.75 sec</td>
<td>1.1833333 sec</td>
</tr>
<tr>
<td>1600 words</td>
<td>2.8333333 sec</td>
<td>3.6166667 sec</td>
</tr>
<tr>
<td>3200 words</td>
<td>14.383333 sec</td>
<td>9.6166667 sec</td>
</tr>
</tbody>
</table>

Increasing the lengths of the random words does not affect the quadratic rate of growth for Insertion Sort. In fact, the numbers for Insertion Sort are not significantly different. Merge Sort, on the other hand, is slowed considerably starting at 400 words. This is due to the excessive amount of copying that is done during the Merge stage. Since the words are now longer, it takes more time to copy them. After 3200 words, however, the $O(N \log N)$ sort will always be faster.
Assignment 9

Pixel-mapped Graphics

One way to represent a picture is to break it down into individual points of light or color called pixels. This is what is done on a computer terminal or television screen, where a picture is drawn one dot (pixel) at a time. For a black-and-white picture, a pixel can either be off (white) or on (black). For a grayscale or color picture, a pixel can have a wider range of values. For example, the values 0 to 255 might be used to represent 256 different shades of gray. A pixel-map is a 2-D matrix of integers, each corresponding to a pixel in a picture. For example, the matrix diagrammed below is a pixel-map which represents a 9x8 (black-and-white) picture of a < sign.

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\end{array}
\]

For this assignment, you will complete the definition of a class for storing and manipulating pixel-maps. Attached at the end of this assignment is a partial definition of the Pixmap class, and a listing of a simple driver program which allows the user manipulate pictures by selecting commands from a menu.

As is, the only Pixmap member functions implemented are Read, Write, and Invert. The Read member function reads in a pixel-map from a file and stores it internally in a Matrix of shorts. The only formats currently recognized by Read is ASCII pbm, either black-and-white or grayscale. A black-and-white file will have the form:

P1
<info about creator of file>
<number of columns> <number of rows>
<pixel values, separated by whitespace>

A grayscale file will have the form:

P2
<info about creator of file>
<number of columns> <number of rows>
<maximum pixel value>
<pixel values, separated by whitespace>

The Write member function writes the stored pixel-map to a file in its current form. Invert turns the picture into a "photo negative" by inverting the value of each pixel. You may test
these member functions by using the supplied driver program pix.cc to read in, invert, and write out a pixel-map. You can then inspect the resulting file, or use a graphics programs (such as XV on UNIX) to view the picture.

You must implement the missing member functions and make the following additions and modifications to the Pixmap class:
• **HorizReflect**: implement this member function which reflects the stored picture horizontally, that is, along the x-axis.

• **VertReflect**: implement this member function which reflects the stored picture vertically, that is, along the y-axis.

• **Compress**: implement this member function which writes the stored picture to a file in a compressed form. In particular, it should use the run-length encoding scheme, listing the number of consecutive 0's at the start of the pixel-map, followed by the number of consecutive 1's, then 0's, then 1's, etc. For example, the pixel-map shown above is encoded as: 5 2 5 2 5 2 5 2 7 2 7 2 7 2 7 2 1.

  Note: the Compress member function should only be applicable to black-and-white pixel-maps. If this function is called on a grayscale pixel-map, it should print an error message and write nothing to the file. When you write a file in compressed format, put the label "C1" on the first line of the file (instead of "P1").

• **Read**: modify the member function so that it can read in files stored in the compressed form described above. When reading a file, the member function should inspect the label on the first line. If it is "P1" or "P2", then it should behave as already defined. If it is "C1", signifying a run-length encoding, then it should read in and convert the picture to pixel-map form for internal storage.

• **Restore**: implement this member function which restores any perturbations in the pixel-map using a median smoothing algorithm. According to this algorithm for removing "bad" pixels, each pixel should be replaced by the median value stored in its "neighborhood". If that pixel is in the interior of the picture, then its neighborhood is all eight of its adjacent pixels plus itself. For a pixel on the edge of the picture, its neighborhood is partial, including only those adjacent pixels that appear in the picture.

Recall that the median of a set of values is that value which is precisely in the middle, i.e., for which half are larger and half are smaller. In other words, the median of N numbers is the (N/2)th largest value. For example, the median of the values 10, 8, 14, 33, and 12 is 12 (the 3rd largest value). Similarly, the median of 13, 9, 18, and 7 is 9 (the 2nd largest value).

For example, applying the median smoothing algorithm to the pixel map on the left produces the map on the right:

```
10 14 18 22 | 10 12 14 16
8 12 16 12 | 10 10 12 12
6 10 10 10 | 8  9 10 10
0  8  9 23 | 6  8 10 10
```
Note: the Restore member function should only be applicable to grayscale pixel-maps. If this function is called on a black-and-white pixel-map, it should simply print an error message.

Advice: Attempt this assignment in stages. First implement the HorizReflect member function, add a case to the driver program, and then test it out. Test your code on small, hand-crafted pictures such as the < pixel-map first, since it is easier to check correctness on small files. Once you are convinced this function works, move on to the next one.
// Pixmap class: store and manipulate 2-D pixel-maps
//
// CONSTRUCTION: with no initializer
//
// *************************PUBLIC OPERATIONS*************************
// void Read(istream & input) --> Read pixel-map from input
// void HorizReflect() --> reflect along y-axis
// void VertReflect() --> reflect along x-axis
// void Invert() --> convert to negative image
// void Restore() --> restore using medians
// void Write(ostream & output) --> Write pixel-map to output
// void Compress(ostream & output) --> Write compressed pixel-map
//_________________________________________________________________

#include "CPstring.h"
#include "matrix.h"

class Pixmap{
public:
    Pixmap();
    void Read(istream & input);
    void HorizReflect();
    void VertReflect();
    void Invert();
    void Restore();
    void Write(ostream & output);
    void Compress(ostream & output);
private:
    string myKind;  // P1, P2, C1, etc.
    string myCreator;  // program that created Pixmap
    int myNumRows, myNumCols;  // dimensions of Pixmap
    short myPixelRange;  // possible number of pixel values
    Matrix<short> myImage;  // 2-D array of "pixels"
};

void Pixmap::Invert()
// postcondition: makes "photo-negative" by inverting each pixel
{
    int row, col;
    for (row = 0; row < myNumRows; row++) {
        for (col = 0; col < myNumCols; col++) {
            myImage[row][col] = myPixelRange - myImage[row][col];
        }
    }
}
void Pixmap::Read(istream & input)
// precondition : input stream contains a pixel-map
// postcondition: reads in and stores pixel-map
{
    if (input){
        getline(input, myKind);       // read P1, P2, C1, etc
        getline(input, myCreator);    // creating program
        input >> myNumCols >> myNumRows; // rows and cols
        myImage.SetSize(myNumRows, myNumCols);
        if (myKind == "P2") {
            input >> myPixelRange;
        } else {
            myPixelRange = 1;
        }
        if (myKind == "P1" || myKind == "P2"){
            int j, k;
            for(j = 0; j < myNumRows; j++){
                for(k = 0; k < myNumCols; k++){
                    input >> myImage[j][k];
                }
            }
        }
    }
}

void Pixmap::Write(ostream & output)
// precondition: output is open for writing
// postcondition: Pixmap information written to output
{
    int count = 0;
    output << myKind << endl;
    output << "# CREATOR: Reed pixmap program v 2.1" << endl;
    output << myNumCols << " " << myNumRows << endl;
    if (myKind == "P2") {
        output << myPixelRange << endl;
    }
    int j, k;
    for(j=0; j < myNumRows; j++){
        for(k=0; k < myNumCols; k++){
            output << myImage[j][k] << " ";
        }
        output << endl;
    }
    output << endl;
}
// Driver program for pixel-map manipulation.

#include <iostream.h>
#include <fstream.h>
#include "CPstring.h"
#include "pixmap.h"

int main()
{
  Pixmap pic;
  string command;

  do {
    cout << "Enter command: " << endl
    << "  (L)oad a picture (pixel-map form)" << endl
    << "  (I)nvert a picture" << endl
    << "  (S)ave a picture (pixel-map form)" << endl
    << "  (Q)uit" << endl << endl;
    getline(cin, command);
    command = command.Downcase();

    switch (command[0]) {
      case 'l': { string filename;
        cout << "Enter input file name: ";
        getline(cin, filename);
        ifstream istr(filename);
        pic.Read(istr);
      }
        break;
      case 'i': pic.Invert();
        break;
      case 's': { string filename;
        cout << "Enter output (pixel-map) "
        << "file name: ";
        getline(cin, filename);
        ofstream ostr(filename);
        pic.Write(ostr);
      }
        break;
      case 'q': cout << "Goodbye" << endl;
        break;
      default : cout << "Unknown command ignored." << endl;
        break;
    }
    cout << endl;
  } while (command[0] != 'q');

  return 0;
}
Assignment 9 Solution

// implementation of Pixmap class for manipulating images
////////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "CPstring.h"
#include "Pixmap.h"

Pixmap::Pixmap()
// constructor: initializes empty pixel-map
{
myKind = "unknown";
myCreator = "unknown";
myNumRows = 0; myNumCols = 0;
}

void Pixmap::Read(istream & input)
// precondition : input stream contains a pixel-map
// postcondition: reads in and stores pixel-map
{
if (input){
    getline(input, myKind); // read P1, P2, C1, etc
    getline(input, myCreator); // creating program
    input >> myNumCols >> myNumRows; // rows and cols
    myImage.SetSize(myNumRows, myNumCols);
    if (myKind == "P2") {
        input >> myPixelRange;
    } else {
        myPixelRange = 1;
    }

    if (myKind == "P1" || myKind == "P2") {
        int j, k;
        for (j=0; j < myNumRows; j++) {
            for (k=0; k < myNumCols; k++) {
                input >> myImage[j][k];
            }
        }
    } else if (myKind == "C1") {
        short currentBit = 0;
        int currentStreak;
        input >> currentStreak;
        int r, c;
        for (r = 0; r < myNumRows; r++) {
            for (c = 0; c < myNumCols; c++) {
                int bit;
                input >> bit;
                if (bit == 1) {
                    currentStreak = currentStreak + 1;
                } else {
                    currentPixelRange = currentStreak;
                    currentStreak = 0;
                }
            }
        }
    }
}

for (c = 0; c < myNumCols; c++) {
    if (currentStreak == 0) {
        input >> currentStreak;
        currentBit = 1 - currentBit;
    }
    myImage[r][c] = currentBit;
    currentStreak--;
}
else {
    cout << "UNKNOWN PICTURE TYPE" << endl;
}
}

void Pixmap::HorizReflect()
// postcondition: reflects picture along x-axis
{
    int row, col;
    for (row = 0; row < myNumRows/2; row++) {
        for (col = 0; col < myNumCols; col++) {
            short temp = myImage[row][col];
            myImage[row][col] = myImage[myNumRows-row-1][col];
            myImage[myNumRows-row-1][col] = temp;
        }
    }
}

void Pixmap::VertReflect()
// postcondition: reflects picture along y-axis
{
    int row, col;
    for (row = 0; row < myNumRows; row++) {
        for (col = 0; col < myNumCols/2; col++) {
            short temp = myImage[row][col];
            myImage[row][col] = myImage[row][myNumCols-col-1];
            myImage[row][myNumCols-col-1] = temp;
        }
    }
}

void Pixmap::Invert()
// postcondition: makes "photo-negative" by inverting each pixel
{
    int row, col;
    for (row = 0; row < myNumRows; row++) {
        for (col = 0; col < myNumCols; col++) {
            myImage[row][col] = myPixelRange - myImage[row][col];
        }
    }
}
static
void Insert(Vector<short> & nums, int & numNums, short newNum) {
    int i;
    for (i = numNums; i > 0 && newNum < nums[i-1]; i--) {
        nums[i] = nums[i-1];
    }
    nums[i] = newNum;
    numNums++;
}

void Pixmap::Restore()
// postcondition: restores picture using median filtering
{
    Matrix<short> copy = myImage;
    for (int row = 0; row < myNumRows; row++) {
        for (int col = 0; col < myNumCols; col++) {
            Vector<short int> neighbor(9);
            int numNeighbors = 0;

            for (int r = row-1; r <= row+1; r++) {
                for (int c = col-1; c <= col+1; c++) {
                    if (r >= 0 && r < myNumRows &&
                        c >= 0 && c < myNumCols) {
                        Insert(neighbor, numNeighbors, copy[r][c]);
                    }
                }
            }
            myImage[row][col] = neighbor[(numNeighbors-1)/2];
        }
    }
}

void Pixmap::Write(ostream & output)
// precondition: output is open for writing
// postcondition: Pixmap information written to output
{
    int count = 0;
    output << myKind << endl;
    output << "# CREATOR: Reed pixmap program v 2.1" << endl;
    output << myNumCols << " " << myNumRows << endl;

    if (myKind == "P2") {
        output << myPixelRange << endl;
    }

    int j, k;
    for(j=0; j < myNumRows; j++){
        for(k=0; k < myNumCols; k++){
            output << myImage[j][k] << " ";
        }
        output << endl;
    }
void Pixmap::Compress(ostream & output)
// precondition: output is open for writing
// postcondition: run-length encoding of pixel-map is written
{
if (myKind != "P1") {
    cout << "COMPRESS ONLY WORKS ON BLACK-AND-WHITE PIXMAPS" << endl;
}
else {
    output << "C1" << endl;
    output << "# CREATOR: Reed pixmap program v 2.1" << endl;
    output << myNumCols << " " << myNumRows << endl;

    int currentStreak = 0;
    short currentBit = 0;

    int r, c;
    for (r = 0; r < myNumRows; r++) {
        for (c = 0; c < myNumCols; c++) {
            if (myImage[r][c] != currentBit) {
                output << currentStreak << " ";
                currentStreak = 0;
                currentBit = 1 - currentBit;
            }
            currentStreak++;
        }
    }
    output << currentStreak;
}
// Driver program for pixel-map manipulation.

#include <iostream.h>
#include <fstream.h>
#include "CPstring.h"
#include "Pixmap.h"

int main()
{
    Pixmap pic;
    string command;

    do {
        cout << "Enter command: " << endl
            << " (L)oad a picture (pixel-map form)" << endl
            << " (I)nvert a picture" << endl
            << " (H)orizontal reflect (along x-axis)" << endl
            << " (V)ertical reflect (along y-axis)" << endl
            << " (R)estore a picture (grayscale only)" << endl
            << " (C)ompress a picture (black-and-white only)" << endl
            << " (S)ave a picture (pixel-map form)" << endl
            << " (Q)uit" << endl << endl;
        getline(cin, command);
        command = command.Downcase();

        switch (command[0]) {
        case 'l':
            { string filename;
              cout << "Enter input file name: ";
              getline(cin, filename);
              ifstream istr(filename);
              pic.Read(istr);
            }
            break;
        case 'i':
            pic.Invert();
            break;
        case 'h': pic.HorizReflect();
            break;
        case 'v': pic.VertReflect();
            break;
        case 'r': pic.Restore();
            break;
        case 'c':
            { string filename;
              cout << "Enter output (compressed) "
                  << "file name: ";
              getline(cin, filename);
              ofstream ostr(filename);
              pic.Compress(ostr);
            }
            break;
        case 's':
            break;
        }
```cpp
{ string filename;
    cout << "Enter output (pixel-map) "
        << "file name: ";
    getline(cin, filename);
    ofstream ostr(filename);
    pic.Write(ostr);
    break;
    case 'q': cout << "Goodbye" << endl;
    break;
    default:
        cout << "Unknown command ignored." << endl;
        break;
    }
    cout << endl;
} while (command[0] != 'q');

return 0;
}
Assignment 10

Linked Lists and Queueing Simulations

For this assignment, you will implement a `Queue` class using a linked list, and use this class for simulating the arrival and servicing of customers at a bank. A queue is known as a FIFO (first-in-first-out) list since items are added at one end (the rear) and removed from the other end (the front). Our everyday lives are full of queues, from the lines at a grocery store to jobs waiting to be printed on a network printer. As such, the queue data structure is commonly used in simulations of real-world events.

PART 1: Implement the templated `Queue` class whose definition is given below. The class has two data fields, pointers to the front and rear of a linked-list of nodes. It is the job of the constructor to initialize the queue to be empty, i.e., set both front and rear to be NULL. The destructor, on the other hand, must deallocate all nodes in the queue. The `IsEmpty` member function returns true if there are no items in the queue, `Enter` adds an item at the rear, `Remove` removes an item from the front, and `Front` returns the value stored at the front of the queue. Be sure to test your class thoroughly.

```cpp
template <class Type>
class Queue
{
    public:
        Queue();                // constructor - init to be empty
        Queue();               // destructor - deallocates nodes
        bool IsEmpty() const;   // return true if no items in queue
        void Enter(const Type & item);  // add item at rear
        void Remove();          // remove item from front
        Type Front() const;     // return value stored at front

    private:
        struct TypeNode {       // node for storing an item
            Type item;          // in a linked list, with
            TypeNode * next;    // pointer to next node
        };
        TypeNode * myFront;     // pointer to front
        TypeNode * myRear;      // pointer to rear


```
PART 2: Using your Queue class, write a program which simulates customers arriving and being served at a bank. In order to simplify this task, we will make some assumptions about the bank and its customers.

- There is only one teller, and customers are served on a first-come-first-served basis.
- The length of the banking day, in minutes, is a constant (call it NUM_MINUTES).
- At any given minute of the day, there is a certain probability (call it ARRIVAL_PROB) of a customer arriving at the bank. For example, if ARRIVAL_PROB is 15, then there is a 15% chance that a customer will arrive at any given minute.
- The length of the transaction for a given customer, in minutes, is a random integer between 1 and some maximum (call it MAX_TRANSACTION).

Your program will need to step through each minute of the day, using a random number generator to determine whether a customer has arrived and if so, the length of their transaction. And since customers may arrive while the teller is busy serving someone else, you will need to be able to store customer information in a queue so that waiting customers can be served in the correct order. You may find it useful to define the following struct type for storing customer information:

```c
struct CustomerInfo {    // struct for repr. a customer:
    int id;              //   stores the customer ID #,
    int arrived;         //   arrival time, and
    int length;          //   length of transaction
};
```

For the customer ID number, you can simply use the order of arrival. That is, the first customer of the day has ID #1, the second customer has ID #2, and so on. Your program should display the appropriate information whenever a customer arrives, is served, or departs. At the end of the simulation, your program should display the number of customers served, the average wait time (time spent waiting in the queue) over all the customers, and the maximum wait time. For example, a simulation with NUM_MINUTES = 20, ARRIVAL_PROB = 15, and MAX_TRANSACTION = 4 might produce the following output.

```
2:  customer 1 arrives (transaction length = 1)
2:  customer 1 served (busy until 3)
3:   customer 1 departs.
6:  customer 2 arrives (transaction length = 3)
6:  customer 2 served (busy until 9)
9:   customer 2 departs.
18: customer 3 arrives (transaction length = 2)
18: customer 3 served (busy until 20)
19:  customer 3 departs.
20: customer 5 arrives (transaction length = 3)
20: customer 4 served (busy until 23)
23:  customer 4 departs.
23:  customer 5 served (busy until 26)
26:   customer 5 departs.
```

Customers served = 5
Average wait time = 0.8 minutes
Maximum wait time = 3 minutes
Note that in this example, customer 4 arrived while customer 3 was still being served. Thus, she had to wait in the queue for 1 minute until the teller was free. Similarly, customer 5 had to wait 3 minutes until the teller was done with customer 4. Also note that the times displayed in this simulation exceed NUM_MINUTES, the number of minutes in the banking day. Your program should certainly stop allowing customers to enter the bank after NUM_MINUTES, but any customers that entered the bank before closing time should be served.

**ANALYSIS:** Once you have your bank simulation program working, answer the following questions:

1. Assuming a banking day of 8 hours, how many customers would you expect to serve if the arrival probability were 20%. Do your simulations support this answer?

2. Assuming a banking day of 8 hours, arrival probability of 15%, and a maximum transaction time of 10 minutes, what is the average wait time for customers at the bank. Run several simulations and comment on the consistency of the results.

3. Assuming a banking day of 8 hours and a maximum transaction time of 8 minutes, what arrival probability can the bank handle and still keep the maximum wait time for customers under 10 minutes. Explain how you came up with this number.

4. What should happen if the arrival probability is 0% ? Do your simulations support this answer?

5. What should happen if the arrival probability is 100% ? Do your simulations support this answer?
Assignment 10 Solution

// templated Queue class: for storing items in a FIFO structure
// CONSTRUCTION: with no initializers

ẫn *************PUBLIC OPERATIONS************
// bool IsEmpty() --> Return true if no items in queue
// void Enter(Type item) --> Add item to rear of queue
// void Remove() --> Remove item at front of queue
// Type Front() --> Return item at front of queue

#ifndef _QUEUE_H
#define _QUEUE_H

#include <iostream.h>

template <class Type>
class Queue
{
  public:
    Queue(); // constructor
    ~Queue(); // destructor
    bool IsEmpty() const;
    void Enter(const Type & item);
    void Remove();
    Type Front() const;

  private:
    struct TypeNode { // node for storing an item
      Type item; // in a linked list, with
      TypeNode * next; // pointer to next node
    };
    TypeNode * myFront; // pointer to front
    TypeNode * myRear; // pointer to rear
};

template <class Type>
Queue<Type>::Queue()
// constructor
// postcondition: myFront and myRear are both set to NULL
{
  myFront = NULL;
  myRear = NULL;
}

template <class Type>
Queue<Type>::~Queue()
// destructor
// postcondition: all nodes in queue are deallocated
{
  while (myFront != NULL) {
template <class Type>
bool Queue<Type>::IsEmpty() const
// postcondition: returns true if queue is empty
{
    return (myFront == NULL);
}

template <class Type>
void Queue<Type>::Enter(const Type & toBeInserted)
// postcondition: adds toBeInserted at rear of queue
{
    myRear->next = new TypeNode;
    myRear = myRear->next;
    myRear->item = toBeInserted;
    myRear->next = NULL;

    if (myFront == NULL) {
        myFront = myRear;
    }
}

template <class Type>
void Queue<Type>::Remove()
// postcondition: removes item at front of queue
{
    TypeNode * temp = myFront;
    myFront = myFront->next;
    delete temp;

    if (myFront == NULL) {
        myRear = NULL;
    }
}

template <class Type>
Type Queue<Type>::Front() const
// postcondition: returns item at front of queue
{
    return myFront->item;
}

#endif
// File : banksim.cc
// Author: Dave Reed
//
// Performs simulation of bank with a single teller (one queue)
// Every minute of the simulation, there is a ARRIVAL_PROB chance
// that a new customer will arrive, and the length of each
// transaction is a random integer from 1 to MAX_TRANSACTION.

#include <iostream.h>
#include "rando.h"
#include "queue.h"

struct CustomerInfo {          // struct for repr. customer info:
    int id;                    //   stores the customer ID #,
    int arrived;               //   arrival time, and
    int length;                //   length of transaction
    CustomerInfo(int ID = 0, int ARR = 0, int LEN = 0)
        : id(ID), arrived(ARR), length(LEN)
    { }                          
};

const int NUM_MINUTES = 480;   // # of minutes in a banking day
const int MAX_TRANSACTION = 8; // max time for a transaction
const int ARRIVAL_PROB = 20;   // prob. a customer will arrive

bool CustomerArrived();        // function prototypes
int TransactionTime();
void DisplayStats(int numCustomers, int totalWait, int maxWait);

int main()
{
    Queue<CustomerInfo> queue;
    int time, busyUntil = 0, numCustomers = 0;
    int totalWait = 0, maxWait = 0;

    for (time = 1;
        time <= NUM_MINUTES || !queue.IsEmpty();
        time++) {

        if ( time == busyUntil ) {
            cout << time << ":     customer " << queue.Front().id
            << " departs." << endl;
            queue.Remove();
        }

        if ( time <= NUM_MINUTES && CustomerArrived() ) {
            numCustomers++;
            CustomerInfo newCust(numCustomers,
                                  time,
                                  TransactionTime());

            cout << time << ":  customer " << newCust.id
            << " arrives (transaction length = "
            << newCust.length << ")" << endl;
            queue.Enter(newCust.length);
    }
if ( time >= busyUntil && !queue.IsEmpty() ) {
    busyUntil = time + queue.Front().length;
    cout << time << ": customer " << queue.Front().id << " served (busy until " << busyUntil << ")" << endl;
}
int currentWait = time - queue.Front().arrived;
totalWait += currentWait;
if (currentWait > maxWait) {
    maxWait = currentWait;
}
DisplayStats(numCustomers, totalWait, maxWait);
return 0;

bool CustomerArrived()
// precondition: 0 <= ARRIVAL_PROB <= 100
// postcondition: returns true with probability ARRIVAL_PROB
{
    RandGen rando;
    return (rando.RandInt(1, 100) <= ARRIVAL_PROB);
}
int TransactionTime()
// precondition: 0 <= MAX_TRANSACTION
// postcondition: returns int between 1 and MAX_TRANSACTION
{
    RandGen rando;
    return rando.RandInt(1, MAX_TRANSACTION);
}
void DisplayStats(int numCustomers, int totalWait, int maxWait)
// precondition: all >= 0
// postcondition: displays # of customers, avg & max wait times
{
    cout << endl << endl;
    if (numCustomers == 0) {
        cout << "There were no customers!" << endl;
    } else {
        cout << "Customers served = " << numCustomers << endl;
        cout << "Average wait time = " << double(totalWait)/numCustomers << " minutes" << endl;
        cout << "Maximum wait time = " << maxWait << " minutes" << endl;
    }
}
ANALYSIS:

1. With an arrival probability of 20%, there should be roughly one customer every twenty minutes. With a banking day of 480 minutes, this means 96 customers. Repeated simulations should come close to this number.

2. A typical result of a simulation yielded, with some variation on the average and maximum wait times:
   - Customers served = 62
   - Average wait time = 6.16179 minutes
   - Maximum wait time = 31 minutes

3. By setting the number of minutes and the maximum transaction time and varying the arrival probability, it would appear that an arrival probability of 7% tends to produce a simulation in which the maximum wait is under 10 minutes. Percentages greater than this more often produce a maximum wait of over 10 minutes.

4. When the arrival probability is 0%, no customers will arrive.

5. When the arrival probability is 100%, a customer arrives every minute. This will result in a horrendously long line (especially if the maximum transaction time is large), and correspondingly large wait times.
5. **Sample Tests**

This chapter contains three sample tests, corresponding to the three main sections of the text. Test 1 focuses on the Foundations of C++ Programming (Chapters 2 through 5), with major topics including variables, functions, and the basic control structures of C++. In addition, some attention is paid to the use (but not design) of classes. Test 2 focuses on Program and Class Construction (Chapters 6 through 8), with major topics including parameters, classes and structs, and Vectors. Test 3 focuses on Design, Use, and Analysis (Chapters 9 through 12), with major topics including strings, recursion, algorithm analysis, and linked lists.

The content of these tests is intentionally generic in order to make them applicable in most any course using the text. You may choose to add course specific questions to tests as well. For example, I often put questions on tests which are similar to homework problems. This allows for deeper questions since the students already have a context in which to understand the question. In addition, exercises from the text often make good test questions, or can be extended to form the basis of new questions. The use of these exercises has the added benefit of encouraging the students to look at the exercises when reading the text and studying for exams. Similarly, the Quiz/Review questions provided in an earlier chapter of this manual also can be used as test questions.

Each of the tests provided is designed for a 50-minute or one hour period. Since each test is scored out of 50 points, the points can be used by students as a guide for time management. For a 50-minute period, I always tell my students to allot roughly one minute per point. If they find that they are spending much more than the allotted time on a specific question, then they should move on to other questions and return if time allows. While 50-minute tests are probably the norm, I find them far from ideal. Designing and writing code is time consuming, so the number of meaningful questions that can be completed in 50-minutes is not very great. Unfortunately, keeping a test short means that each question is worth more (relatively speaking). Allowing for additional time, if possible, will allow for more questions with less pressure on the student since each question is worth fewer points. One way to accomplish this is to add a take-home component to the test. For example, ask fewer questions on the test, but then assign a problem that is to be completed and handed in at the next class period. Since there is no time pressure, the take-home question allows for more depth than is possible in an in-class test.
## Test 1 (Chapters 1 - 5)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>(5 pts) :</td>
</tr>
<tr>
<td>Problem 2</td>
<td>(5 pts) :</td>
</tr>
<tr>
<td>Problem 3</td>
<td>(8 pts) :</td>
</tr>
<tr>
<td>Problem 4</td>
<td>(4 pts) :</td>
</tr>
<tr>
<td>Problem 5</td>
<td>(14 pts) :</td>
</tr>
<tr>
<td>Problem 6</td>
<td>(14 pts) :</td>
</tr>
<tr>
<td>TOTAL</td>
<td>(50 pts) :</td>
</tr>
</tbody>
</table>
1. True or False? (5 points)

___________ The scope of a variable is that section of the program in which the variable exists.

___________ A #include statement is used to insert the contents of a separate file into a program.

___________ A constructor is a class member function that is automatically called whenever an object of that class is created.

___________ When C++ evaluates the boolean expression

        (num >= 0 && num <= 100)

the final value of the expression is determined only after both sub-expressions (num >= 0, num <= 0) are evaluated.

___________ Consider the following declaration which appears at the top of a program:

        const int TAX = 0.05;

The value of the variable TAX cannot subsequently be altered in the program, and any attempt to do so will be caught by the compiler.

2. Expressions (5 points)

Assume the following variables have been declared and initialized:

        int i = 5, j = 3;
        double d = 2.5;

What is the value of each of the following expressions? Be specific.

___________ 3+i * j

___________ i % j

___________ j / 2

___________ d/2
\text{___________} \quad \text{double}(i)/2
3. Program Execution  (8 points)

What is output as a result of executing the following program? Be specific.

```cpp
#include <iostream.h>

void foo()
{
    cout << "foo";
}

void bar(int n)
{
    cout << "bar" << endl;
    int k;
    for (k = 1; k <= n; k++) {
        foo();
    }
    cout << endl;
}

void biz(int foo)
{
    cout << "biz" << endl;
    int k;
    for (k = 1; k <= foo/2; k++) {
        bar(k);
    }
}

int main()
{
    foo();
    bar(3);
    int n = 4;
    biz(n);
}
```

output

4. Short Answer (4 points)

Throughout this course, we have stressed the importance of abstraction as a tool for managing complexity when solving problems. Describe one feature of C++ which supports abstraction, and how that feature makes problem solving easier.
5. C++ Code (14 points)

You have been hired by the Psychic Strangers Network in their numerology division. You have been supplied two boolean functions that are able to identify "lucky" and "unlucky" numbers. That is, the call `lucky(12)` will return true if 12 happens to be a lucky number (whatever that means), and false otherwise. Similarly, a call to `unlucky` will return true if its argument is an unlucky number, and false otherwise.

A. If we allow for some inconsistency on the part of psychics, it may be possible for a number to be both lucky and unlucky at the same time. Assuming this is the case, the if-statement below is intended to classify numbers as being lucky, unlucky, or both. However, this code does not produce the correct result in all cases. Explain why.

```cpp
if (lucky(x)) {
    cout << x << " is lucky" << endl;
} else if (unlucky(x)) {
    cout << x << " is unlucky" << endl;
} else {
    cout << x << " is both" << endl;
}
```

B. In order to check the consistency of your psychics, you wish to be able to test some range of numbers and see how many of them are both lucky and unlucky. Complete the definition of the `NumBoth` function whose header is given below. This function takes two integers as arguments, representing the low and high ends of the range to be tested, and returns how many numbers in that range were both lucky and unlucky.

```cpp
int NumBoth(int low, int high)
// precondition : low <= high
// postcondition: returns the number of mutually lucky and unlucky numbers in the range low to high
```
6. Classes (14 points)

Consider the Dice class whose definition is listed below:

```cpp
#include "rando.h"

class Dice
{
    public:
        Dice(int sides); // constructor
        int Roll(); // return random roll
        int NumSides(); // # sides this die has
        int NumRolls(); // # times this die rolled
    
    private:
        RandGen myGenerator; // random number generator
        int myRollCount; // # times die rolled
        int mySides; // # sides on die
};
```

A. What is the purpose of the line `#include "rando.h"` before the class definition? Is this line necessary? Explain.

B. Why are the member functions NumSides and NumRolls provided as part of the class? Why can't the client program simply inspect the values of the data fields myRollCount and mySides?

C. Utilizing this class, write a void function called RollUntilConsec which has no parameters. The function should repeatedly simulate the rolling of a 6-sided die and print out each roll, until the same value is obtained on consecutive rolls. It should then print out how many rolls it took to achieve this. For example, the output of the call RollUntilConsec() might be:

```
3
5
2
4
4
```

It took 5 rolls.

Write your function on the next page.
Test 1 Key

1. TRUE
   TRUE
   TRUE
   FALSE
   TRUE

2. 18
   2
   1
   1.25
   2.5

3. foobar
   foofloofoo
   biz
   bar
   foo
   bar
   foofoo

4. Functions provide one means for building abstractions, since the details of the internal code can be abstracted away. Instead of having to remember the low-level details, the programmer only needs to write the function once and then call it when needed. Similarly, C++ classes provide for data abstractions since they encapsulate data and operations to create new types. The details (data fields, implementations of the operations) need not be recalled, only the interface to the class.

5. A. In the case where a number is both lucky and unlucky, the first if-test will succeed, and the message "x is lucky" will be displayed.

   ```c
   int NumBoth(int low, int high)
   // precondition : low <= high
   // postcondition: returns the number of mutually lucky and unlucky numbers in the range low to high
   {
       int i, count = 0;

       for (i = low; i <= high; i++) {
           if (lucky(i) && unlucky(i)) {
               count++;
           }
       }

       return count;
   }
   ```
6. A. Since the Dice class has a data field of type RandGen, the file containing the definition of this class must be included.

B. Since the data fields are declared in the private section, they can only be accessed by the member functions of the class. The public member functions then provide access to the client program.

```cpp
void RollUntilConsec()
// postcondition: simulates rolls of the dice (printing each)
// until consecutive values are obtained, then
// prints total number of rolls
{
    Dice d(6);
    int previous, roll = 0;

    do {
        previous = roll;
        roll = d.Roll();
        cout << roll << endl;
    } while (roll != previous);

    cout << endl;
    cout << "It took " << d.NumRolls() << " rolls" << endl;
}
```
Test 2  (Chapters 6 - 8)

<table>
<thead>
<tr>
<th>Problem 1</th>
<th>(8 pts) :</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 2</td>
<td>(4 pts) :</td>
</tr>
<tr>
<td>Problem 3</td>
<td>(12 pts) :</td>
</tr>
<tr>
<td>Problem 4</td>
<td>(12 pts) :</td>
</tr>
<tr>
<td>Problem 5</td>
<td>(14 pts) :</td>
</tr>
<tr>
<td>TOTAL</td>
<td>(50 pts) :</td>
</tr>
</tbody>
</table>
1. True or False? (8 points)

___________ By default, parameters in C++ are passed by-reference.

___________ A sentinel is a special value that is placed at the end of an input sequence to mark the end of the data.

___________ While the get operator >> suffices for reading values from the standard input stream (cin), it cannot be applied to input file streams (ifstream).

___________ A copy constructor is a special constructor that defines how to copy an object when it is passed by value.

___________ The only difference between a struct and a class is that by default all data and functions in a struct are public whereas the default in a class is that everything is private.

___________ If binary search requires \( n \) inspections to search a list (worst case), then it will require \( 2n \) inspections (worst case) to search a list that is twice as large.

___________ The relational operators \( == \) and \( != \) are automatically defined for struct and class objects.

___________ The C++ statement

\[
\text{enum Color \{RED, WHITE, BLUE\};}
\]

defines a new type called Color whose only possible values are RED, WHITE, and BLUE.

2. Short Answer (4 points)

Throughout this course, we have predominantly used the Vector class to represent lists, as opposed to the built-in C++ array. Describe two differences between Vectors and arrays that make Vectors safer or more flexible.
3. Parameters (12 points)

Consider the following function:

```c
void foo(int x, int & y)
{
    x++;
    y = y - x;
    cout << "Inside: " << x << " , " << y << endl;
}
```

What is output as a result of each of the following code segments? Be specific.

```
output

int a = 5, b = 13;
foo(a, b);
cout << "Outside: " << a << " , " << b << endl;
```

```
output

int x = 6, y = -4;
foo(y, x);
cout << "Outside: " << x << " , " << y << endl;
```

```
output

int q = 6;
foo(q, q);
cout << "Outside: " << q << endl;
```
4. C++ Code (12 points)

Consider the header below for a function which computes the average length of words stored in a Vector.

```cpp
double AvgLength(const Vector<string> & words, int numWords)
// precondition: words contains numWords (>= 0) strings
// postcondition: returns avg word length (0.0 if no words)
```

A. What is the purpose of passing words by-const-reference as opposed to by-value or by-reference?

B. Complete the definition of this function. Recall that the string member function `length` can be used to determine the number of characters in a particular string.

```cpp
double AvgLength(const Vector<string> & words, int numWords)
// precondition: words contains numWords (>= 0) strings
// postcondition: returns avg word length (0.0 if no words)
```
5. Classes (14 points)

Consider the following struct and class for representing and manipulating playing cards. Each card is encoded as a struct with two character data fields, one for the suit and one for the card value.

```cpp
struct Card {
    char suit; // 'S', 'H', 'D' or 'C'
    char value; // '1', '2',..., 'T', 'J', 'Q', 'K' or 'A'
};

class DeckOfCards {
    public:
        DeckOfCards(); // constructs & shuffles a deck
        Card DealCard(); // returns card from "top" of the deck
        bool IsEmpty(); // returns true if no cards left
    private:
        Vector<Card> cards; // vector of card structs
        int numCards; // number of cards in deck
};

A. Give a C++ declaration that creates a DeckOfCards called deck.

B. The constructor for the DeckOfCards class has the following form:

```cpp
DeckOfCards::DeckOfCards() : numCards(52), cards(52)
{
    // code for shuffling the deck
}
```

What is the purpose of the code that appears after the colon in the first line? Why is this code necessary for constructing a DeckOfCards?
C. Complete the definition of the `DealUntilSpade` function, whose header is given below. This function deals cards, displaying each card, until a Spade is dealt. It then displays how many cards it took to get a Spade. If it so happens that the deck of cards is exhausted without dealing a Spade (i.e., it is only a partial deck), then the function should stop and display an error message.

For example, consider the following sample executions:

```
JH  AD
3C  TH
4D  9H
AS
It took 4 cards.
```

On the left, a Spade (the Ace) is dealt on the fourth card. On the right, the deck is exhausted before dealing a Spade (there were only three cards to start with), so an error message is displayed.

```cpp
void DealUntilSpade(DeckOfCards & deck)  
// postcondition: deals cards and displays them until a 
//                Spade is dealt (or deck is emptied)
```
Test 2 Key

1. FALSE
   TRUE
   FALSE
   TRUE
   TRUE
   FALSE
   FALSE
   TRUE

2. Advantages of Vectors over arrays:
   1. automatic bounds checking is performed
   2. can specify the size at run time
   3. can assign as a unit
   4. pass by-value works as expected (a safe copy is made)
   5. constructor can initialize all of the elements at once

3. Inside: 6, 7
   Outside: 5, 7

   Inside: -3, 9
   Outside: 9, -4

   Inside: 7, -1
   Outside: -1

4. A. By passing the Vector by-const-reference, the efficiency of by-reference is obtained (no copy of the Vector is made), but the safety of by-value is maintained (the compiler will catch any attempt to modify the contents of the Vector).

   ```cpp
   double AvgLength(const Vector<string> & words, int numWords)
   // precondition : words contains numWords (>= 0) strings
   // postcondition: returns avg word length (0.0 if no words)
   {
     if (numWords == 0) {
       return 0.0;
     }

     int i, charSum = 0;
     for (i = 0; i < numWords; i++) {
       charSum += words[i].length();
     }

     return double(charSum) / numWords;
   }
   ```
5. A. DeckOfCards deck;

B. Following the colon is an initializer list, which is used to initialize the data fields of the DeckOfCards. Although it would be possible to initialize the int field numCards inside the constructor, the Vector field must be initialized this way since its constructor must be called prior to the execution of the DeckOfCards constructor.

```cpp
void DealUntilSpade(DeckOfCards & deck)
// postcondition: deals cards and displays them until a Spade is dealt (or deck is emptied)
{
    int numDealt = 0;
    bool found = false;

    while (!deck.IsEmpty() && !found) {
        Card top = deck.DealCard();

        numDealt++;
        cout << top.suit << top.value << endl;

        if (top.suit == 'S') {
            found = true;
        }
    }

    if (found) {
        cout << "It took " << numDealt << " cards."
             << endl;
    }
    else {
        cout << "The deck is exhausted -- no spades."
             << endl;
    }
}
```
Test 3  (Chapters 9 - 12)

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>(6 pts)</td>
</tr>
<tr>
<td>Problem 2</td>
<td>(12 pts)</td>
</tr>
<tr>
<td>Problem 3</td>
<td>(8 pts)</td>
</tr>
<tr>
<td>Problem 4</td>
<td>(10 pts)</td>
</tr>
<tr>
<td>Problem 5</td>
<td>(14 pts)</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>(50 pts)</strong></td>
</tr>
</tbody>
</table>
1. True or False? (6 points)

___________  The expression `new int` returns a pointer to a dynamically allocated integer.

___________  If a member function is declared in the private section of a class, then that it can only be called by other member functions of that class.

___________  Unlike a linked list, a Vector provides random access to its elements.

___________  Declaring a variable inside a function to be static affects its scope, but not its lifetime.

___________  In C++, it is legal to overload a function name by using the same name for different functions.

___________  Suppose Algorithm A can sort a list of n items in $O(n \log n)$ time, while Algorithm B requires $O(n^2)$ time. Then A will always be faster than B when applied to the same list.

2. Short Answer (12 points)

A. Recall that one of the advantages of Vectors over arrays is that a Vector behaves as expected when passed by-value. That is, the formal parameter is assigned to be a copy of the argument, and changes to the parameter do not affect the original Vector. Alternatively, arrays do not behave this way. Even when an array is passed by-value, its contents can be altered inside the function. Explain why this is the case.
B. In practice, Quicksort tends to be the fastest of all the sorting algorithms. On average, it is an $O(n \log n)$ sort, where $n$ is the number of items being sorted. However, there are cases where Quicksort does not live up to its name, and is in fact $O(n^2)$. Describe the Quicksort algorithm. When and why does it degenerate to $O(n^2)$?

C. The idea of a class constructor, which is automatically called when an object of that class is created, was introduced early in this course (at the time classes were first introduced). However, the idea of a class destructor, which is automatically called upon destruction of the object, was introduced only recently. Why is it that destructors have not been useful until now?
3. Recursion (8 points)

Consider the following recursive function:

```cpp
void recurse(string str, int n) {
    cout << n << endl;
    if (n != 0) {
        recurse(str, n-1);
        int i;
        for (i = 0; i < n; i++) {
            cout << str[i];
        }
        cout << endl;
    }
}
```

If the string variable `str` has been assigned as follows: `str = "abc"`, what is output as a result of the following function calls? Be specific.

- `recurse(str, 0);`
- `recurse(str, 1);`
- `recurse(str, str.length());`
4. Matrices (10 points)

Consider a Matrix of integers representing altitudes (as in a topographical map). For example,

\[
\begin{bmatrix}
1000 & 1300 & 1100 & 1000 & 1100 \\
500 & 600 & 1200 & 900 & 1000 \\
700 & 800 & 800 & 1000 & 1100 \\
800 & 600 & 1000 & 900 & 1000
\end{bmatrix}
\]

In this map, the location specified by row 0, column 0 has an altitude of 1000 feet, the location specified by row 0, column 1 has an altitude of 1300 feet, and so on.

You are to complete the definition of the function `PrintPeaks` which has one parameter, a Matrix of integers representing a map, and which prints out all of the "peaks" in the map. A peak is defined to be a location that is at least as high as its four neighbors (up, down, left, and right). To make the problem easier, you do not need to check for peaks on the map border (i.e., only print peaks that occur in the interior of the map).

For example, given the above map, your function should output:

peak at (1,2) -- altitude 1200
peak at (2,1) -- altitude 800

```cpp
void PrintPeaks(const Matrix<int> & map)
// postcondition: prints all peaks in map (interior only)
```

Recall the basic operations defined on a Matrix:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows()</td>
<td>returns number of rows in Matrix</td>
</tr>
<tr>
<td>Cols()</td>
<td>returns number of columns in Matrix</td>
</tr>
<tr>
<td>SetSize(int r, int c)</td>
<td>resizes Matrix to r rows and c columns</td>
</tr>
<tr>
<td>[] operator</td>
<td>accesses Matrix element (a Vector)</td>
</tr>
<tr>
<td>= operator</td>
<td>performs assignment</td>
</tr>
</tbody>
</table>
This two-part problem deals with linked lists of words, implemented using the `WordNode` struct below.

```c
struct WordNode {
    string word;
    WordNode * next;
};
```

A. Complete the function `RemoveAfter` whose header is given below. This function should remove the node following the one pointed to by `ptr` (if there is such a node). No memory leakage should occur. For example,

```c
void RemoveAfter(WordNode * ptr)
// precondition: ptr points to node in linked list (or NULL)
// postcondition: removes node after the one pointed to by ptr
```
B. Complete the function `RemoveSortedDupes` whose header is given below. This function should remove any duplicate words from the linked list pointed to by `ptr`. Assume that the list is in sorted order, so any duplicates in the list will be adjacent. Again, no memory leakage should occur. For example,

before `RemoveSortedDupes(ptr);`

![Diagram before RemoveSortedDupes](image)

after `RemoveSortedDupes(ptr);`

![Diagram after RemoveSortedDupes](image)

Note: assume that the function `RemoveAfter`, as described earlier, is available for use here.

```c
void RemoveSortedDupes(WordNode * ptr)
// precondition : ptr points to a sorted linked list (or NULL)
// postcondition: removes all duplicates from the list
```
Test 3 Key

1. TRUE
   TRUE
   TRUE
   FALSE
   TRUE
   FALSE

2. A. Although we think of an array as a sequence of elements, C++ thinks of an array as a pointer to the first element. The [] operator accesses an array element by adding the specified index to the array pointer, and then dereferencing. When an array is passed by-value, a copy of the array (the pointer) is made as with any other type. However, dereferencing the copy of the pointer via the [] operator provides access to the original array elements, and so changes can be made.

   B. Quick sort is a recursive algorithm for sorting a list. The first step is to pick a particular element of the list as the pivot. Then, the list is partitioned into two parts: those elements ≤ the pivot and those > the pivot. Each of these partitions is then sorted recursively, resulting in a sorted list. If the pivot partitions the list (roughly) in half, then the number of recursions is logarithmic, resulting in $O(n \log n)$ cost. If, however, the pivot leads to unbalanced partitions, then a linear number of recursions may be necessary (e.g., if pivot is always the smallest element). Thus, the cost is $O(n^2)$.

   C. The common purpose of a destructor is to avoid memory leakage by deallocating memory when the class object is destroyed. Since the memory for statically allocated data fields is automatically reclaimed by the compiler, destructors only became necessary once we started performing dynamic memory allocation. Since memory allocated via new persists until explicitly deleted, a destructor is needed to take care of this memory management.

3. 0 1 3
   0 2
   a 1
   a
   ab
   abc
4. void PrintPeaks(const Matrix<int> & map)
   // postcondition: prints all peaks in the map (interior only)
   {
      int row, col;
      for (row = 1; row < map.Rows()-1; row++) {
         for (col = 1; col < map.Cols()-1; col++) {
            if (map[row][col] >= map[row-1][col] &&
                map[row][col] >= map[row][col-1] &&
                map[row][col] >= map[row][col+1] &&
                map[row][col] >= map[row+1][col] ) {
               cout << "peak at (" << row << ", " << col << ") -- altitude " << map[row][col] << endl;
            }
         }
      }
   }

5. void RemoveAfter(WordNode * ptr)
   // precondition: ptr points to node in linked list (or NULL)
   // postcondition: removes node after one pointed to by ptr
   {
      if (ptr != NULL && ptr->next != NULL) {
         WordNode * temp = ptr->next;
         ptr->next = temp->next;
         delete temp;
      }
   }

void RemoveSortedDupes(WordNode * ptr)
   // precondition: ptr points to sorted linked list (or NULL)
   // postcondition: removes all duplicates from the list
   {
      while (ptr != NULL && ptr->next != NULL) {
         if (ptr->word == ptr->next->word) {
            RemoveAfter(ptr);
         } else {
            ptr = ptr->next;
         }
      }
   }