3RD EDITION

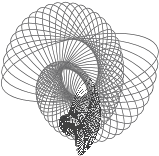
COMPUTING

FUNDAMENTALS

WITH C++

Rick Mercer

University of Arizona

Franklin, Beedle & Associates Inc.

Portland, Oregon

1/800-322-2665 https://fbeedle.com

©

*Th is textbook is dedicated to my publisher and my friend, Jim Leisy (1950–2014).*

*—R.M.*

Publisher Tom Sumner (tsumner@fbeedle.com)

Cover Photography Jim Leisy

Production Editor Jaron Ayres

Text Editor Brenda Jones

©2018 Franklin, Beedle & Associates Incorporated. No part of this book may be reproduced, stored in a retrieval system, transmitted, or transcribed, in any form or by any means—electronic, mechanical, telepathic, photocopying, recording, or otherwise—without prior written permission of the publisher. Requests for permission should be addressed as follows:

Rights and Permissions

Franklin, Beedle & Associates Incorporated

2154 NE Broadway, Suite 100

Portland, OR 97232

# **Library of Congress Cataloging-in-Publication data**

Names: Mercer, Rick.

Title: Computing fundamentals with C++ / Rick Mercer, University of Arizona.

Description: Portland, OR : Franklin, Beedle & Associates Inc., [2017]

Identifi ers: LCCN 2016027056 (print) | LCCN 2016026262 (ebook) | ISBN

9781590282762 | ISBN 9781590282793 (ebook)

Subjects: LCSH: C++ (Computer program language) | Computer programming.

Classifi cation: LCC QA76.73.C153 M46 2017 (ebook) | LCC QA76.73.C153 (print)

| DDC 005.13/3--dc23

LC record available at https://lccn.loc.gov/2016027056

## P R E F A C E

*Computing Fundamentals with C++, 3rd Edition* is written for students in a first course in a Computer Science curriculum using the C++ programming language. It is appropriate for students with no programming experience as well as those with programming experience in another language.

*Computing Fundamentals with C++, 3rd Edition* emphasizes computing fundamentals while recognizing the relevance and validity of object-oriented programming. Th is book is the result of decades of reasoning about how best to facilitate student learning in the fi rst course of the computer science curriculum, how best to integrate objects and classes into it, and how best to prepare students for the next course.

# **FEATURES**

**Book Resources.** Much of the C++ code from this textbook and chapter outlines written as presentations can be found at https://www2.cs.arizona.edu/people/mercer/compfun3/.

**Traditional Topics.** Th is textbook recognizes the relevance and validity of object-oriented programming while emphasizing traditional computing fundamentals. It also presents some C++ features that could well become traditional topics, such as templates for generic classes and standard containers with iterators, during the fi rst two courses.

**Standard C++.** Because the International Standards Organization (ISO) approved the C++ standard document many years ago, students can now study C++ as a language that has an internationally accepted standard. At the time of this writing, not all compilers were completely C++14 compliant. Because of this and the fact that C++14 only added a few things that are beyond the scope of this text book, this textbook only uses elements up through C++11. However, because any newer release would be backwards compatible, you may certainly use C++14 compliant compiler or any newer release.

**Gentle Objects-Early Approach.** Th is third edition maintains the objects-early approach of the fi rst and second editions. Students begin by using existing objects such as string, cin, cout, BankAccount, and Grid for Karel-like (Rich Pattis) programming while honing problem-solving

xiii

Preface

and program-development skills. Students will then modify, enhance, and ultimately design and implement their own classes of increasing complexity.

**Carefully Chosen Subset of Analysis, Design, and C++.** Because students using this textbook might have little or no programming or design experience, several C++ features and subtleties are not presented. Students concentrate on a solid subset of this feature-rich language. Some trickier topics are delayed until the later chapters. For example, nested loops with vectors of vectors, pointers, dynamic memory management, and the singly linked data structure are in the fi nal two chapters. Advanced topics such as copy constructors, destructors, and operator overloading have been removed from this third edition.

**Not Tied to a Specifi c System.** Th ere is no bias toward a particular operating system or compiler. Th is textbook presents standard #includes and namespaces according to the ISO standard. All material applies to any computer system using standard C++. All code has been tested with Microsoft Visual C++ on Windows, and GNU g++ on Unix.

**Algorithmic Patterns.** Algorithmic patterns help beginning programmers design algorithms around a set of common algorithm generalities. Th e fi rst algorithmic pattern, and perhaps one of the oldest—Input/Process/Output (IPO)—is introduced in Chapter 1. It is reused in subsequent chapters. Th e IPO pattern is especially useful to students with no programming experience and to the lab assistants helping them. Other algorithmic patterns introduced in the appropriate places include Alternative Action and Indeterminate Loop.

**Extensively Tested in the Classroom and Lab.** Th is textbook was six years in the making. Students supplied many useful comments and suggestions concerning manuscript clarity, organization, projects, and examples. Th e tremendous personal contact and testability was made possible with closed lab sections for all students.

# **PEDAGOGY**

Th is textbook has many pedagogical features that make this introduction to programming, design, and object technology accessible to students.

**Self-Check Questions.** Th ese short questions and answers allow students to evaluate whether they understand the details and terms presented in the reading. Th e answers to all self-check questions are included at the end of this textbook.

**Exercises.** Th ese transitional problems examine the major concepts presented in the chapter. Answers are available only to instructors to encourage students to write down the answers with paper and pencil, as if they were practice test questions.

Preface

**Programming Tips.** Each set of weekly programming projects is preceded by a set of programming tips intended to help students complete programs while warning of potential pitfalls, and promoting good programming habits.

**Programming Projects.** Many relatively small-scale problems have been extensively lab tested to ensure that projects can be assigned and completed with little or no instructor intervention. Th e programming projects are strategically positioned to occur every week of lecture to reinforce the concepts just presented.

**WHAT’S NEW IN THIS THIRD EDITION?**

Th is third edition contains dramatic improvements in programming projects, making them more interesting and challenging. Included are projects I have developed and class-tested at the University of Arizona, and, in fact, students have rated these “outside” assignments very highly. Th is third edition is smaller. We removed chapters on inheritance, object-oriented programming and design, operator overloading, and recursion. Adopters weren’t using these chapters and we felt the second edition was too big at 830 pages. Th e topics in this edition refl ect everything used in a traditional CS1 course while adding a few topics from CS2, such as generic collections with templates.

Th is third edition has been updated to make sure everything works using the current standard, C++14. Th is edition also incorporates a few appropriate additions to C++, such as the longoverdue literal nullptr. Most of the major additions to the C++ programming language, such as threads, are beyond the scope of this textbook.

# **ACKNOWLEDGMENTS**

Critical feedback from students and other instructors is essential to creating a solid textbook. For the fi rst and second editions, I was fortunate enough to have small lecture sizes (20 to 35 students) and to be in all labs with all of my students for 10 years. Th is enabled me to keep track of their progress and of their problems, which has dramatically helped produce a textbook that is accessible to the intended audience. I acknowledge and thank those Penn State students. I have been fortunate to encounter many excellent educators and industry people who care and think about the same issues. Th e debates and new ideas generated in discussions, both live and by email, have allowed me to make the plethora of informed decisions necessary for producing a high-quality textbook. I wish to acknowledge the following people (listed in reverse alphabetical order) with apologies to those whom I have unintentionally left out: Gene Wallingford, Doug Van Weiren, David Teague, Marty Stepp, Dave Richards, Stuart Reges, Margaret Reek,

Ken Reek, Rich Pattis, Allison Obourn, Linda Northrop, Zung Nguyen, John McCormick,

Carolina McCluskey, Lester McCann, Mary Lynn Manns, Mike Lutz, David Levine, Patrick Homer, Jim Heliotis, Peter Grogono, Adele Goldberg, Michael Feldman, Ed Epp, Robert Duvall

xv

Preface

(the lecturer from Duke, not the actor), Ward Cunningham, Alistair Cockburn, Mike Clancy, Tim Budd, Barbara Boucher-Owens, Mike Berman, Joe Bergin, Owen Astrachan, and Erzebet Angster.

Th ough too numerous to mention, I also acknowledge the many authors and presenters who have infl uenced me during my thirty-year career. In addition, my thanks go to the people at Franklin, Beedle & Associates: the late Jim Leisy, Jaron Ayres, Brenda Jones, and Tom Sumner.

# **REVIEWERS**

Reviewers spend many hours poring over material with critical eyes and useful comments. Because of the high quality of their work, criticisms and recommendations were always considered seriously. I thank the reviewers of this and previous editions.

|  |  |
| --- | --- |
| Kristin Roberts | *Grand Rapids Community College* |
| Rich Pattis | *UC Irvine* |
| Michael Berman | *Rowan University* |
| Seth Bergman | *Rowan University* |
| Robert Duvall | *Duke University* |
| Tom Bricker | *University of Wisconsin, Madison* |
| David Teague | *Western Carolina University* |
| Ed Epp | *University of Portland* |
| James Murphy | *California State University, Chico* |
| Jerry Weltman | *Louisiana State University, Baton Rouge* |
| John Miller | *St. John’s University* |
| Stephen Leach | *Florida State University* |
| Alva Th ompson | *University of South Florida* |
| Norman Jacobson | *University of California, Irvine* |
| David Levine | *Gettysburg College* |
| H. E. Dunsmore | *Purdue University* |
| Howard Pyron | *University of Missouri at Rolla* |
| Lee Cornell | *Mankato State University* |
| Eugene Wallingford | *University of Northern Iowa* |
| David Teague | *Western Carolina University* |
| Clayton Lewis | *University of Colorado* |
| Tim Budd | *Oregon State University* |
| Jim Miller | *University of Kansas* |
| Art Farley | *University of Oregon* |
| Richard Enbody | *Michigan State University* |
| Van Howbert | *Colorado State University* |
| Joe Burgin | *Texas Tech University* |

Preface

Jim Coplien *Bell Labs*

Dick Weide *Ohio State University*

Gene Norris *George Mason University*

# **SPECIAL THANKS**

I would like to thank super-reviewer Kristin Roberts from Grand Rapids Community College. Not only did she provide tremendous feedback, Kristin encouraged me to complete this third edition. Th ere have been several major changes, some reorganization of chapters, many refi nements, some updates, and many new programming projects. Th is textbook is now current, in large part because of Kristin.

xvii

© 2018 Franklin, Beedle & Associates All rights reserved

**C H A P T E R O N E**

# Problem Solving with C++

## COMING UP

We begin with a need for a computer-based solution to a problem. Th e need may be expressed in one or two paragraphs as a problem specifi cation. Th e progression from understanding a problem specifi cation to achieving a working computer-based implementation is known as problem solving. After studying this chapter, you will understand

* one example of problem solving
* the characteristics of an algorithm
* how algorithmic patterns help in program design
* the relationship between a class and its many objects that objects have a name, state, and set of operations
* the categories of errors that occur during the implementation phase of software development

### 1.1 PROBLEM SOLVING

Th ere are many approaches to problem solving. Th is chapter begins by examining a strategy with these three steps: analysis, design, and implementation.

#### Phase Activity

**Analysis:** Understand the problem.

**Design:** Design an algorithm that outlines a solution. **Implementation:** Code an executable program.

Our study of computing fundamentals begins with an example of this particular approach to problem solving. Each of these three phases will be exemplifi ed with a case study of one particular problem: computing a course grade.

1

©

#### 1.1.1 ANALYSIS (INQUIRY, EXAMINATION, STUDY)

Program development may begin with a study, or analysis, of a problem. Obviously, to determine what a program is to do, you must fi rst understand the problem. If the problem is written down, you can begin the analysis phase by simply reading the problem.

While analyzing the problem, it is helpful to name the data that represents the information. For example, you might be asked to compute the maximum weight allowed for a successful liftoff of a particular airplane from a given runway under certain thrust-aff ecting weather conditions such as temperature and wind direction. While analyzing the problem specifi cation, you might name the desired information maximumWeight. Th e data required to compute that information could have names such as temperature and windDirection.

Although such data do not represent the entire solution, they do represent an important piece of the problem. Th e data names are symbols for what the program will need and what the program will compute. One value needed to compute maximumWeight might be 19.0 for temperature. Such data values must often be manipulated—or processed—in a variety of ways to produce the desired result. Some values must be obtained from the user, other values must be multiplied or added, and still other values must be displayed on the computer screen*.* At some point, these data values will be stored in computer memory. Th e values in the same memory location can change while the program is running. Th e values also have a type, such as integers, numbers with decimal points, strings of characters, or types that store several diff erent types of values. Th ese named pieces of memory that store values that change while the program is running are known as *variables*.

You will see that there also are operations for manipulating those values in meaningful ways. It helps to distinguish the data that must be displayed—*output*—from the data required to compute that result—*input*. Th ese variables summarize what the program must do.

**Input:** Information a user must supply to solve a problem.

**Output:** Information the computer must display.

A problem can be better understood by answering this question: What would the output be given certain input? Th erefore, it is a good idea to provide an example of the problem. Here are two problems with variable names selected to accurately describe the stored values:

|  |  |  |
| --- | --- | --- |
| **Problem** | **Variable Input or Output** | **Sample Problem** |
| Compute a monthly | amount Input | 12500.00 |
| loan payment | rate Input | 0.08 |
|  | months Input | 48 |
|  | payment Output | 303.14 |

|  |  |  |
| --- | --- | --- |
| **Problem** | **Variable Input or Output** | **Sample Problem** |
| Count how often | theWork Input | Much Ado About Nothing |
| Shakespeare wrote | theWord Input | thee |
| a particular word in a particular play | howOften Output | 74 |

In summary, problems can be analyzed by:

1. Reading and understanding the problem specifi cation.
2. Deciding what data represent the answer—the output.
3. Deciding what data the user must enter to get the answer—the input.
4. Creating some sample problems (like those above) that summarize.

Textbook problems sometimes provide the variable names and types of values such as strings, integers, or numbers with a decimal point that must be input and output. If not, they are relatively easy to recognize. In real-world problems of signifi cant scale, a great deal of eff ort is expended in the analysis stage.

#### SELF-CHECK

1-1 Given the problem of converting British pounds to U.S. dollars, provide a meaningful name for a variable to store the value that must be input by the user. Also give a meaningful name for the variable to store a value that is to be output.

1-2 Given the problem of selecting one CD from a 200-CD player, what variable name would represent all of the CDs? What name would be appropriate to represent one CD selected by the user?

##### An Example of Analysis

*Problem:* Using the grade assessment scale to the right, com- **Item Weight** pute a course grade as a weighted average of projects, a mid- Projects 50%

term, and one fi nal exam. Midterm 20%

Final Exam 30%

Analysis begins by reading the problem specifi cation and establishing the desired output and the required input to solve the problem. Determining and naming the output is a good place to start. Th e output stores the answer to the problem. It provides insight into what the program must do.

Once the need for a data value is discovered and given a meaningful name, the focus can shift to what must be accomplished. For this particular problem, the desired output is the actual course grade. Th e name courseGrade represents the requested information to be output to the user. Th is problem becomes more generalized when the user enters values to produce the result. If the program asks the user for data, the program can be used later to compute course grades for many students with any set of grades. So let’s decide on and create names for the values that must be input. To determine courseGrade, three values are required: projects, midterm, and nalExam. Th e fi rst three analysis activities are now complete:

1. Problem understood.
2. Information to be output: courseGrade.
3. Data to be input: projects, midterm, and nalExam.

It helps to have a sample problem, a test case. Th is involves having the input values and the expected output result. For example, when projects is 74.0, midterm is 79.0, and nalExam is 84.0, the weighted average should be 78.0:

1. 50 × projects) + (0.20 × midterm) + (0.30 × nalExam)

(0.5 × 74.0) + (0.2 × 79.0) + (0.30 × 84.0)

37.0 + 15.8 + 25.2

78.0

Th e problem has now been analyzed, the input and output variables have been identifi ed, it is understood what the computer-based solution is to do, and one test case exists.

|  |  |  |  |
| --- | --- | --- | --- |
| **Problem** | **Variable** | **Input or Output** | **Test Case** |
| Compute a course grade | projects | Input | 74.0 |
|  | midterm | Input | 79.0 |
|  | nalExam | Input | 84.0 |
|  | courseGrade | Output | 78.0 |

#### SELF-CHECK

1-3 Complete an analysis for the following problem. You will need a calculator to determine output.

*Problem:* Show the future value of an investment given its present value, the number of periods (years, perhaps), and the interest rate. Be consistent with the interest rate and the number of periods; if the periods are in years, then the annual interest rate must be supplied (0.085 for 8.5%, for example). If the period is in months, the monthly interest rate must be supplied (0.0075 per month for 9% per year, for example). Th e formula to compute the future value of money:

future value = present value \* (1 + rate)periods

##### 1.1.2 DESIGN (MODEL, THINK, PLAN, DEVISE, PATTERN, OUTLINE)

*Design* refers to the set of activities that includes specifying an algorithm for each program component. An *algorithm* is a step-by-step procedure for solving a problem or accomplishing some end, especially by a computer. A good algorithm must

* list the activities that need to be carried out
* list those activities in the proper order

Consider an algorithm to bake a cake taken from directions for carrot cake:

* + Preheat oven to 350° F.
  + Grease sides and bottom of each pan.
  + Blend ingredients in a large bowl.
  + Pour batter into the pan and bake immediately. For cupcakes, fi ll to 2/3 full.
  + Bake following the chart below.
  + Cake is done when toothpick inserted in center comes out clean.

If the order of the steps is changed, the cook might get a very hot cake pan with raw cake batter in it. If one of these steps is omitted, the cook probably won’t get a baked cake—or there might be a fi re. An experienced cook may not need such an algorithm. However, cake-mix marketers cannot and do not presume that their customers have this experience. Good algorithms list the proper steps in the proper order and are detailed enough to accomplish the task.

#### SELF-CHECK

1-4 Cake recipes typically omit a very important activity. Describe an activity that is missing from the algorithm above.

An algorithm often contains a step without much detail. For example, “Blend ingredients in a large bowl” isn’t very specifi c. What are the ingredients? If the problem is to write a recipe algorithm that humans can understand, this step could be refi ned a bit to instruct the cook on how to blend the ingredients. Th e refi nement to this step could be “Empty the cake mix into the bowl and mix in the milk until smooth,” or for scratch bakers:

* Sift the dry ingredients.
* Place the liquid ingredients in the bowl.
* Add the dry ingredients a quarter-cup at a time, whipping until smooth.

Algorithms may be expressed in *pseudocode*—instructions expressed in a language that even nonprogrammers understand. Pseudocode is written for humans, not for computers. Pseudocode algorithms are an aid to program design.

Pseudocode is very expressive. One pseudocode instruction may represent many computer instructions. Pseudocode algorithms are not concerned about issues such as misplaced punctuation marks or the details of a particular computer system. Pseudocode solutions make design easier by allowing details to be deferred. Writing an algorithm can be viewed as planning. A program developer can design with pencil and paper and sometimes in their head.

##### 1.1.3 ALGORITHMIC PATTERNS

Problems often require input from the user in order to compute and display the desired information. Th is particular fl ow of activities—input/process/output—occurs so often, in fact, that it can be viewed as a pattern. It is one of several algorithmic patterns you will fi nd helpful in the design of programs.

A *pattern* is anything shaped or designed to serve as a model or a guide in making something else. An algorithmic pattern serves as a guide to help solve problems. For instance, the following Input/Process/Output (IPO) algorithmic pattern can be used to help design our fi rst problem. In fact, this IPO pattern can be used to help design almost all of the programs in the fi rst fi ve chapters of this textbook.

###### Algorithmic Pattern *Input/Process/Output*

|  |  |
| --- | --- |
| Pattern: | Input/Process/Output (IPO) |
| Problem: | Th e program requires input from the user in order to compute and display the |
|  | desired information. |
| Outline: | 1. Obtain the input data. |
|  | 2. Process the data in some meaningful way. |
|  | 3. Output the results. |

Code Example:

int n1, n2, n3;

oat average;

// Input

cout << "Enter three numbers: "; cin >> n1 >> n2 >> n3;

// Process

average = (n1 + n2 + n3) / 3.0;

// Output cout << "Average = " << average;

Th is algorithmic pattern is the fi rst of several. In subsequent chapters, you’ll see other algorithmic patterns such as Guarded Action, Alternative Action, and Indeterminate Loop. To use an algorithmic pattern eff ectively, you should fi rst become familiar with it. Register this Input/Process/Output algorithmic pattern and look for this pattern while developing programs. Th is allows you to design programs more easily. For example, if you discover you have no meaningful values for the input data, it may be that you have placed the process step *before* the input step. Or you may have skipped the input step altogether.

Patterns help solve other kinds of problems. Consider this quote from Christopher Alexander’s book, *A Pattern Language* [Alexander 77]:

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Even though Alexander was describing patterns in the design of furniture, gardens, buildings, and towns, his description of a pattern can also be applied to computational problem solving. Th e IPO pattern frequently pops up during program design. It guides the solution to many problems.

##### 1.1.4 AN EXAMPLE OF ALGORITHM DESIGN

Th e Input/Process/Output pattern guides the design of the algorithm that relates to our course grade problem.

|  |  |
| --- | --- |
| **Th ree-Step Pattern** | **Pattern Applied to a Specifi c Algorithm** |
| 1. Input | 1. Read in projects, midterm, and nalExam |
| 2. Process | 2. Compute courseGrade |
| 3. Output | 3. Display courseGrade |

Although algorithm development is usually an iterative process, a pattern provides an outline of the activities necessary to solve this problem.

#### SELF-CHECK

1-5 Read the three activities of the algorithm above. Do you detect a missing activity?

1-6 Read the three activities of the algorithm above. Do you detect any activity out of order?

1-7 Would this previous algorithm work if the fi rst two activities were switched?

1-8 Is the algorithm detailed enough to compute courseGrade?

Th ere currently is not enough detail in the process step of the course grade problem. Th e algorithm needs further refi nement. Specifi cally, exactly how should the input data be processed to compute the course grade? Th e algorithm omits the weighted scale described in the problem specifi cation. Th e process step should be refi ned a bit more as shown in step 2:

1. Obtain projects, midterm, and nalExam from the user
2. Compute courseGrade = (50% projects) + (20% midterm) + (30% nalExam)
3. Display the value of courseGrade

It has been said that good artists know when to put down the brush. Deciding when a painting is done is critical for its success. By analogy, a designer must decide when to stop designing.

Th is is a good time to move on to the third phase of problem solving, which is implementation.

In summary, here is what has been accomplished so far:

Th e problem is understood

Variables have been identifi ed

Expected output of a sample problem is known (78.0%)

An algorithm has been developed

##### 1.1.5 IMPLEMENTATION (FULFILLMENT, OPERATION, USAGE)

A computer is a programmable electronic device that can store, retrieve, and process data. Programmers can simulate an electronic version of an algorithm by following the algorithm and manually performing the activities of storing, retrieving, and processing data using pencil and paper. Th e following algorithm walkthrough is a human (non-electronic) execution of the algorithm:

* 1. Retrieve some example values from the user and store them as shown:

projects = 80 midterm = 90

nalExam = 100

* 1. Retrieve the values and compute courseGrade as follows:

courseGrade = (0.5 × projects) + (0.2 × midterm) + (0.3 × nalExam) (0.5 × 80.0) + (0.2 × 90.0) + (0.3 × 100.0)

40.0 + 18.0 + 30.0 courseGrade = 88.0

3. Display the value stored in courseGrade to show 88%

##### 1.1.6 A C++ PROGRAM

Th e following complete C++ program previews many programming language details presented in the next chapter. You are not expected to understand this C++ source code. For now, just peruse the source code as an implementation of the pseudocode algorithm. Th e three variables—projects, midterm, and nalExam—represent user input. Th e output variable is named courseGrade. Th e object cout, pronounced "see-out," stands for *common output* and allows a program to generate output. Input is made possible using the object cin, pronounced "see-in," which stands for *common input*.

/\*

* This program computes and displays a nal course grade as a \* weighted average after the user enters the appropriate input.

\*

* File name: CourseGrade.cpp

\*/

#include <iostream> // for cin and cout

#include <string> // for string

using namespace std; // avoid writing std::cin std::cout std::string

int main() {

// Explain what this program does. cout << "This program computes a weighted course grade." << endl;

// Read in a string

cout << "Enter the student's name: "; string name; cin >> name;

// I)nput projects, midterm, and nalExam double projects, midterm, nalExam;

cout << "Enter project score: "; cin >> projects;

cout << "Enter midterm: "; cin >> midterm;

cout << "Enter nal exam: "; cin >> nalExam;

// P)rocess

double courseGrade = (0.5 \* projects) + (0.2 \* midterm) + (0.3 \* nalExam);

// O)utput the results

cout << name << "'s grade: " << courseGrade << "%" << endl; }

###### Dialogue

This program computes a weighted course grade.

Enter the student's name: ***Dakota***

Enter project score: ***80***

Enter midterm: ***90***

Enter nal exam: ***100***  Dakota's grade: 88%

##### 1.1.7 TESTING

Th e important process of testing may, can, and should occur at any phase of problem solving. Th e actual work can be minimal, and it’s worth the eff ort. However, you may not agree until you have experienced the problems incurred by *not* testing. Testing activities can occur during all phases of program development:

During analysis, establish test case problems to confi rm your understanding of the problem.

During design, walk through the algorithm to ensure that it has the proper steps in the proper order.

During testing, run the program several times with diff erent sets of input data. Confi rm that the results are correct.

Review the problem specifi cation. Does the running program do what was requested?

You should have one or more test case problems before the program is coded—not after. Determine the input values and what you expect for output. Using 80, 90, and 100 as input and expecting the output to be 88% is one such test case. When the program fi nally does generate output, the expected result can then be compared to the output of the running program. Adjustments must be made any time the predicted output does not match the program output. Such a confl ict indicates that the problem example, the program output, or perhaps both are incorrect. Testing with several test cases helps avoid the misconception that a program is correct just because the program runs successfully and generates output. Th e output could be wrong! Simply executing a program does not make that program correct. Test cases provide confi dence that the program does work.

However, even exhaustive testing does not prove a program is correct. E. W. Dijkstra has argued that testing only reveals the presence of errors, not the absence of errors. Even with correct program output, the program is not proven correct. Testing reduces errors and increases confi dence that the program works correctly.

##### SELF-CHECK

1-9 If the programmer predicts courseGrade should be 100.0 when all three inputs are 100.0 and the program displays courseGrade as 75.0, what is wrong: the predicted output, the program, or both?

1-10 If the programmer predicts courseGrade should be 90.0 when projects is 80.0, midterm is 90.0, and nalExam is 100.0 and the program outputs courseGrade as 88, what is wrong: the prediction, the program, or both?

1.2: Objects, Types, and Variables

1-11 If the programmer predicts courseGrade should be 88 when projects is 80.0, midterm is

90.0, and nalExam is 100.0 and the program outputs courseGrade as 90.0, what is wrong: the prediction, the program, or both?

### 1.2 OBJECTS, TYPES, AND VARIABLES

To input something that can be used by a program, there must be a place to store it in the memory of the computer. Bjarne Stroustrup, the creator of C++, writes

We call such a “place” an object. An object is a region of memory with a type that specifi es what kind of information can be placed in it. A named object is called a variable. For example, character strings are put into string variables and integers are put into int variables. You can think of an object as a “box” into which you can put a value of the object’s type.

For example, the int type used in the previous program stores whole numbers, or integers. Some operations on int variables include addition, subtraction, multiplication, and division. Note that C++ uses \* for the multiplication operator (using x would be confusing).

double courseGrade = 0.5\*projects + 0.2\*midterm + 0.5\* nalExam;

Th e oat and double types (double is twice the size of a oat) store numeric values with a fractional part. Th e C++ string type stores character sequences as "Firstname I. Lastname" along with an integer to maintain the number of characters in that string.

*Objects* are entities stored in computer memory. An object is understood by the types of values the object stores—its *attributes*—and the operations that can be applied to that object—its *behavior* [Booch]. Every object has

a name to store and retrieve the values of that object

values stored in computer memory, which is known as the object’s state

a set of operations such as addition, input, output, assignment

Th ese three characteristics of objects—name, state, and operations—were all illustrated in the course grade program. It used three numeric objects stored as projects, midterm, and nalExam that were read in from the keyboard. Each of these objects is capable of storing the value of an integer such as 79 or 90. Th ese objects, along with available operations such as input, multiplication, and addition, computed the courseGrade. An assignment operation was used to store it as a numeric object. An output operation with cout << was used so the user could see the results of the processing.

#### Characteristics of Objects in the First Program

Name: Each of the four numeric objects has its own identity because each has its own name. Th e fi rst of the four numeric objects was stored as the variable named projects.

State: Th e value of projects was set with an input operation using cin >>. Th e state of courseGrade was defi ned with an assignment operation using the = operator.

Th e state of courseGrade was retrieved during output with cout.

Operations: Other operations available for the int objects included addition (+) and multiplication (\*).

C++ has fundamental types and compound types. Th e *fundamental* types store one value that corresponds directly to hardware with a fixed size. Th e type determines what values can be stored into that object and what operations can be performed. With numeric types such as int and double, the number of bytes, which varies for diff erent computers, determines the range of values that can be stored in it.

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Size** | **Typical Range of Values (varies)** |
| short | 2 bytes (16 bits) | -32768 to 32767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| int | 4 bytes | -2147483648 to 2147483647 |
| unsigned int | 4 bytes | 0 to 4294967295 |
| unsigned long | 8 bytes | 0 to 18446744073709551615 |
| fl oat | 4 bytes | 3.4E +/- 38 (7 digits) |
| double | 8 bytes | 1.7E +/- 308 (15 digits) |
| char | 1 byte | 0 to 255 |
| bool | 1 byte | true or false |

A *compound* type is a type that is defi ned in terms of another type. C++ has these compound types presented in this textbook: references, functions, classes, arrays, and pointers. For example, string is a reference type made up of characters and other data. It has operations to fi nd the length of the sequence of characters and another operation to create a substring from a string given the starting and ending indexes (there are many more operations that will be explored in a later chapter):

string aString = "A sequence of characters"; // Output: cout << aString.length() << endl; // 24 cout << aString.substr(2, 8) << endl; // sequence Chapter Summary

In addition to the string type, two other reference types are used immediately. Th e istream type object named cin has operations to read from an input source such as the keyboard or a fi le on disk. Th e ostream type object named cout helps generate output.

#### SELF-CHECK

1-12 Describe the values stored in objects of type double.

1-13 Name two operations for double objects.

1-14 Describe the values stored in objects of the int type.

1-15 Name two operations for int objects.

1-16 Describe the values stored in string objects.

1-17 Which of the types above store precisely one value?

### CHAPTER SUMMARY

Th is chapter presented a three-step problem-solving strategy of analysis, design, and implementation. Th e table below shows some of the activities performed during each of these three phases. Th e maintenance phase has been added to show how the three steps fi t into the complete program life cycle. Th e maintenance phase actually requires the majority of the time, energy, and money of the program’s life cycle.

|  |  |
| --- | --- |
| **Phase** | **Activities You Might Perform** |
| Analysis | Read and understand the problem statement.  Determine the input and output objects.  Solve a few sample problems. |
| Design | Look for patterns to guide algorithm development.  Write an algorithm—steps needed to solve the problem.  Refi ne the steps in the algorithm and walk through it. |
| Implementation | Translate the design into a programming language. Fix errors.  Create an executable program.  Test the program. |
| Maintenance | Update the program to keep up with a changing world.  Enhance it.  Correct bugs as they are found. |

Some analysis and design tools were introduced:

naming the objects that help solve a problem

developing algorithms

refi ning one or more steps of an algorithm

using the Input/Process/Output pattern

Th e sample program we presented previews the details to be discussed in the next chapter.

C++ types were previewed: fundamental and compound.

Testing is important, but it does not prove the absence of errors. Testing can and does detect errors, but it can only build confi dence that the program appears to work.

### EXERCISES

1. What activities can be performed when analyzing problems?
2. What are the characteristics of a good algorithm?
3. What is the diff erence between objects used to store output values and objects that store the values input by the user?
4. List the three characteristics of objects.
5. What activities can be performed when designing programs?
6. What is one “deliverable” of design?
7. What type of object would you use to store the number of students registered in a course?
8. What type of object would you use to store π?
9. What type of object would you use to store the words of a Shakespeare play?
10. What is the deliverable from the implementation phase of program development?
11. Does a program that runs work correctly? Justify your answer.
12. Write an algorithm that describes how to get to where you live.
13. Write an algorithm for fi nding any phone number in the phone book. Will the search always be successful?
14. Write an algorithm that instructs someone to arrive at your home on foot.
15. Obtain the instructions necessary to create, compile, link, and execute a C++ program on your system. You may need to seek out a login procedure and/or basic editing commands and compiling commands. After this, write a complete algorithm that provides all necessary steps to successfully guide a novice to complete a program through testing. Your

Problem Solving: Writing Algorithms

algorithm may contain steps such as “Compare example output to program output,”

“Create a new fi le,” and “Compile the program.”

**PROBLEM SOLVING: WRITING ALGORITHMS**

**1A SIMPLE AVERAGE**

Write an algorithm that will compute the average of three test scores of equal weight.

## 1B WEIGHTED AVERAGE

Write an algorithm that will compute the course grade using this weighted scale:

|  |  |
| --- | --- |
| Assessment | Weight |
| Quiz average | 20% |
| Midterm | 20% |
| Lab grade | 35% |
| Final exam | 25% |

## 1C WHOLESALE COST

You happen to know that a store has a 25% markup on compact disc (CD) players. If the retail price (what you pay) of a CD player is $189.98, how much did the store pay for that item (the wholesale price)? In general, what is the wholesale price for any item given its retail price and markup? Analyze the problem and design an algorithm that computes the wholesale price for any given retail price and any given markup. Use this formula and a little algebra to solve for wholesale price: retail price = wholesale price \* (1 + markup).

## 1D TIME DIFFERENCES

Write an algorithm that takes two diff erent train departure times (where 0 is midnight, 0700 is 7:00 a.m., 1314 is 14 minutes past 1:00 p.m., and 2200 is 10 p.m.) and prints the diff erence between the two times in hours and minutes. Assume both times are on the same date and that both times are valid. For example, 1099 is not a valid time because the last two digits are minutes, which should be in the range of 00 through 59. 2401 is not valid because the hours (the fi rst two digits) must be in the range of 0 through 23 inclusive. For example, if train A departs at 1255 and train B departs at 1305, the diff erence would be 0 hours and 10 minutes.

© 2018 Franklin, Beedle & Associates All rights reserved

**C H A P T E R T W O**

# C++ Fundamentals

## SUMMING UP

Th e fi rst chapter introduced a real-world program development strategy of analysis, design, and implementation. You were encouraged to do some analysis and design before writing C++ code. However, many problems encountered in this textbook will not require much eff ort to analyze and design. Analysis may simply be “Read the problem.” Design might end up as “I can picture an algorithm in my head.”

## COMING UP

In this chapter, the emphasis will be on translating algorithms into programs using the C++ programming language. Th e resulting source code you type is the input to the compiler. Th e compiler translates the source code into machine code that your particular computer understands. However, the compiler expects source code to follow the precise rules of the programming language. Understanding how to translate a pseudocode algorithm into its programming language equivalent requires understanding the smallest pieces of a program and how to correctly gather them together to create statements. Th is chapter also examines operations that can be performed on many objects. After studying this chapter, you will be able to

* understand how to include existing source code in your programs
* obtain data from the user and display information to the user
* evaluate and create arithmetic expressions
* understand that these common operations are available to many objects such as initialization, input, assignment, and output
* solve problems using the C++ programming language

### 2.1 THE PIECES OF A C++ PROGRAM

A C++ program begins as a sequence of characters stored in a fi le. Th e name of the fi le holding a

C++ program typically ends with either .cc, .c, .cp, or .cpp ( rst.cc, rst.C, or rst.cpp, for example). Some programming environments require or assume certain fi le-naming conventions.

17

Th erefore, when you create a fi le to translate an algorithm into its C++ programming language equivalent, create the fi le with the extension you should—or must—use.

Th e text contained in the fi le is introduced as the *general form* of a C++ program (below). A general form describes the *syntax*—the correct language—necessary to write legal programming language constructs. Th is general form, like all others in this textbook, follows these conventions:

1. Elements written in *monospace* are used exactly as shown. Th is includes certain words such as int main*,* cout*,* cin*,* and symbols such as << >> ;.
2. Th e portions of a general form written in *italic* must be supplied by the programmer—for example, *expression* means you must supply a valid expression.
3. An item in italic is defi ned somewhere else.

**General Form 2.1** *Standard C++ Program*

// A comment

#*include-directives* using namespace std; int main() {  *statements*  return 0; }

Th e parts of a general form in boldface must be written exactly as shown. Th e statements part refers to a collection of diff erent statements. A statement is the smallest standalone element that expresses some action to be carried out. Semicolons terminate statements. A few statements are described in this chapter. Although not necessary with standard C++, the last statement in the C++ programs of this textbook will be return 0;. Th e curly braces { and } mark the extent of the main function. A function is a construct that allows all of the code to be treated as one entity. Before getting into the details, here is a syntactically correct standard C++ program. To run code as a program, it must have a function named main. (*Note:* std is an abbreviation for standard.)

// This program prompts for a name and prints a friendly message

#include <iostream> // For cout, cin, and endl

#include <string> // for the string type

using namespace std; // Allow programmers to write cin and cout

// rather than std::cin and std::cout int main() { string name;

cout << "What is your name: "; cin >> name; cout << "Hello " << name;

cout<< ", I hope you're feeling well." << endl;

return 0; }

#### Dialogue

What is your name: ***Casey***

Hello Casey, I hope you’re feeling well.

Th is source code represents input to the compiler. Th e compiler translates source code like this into machine code. Along the way, the compiler may generate error and warning messages. Th e errors are detected as the compiler scans the source code of the program and any #include fi les that represent additional source code. For example, the fi le named iostream precedes the code beginning at int main() and so the source code in the fi le becomes part of the program.

Th e #include directive is conceptually replaced by the text contained in the #included fi le. Every C++ program uses more than one fi le to take advantage of the code produced by other programmers. In fact, C++ compilers are delivered with a large number of fi les. Below is the general form that adds source code from other fi les to your program.

#### General Form 2.2 *Include Directive*

#include < *include- le* >

- or -

#include " *include- le* "

Th e #include and angle brackets < > or double quote marks " " must be written exactly as shown. Th e include-fi le is the name of an existing fi le. For example, the previous program contains the following #include directive in order to furnish cout, cin, and endl:

#include <iostream>

However, this #include directive actually provides std::cout, std::cin, and std::endl. Th e C++ standard library, of which iostream is a part, is defi ned in a namespace called std. So to avoid repetitiously writing std::, this line should accompany #include <iostream> and other #includes seen later:

using namespace std; // Can now write cout instead of std::cout

Care should be taken to avoid any blank spaces between the < > or " ".

#include <iostream > // ERROR, space at end

#include " BankAccount.h" // ERROR, space up front

Any included fi le with angle brackets < > must be part of the system. Your system should be able to fi nd those fi les automatically. However, the fi le names included within double quote marks " " may need to be stored in the same directory as the program that includes them.

#### 2.1.1 TOKENS: THE SMALLEST PIECES OF A PROGRAM

Before looking at the general forms for object initializations and statements, consider the smallest pieces of the programming language that make up the larger constructs. Th is should help you

* more easily code syntactically correct statements
* better understand how to fi x errors detected by the compiler
* understand general forms

As the C++ compiler reads the source code, it identifi es individual *tokens*. A token is the smallest recognizable component of a program. Tokens fall into four categories:

|  |  |
| --- | --- |
| **Category** | **Examples** |
| Special symbols | ; ( ) << >> |
| Keywords | return double int |
| Identifi ers | main test2 rstName |
| Literals | "Hello" 507 -2.1 true 'c' nullptr |

#### 2.1.2 SPECIAL SYMBOLS

A *special symbol* is a sequence of one or two characters with one or possibly many specifi c meanings. Some special symbols such as {, ;, and , separate other tokens. Other special symbols such as +, -, and << represent operators in expressions. Here is a partial list of single-character and double-character special symbols frequently seen in C++ programs:

( ) . + - / \* =< >= // { } == ; << >>

#### 2.1.3 IDENTIFIERS

*Identifi ers* are names given to a variety of things. Th ey all follow these rules that govern the creation of C++ identifi ers.

* Identifi ers begin with upper- or lowercase letters a through z or A through Z, dollar sign $, or the underscore character \_.
* Th e fi rst letter may be followed by a number of upper- and lowercase letters, digits (0 through 9), and underscore characters.
* Identifi ers are case-sensitive. For example, Ident, ident, and iDENT are three diff erent identifi ers.

##### Valid Identifi ers

main cin incomeTax i MAX\_SIZE Maine cout employeeName x all\_4\_one miSpelte string A1 n $motion$

##### Invalid Identifi ers

1A Begins with a digit miles/Hour / is unacceptable

rst Name Th e blank space is unacceptable pre-shrunk Th e - operator means subtraction

C++ comes with a large number of standard—must be part of the language—identifi ers. For example, cin is the name of an object used to obtain input from the keyboard. Another standard identifi er, cout, is the name of the object used to prompt generate output. Here are a few other standard C++ identifi ers. (*Note:* Th e fi rst identifi er, pronounced “end-ell,” is used for end line.)

endl sqrt fabs pow string vector width precision queue

Programmer-defi ned identifi ers have meaning for the programmer who created the program, for others who might later use it, and for those who must maintain the program. For example, test1, nalExam, and courseGrade are programmer-defi ned. When creating your own identifi ers, give them meaningful names that reveal their purposes.

C++ is case-sensitive, which means an uppercase letter is diff erent from the same letter in lowercase; “A” is not the same as “a.” For example, every complete program must include the identifi er main. MAIN or Main won’t do. Also note that several conventions may be used for upper and lowercase letters. Some programmers prefer avoiding uppercase letters; others prefer to use uppercase letters for each new word. Th e convention used in this textbook is the “camelBack” style where each word after the fi rst has an uppercase letter. For example, you will see letterGrade rather than lettergrade, LetterGrade, or letter\_grade. Diff erent programmers use diff erent styles.

#### 2.1.4 KEYWORDS

*Keywords* are identifi ers that have a specifi c purpose whose meaning is reserved by the standard language defi nition, such as the keywords double and int.

**C++ KEYWORDS**

break do for operator switch case double if return typedef char else int sizeof void class oat long struct while

Th e case sensitivity of C++ applies to keywords. For example, there is a diff erence between double (a keyword) and Double (not a keyword). C++ keywords are always in lowercase.

#### 2.1.5 COMMENTS

*Comments* are portions of text that annotate a program. Comments fulfi ll any or all of the following expectations:

* Provide internal documentation to help one programmer read another’s program—assuming those comments clarify the meaning of the program
* Explain certain code fragments or the purpose of an object
* Indicate the programmer’s name and the goal of the program
* Describe a wide variety of program elements and other considerations

Comments may be added anywhere throughout a program, including to the right of any C++ statement, on a separate line, or over several lines. Th ey may begin with the two-character special symbol /\* when closed with \*/.

/\*

A comment may extend over several lines \*/

An alternate form for comments is to use // before the text. Such a comment may appear on a line by itself or at the end of a line:

// A complete C++ program int main() {

return 0; // This program returns 0 to the operating system }

Within the context of the programs in this textbook, comments are most often written as one-line comments like // Comment rather than /\* Comment \*/. All code after /\* is a comment until \*/ is encountered, so a large portion of the program can accidentally be turned into a comment by forgetting \*/ at the end. Th e one-line comments make it more diffi cult to accidentally “comment out” large sections of code.

Comments are added to help clarify and document the purpose of the source code. Th e goal is to make the program more understandable, easier to debug (correct errors), and easier to maintain (change when necessary). Programmers need comments to understand programs that may have been written days, weeks, months, years, or even decades ago.

#### 2.1.6 C++ LITERALS

Th e C++ compiler recognizes string, integer, Boolean (true/false), and fl oating-point literals. A *string literal* is zero or more characters enclosed within double quotes and fi nished on the same line:

"Double quotes are used to delimit string constants."

"Hello, World!"

*Integer constants* are numbers without decimal points. *Floating-point constants* are written with decimal points or using exponential notation: 5e3 = 5 \* 103 = 5000.0 and 1.23e-4 = 1.23 \* 10-4 = 0.000123, for example. *Boolean literals* are true and false. Here are some examples of C++ literals and the C++ types used in this textbook to store those literal objects.

##### Type

**Example Literals**

|  |  |
| --- | --- |
| int | 0 1 999 -999 -2147483647 2147483647 |
| char | 'a' '#' '9' '\t' (tab) '\n' (new line) |
| double | 1.23 0.5 .5 5. 2.3456e9 1e-12 |
| bool | true false |
| string | "Double quoted" "Kim's" "\n" "" (empty string) |

// Print a few C++ literals

#include <iostream> // For cout and endl using namespace std;

int main() { cout << 123 << endl; cout << 'a' << '\t' << 'm' << endl; cout << 1.23 << endl;

// true prints as 1 and false as 0 cout << true << " and " << false << endl; cout << "Hello \n world" << endl;

return 0; }

##### Output

123 a m 1.23 1 and 0

Hello world

#### SELF-CHECK

2-1 How many special symbols are there in the preceding program?

2-2 List each of the following as a valid identifi er or explain why it is not valid.

1. abc l. H.P.
2. 123 m. double
3. ABC n. 55\_mph
4. #include o. sales Tax
5. my Age p. main
6. #de ne q. a
7. Abc! r. å)
8. identi er s. \_\_\_1\_\_\_
9. (identi er) t. Mile/Hour
10. Double u. os
11. mispellted
    1. List two special symbols that are one character long.
    2. List two special symbols that are two characters long.
    3. List two standard identifi ers.
    4. Create two programmer-defi ned identifi ers.
    5. Given the following tokens:

'\n' false 234 1.0 'H' "'" -123 1.0e+03 "H" true

1. Which are valid string literals?
2. Which are valid integer literals?
3. Which are valid fl oating-point literals?
4. Which are valid Boolean literals?
5. Which are valid char literals?

2-8 Which of the following are valid C++ comments?

1. // Is this a comment?
2. / / Is this a comment?
3. /\* Is this a comment?
4. /\* Is this a comment? \*/

### 2.2 STATEMENTS

A *declaration* introduces one or more object names into a program. An *initialization* also introduces object names into a program with the additional feature of setting the initial *value* to whatever the programmer wants. Th ese variable names are used later when the programmer is interested in the current value or needs to change that value. Here are the general forms for declaring or initializing fundamental and compound types of variables:

**General Form 2.3** *Declaration (some classes have a default initial state)*

*type identi er* ;

**General Form 2.4** *Initialization (declare a variable and assign it a value)*

*type identi er* = *initial-state* ;

Th e type may be double—to store numbers with a decimal point—or a compound type class such as string to store a collection of characters (many other compound types exist).

Th e following code declares some variables and initializes others. Th e semicolons (;) are used to terminate statements.

int credits; // credits is some random integer double points; // credits is some random oating point number double GPA = 0.0; // GPA is initialized to 0.0

bool boolOne; // boolOne could be either true or false bool boolTwo = true; // boolOne is true string rstName; // rstName is the empty string "" string middleName = "James"; // middleName.length() is 5 string lastName = "Potter"; // lastName.length() is 6

Th e fundamental types int, double, and bool diff er from string and other compound types in several ways. When declared, numeric types have an unknown value. However, the default initial value of string objects is the empty string "" when it is not explicitly initialized. Th e following table summarizes the initial state of these objects where some have an unknown state. Th ose variables were declared, but not initialized. Th e value is whatever bits happen to be there when the program runs. Th ese variables can actually have diff erent value during diff erent program runs.

#### Variable Name Object’s State

credits unknown points unknown boolOne unknown

boolTwo true (would print as 1)

GPA 0.0

|  |  |
| --- | --- |
| rstName | "" |
| middleName | "James" |
| lastName | "Potter" |

#### 2.2.1 OUTPUT WITH cout

Programs communicate with users. Such communication is provided through, but not limited to, keyboard input and screen output. Th is two-way communication is a critical component of many of the programming projects in this textbook.

##### General Form 2.5 *Th e* cout *statement*

cout<< *expression-1* **<<** *expression-2* , . . . , << *expression-n* << endl;

Th e object named cout (pronounced “see-out” and short for **c**ommon **out**put) represents where the output will go: the console. *expression-1* through *expression-n* may take the form of object names such as GPA and rstName or constants such as "Credits: " and 99.5. Th e output operator << indicates the direction in which data are fl owing. Finally, a semicolon (;) terminates each statement. Here are some legal output statements that use the endl identifi er (pronounced “end-ell”) to generate a new line:

cout << 99.5 << endl;

cout << "Show me literally too" << endl; cout << "First Name: " << rstName << endl; cout << "Credits: " << credits << endl;

When a cout statement is encountered, the expressions are inserted into a data stream going toward the computer screen. Th e expressions are output in the same order as they are encountered in the statement—in a left-to-right order. When the expression endl is encountered, a new line is generated, so any subsequent output starts at the beginning of a new line.

cout << 'A' << " line " << true << " " << 123 << 4.56 << endl;

##### Output

A line 1 1234.56

#### SELF-CHECK

2-9 Initialize two objects that represent numbers with an initial value of -1.5.

2-10 Declare one object named address that could store a street address.

2-11 Write a complete C++ program that displays any names you have on separate lines.

##### 2.2.2 ASSIGNMENT AND TYPE CONVERSIONS

*Assignment* statements set the state of an object. Th e value of the expression to the right of = replaces whatever value was in the object to the left of =.

**General Form 2.6** *Th e Assignment Statement*

*object-name* **=** *expression*;

Th e *expression* must be a value that can be stored by the object to the left of the assignment operator =. For example, an expression that results in a fl oating-point value can be stored in a numeric object, and a string expression (characters between double quotes " ") can be stored in a string object. Here are some other examples of assignment statements:

double aNumber = -999.9; string aString = "Initial state";

aNumber = 456.789;

aString = "Modi ed state";

After the four assignment operations execute, the state of both objects is modifi ed and the state of these objects can be shown like this:

|  |  |
| --- | --- |
| **Object** | **State** |
| aNumber | 456.789 |
| aString | "Modi ed State" |

Th e value to the right of = must be assignment-compatible with the variable’s type on the left for the assignment to work correctly. For example, a string literal cannot be assigned to a numeric variable.

aNumber = "Ooooohhhh no, you can't do that"; // ERROR

A double literal cannot be assigned to a string object.

aString = 12.34; // ERROR

Th e compiler will report errors at both attempted assignment statements. However, type conversions happen automatically when an object of one type is used when an object of another type is expected. Th ere are no warnings or reported errors, just unexpected values assigned to the variables.

char c = 65; // c becomes 'A' bool b = 0; // b becomes false b = 42; // b becomes true, actually 1 int n = b; // n becomes 1, the integer for true n = 5.9999; // n becomes 5 due to truncation double x = n; // x becomes 5.0, but prints as 5 long l = n; // l becomes 5, int promotes to long

#### SELF-CHECK

2-12 Given the variables initialized above, write the assigned value or report as an error.

|  |  |
| --- | --- |
| a. b = -123; | d. l = x; |
| b. n = 123.495678; | e. c = 66; |
| c. x = 123; | f. ui = "abcde"; |

Be wary of a meaningless object values. Th ey can cause unpredictable errors. Make sure you defi ne all objects either through initialization, an assignment operation, or keyboard input. Also be wary of type conversions, especially when there will be a conversion error that can result in a diff erent value. Th is will occur when mixing signed and unsigned types, so don’t do that. To properly use objects in a program, all three characteristics must be considered:

* An object must be given a name with a declaration or initialization.
* An object must be declared as an instance of a specifi c type.
* At some point, an object should be given a meaningful value.

##### 2.2.3 INPUT WITH cin

To make programs more general—for example, to fi nd a course grade for any student—the state of objects is often set through keyboard input. Th is allows the user to enter any data desired. Input happens with the input stream object named cin (pronounced “see-in” and short for **c**ommon **in**put) and the stream extraction operator >>. For example, the following statements modify the state of two objects with data supplied by the user:

cin >> rstName; // User must input a string cin >> credits; // User must input a number Here is the general form of the input statement with cin:

###### General Form 2.7 *Th e* cin *statement*

cin >> *object-name* ;

- or cin >> *object-name-1* >> *object-name-2* >> *object-name-n*;

Th e object-name must be an instance of a class whose value can be typed in at the keyboard. Th is form of input operation is defi ned for many but not all objects in this textbook. Input with cin is defi ned for the int, double, and string types.

When a cin statement is encountered, the program pauses until the user types the proper input value and presses the Enter key. If everything goes okay, the value typed by the user is converted into the proper machine representation and stored as the state of that object.

In addition to the Enter key, input data is also separated by one or more blank spaces. Th is makes it diffi cult to read in a string with blank spaces, such as a person’s full name or address. Given the following code:

string name;

cout << "Enter your name: "; cin >> name;

and this dialogue:

Enter your name: ***Kim McPhee***

Kim is stored into name, not Kim McPhee as one would hope. Th e blank space after Kim terminates the input value. To get all characters from one line even with spaces, use the getline operation:

getline(cin, name);

You may write the cin statement with more than one object for input. If you do, you must assume that the user knows to separate each input from the preceding one with a blank space (press the Spacebar), a new line (press Enter or Return), or a tab (press the Tab key). Th e following program was run several times to show the various ways input is separated.

#include <iostream> using namespace std;

int main() { int a, b, c, d;

cout << "Enter four integers: ";

// Just need to separate input by a space, tab, or new line. cin >> a >> b >> c >> d; cout << a << endl; cout << b << endl; cout << c << endl; cout << d << endl; return 0;

}

|  |  |  |
| --- | --- | --- |
| **Th ree Possible Dialogues** |  |  |
| Enter four integers:  ***1 2 3 4***  1  2  3  4 | Enter four integers:  ***1 2***  ***3***  ***4***  1  2  3  4 | Enter four integers: ***1***  ***2***  ***3***  ***4***  1  2  3  4 |

A simple alternative would be to use one cin statement for each input.

### 2.3 ARITHMETIC EXPRESSIONS

Many of the problems in this chapter require you to write arithmetic expressions. Arithmetic expressions are made up of two components: operators and operands. An arithmetic *operator* is one of the C++ special symbols +, -, /, or \*. Th e *operands* of an arithmetic expression may be a numeric object name such as test1 or a numeric constant such as 0.25. Assuming x is an instance of the double class, the following expression has operands x and 4.5. Th e operator is +.

x + 4.5

Together, the operator and operands determine the value of the arithmetic expression.

Th e simplest arithmetic expression is a numeric constant or numeric object name. Arithmetic expressions may also have two operands with one operator (see the table below).

|  |  |
| --- | --- |
| **An Arithmetic Expression May Be** | **Example** |
| *Numeric object* | x |
| *Numeric constant* | 100 *or* 99.5 |
| *Expression* + *expression* | x + 2.0 |
| *Expression* - *expression* | x - 2.0 |
| *Expression* \* *expression* | x \* 2.0 |
| *Expression* / *expression* | x / 2.0 |
| ( *Expression* ) | (x + 2.0) |

Th e previous defi nition of expression also suggests that more complex arithmetic expressions are possible, such as this:

1.5 \* ((x - 99.5) \* 1.0 / x)

Since arithmetic expressions may be written with many constants, numeric object names, and operators, rules are put into force to allow a consistent evaluation of expressions. Th e fol-

2.3: Arithmetic Expressions

lowing table lists the fi ve C++ arithmetic operators and the order in which they are applied to numeric objects.

#### Binary Arithmetic Operators

|  |  |
| --- | --- |
| **Operators** | **Precedence Rule** |
| \* / % | In the absence of parentheses, multiplication, division, and remainder (%), operators evaluate before addition and subtraction. In other words, \*, /, and % (for the int remainder) have precedence over + and -. If more than one of these operators appear in an expression, the leftmost operator evaluates fi rst. |
| + - | In the absence of parentheses, + and - evaluate after all \*, /, and % operators, with the leftmost evaluating fi rst. Parentheses may override these precedence rules. |

Th e operators of the following expression are applied to the operands in this order: /, +, and lastly -.

2.0 + 5.0 - 8.0 / 4.0 // Evaluates to 5.0

Parentheses may alter the order in which arithmetic operators are applied to their operands.

(2.0 + 5.0 - 8.0) / 4.0 // Evaluates to -0.25

With parentheses, the / operator evaluates last, rather than fi rst.

Th ese precedence rules apply to binary operators only. A *binary operator* is one that requires one operand to the left and one operand to the right. A *unary operator* only requires one operand on the right. Consider this expression, which has the binary operator \* and the unary minus operator -.

3.5 \* -2.0 // Evaluates to -7.0

Th e unary operator evaluates before the binary \* operator: 3.5 times negative 2.0 (-2.0) results in negative 7.0 (-7.0).

Arithmetic expressions usually have object names as operands. When C++ evaluates an expression with double objects, the object name is replaced with its state. Consider the following code:

double x = 1.0; double y = 2.0; double z = 3.0;

double answer = x + y \* z / 4.0;

When the program is running, the values stored in the variables are retrieved to get this equivalent expression:

double answer = 1.0 + 2.0 \* 3.0 / 4.0; // store 2.5 into answer

#### SELF-CHECK

2-13 Evaluate the following arithmetic expressions:

double x = 2.5; double y = 3.0;

|  |  |
| --- | --- |
| a. x \* y + 3.0 | d. 1.5 \* (x - y) |
| b. 0.5 + x / 2.0 | e. y + -x |
| c. 1 + x \* 3.0 / y | f. ( x - 2) \* (y - 1) |

##### 2.3.1 int ARITHMETIC

Th e C++ language provides several numeric types. Perhaps the two most often used are double and int. An int object represents a limited range of whole numbers. Th ere are times when int is the correct choice over double. An int object has operations similar to double (+, \*, -, =, <<, >>), but some diff erences do exist. For example, a fractional part cannot be stored in an int object. Th e fractional part is lost during an assignment statement.

int anInt = 1.999; // The state of anInt is 1, not 1.999

Th e / operator has diff erent meanings for int and double operands. Whereas the result of

3.0 / 4.0 is 0.75, the result of 3 / 4 is 0. Two integer operands with the / operator have an integer result—not a fl oating-point result. So what happens? An integer divided by an integer results in the integer quotient. For example, the quotient obtained from dividing 3 by 4 is 0. Th is implies that the same operator (/ in this case) has a diff erent meaning when it has two integer operands.

Another diff erence is that int objects have a remainder operation symbolized by the % operator. For example, the result of 18 % 4 is the integer remainder after dividing 18 by 4, which is 2. Th ese diff erences are illustrated in the following program, which shows % and / operating on integer expressions and / operating on fl oating-point operands. In this example, the integer results describe whole hours and whole minutes rather than the fractional equivalent.

2.3: Arithmetic Expressions

// This program provides an example of int division with '/' for

// the quotient and '%' for the remainder

#include <iostream> using namespace std;

int main() {

// Declare objects that will be given meaningful values later int totalMinutes, minutes, hours; double fractionalHour;

// Input

cout << "Enter total minutes: "; cin >> totalMinutes;

// Process

fractionalHour = totalMinutes / 60.0; hours = totalMinutes / 60; minutes = totalMinutes % 60;

// Output

cout << totalMinutes << " minutes can be rewritten as "

<< fractionalHour << " hours " << endl; cout << "or as " << hours << " hours and " << minutes << " minutes" << endl;

return 0; }

###### Dialogue

Enter total minutes: ***254***

254 minutes can be rewritten as 4.23333 hours or as 4 hours and 14 minutes

Th e preceding program indicates that even though int objects and double objects are similar, there are times when double is the more appropriate class than int, and vice versa. Th e double class should be specifi ed when you need a numeric object with a decimal component. If you need whole numbers, select the int class. Also, once the class is chosen, you should consider the diff erences in some of the arithmetic operators. For example, although the +, -, /, and \* operations can be applied to double operands, the % operator may only be used with two integer operands.

#### SELF-CHECK

2-14 What value is stored in nickel?

int change = 97; int nickel = 0; nickel = change % 25 % 10 / 5;

2-15 What value is stored in nickel when change is initialized to:

|  |  |
| --- | --- |
| a. 4 | d. 15 |
| b. 5 | e. 49 |
| c. 10 | f. 0 |

##### 2.3.2 MIXING INTEGER AND FLOATING-POINT OPERANDS

Whenever integer and fl oating-point values are on opposite sides of an arithmetic operator, the integer operand is promoted to its fl oating-point equivalent (3 becomes 3.0, for example). Th e expression then results in a fl oating-point number. Th e same rule applies when one operand is an int object and the other a double.

// Display the value of an expression with a mix of operands

#include <iostream> using namespace std;

int main() { int n = 10; double sum = 567.9;

// n will be promoted to a double and use the oating point / cout << (sum / n) << endl;

return 0; }

###### Output

56.79

#### SELF-CHECK

2-16 Evaluate the following expressions:

|  |  |
| --- | --- |
| a. 5 / 9 | d. 2 + 4 \* 6 / 3 |
| b. 5.0 / 9 | e. (2 + 4) \* 6 / 3 |
| c. 5 / 9.0 | f. 5 / 2 |

##### 2.3.3 CONSTANT OBJECTS

Th e state of any object can be, and usually is, altered during program execution. However, it is sometimes convenient to have data with values that cannot be altered during program execution. C++ provides the keyword const for this purpose. Constant objects are created by specifying and associating an identifi er with a value preceded by the keyword const. In essence, this is an object

2.4: Prompt Then Input

whose state cannot be changed through assignment or stream extraction operations. Th e general form used to initialize a constant object is a combination of an initialization preceded by the keyword const. Const objects are usually written in upper case.

**General Form 2.8** *Initializing a constant object*

const *type* *IDENTIFIER* = *expression*;

For example, the value stored in the constant object PI is the fl oating-point number 3.1415926, and TAX\_RATE is 7.51%.

const double PI = 3.1415926; const double TAX\_RATE = 0.0751; const string PAUSE\_MESSAGE = "Press any key to continue . . .";

Th ese constant objects represent values that cannot be changed while the program is executing; therefore a statement such as PI = PI \* r \* r; generates an error because PI is declared as constant. Th e value cannot be destroyed with an input statement such as cin >> PI;.

### 2.4 PROMPT THEN INPUT

Th e output and input operations are often used together to obtain values from the user of the program. Th e program informs the user what must be entered with an output statement and then performs an input operation to set the state of the object. Th is happens so often that this activity can be considered to be a pattern. Th e Prompt then Input algorithmic pattern has two activities:

1. Ask the user to enter a value (prompt).
2. Obtain the value for the object (input).

|  |  |
| --- | --- |
| **Algorithmic Pattern** | *Prompt then Input* |
| **Pattern:** | Prompt then Input |
| **Problem:** | Th e user must enter something |
| **Outline:** | 1. Prompt the user for input 2. Obtain the input |
| **Code Example:** | cout << "Enter your rst name: "; cin >> rstName; |

Strange things can happen if the prompt is left out. Th e user will not know what must be entered. So whenever you require user input, make sure you prompt for it fi rst. Write the code that tells the user precisely what you want. First output the prompt, then obtain the user input.

Here is one instance of the Prompt then Input pattern:

cout << "Enter test #1: "; cin >> test1;

and another:

cout << "Enter credits: "; cin >> credits;

In general, tell the user the value needed, then read it in with cin.

cout << "*the prompt for the\_object:* "; cin >> *the\_object*;

Th e following program uses the Prompt then Input pattern four times. It also reviews operations such as object initialization, assignment, input, and output. Th is program illustrates a more general approach to computing any grade point average. By requesting input data from the user, it can be used over and over again with diff erent sets of input to produce diff erent results. Also notice the presence of the IPO pattern in the implementation.

// This program uses input statements to produce a meaningful

// result that can be used in a variety of examples

#include <iostream> // For input and output #include <string> // For the string class using namespace std;

int main() {

// 0. Initialize some objects double credits = 0.0; double points = 0.0; double GPA = 0.0; string rstName; string lastName;

// 1. Input

cout << "Enter rst name: "; cin >> rstName; cout << "Enter last name: "; cin >> lastName; cout << "Enter credits: "; cin >> credits; cout << "Enter points: "; cin >> points;

// 2. Process

GPA = points / credits;

// 3. Output cout << "Name : " << rstName << " " << lastName << endl; cout << "Credits : " << credits << endl; cout << "Points : " << points << endl; cout << "GPA : " << GPA << endl;

2.4: Prompt Then Input

return 0; }

#### Dialogue

Enter rst name: ***Pat***

Enter last name: ***McCormick***

Enter credits: ***97.5***

Enter points: ***323.75***

Name : Pat McCormick

Credits : 97.5

Points : 323.75

GPA : 3.32051

Care must be taken when entering numeric data. If you enter a non-digit instead of valid numeric input, the input object cin may no longer be in a “good” state and all subsequent cin statements will be ignored.

#### SELF-CHECK

2-17 Write the value for GPA given each of the dialogues shown below.

// This program uses input statements to produce a

// meaningful result that can be used for a variety of examples

#include <iostream> // For cin, cout, and endl

#include <string> // For the string class using namespace std;

int main() {

// 0. Initialize some numeric objects double c1 = 0.0; double c2 = 0.0; double g1 = 0.0; double g2 = 0.0; double GPA = 0.0;

// 1. Input

cout << "Credits for course 1: "; cin >> c1;

cout << " Grade for course 1: "; cin >> g1;

cout << "Credits for course 2: "; cin >> c2;

cout << " Grade for course 2: "; cin >> g2; // 2. Process GPA = ( (g1\*c1) + (g2\*c2) ) / (c1 + c2);

// 3. Output

cout << "GPA: " << GPA << endl; return 0;

}

Dialogue 1:

Credits for course 1: ***2.0***

Grade for course 1: ***2.0***

Credits for course 2: ***3.0***

Grade for course 2: ***4.0***

1. \_\_\_\_\_\_\_\_ GPA

Dialogue 2:

Credits for course 1: ***4.0***

Grade for course 1: ***1.5***

Credits for course 2: ***1.0***

Grade for course 2: ***3.5***

1. \_\_\_\_\_\_\_\_ GPA

Dialogue 3:

Credits for course 1: ***1.0***

Grade for course 1: ***2.0***

Credits for course 2: ***4.0***

Grade for course 2: ***3.0***

1. \_\_\_\_\_\_\_\_ GPA

### 2.5 IMPLEMENTATION ERRORS AND WARNINGS

Th ere are several types of errors and warnings that occur during the implementation phase of problem solving:

* compile time—errors that occur during compilation
* warnings—code that is risky, suggesting there may be a future error
* linktime—errors that occur when the linker cannot fi nd what it needs
* runtime—errors that occur while the program is executing
* intent—the program does what was entered, not what was intended

**Compiler**

**Linker**

**Executable**

**Program**

Source Code

iostream

Machine Code

Machine Code

Machine Code

Machine Code

010101010

110010100

101010010

010101001

001010100

101010010

#include <iostream>

int main()

{

double x(0.0) ;

cout << "x? " ;

cin >> x;

cout << (x\*x) ;

Machine Code

#### 2.5.1 ERRORS AND WARNINGS DETECTED AT COMPILE TIME

A programming language requires strict adherence to its own set of formal syntax rules. As you have probably noticed, it is easy to violate these syntax rules while translating algorithms into their programming language equivalents. All it takes is a missing { or ; to really foul things up. As the C++ compiler translates source code into code that can run on the computer, the compiler locates and reports as many errors as possible

warns of potential problems that are syntactically legal, but might cause errors later

A compile time error occurs when the C++ compiler recognizes the violation of a syntax rule. Th e machine code cannot be created until all compile time errors have been removed from the program. If the machine code is not created, the linker cannot create an executable program. Many strange-looking error messages will be generated by the compiler as it reads the source code. Unfortunately, deciphering these compile time error messages takes practice, patience, and a reasonable knowledge of the C++ programming language. So in an eff ort to improve this situation, here are some examples of common compile time errors you will soon see and explanations of how they are corrected. (*Note:* Your compiler will generate diff erent error messages.)

|  |  |  |
| --- | --- | --- |
| **Error Detected by a Compiler Incorrect Code** | | **Corrected Code** |
| Splitting a variable name | int Total Weight; | int totalWeight; |
| Misspelling a name | integer sum = 0 ; | int sum; |
| Omitting a semicolon | double x | double x; |
| Not closing a string | cout << "Hello; | cout << "Hello"; |
| Failing to declare the variable | cin >> testScore; | double testScore; cin >> testScore; |
| Ignoring case sensitivity | double X; | double x; |
|  | X = 5.0; | x = 5.0; |
| Forgetting an argument | cout << sqrt; | cout << sqrt(x); |
| Using wrong type | cout << sqrt("12"); cout << sqrt(12.0); | |
| Using too many arguments | cout << sqrt(1.2, 3); cout << sqrt(1.2); | |
| Forgetting namespace std; | // cout is unknown using namespace std; | |

Compilers generate many error messages. However, it is your source code that is the source of these errors. Whenever your compiler appears to be nagging you, remember that the compiler is trying to help you correct your errors as much as possible. Th e compiler is your friend. Th e following code shows several errors the compiler should eventually detect and report. Because error messages generated by compilers vary among systems, the comments here indicate the reason for the error—they are not an attempt to match compile time error messages for any particular compiler (and there are many compilers). Your system will certainly generate quite different error messages.

// This attempt at a program contains many errors--over a

// dozen. Add #include <iostream>, and there are only eight. using namespace std;

int main { // 1. No () after main. // 2. Every cin and cout will generate an error

// because #include <iostream> is missing. int pounds;

cout << "Begin execution" << endl // Missing ; after endl cout >> "Enter weight in pounds: "; // >> should be << cin << pounds; // << should be >> cout << "In the U.K., you"; // Extra ;

<< " weigh " << (Pounds / 14) // Pounds is not declared

<< " stone. " << endl // Missing ; return 0; // Missing right brace }

Compilers can generate some rather cryptic messages. When the program shown above compiled with one particular compiler, six errors occurred (other compilers found seven and two), all reporting a type name was expected. Other systems generate a diff erent crop of errors. Another Unix compiler generated eight completely diff erent errors. Compile time error messages take some getting used to, so try to be patient and observe the location where the compile time error occurred. Th e error is usually in the vicinity of the line where the error was detected, although you may have to fi x preceding lines. Always remember to fi x the fi rst error fi rst. Th e error not reported until line 23 may be the result of a forgotten semicolon on line 4.

Th e corrected source code, without error, is given next, followed by an interactive dialogue:

// There are no compile time errors in this program

#include <iostream> using namespace std;

int main() { int pounds;

cout << "Begin execution" << endl; cout << "Enter your weight in pounds: "; cin >> pounds; cout << "In the U.K., " << "you weigh " << (pounds/14.0) << " stone." << endl;

return 0; }

##### Dialogue

Begin execution

Enter your weight in pounds: ***162*** In the U.K., you weigh 11.5714 stone.

It should also be noted that one small compile time error can result in a cascade of errors. For example, omitting { after int main() in an otherwise error-free program caused the Clang C++ compiler to generate 11 compile time errors!

#include <iostream> // For cin and cout

#include <string> // For the string class using namespace std;

int main() // <- Without the left curly brace, there were 11 errors!

double x; string str;

cout << "Enter a double: "; cin >> x;

cout << "Enter a string: "; cin >> str; return 0; }

##### Compile time error messages from one compiler

main.cpp:5:11: error: expected ';' after top level declarator main.cpp:9:3: error: unknown type name 'cout' main.cpp:9:8: error: expected unquali ed-id main.cpp:10:3: error: unknown type name 'cin' main.cpp:10:7: error: expected unquali ed-id main.cpp:11:3: error: unknown type name 'cout' main.cpp:11:8: error: expected unquali ed-id main.cpp:12:3: error: unknown type name 'cin' main.cpp:12:7: error: expected unquali ed-id main.cpp:13:3: error: expected unquali ed-id main.cpp:14:1: error: expected external declaration

Th e SunOS C++ compiler reported one error, which is more decipherable:

"{" expected not double

So it is possible that fi xing the fi rst error might correct many other errors. It is also true that fi xing one error might let the compiler fi nd new ones! Try to concentrate on the fi rst error your compiler reports. Th e compiler usually, but not always, will be able to approximate the location of the error in your source code. Th e error may be on the line above, or many lines above. Also, realize that all statements must be terminated by a semicolon “;”. Excluding this statement terminator, or putting it in where it doesn’t belong, causes compile time errors. Missing semicolon errors are not usually detected until the compiler has already gone past the line with the off ense, so look at the statement above the location of the error.

#### 2.5.2 WARNINGS GENERATED AT COMPILE TIME

Compilers also generate warnings, which are messages intended to help programmers avoid errors later on. Consider the following code:

#include <iostream> using namespace std;

int main() { double x, y; y = 2 \* x; cout << y << endl; }

#### SELF-CHECK

2-18 What is the output of the preceding program?

Th e preceding program has an error, but none that the compiler will catch. Th e compiler happily translated the source code into machine code and the linker created an executable program. However, the output from two program runs were the rather inexplicable numbers

1.09087e+82 and later 1.39064e-309. With another compiler, the output was 0. Fortunately some compilers generate warnings like these:

Warning: Possible use of 'x' before de nition in main()

Warning: 'x' is used uninitialized in this function

Th e warning states that x has been used before it was defi ned (initialized, actually). Th is is a good warning that should not be ignored. Th e program does not initialize x. It has an unknown state sometimes referred to as garbage. Unfortunately, not all compilers will warn you of this potential error.

Th is is not a violation of any C++ rule. It is legal to declare a variable without an initial value. However, this warning should not be ignored. You should read it and make sure x has an initialized state before using the object in an arithmetic expression.

Th is is but one example of a warning. You will see more. You will likely ignore many. However, warnings are hints that something may go wrong and the program will not be correct. If you are getting incorrect results, look to see if there are any warnings—they may be clues to the source of the error.

##### 2.5.3 LINKTIME ERRORS

Computer systems use a linker to combine pieces of machine code to create executable programs. Among other things, the linker must resolve details such as locating the identifi er main in one of these fi les. If main is not found during the linking process, the linker generates an error that displays main is an undefi ned symbol. If this takes place, verify that your program starts with int main.

int main() { // . . .

}

Make sure that main is not typed as mane, Main, or MAIN.

Another linktime error occurs when you have two fi les with the required int main(). For example, you may soon try to have two programs in the same folder when completing programming projects. Th e following linktime error message indicates initials.cpp and average.cpp both had int main() functions in the same directory named src:

ld: duplicate symbol main () in ./src/initials.o and ./src/average.o

One solution is to not link both of them. But if you are using an integrated development environment such as Eclipse, Visual Studio, or XCode, make sure you have only one fi le in your project with the main method.

##### 2.5.4 RUNTIME ERRORS

A program may execute after all compile time errors have been removed and the linker has created an executable program. But errors may still occur while the program is running. A runtime error may cause the program to terminate before it should because some event occurs that the computer cannot handle.

For example, when your program is expecting an integer, and the user enters a fl oating-point number, the input stream cin becomes corrupted. Consider the same program run twice, the fi rst time with good input and once with “bad” input; a fl oating-point number such as 1.2 instead of an integer.

#include <iostream> using namespace std;

int main() {

int anInt, anotherInt;

cout << "Enter anInt: "; cin >> anInt;

cout << "anInt: " << anInt << endl;

cout << "Enter anotherInt: "; cin >> anotherInt;

cout << "anotherInt: " << anotherInt << endl; return 0; }

With good input, the user can enter two integers:

Enter anInt: ***7*** anInt is 7 Enter anotherInt: ***9*** anotherInt is 9

With non-integer input, 1.2 cannot be assigned to an int, so it is not. Th en the second input is not allowed and the user cannot even try to enter a number (the output of 0 will vary since anotherInt is undefi ned):

Enter anInt: ***1.2*** anInt: 1 Enter anotherInt: anotherInt: 0

##### 2.5.5 INTENT ERRORS

Even when no compile time errors are found and no runtime errors occur, the program still may not execute properly. A program may run and terminate normally, but it may not be correct. Let’s make one small change to an earlier program to get an incorrect program.

cout << "Average: " << (n / sum); Th e interactive dialogue may now look like this:

Enter sum: ***291***

Enter n: ***3***

Average: 0.010309

Such intent errors occur when the program does what was typed, not what was intended. Unfortunately, the compiler does not locate intent errors. Th e expression n/sum is syntactically correct—the compiler just has no way of knowing that this programmer intended to write sum/n instead.

Intent errors are the most insidious and usually the most diffi cult to correct. Th ey may also be diffi cult to detect—the user, tester, or programmer may not know they even exist.

#### SELF-CHECK

2-19 Assuming a program is supposed to fi nd an average given the total sum and number of values in a set, then the following dialogue is generated. What clue reveals the presence of an intent error?

Enter sum: ***100***

Number : ***4***

Average : 0.04

2-20 Assuming the following code was used to generate the dialogue above, how is the intent error to be corrected?

cout << "Enter sum: "; cin >> n; cout << "Number : "; cin >> sum; average = sum / n;

cout << "Average : " << average << endl;

2-21 List the type of error (compile time, runtime, linktime, or intent) or warning that exists when the last statement in the preceding program is changed to:

Chapter Summary

1. cout << "Average: " << "sum / n";
2. cout << "Average: ", sum / n;
3. cout << "Average: " << sum / n

##### 2.5.6 WHEN THE SOFTWARE DOESN’T MATCH THE SPECIFICATION

Even when a process has been automated and delivered to the customer in working order according to the developers, there may still be errors. Th ere have been many instances of software working, but not doing what it was supposed to do. Th is could be from a failure to meet the problem statement, which occurs when the software developers don’t understand the customer’s problem statement. Something could have been missed. Something could have been misinterpreted. A related error occurs when the client specifi es the problem incorrectly. Th is could be the case when the requester isn’t sure what they want. A trivial or critical omission in specifi cation may occur, or the request may not be written clearly. Also, the requester may change their mind after problem solving has begun.

For the most part, the end-of-chapter programming projects in this textbook ask you to fulfi ll the problem specifi cation. If you think there is an omission or there is something you don’t understand, don’t hesitate to ask questions. It is better to understand the problem and know what it is that you are to trying to solve before getting to the design and implementation phases of problem solving. Although not intended, the problem may be incorrectly or incompletely specifi ed—this does actually happen in the real world!

### CHAPTER SUMMARY

* Th e smallest pieces of a program (tokens) were shown to help you understand general forms and fi x errors.
* Objects are entities that have a name, state (value), and operations. Output (cout <<) and input (cin >>) operations are used in concert with double and string objects to set their states. Th ere are at least three techniques for modifying the state of an object:
* initialization with double x = 0.0;
* input with cin >>
* assignment with =
* Knowledge of existing objects aids the program development process. For example, knowing about cin, cout, string, and numeric objects such as int and double precludes the necessity of implementing many intricate operations such as input, output, addition, and multiplication. Fortunately, other programmers have already built them.
* Th e objects named std::cout and std::cin are so frequently used that they are automati-

cally made available for easy screen output and keyboard input with #include <iostream>. If you add using namespace std; you do not have to precede cout, cin, and endl with std:: .

* Arithmetic expressions are made up of operators +, -, \*, /, and % (remainder). A binary arithmetic operator requires two operands, which may be numeric constants such as 1 and 2.3, numeric objects, or other arithmetic expressions.
* Instances of the Prompt then Input pattern will occur in many programming projects. Use it whenever a program needs to get some input from the user.
* When / has two integer operands, the result is an integer, so 5 / 2 is 2.
* When / has at least one fl oating-point operand, the result is a fl oating-point number, so

5 / 2.0 is 2.5.

Th e % operator returns the integer remainder of one integer operand divided by another, so

5 % 2 is 1.

* Th e % operator cannot have a fl oating-point argument, so 5 % 2.0 is a compile time error.
* Be careful in choosing int and double. Always use double to store numbers unless it makes sense that the object can only store integers.
* Th is chapter ended with a discussion of the variety of errors that occur during implementa-tion. You will continue to encounter errors. It is part of the process.
* Errors may be present because the problem statement was incorrect or incomplete.
* Intent errors eventually prove to be the most diffi cult to fi x—they can be diffi cult to detect.

### EXERCISES

1. List three operations that may be applied to numeric types like double.
2. Describe the value(s) stored in string objects.
3. List three operations that may be applied to any string object.
4. List four types of C++ tokens and give two examples of each.
5. Which of the following are valid identifi ers?

|  |  |
| --- | --- |
| a. a-one | g. 1\_2\_3 |
| b. R2D2 | h. A\_B\_C |
| c. registered\_voter | i. all right |
| d. BEGIN | j. 'doubleObject' |
| e. 1Header | k. {Right} |
| f. $money | l. Mispelt |

1. Declare totalPoints as an object capable of storing a number.

Exercises

1. Write a statement that sets the state of totalPoints to 100.0.
2. Write the entire dialogue generated by the following program when 5.2 and 6.3 are entered at the prompt. Make sure you write the user-supplied input as well as all program output including the prompt.

#include <iostream> using namespace std; int main() { double x = 0.0; double y = 0.0; double answer = 0.0; cout << "Enter a number: "; cin >> x;

cout << "Enter another number: "; cin >> y; answer = x \* (1.0 + y); cout << "Answer: " << answer << endl; return 0; }

1. Write C++ code that declares tolerance as a numeric object set to 0.001 that cannot be changed while the program is running.
2. Write a statement that displays the value of a numeric object named total.
3. Given these two object initializations, either write the value that is stored in each object or report the attempt as an error.

string aString; double aNumber = 0.0;

|  |  |
| --- | --- |
| a. aString = "4.5"; | c. aString = 8.9; |
| b. aNumber = "4.5"; | d. aNumber = 8.9; |

1. With paper and pencil, write an entire C++ program that prompts for a number from 0.0 to 1.0 and stores this input value into the numeric object named relativeError. Echo the input (output the input). Th e dialogue generated by your program should look like this:

Enter relativeError [0.0 through 1.0]: 0.341

You entered: 0.341

1. Assuming x is 5.0 and y is 7.0, evaluate the following expressions:
   1. x / y c. 2.0 - x \* y
   2. y / y d. (x\*y)/(x+y)
2. Predict the output generated by these two programs:

a. b.

#include <iostream> using namespace std; int main() { double x = 1.2; double y = 3.4; cout << (x + y) << endl; cout << (x - y) << endl; cout << (x \* y) << endl; cout << (x / y) << endl; return 0;

}

#include <iostream> using namespace std; int main() { double x = 0.5; double y = 2.3; double answer = 0.0; answer = x \* (1 + y); cout << answer << endl; answer = x / (1 + y); cout << answer << endl; return 0; }

1. What is the value of quarter when change is initialized as follows:
   1. 0 c. 49
   2. 74 d. 549

int change = (one of the following: 0 , 74, 49, and 549); int quarter = change % 50 / 25;

1. Is this code correct?

const double EPSILON = 0.000001; EPSILON = 999999.9;

1. Write C++ code that generates a runtime error and give the reason for the error.
2. At what time will the error in this code be detected?

#include <iostream> using namespace std; int Main() { cout << "Hello world"; return 0; }

1. Explain how to fi x the error in each of these lines:
   1. cout << "Hello world" c. cout "Hello World";
   2. cout >> "Hello world"; d. cout << "Hello World;
2. Explain the error in this attempt at a program:

int main() { cout << "Hello world";

Exercises

return 0; }

1. Describe the phrase *intent error*.
2. Does the following code always correctly assign the average of the three doubles x, y, and z to average? double average = x + y + z / 3.0;
3. Evaluate the following expressions. Use a decimal point to distinguish integer and fl oatingpoint values.
   1. 5 / 2 d. 5.0 / 2.0
   2. 5 / 2.0 e. 1.0 + 2.0 - 3.0 \* 4.0
   3. 101 % 2 f. 100 % 2
4. Write the output generated by the following programs:

|  |  |  |
| --- | --- | --- |
| a. | #include <iostream> using namespace std; int main() { const int MAX = 5; cout << (MAX / 2.0) << endl; cout << (2.0 / MAX) << endl; cout << (2 / MAX) << endl; cout << (MAX / 2) << endl; return 0;  } | c.  #include <iostream> using namespace std;  #include <string> int main() {  const string pipe = " ¦ "; cout << pipe << (1 + 5.5) << pipe << (3 + 3 / 3)  << pipe << (1 + 2) / (3 + 4)  << pipe << (1 + 2 \* 3 / 4); return 0;  } |
| b. | #include <iostream> using namespace std; int main() { int j = 14; int k = 3;  cout << "Quotient: "  << (j / k) << endl; cout << "Remainder: " << (j % k) << endl; return 0;  } | d.  #include <iostream> using namespace std; int main() { int j = 11;  cout << " " << (j % 2) << " " << (j / 2)  << " " << ((j - j) / 2); return 0;  } |

### PROGRAMMING TIPS

1. Semicolons terminate statements. Make sure you terminate statements with ;. However, do not place semicolons after #includes and int main().

#include <iostream> ; // Error found on this line int main() ; // Error found on this line {

1. Fix the fi rst error fi rst. When you compile, you may get dozens of errors. Don’t panic. Try to fi x the very fi rst error fi rst. Th at may fi x several others. Sometimes fi xing one error causes others. After fi xing one error, the compiler may generate errors that went undetected before.
2. Integer arithmetic behaves unexpectedly for some students. Integer division results in an integer. Th erefore 5 / 2 is 2, not the 2.5 your brain and calculator feel are so right.
3. Th e % arithmetic operator returns an int remainder. Experience shows some students never understand %, or at least they still get the wrong answers on the fi nal exam. Th e expression a % b is the integer remainder after dividing a by b.

|  |  |  |
| --- | --- | --- |
| 99 % 50 = 49 |  | 101 % 2 = 1 |
| 99 % 50 % 25 = 24 |  | 102 % 2 = 0 |
| 4 % 99 = 4 |  | 103 % 2 = 1 |

1. If you do not have the line using namespace std; you will have to prepend std:: before every occurrence of the cin, cout, and endl.

#include <iostream> // For cout, cin, and endl

// using namespace std; Without this, prepend with std::

int main() { std::string name; std::cin >> name;

std::cout << "Hello" << std::endl; std::cout << name << std::endl; }

**PROGRAMMING PROJECTS**

## 2A THE CLASSIC “HELLO WORLD!” PROGRAM

While designing the C language at AT&T, Dennis Ritchie suggested that a fi rst program in any language be one that displays Hello World! Many fi rst programs have continued this “Hello World!” tradition. Create a new fi le called hello.cpp and retype the following code as shown. Save this fi le and use the instructions particular to your setup to compile, link, and run this program.

Programming Projects

// Programmer: Firstname Lastname // This programs displays a simple message.

#include <iostream> // For cout

using namespace std; // Allow cout instead of std::cout

int main() {

cout << "Hello World!" << endl; return 0; }

## 2B EXPERIENCE ERRORS GENERATED BY THE COMPILER

One small coding error may cause the report of many errors at compile time—this can be misleading. For example, a missing semicolon may result in dozens of errors throughout a program. Remember to fi x the fi rst error fi rst. Start by fi xing the earliest discovered error in the source code. You are now asked to observe what happens when a left curly brace is left out of a program. Carefully retype the following program exactly as shown.

// Observe how many errors occur when { is missing

#include <iostream> // For cout using namespace std; // To make cout known

int main() // <- Leave off { double x = 2.4; double y = 4.5; cout << "x: " << x << endl; cout << "y: " << y << endl; return 0; }

1. Compile your source code and write the number of errors that occur.
2. Add { after int main() and compile again. Make corrections until you have no errors.
3. Now remove the #include directive #include <iostream> and compile the program. How many errors do you get?
4. Replace #include <iostream> and remove the () after main. How many errors do you get?
5. Comment out using namespace std;. How many errors do you get now?
6. If necessary, edit and compile this program until there are no compile time errors. Link and run the program.

## 2C BIG INITIALS

Write a C++ program that displays your initials on the screen in large letters. Th ere are no input or process steps, only output. For example, if your initials are E. T. M., the output should look like this generated by fi ve cout statements:

EEEEE TTTTTTT M M

E T M M M M

EEEEE T M M M

E T M M

EEEEE o T o M M o

## 2D YODA

Write a C++ program that obtains any three strings from the user and outputs them in reverse order with one space between them. (*Hint:* Th ere is no process step; only input followed by output.)

Enter string one: ***happy***

Enter string two: ***am***

Enter string three: ***I*** I am happy

## 2E WEIGHTED AVERAGE

Implement and test a C++ program that will compute the course grade using this weighted scale:

|  |  |
| --- | --- |
| Assessment | Weight |
| Quiz average | 20% |
| Midterm | 20% |
| Lab grade | 35% |
| Final exam | 25% |

One dialogue should look like this:

Enter Quiz Average: ***90.0***

Enter Midterm: ***90.0***

Enter Lab Grade: ***90.0***

Enter Final Exam: ***90.0*** Course Average = 90

## 2F SECONDS

Write a program that reads a value in seconds and displays the number of hours, minutes, and seconds represented by the input. Here are two sample dialogues:

Enter seconds: ***32123*** Enter seconds: ***61***

8:55:23 0:1:1

## 2G MINIMUM COINS

Write a C++ program that prompts for an integer that represents the amount of change (in cents) to be handed back to a customer in the United States. First, display the minimum number of half dollars, quarters, dimes, nickels, and pennies that will make the correct change. (*Hint*: With increasingly longer expressions, you could use / and % to evaluate the number of each coin. Or

Programming Projects

you could calculate the total number of coins with / and the remaining change with %.) Verify that your program works correctly by running it with a variety of input. Here are two sample dialogues:

|  |  |
| --- | --- |
| Enter change [0...99]: ***83***  Half(ves) : 1 Quarter(s) : 1 Dime(s) : 0 Nickel(s) : 1 Penny(ies) : 3 | Enter change [0...99]: ***14***  Half(ves) : 0 Quarter(s) : 0 Dime(s) : 1 Nickel(s) : 0 Penny(ies) : 4 |

## 2H EINSTEIN’S NUMBER

It is said that Albert Einstein used to take great delight in baffl ing friends with the following puzzle. It could be repeated something like this:

* Write 1089 on a piece of paper, fold it, and hand it to another for safekeeping.
* Ask someone else to write down any three-digit number, emphasizing that the fi rst and last digits must diff er: 654 is okay, while 454 and 656 are not allowed.
* Reverse that written down number: if starting with 654, write 456.
* Compute the diff erence of the written down three digit number and its reverse: use

abs(456-654) = 198

* Once this is done, reverse the new number: 198 becomes 891.
* Th en add the new number to its reverse: 198 + 891 = 1089.

If all goes as planned, observers will be amazed. Th e number written down at the start, 1089, will always be the same as the end result of this mathematical trick. Replicate this puzzle as a C++ program. Your program dialogue must look like this when the user enters 541.

Enter a 3 digit number ( rst and last digits must differ): ***541***

541 -- original

145 -- reversed

396 -- difference

693 -- reverse of the difference

1089 -- difference + reverse of the difference

You don’t need to error check the three-digit number. Assume input is in the range of 100 to 998 where the fi rst and third digits are not the same. 101, 252, or 989 are not expected to generate 1089. (*Hint:* To fi nd the diff erence between two numbers, use the absolute value function abs. Th e argument is an expression that subtracts two numbers. You may need to #include <cstdlib>.)

#include <cstdlib> // A new include

#include <iostream> using namespace std; int main() {

// abs is a new function that can return the difference

// between two numbers by subtracting one from the other.

cout << abs(541 - 145) << endl; // 396 cout << abs(145 - 541) << endl; // 396 return 0; }

## 2I TIME DIFFERENCE

Write a C++ program that takes two diff erent train departure times (where 0 is midnight, 0700 is 7:00 a.m., 1314 is 14 minutes past 1:00 p.m., and 2200 is 10 p.m.) and prints the diff erence between the two times in hours and minutes. Assume both times are on the same date and that both times are valid. For example, 1099 is not a valid time because the last two digits are minutes, which should be in the range of 00 through 59. 2401 is not valid because the hours (the fi rst two digits) must be in the range of 0 through 23 inclusive. For example, if train A departs at: 1255 and train B departs at: 1305 the diff erence would be 0 hours and 10 minutes. One dialogue should look like this, but run your program several times for several test cases.

Train A departs at: ***1255***

Train B departs at: ***1305***

Difference: 0 hours and 10 minutes

**C H A P T E R T H R E E**

# Using Free Functions

## SUMMING UP

You should have now gained hands-on experience with your system, the syntax of the language, error messages, and program development—from beginning to end. Most programming projects in the early chapters of this textbook are instances of the IPO algorithmic pattern. You should now be able to put these three steps in the proper order and understand the ramifi cations of omitting one or mixing up the order.

## COMING UP

Software developers often use existing software, a practice that saves time and money. Th is chapter introduces one way to reuse existing software. Programmers begin with a substantial base of tested software. You will learn how to use existing functions by reading function headings and to determine what the functions do by reading their pre- and postconditions—the contracts for using those functions. Th e chapter ends with a review of the categories of errors you will probably have encountered. After studying this chapter, you will be able to

* evaluate some mathematical and trigonometric functions
* use arguments in function calls
* appreciate why programmers divide software into functions
* read function headings so you can use existing functions

### 3.1 cmath FUNCTIONS

C++ defi nes a large collection of mathematical and trigonometric functions that may be used with doubles. Here are two:

sqrt(x) // Return the square root of x pow(x,y) // Return x to the yth power

Th ese functions are *called* by specifying the name of the function, followed by the appropriate number and type of *arguments* within the parentheses. Here is one general form to call certain functions.

55

#### General Form 3.1 *Function call*

function-name(*arguments*)

Th e *function-name* is a previously declared identifi er representing a function name. Th e *arguments* represent a set of zero or more expressions separated by commas. In the following function call, the function name is sqrt (square root) and the argument is 81.0:

sqrt(81.0) // An example of a function call

Functions may have zero, one, or even more arguments. Although most math functions require exactly one argument, the pow function requires exactly two arguments. In the following function call, the function name is pow (for power), the arguments are base and power, and the function call pow(base, power) is replaced with basepower:

double base = 2.0; double power = 3.0; cout << pow(base, power); // Output: 8.0

Any argument used in a function call must be an expression from an acceptable class. For example, the function call sqrt("Bobbie") results in an error because the argument is not one of the numeric classes.

Th e function must also be supplied with reasonable arguments. For example, the function call sqrt(-4.0) could be a problem because -4.0 is not in the domain of sqrt. Th e square root function is not defi ned for negative numeric values. Th e sqrt function operates correctly only if certain conditions are met. For sqrt, the argument must be greater than or equal to 0.0. Here are some mathematical and trigonometric functions available when you include the cmath function library.

**A Partial List of** cmath **Functions.** *Note:* double *before the function name is the return type*

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **What it Returns** | **Example Call** | **Result** |
| double ceil(double x) | Smallest integer ≥ x | ceil(2.1) | 3.0 |
| double cos(double x) | Cosine of x radians | cos(1.0) | 0.5403 |
| double fabs(double x) | Absolute value of x | fabs(-1.5) | 1.5 |
| double oor(double x) | Largest integer ≤ x | oor(2.9) | 2.0 |
| double pow(double x, | xy | pow(2, 4) | 16.0 |
| double y) double sin(double x) | Sine of x radians | sin(1.0) | 0.84147 |
| double sqrt(double x) | Square root of x | sqrt(4.0) | 2.0 |

3.2: Problem Solving with cmath Functions

With #include <cmath> at the “top” of the program near #include <iostream>, the programmer can successfully compile a program with calls to the functions declared in cmath. Th is means that the following program compiles successfully:

// Show some mathematical functions available from cmath

#include <cmath> // For fabs, ceil, oor, and pow

#include <iostream> // For cout using namespace std;

int main() { double x = -2.1;

cout << "fabs(-2.1): " << fabs(x) << endl << "ceil(-2.1): " << ceil(x) << endl

<< " oor(-2.1): " << oor(x) << endl << "pow(-2.1, 2.0): " << pow(x, 2.0) << endl; return 0; }

#### Output

fabs(-2.1): 2.1 ceil(-2.1): -2 oor(-2.1): -3 pow(-2.1, 2.0): 4.41

It should be noted that integer expressions may also be used as arguments to cmath functions. As with assignment, the integer value will be promoted to a double. So sqrt(4) returns the same result as sqrt(4.0) without error.

#### SELF-CHECK

3-1 Evaluate pow(4.0, 3.0)

3-2 Evaluate pow(3.0, 4.0)

3-3 Evaluate oor(1.6 + 0.5)

3-4 Evaluate ceil(1.6 - 0.5)

3-5 Evaluate fabs(1.6 - 2.6)

3-6 Evaluate sqrt(16.0)

### 3.2 PROBLEM SOLVING WITH cmath FUNCTIONS

*Problem:* Write a program that rounds a number to a specifi c number of decimal places. For example, 3.4589 rounded to two decimal places should be 3.46 and 3.4589 rounded to one decimal place should be 3.5.

#### 3.2.1 ANALYSIS

Th e analysis/design/implementation software development strategy begins with these analysis activities:

1. Read and understand the problem.
2. Decide what object(s) represent the answer—the output.
3. Decide what object(s) the user must enter to get the answer—the input.
4. Write test cases (two were given above).

#### 3.2.2 DESIGN

Th e deliverable from this design phase is the algorithm. A pseudocode algorithm can be developed with the help of the Input/Process/Output pattern. Th at pattern is repeated here for your convenience:

**Pattern:** Input/Process/Output (IPO)

**Problem:** Th e program requires input from the user in order to compute and display the desired information.

**Outline:** 1. Obtain the input data.

1. Process the data in some meaningful way.
2. Output the results.

**Example:** See the problem of rounding x to n decimals that follows.

Th e Input/Process/Output pattern helps guide placement of the appropriate activities in the proper order. Th e algorithm represents the general design—an outline of the solution. Adding two usages of the Prompt then Input pattern and a more detailed algorithm might now look like this:

1. Prompt for the number to round (call it x).
2. Input x.
3. Prompt for the number of decimal places (call it n).
4. Input n.
5. Round x to n decimals.
6. Display the modifi ed state of x.

Steps 1, 2, 3, 4, and 6 have straightforward C++ implementations. Th ey can be implemented as input and/or output statements. However, the details of step 5, “Round x to n decimals,” are not present. Step 5 needs refi nement. With the rest of the problem out of the way, you can focus on the more diffi cult process of rounding x to n decimals. A solution is a bit tricky, so one method is provided.

3.2: Problem Solving with cmath Functions

To round a number x to n decimal places, fi rst multiply x by 10n. Th en add 0.5 to the new state of x. Th en store oor(x) in x. Finally, divide x by 10n. Th e refi ned algorithm adds this fourstep refi nement:

1. Prompt for x, the number to round.
2. Input x.
3. Prompt for n, which is the number of decimal places.
4. Input n.
5. Round x to n decimals like this:
   1. Let x become x \* 10n.
   2. Add 0.5 to x.
   3. Let x become oor(x).
   4. Let x become x divided by 10n.
6. Display the modifi ed state of x.

Th e following trace of program execution simulates what will happen to x when it starts at

|  |  |
| --- | --- |
| 3.4567 and is rounded to two decimal places.  **Round 3.4567 to Two Decimal Places** |  |
| x = x \* 10n = 3.4567 \* 102 | = 345.67 |
| x = x + 0.5 = 345.67 + 0.5 | = 346.17 |
| x = oor(x) = oor(346.17) | = 346 |

x = x / 10n = 346.17 / 100.0 = 3.46

#### SELF-CHECK

3-7 Trace the same algorithm with the diff erent example problem of rounding 9.99 to one decimal place. What is the result? Write the new value for x in the space provided (x changes state four times after being input).

Algorithm

x n 1. Prompt for the number to round (call it x). ? ?

1. Input x. 9.99 ?
2. Prompt for the number of decimal places (call it n). 9.99 ?
3. Input n. 9.99 1
4. Let x become x \* 10n. \_\_\_\_\_ 1
5. Add 0.5 to x. \_\_\_\_\_ 1
6. Let x become oor(x). \_\_\_\_\_ 1
7. Let x become x divided by 10n. \_\_\_\_\_ 1
8. Display the modifi ed state of x. \_\_\_\_\_ 1

##### 3.2.3 IMPLEMENTATION

Th e complete C++ source code version is a translation of the previous algorithm. Notice that the algorithm steps are embedded as comments in the source code to show how each was translated into C++.

// Round a given number to a speci c number of decimal places

#include <iostream> // For cin and cout #include <cmath> // For pow(10, n) and oor(x) using namespace std;

int main() {

// Declare objects identi ed during analysis double x = 0.0; double n = 0.0;

// Algorithm step number:

// Input cout << "Enter number to round : "; // 1. cin >> x; // 2. cout << "Enter number of decimal places : " ; // 3. cin >> n; // 4.

// Process (Round x to n decimals) x = x \* pow(10, n); // 5a. x = x + 0.5; // b. x = oor(x); // c. x = x / pow(10, n); // d.

// Output (Display the modi ed state of x) cout << "Rounded number : " << x << endl; // 6.

return 0; }

###### Dialogue

Enter number to round : ***3.4567***

Enter number of decimal places : ***2***

Rounded number : 3.46

#### SELF-CHECK

3-8 List three more test cases for the rounding program above.

3-9 What is the fi nal state of x after the user enters 3.15 for x and 1 for n?

3-10 Given the table “A Partial List of cmath Functions" on p. 56, fi nd a slightly diff erent algorithm that accomplishes the same task where 3.15 rounded to one decimal place would be

3.1 instead of 3.2. (*Hint*: Consider subtracting 0.5 rather than adding it.) 3-11 What values are returned with these function calls?

|  |  |
| --- | --- |
| a. pow(2.0, 4.0) | d. oor(1.0) |
| b. sqrt(16.0) | e. fabs(-23.4) |
| c. ceil(-1.7) | f. pow(4.0, 2.0) |

### 3.3 CALLS TO DOCUMENTED FUNCTIONS

All functions must fi rst be declared with a function heading so the compiler can determine whether the function calls are correct. Th ese function headings also help the programmer properly call those functions. If you peruse the fi le cmath, you will see many such function headings. Th is section concentrates on how to read these function headings and how to use other documentation describing what a particular function expects and what that function will do. Th ese are called the function’s preconditions and postconditions, respectively.

#### 3.3.1 PRECONDITIONS AND POSTCONDITIONS

For a function to behave properly, certain conditions are presumed. Consider the sqrt function, which presumes that the argument is greater than or equal to 0.0. Th e *preconditions* of a function state assumptions made about arguments to the function. If the preconditions are not met, all bets are off —the function’s behavior is undefi ned. Some systems cause program termination with an arithmetic overfl ow error. Other systems may return values such as -1.#IND or nan, which represents the value “Not a Number.” Th e function call must satisfy the preconditions in order to have predictable results.

Th e other part of the contract is the *postconditions*—the statements that describe what the function does if the preconditions were met. Th e pre- and postconditions are often written as part of the function documentation. For example, here is the sqrt function documented with preconditions and postconditions:

double sqrt(double x)

// precondition: x is not negative (x >= 0)

// postcondition: Square root of x replaces the function call

Th e comments indicate the argument must be greater than or equal to 0.0. If this precondition is met, the square root of that argument is returned to the *client*—the code that called the function. If the precondition is not met, the result is undefi ned.

|  |  |
| --- | --- |
| **Function Call** | **Return Result** |
| sqrt(4.0) | Th e precondition was met: 2.0 is returned. |
| sqrt(-1.0) | Th e precondition was not met. Th is function call may return nan (not a |
|  | number) |

Another implied precondition to calling a function and getting predictable results is this: the client code must supply a proper type of argument. For example, the ceil function takes one double argument. Th is implies the argument must be convertible to a double, which includes short, int, oat, and even char. For example, ceil will not accept a string argument. Th is could be stated as an obvious precondition like this:

double ceil(double x)

// precondition: Argument must be convertible to a double

// postcondition: Return the smallest integer >= x

However, this information is implied in the parameter declaration. Th e compiler detects improper argument. From now on, such function preconditions will not be written.

Th e compiler does not detect preconditions. For example, it is syntactically correct to have a program with this:

cout << sqrt(-1.0); // Return depends on the system in use

From now on, the label for preconditions will be abbreviated as pre: and for postconditions as post:. Th e same function (ceil) may now be documented as follows:

double ceil(double x) // post: Return the smallest integer >= x

It should be noted that the use of pre: and post: after the function headings is not required. Diff erent people document functions in their own ways. Th is documentation is particular to this textbook.

#### 3.3.2 FUNCTION HEADINGS

Pre- and postconditions help programmers determine the proper use of functions. If provided as documentation, they are usually listed after the function heading. Th e function heading also provides very important usage information such as the type of value returned and the number of arguments required by the function. Here is the general form of a function heading:

##### General Form 3.2 *Function Heading*

*return-type function-name* (*parameter-1, parameter-2, parameter-n*)

Th e *return-type* may be any valid C++ class or the keyword void. A void function does not return anything. Th e parameters between ( and ) may either be value parameters, reference parameters, or const reference parameters. Value parameters are up fi rst.

A function may require one or more arguments. Values are passed to functions by adding value parameters of this form:

©

##### General Form 3.3 *Value Parameter*

*class-name identi er*

**Examples of Standard C++ Function Headings:**

int isapha(int c) int tolower(int c); double round (double x); double remainder(double numerator, double denom);

Function headings specify the type of value returned by the function, the function name, and the number of arguments the programmer must supply. Th e class of arguments required is specifi ed as the *class-name* of each parameter between the parentheses. For example, because the parameters in pow below, x and y, are declared as double, one can ascertain that the type of each argument in calls to pow must be double, or at least convertible to double—an integer for example.

double pow(double x, double y)

// pre: When y has a fractional part, x must be positive

// When y is an integer, x may be negative

// post: Returns x to the yth power

Also note that the function name is pow and its return type is double.

Although the complete implementation of the pow function is not present, the information supplied by the preconditions, the postconditions and the function heading are enough to eff ectively utilize the function.

To summarize, a function heading with pre- and postconditions provides the following information:

1. the *return-type* that provides the type of value returned by the function
2. the *function-name* that begins a valid function call
3. the *parameter-list* that provides the number and class of arguments required in the function call
4. pre:, which describes what must be true before calling this function
5. post:, which describes what the function does if the preconditions are met

In addition to revealing information to programmers, function headings supply information to the compiler to verify the validity of every attempt to call that function. Th e compiler informs you if a function is not called properly. Consider the oor function heading:

double oor(double x)

// post: Returns the largest integer <= x

Th e return type is double. Th is means that a double replaces any valid call to oor. Th erefore, the function call can be used wherever a double value is legal—in an arithmetic expression, for instance. Also present is the function name oor—very important information for eff ectively calling this or any particular function. Th e parameter list shows one double parameter named x. So the client code must supply exactly one numeric argument to properly call oor. For example, the following is a valid call to double and it is used in a proper spot in the code:

double x; x = oor(5.55555); // This assignment is okay However, these function calls are invalid:

string s; s = oor(5.5555); // Error: oor doesn't return a string cout << oor(1.0, 2.0); // Error: too many arguments cout << oor("wrong type"); // Error: wrong type argument cout << oor(); // Error: too few arguments

#### SELF-CHECK

3-12 Given the following function heading, write “valid” for each correct function call or explain why it is not valid.

double ceil(double x)

|  |  |
| --- | --- |
| a. ceil(1.1) | d. ceil("Ceila") |
| b. oor(2.9) | e. ceil -0.1 |
| c. ceil(1.2, 3.0) | f. ceil(-3) |

3-13 Describe the error in each of the following attempts at function headings:

1. double f ( x )
2. int smaller(int n1 int n2)
3. toUpper(string s)
4. myClass g()
5. int twoStrings(string s1, string s2,)
6. unknownType initialize(" lename.dat")

Use the following documentation for the questions that follow:

double oor(double x)

// post: The oor function returns a oating-point value

// representing the largest integer that is less than or

// equal to x

* 1. Write four function calls (with diff erent arguments) that would help explain how oor works to someone who has never seen it before.
  2. Write the values returned from each of the four function calls in your answer to the preceding question.

##### 3.3.3 ARGUMENT / PARAMETER ASSOCIATIONS

A function heading may list zero, one, two, and sometimes more parameters. If there is more than one, the parameters must be separated by commas. Th e next function heading has two parameters—str and x.

double twoParameters(string str, double x)

Exactly one argument of an acceptable class is required for each parameter listed in a function heading. Th erefore, precisely two arguments must be present in every call to twoParameters. Th e compiler will report an error if you call this particular function with any other number of arguments than two. Additionally, the type and position of the arguments must match the type and position of the parameters. For example, a double argument cannot be associated with a string parameter. Here are some examples of correct calls of twoParameters:

###### Valid Calls to the twoParameters Function

twoParameters("abc", 1.2); twoParameters("another string", 15); twoParameters("$", 3.4);

Th e following attempts to call twoParameters result in compile time errors:

|  |  |
| --- | --- |
|  | |
| **Error** | **Reason for Error** |
| twoParameters("a"); | Needs two arguments. |
| twoParameters("1.1", "2.2"); | Th e string “2.2” can’t be assigned to a double. |
| twoParameters(1.1, 1.1); | Th e number 1.1 can’t be assigned to a string. |
| twoParameters("a", 2.2, 3.3); | One too many arguments. |
| twoParameters; | Generate a warning. Statement has no eff ect. |

Arguments associate with parameters by position—fi rst argument to the fi rst parameter, second argument to the second parameter, and so on. For example, when twoParameters is called, the fi rst parameter is assigned the value of the fi rst argument and the second argument to the

function is copied into the second parameter x. When twoParameters is called with arguments "abc" and 1.2, like this:

int twoParameters(string str, double x)

result4 = twoParameters ("abc", 1.2); it’s as if these two assignment operations occur:

str = "abc"; x = 1.2;

Whatever happens inside twoParameters now depends on the values of these two parameters. Th e parameters are used inside the function to produce the return result.

#### SELF-CHECK

3-16 What value is sent to parameter str with twoParameters("1st", 1.2)?

3-17 What value is sent to parameter x with twoParameters("2nd", 3.4)?

Much can be deduced from a function heading when it is accompanied by the function pre- and postconditions. As review, here is the sin function heading complete with pre- and postconditions:

double sin(double x) // post: Returns the sine of x radians

Th e following information is ascertained:

* What happens: returns the sine of x radians
* Return type: double
* Function name: sin
* Number of arguments: one
* Type of argument: double (or an expression convertible to double)

Th e return results can now be determined (with the help of a scientifi c calculator in radian mode).

##### Function Call Return Result

sin(3.1415926/2.0) 1.0 sin(1.0) 0.8421 // Approximately sin(3.1415926) 5.35898e-08 // close to 0.0

#### SELF-CHECK

3-18 Given the following pow function from cmath, complete with precondition and postcondition documentation, determine the information below:

double pow(double x, double y) // pre: When y has a fractional part, x must be positive.

// When y is an integer, x may be negative.

// post: Returns x to the yth power

1. return type d. class of fi rst argument
2. function name e. class of second argument
3. number of arguments f. class of third argument
   1. Write one proper function call to pow.
   2. Is pow(-81.0, 0.5) a valid function call? What is the return value?
   3. Is pow(-10.0, 2) a valid function call? What is the return value?
   4. Is pow(2, 5) a valid function call? What is the return value?
   5. Is pow(4.0, 0.5) a valid function call? What is the return value?
   6. Is pow(5.0) a valid function call? What is the return value?
   7. Write a function heading that returns the fractional component of the fi rst number divided by the second number. Write appropriate pre- and postconditions. For example, remainder(5.0, 2.0) must return 0.5 and remainder(1, 3) must return 0.3333333.

##### 3.3.4 A FEW FUNCTIONS FOR int, char, AND bool

Some free functions work with the other primitive types. For example, the standard C++ library has free functions that can be used in an end-of-chapter programming project: min, max, and abs.

#include <iostream> using namespace std;

int main() {

cout << min(5, 7) << endl; cout << min(5.5, 7.7) << endl;

cout << max(5, 7) << endl; cout << max(5.5, 7.7) << endl; cout << abs(5 - 7) << endl;

return 0;

}

###### Output

5

5.5 7

7.7

2

Th e min and max functions are defi ned in such a way that the same function name can be used with diff erent types. Th ey can take either two int arguments or two double arguments, but not a mix.

C++ also has some methods that sound Boolean in nature because of names like islower and isdigit. Other functions seem as though they should have char parameters and return types because they convert characters to their upper or lower case equivalents. Consider the function heading for the free function islower when you #include<cctype> to have access to its set of functions to classify and transform individual characters.

int islower(int c);

Th is function checks whether c is a lowercase letter. It would seem that the parameter should be char and the return type bool like this:

bool islower(char ch); // This is not the function heading

However, C++ allows an int to be assigned to a char and vice versa. Arithmetic operations can have a mix of integers and character operands.

#include <iostream> using namespace std;

int main() {

int anInt = 'A'; // 'A' equals 65 char aChar = 67; // 67 equals 'C'

cout << "anInt: " << anInt << endl; cout << "aChar: " << aChar << endl;

cout << "aChar + anInt: " << (aChar + anInt) << endl; cout << "anInt % aChar: " << (anInt % aChar - 2) << endl;

return 0; }

###### Output

anInt: 65 aChar: C aChar + anInt: 132 aChar % anInt: 63

More confusion may occur because C++ considers true to be 1 and false to be 0.

#include <iostream> using namespace std;

int main() {

bool aBool = 1; // C++ allows assignment of int to bool int anotherBool = false; // and a bool literal to an int cout << aBool << " " << anotherBool << endl;

return 0; }

###### Output

1 0

Th e output shows true prints as 1 and false as 0.

If you need to classify if a char is an alphabetic letter like “A” or “a”, or a digit such as “9” or “3”, you can use one of the free functions from <cctype>. Th is program shows three more cctype functions:

#include <iostream>

#include <cctype> // For isalpha isblank isdigit using namespace std;

int main() { char ch = 'a';

cout << "isalpha('" << ch << "')? " << isalpha(ch) << endl; ch = '?';

cout << "isalpha('" << ch << "')? " << isalpha(ch) << endl;

ch = ' ';

cout << "isblank('" << ch << "')? " << isblank(ch) << endl; ch = 'N';

cout << "isblank('" << ch << "')? " << isblank(ch) << endl;

ch = 'P'; // Oh, not zero

cout << "isdigit('" << ch << "')? " << isdigit(ch) << endl; ch = '5';

cout << "isdigit('" << ch << "')? " << isdigit(ch) << endl;

return 0;

}

###### Output

isalpha('a')? 1 isalpha('?')? 0 isblank(' ')? 1 isblank('N')? 0 isdigit('P')? 0 isdigit('5')? 1

Th e toupper and tolower functions convert a character to its lowercase or uppercase equivalent. Because the return type for both is int instead of char, the functions are cast to char with the code (char). Otherwise the output from this program would have been 88 97.

#include <iostream>

#include <cctype> // For toupper and tolower using namespace std;

int main() { char lower = 't'; char upper = 'A';

// (char) makes sure we the character, not the int cout << (char)toupper(lower) << endl; // Cast required cout << (char)tolower(upper) << endl; // to see chars

return 0; }

###### Output

T a

### CHAPTER SUMMARY

* You have been confronted with a large variety of details concerning the C++ programming language, expressions, program development, function calls, and the types of errors that occur during program development. Th is can be somewhat overwhelming at fi rst, especially if you have never programmed before. However, most of these details are necessary for implementation of even the simplest program.
* #include<cmath> provides access to many mathematical and trigonometric functions. #include<cctype> provides access to a set of functions that classify and transform individual characters.

Exercises

* Functions that have a return type of double can be used wherever a double (or fl oating-point expression) can be used. Many of the cmath functions return double.
* Most cmath functions require one numeric argument; pow requires two.
* Preconditions and postconditions represent a contract between the function and the client code that calls the function. Th is documentation and other forms of documentation are intended to help someone understand what the function does.
* Th e function heading itself provides vital usage information such as the return type, the func-tion name, and the number of parameters so the programmer knows how many arguments to include in the call.
* Arguments are associated with parameters by position. It doesn’t matter what names are used. Th e fi rst argument is associated with the fi rst parameter, the second argument with the second parameter, and so on.
* Arguments passed to parameters are like assignment statements. Th e argument must be com-patible with the parameter (the same type). Passing a double to an int results in loss of value.

### EXERCISES

1. Write the return result for each function call or explain the error.

|  |  |
| --- | --- |
| a. pow(3.0, 2.0) | g. fabs(-123.4) |
| b. pow(-2, 5) | h. sqrt(-1.0) |
| c. ceil(1.001) | i. sqrt(sqrt(16.0)) |
| d. ceil(-1.2) | j. ceil 1.1 |
| e. pow(16.0, 0.5) | k. oor() |
| f. pow(-16.0, 2) | l. sqrt(0) |

1. Use these initializations to evaluate the expressions that follow:

double x = 5.0; double y = 7.5;

|  |  |
| --- | --- |
| a. sqrt(x - 1.0) | e. oor(y + 0.5) |
| b. ceil(y - 0.5) | f. pow(x, 3.0) |
| c. sqrt(y - x + 2.0) | g. fabs(y - x) |
| d. pow(10, 2) | h. pow(10, 3) |

1. What is the value of pow(4, pow(2, 3))?
2. Write an algorithm that shows the range of a projectile. Th e formula is

*range* = sin(2 \* *angle*) \* *velocity*2 / *gravity*

where *angle* is the angle of the projectile’s takeoff (in radians), *velocity* is the initial velocity of the projectile (in meters per second), and *gravity* is acceleration due to gravity at 9.8 meters per second.

1. What happens if the client program does not satisfy the preconditions of a called function?
2. What information do postconditions provide?
3. Which of the following represent valid function headings?

|  |  |
| --- | --- |
| a. int large(int a, int b) | d. int f(a, int b) |
| b. double(double a, double b) | e. double f() |
| c. int f(int a; int b;) | f. string c(string a) |

1. Name three possible return types from a C++ function (there are many).
2. Given the following function heading with pre- and postconditions, write six function calls (with diff erent arguments) that would adequately test fmod and would also help explain how fmod works to someone who has never seen it before.

double fmod(double x, double y)

// post: Calculates the oating-point remainder.

// fmod returns the oating-point remainder of x / y.

// If the value of y is 0.0, fmod returns Not a Number.

// Header required: <cmath>

1. Write the values returned from each of the six function calls in your answer to the preceding question.

### PROGRAMMING TIPS

1. When calling existing functions, supply the correct number and type of arguments. Th e function heading and documentation, if present, provide this information. Count the number of parameters between ( and ). Make sure each associated argument is the same type, or convertible to that type. An int can be assigned to a oat, a oat to a double, an int to a long, for example.
2. Don’t mix arguments types with min and max functions. max(2, 3.0) and min(1.0, 4) are compile time errors.
3. Th ree C++ types appear to be the same. C++ allows integer literals to be treated as character literals and vice versa. Also, underneath, false is 0 and true is 1. Th e reason there is no output here is because aChar is storing the non-printable char value of 1.

Programming Projects

char aChar = true; // assign 1

cout << ">" << aChar << "<" << endl; // Output: ><

1. If you do not have the line using namespace std; you will have to prepend std:: before every occurrence of the cmath function you use.

#include <iostream> // For cout

#include <cmath> // for ceil and oor

// using namespace std; Without this, prepend with std::

int main() {

std::cout << std::ceil(5.99) << std::endl; // 6 std::cout << std:: oor(5.99) << std::endl; // 5 }

**PROGRAMMING PROJECTS**

## 3A cmath FUNCTIONS

Write a program that allows the user to enter any number. After an appropriate label, show the return value from each of the following functions (assume x represents the number input by the user):

1. the square root of x
2. x to the 2.5 power
3. the ceiling of x
4. the fl oor of x
5. the absolute value of x

Your dialogue should look like this:

Enter a number for x: ***2.5*** sqrt(x) : 1.5814 pow(x, 2.5) : 6.25 ceil(x) : 3 oor(x) : 2 fabs(x) : 2.5

## 3B CIRCLE

Write a C++ program that inputs a value for the radius of a circle (r) from the keyboard and then outputs the diameter, circumference, and area of the circle. Use the pow function to compute the area.

* *diameter* = 2 \* *radius*
* *circumference* = *pi* \* *diameter*
* *area* = *pi* \* *radius* 2

Initialize PI as a constant object with the value of 3.14159. Your dialogue should look like this (*Note:* Output of fl oating-point numbers varies among C++ compilers, so your output might be slightly diff erent—especially in the number of decimal places shown for Circumference and Area):

Enter Radius: ***1.0***

Diameter: 2.0

Circumference: 6.28318

Area: 3.14159

Run your program with radius = 1.0. Verify that your values for circumference and area match the preceding dialogue. After this, run your program with the input radii of 2.0 and 2.5 and verify that the output is what you expect.

## 3C MORE ROUNDING

Write a program that asks the user for a number and displays that number rounded to zero, one, two, and three decimal places. Your dialogue should look like this:

Enter the number to round: ***3.4567***

3.4567 rounded to 0 decimals = 3

3.4567 rounded to 1 decimal = 3.5

3.4567 rounded to 2 decimals = 3.46

3.4567 rounded to 3 decimals = 3.457

## 3D RANGE

Write a program that determines the *range* of a projectile using this formula: *range* = sin(2 \* *angle*) \* *velocity*2 / *gravity*

where *angle* is the angle of the projectile’s path (in radians), *velocity* is the initial velocity of the projectile (in meters per second), and *gravity* is acceleration at 9.8 meters per second (a constant). Th e takeoff angle must be input in degrees. Th erefore you must convert this angle to its radian equivalent. Th is is necessary because the trigonometric function sin(x) assumes the argument (x) is an angle expressed in radians. An angle in degrees can be converted to radians by multiplying the number of degrees by π/180 where π ≈ 3.14159. For example, 45° = 45 \* 3.14159/180, or 0.7853975 radians. Th e velocity is presumed to be input in meters per second. Make your interactive dialogue look like this:

Takeoff angle (in degrees)? ***45.0***

Initial velocity (meters per second)? 100.0

Range = 1020.41 meters

Programming Projects

## 3E TIME TRAVEL

For astronauts approaching the speed of light in a spaceship, time passes more slowly. Also, the weight of their spaceship increases. Th e Lorentz factor indicates this change in time and weight depends on the spaceship velocity v

1 *factor* =

*c*

2

*v*

2

1

–

where *v* is velocity and *c* is the speed of light (299,792,458 meters/second). Th e *factor* can be used to compute how much perceived time is decreased for the astronauts and by how much the weight of their spaceship is increased. For example, at 74948114.5 meters per second (1/4 the speed of light) the factor is 1.038. Time’s passage is reduced by the factor and the weight is increased by that factor.

Write a program that reads the weight of the spaceship on earth (90,000 kilograms), the fraction of the speed of light (0.25, less than 1.0), and the distance to travel in light years (Alpha Centauri is about 4.35 light years away).

**Dialog 1:**

Weight of spaceship on earth in kilograms? ***90000***

Velocity as a fraction of the speed of light 0.0 to 1.0? ***0.25***

Distance to travel in light years? ***4.35***

Travel time: 4.35 light years

Perceived time: 4.21187 years

Earth weight of spaceship: 90000 kg

Weight of spaceship: 92951.6 kg traveling at 7.49481e+07 m/s **Dialog 2:**

Weight of spaceship on earth in kilograms? ***90000***

Velocity as a fraction of the speed of light 0.0 to 1.0? ***0.9***

Distance to travel in light years? ***4.35***

Travel time: 4.35 light years

Perceived time: 1.89612 years

Earth weight of spaceship: 90000 kg

Weight of spaceship: 206474 kg tra veling at 2.69813e+08 m/s

© 2018 Franklin, Beedle & Associates All rights reserved

**C H A P T E R F O U R**

# Implementing Free Functions

## SUMMING UP

C++ has many free functions that any programmer can reuse. Th ese documented functions can be found at **http://www.cplusplus.com/reference**. It is considered good practice to compose code into well-defi ned functions, test them, and then call them from our programs. Th is makes for more readable programs. However, C++ does not provide all functions needed by everyone in all situations.

## COMING UP

Th is chapter shows how to write your own functions. After studying this chapter, you will be able to

* implement free functions
* pass values to your functions as input
* return values from your functions as output
* test your new functions
* begin to understand the scope of objects and functions

### 4.1 IMPLEMENTING YOUR OWN FUNCTIONS

Functions, such as those presented in the previous chapter like min, max, abs, round, and sqrt, are defi ned as a function heading followed by a block.

#### General Form 4.1 *Free Function*

*function-heading block*

A block begins with { and ends with }. It contains components such as variable declarations and executable statements.

77

#### General Form 4.2 *Block*

{ *object-initializations*

*statements*

}

Functions get their input via the arguments in the function call. Th e function uses these input values to compute a result, which is then returned to the caller. You have seen how arguments are associated with parameters to get input into the function. Functions communicate values to the calling code through the return statement.

#### General Form 4.3 *Return Statement*

return *expression* ;

Example of returning a value back to the calling code:

int minOf3(int a, int b, int c) {

// post: Return the smallest value amongst the 3 arguments return min(a, min(b, c)); }

When the return statement is encountered, the expression that follows return replaces the function call in the client code as program control returns to the place where the function was called. Th e following function named f implements the function *f* (*x*) = 2*x*2−1. Notice that the function must be coded before it can be called—the entire function f is located before the call to it from main.

#include <iostream> // For cout

#include <cmath> // For pow using namespace std;

double f(double x) { // post: Return 2 \* x \* x - 1 double result; result = 2 \* pow(x, 2) - 1.0; return result; }

int main() { double x, y; cout << "Input x: "; cin >> x;

// Call function f:

y = f(x);

cout << "f(" << x << ") = " << y << endl; return 0;

}

4.1: Implementing Your Own Functions

#### Dialogue

Input x: ***1.01*** f(1.01) = 1.0402

#### SELF-CHECK

4-1 What value is returned for each of these function calls? If there is an error, explain it. Use *f* (*x*) = 2*x*2−1 from the previous example.

|  |  |
| --- | --- |
| a. f(0.0) | d. f(1, 2) |
| b. f(-2.0) | e. f() |
| c. f(3) | f. f(5.8) |

In the next example, the function serviceCharge is declared with the double return type. Th e call to serviceCharge is replaced by a double value that depends on the values of the arguments.

// Call serviceCharge to determine a bank debit

#include <iostream> using namespace std; const double MONTHLY\_FEE = 5.00;

double serviceCharge(int checks, int ATMs) {

// pre: checks >= 0 and ATM >= 0

// post: Return a banking fee based on local rules double result;

result = 0.25 \* checks + 0.10 \* ATMs + MONTHLY\_FEE; return result;

int main() {

// 0. Initialize objects int checks; int ATMs;

}

double fee; // Stores the function return result

// 1. Input

cout << "Checks this month? "; cin >> checks;

cout << "ATMs this month? "; cin >> ATMs;

// 2. Process

fee = serviceCharge(checks, ATMs); //Call to serviceCharge

// 3. Output

cout << "Fee: " << fee << endl;

return 0; }

##### Dialogue

Checks this month? ***17***

ATMs this month? ***9*** Fee: 10.15

Here is what happens when the preceding program runs:

1. Th e user is asked to supply input for the number of checks and ATM transactions.
2. Th e values of the arguments (17 and 9) are assigned to the parameters of serviceCharge (checks = 17 and ATMs = 9). Th ese particular values will be used by the function to return the proper monthly bank fee.
3. Th e statements in serviceCharge execute.
4. Th e return is encountered in serviceCharge.
5. Th e function call serviceCharge(checks, ATMs) in main is replaced by the returned value of 10.15.
6. Th e function’s return value is assigned to fee.
7. Th e fee is displayed.

##### 4.1.1 TEST DRIVERS

When a function requires arguments, it is not unusual to have the same variable name declared in two diff erent places. Consider the previous program that declares checks and ATMs in main and also as parameters within the function serviceCharge. Th e objects declared in main are used to obtain user input. Th e parameters declared in serviceCharge obtain input from main. Although they have the same names, they are diff erent variables.

Sometimes the duplication of parameter names in main is not required. In the next program, you’ll see there is no user input, so the duplicated objects are not necessary. Instead, the arguments used to test the function are constants. Rather than being assigned to another object, the program simply displays the return results. Th e only purpose of this program is to test the function—to verify that the return values are what was expected. Th is is a good thing to do before the function becomes incorporated into a larger program. In fact, many of the programming problems ask you to carry out this form of testing.

4.1: Implementing Your Own Functions

// The main function makes several calls to test a new function

#include <iostream> using namespace std; const double MONTHLY\_FEE = 5.00;

double serviceCharge(int checks, int ATMs) {

// pre: checks >= 0 and ATM >= 0

// post: Return a banking fee based on local rules double result;

result = 0.25 \* checks + 0.10 \* ATMs + MONTHLY\_FEE; return result; }

int main() {

// Test drive serviceCharge // Sample problems:

cout << serviceCharge(0, 0) << endl; // 5.0 cout << serviceCharge(1, 0) << endl; // 5.25 cout << serviceCharge(0, 1) << endl; // 5.1 cout << serviceCharge(1, 1) << endl; // 5.35 return 0; }

###### Output

5

5.25 5.1

5.35

Th is version of main is called a test driver. A *test driver* is a program with the sole purpose of testing a new function. Functions like serviceCharge, sqrt, and pow are intended to be small parts of much bigger programs. Th erefore all functions should be thoroughly tested before they are reused. Th e four sample problems shown above were predicted and documented in comments. Th is has been a successful test of the serviceCharge function.

##### 4.1.2 FUNCTIONS WITH ONLY A RETURN STATEMENT

Some functions are so simple, they may contain only a return statement.

double serviceCharge(int checks, int ATMs) {

// pre: checks >= 0 and ATM >= 0

// post: Return a banking fee based on local rules return 0.25 \* checks + 0.10 \* ATMs + MONTHLY\_FEE; }

However, this textbook will often use the following convention in addition to the above shortcut (one return statement):

1. Declare a local variable named result to be the same type as the function’s return type.
2. Store the desired value in result.
3. Return result.

Th is is extra work for simple functions. However, this pattern will help when the processing gets more complex, beginning in Chapter 7, “Selection.”

Also, the extra two lines of code are likely to prevent you from making a very common mistake. Perhaps because other languages use this technique to return values or perhaps because it simply appears to be the right thing to do, it is common to try to assign a value to the function name. Th is is a compile time error. You can only assign values to variables.

double serviceCharge(int checks, int ATMs) {

// You cannot assign a value to a function name serviceCharge = 0.25 \* checks + 0.10 \* ATMs + 5.00; // ERROR return serviceCharge; // ERROR, attempt to return function }

If you do make this common mistake, the compiler will tell you. Fix the error by placing an expression of the correct type, double, after the return.

#### SELF-CHECK

4-2 Given the following function f1, what value is returned with f1(9.0)?

double f1(double x) {

// pre: x is zero or positive, but not 1.0

// post: Return f(x) = (square root of x) / ( x - 1.0 ) return sqrt(x)/(x - 1.0);

}

4-3 Does the function call f1(-1.5) satisfy the previous function’s precondition? What happens during a call to f1 with a negative number for an argument?

4-4 Describe how to fi x the error in each function.

1. double f1(int j);{ d. double f4(double x){ return 2.5 \* j; f4 = 2.5 \* x;

} }

1. double f2(int) { e. double f5(double x) { return 2.5 \* j; return double;

} }

1. double f3(int x) { f. int f6(string s) { return 2.5 \* j; return s;

} }

4-5 Write a function times3 that returns a value that is three times greater than the argument (times3(2.0) should return 6.0).

### 4.2 ANALYSIS, DESIGN, AND IMPLEMENTATION

Rather than writing a program, consider a problem that implements a function that may be a very small part of a large program. It may represent just one step of an algorithm, but it is frequently called.

*Problem:* Compute the distance between two points.

#### 4.2.1 ANALYSIS

Recall that the analysis phase of program development involves determining input and output. Also recall that while developing computer-based solutions to problems involving the IPO algorithmic pattern, the developer determines the output that must be sent to the user and also determines the input required from the user. Replace the word *user* with *client* in the preceding sentence, and the IPO pattern can be applied again to assist the design of functions. Except now the output from the function is expressed in the return statement and the input is expressed in terms of the argument/parameter associations. Here is a generalized IPO algorithm as it relates to functions instead of programs.

##### IPO Pattern Applied to Functions

Input: Input values to the function via argument/parameter associations.

Process: Compute the result to be returned.

Output: Return the result.

Sample problems are a good way to confi rm understanding of a problem. Sample problems also provide expected results that can be compared to program output during program testing. It is a good idea to develop sample problems for new functions. Th is will help you decide what the function needs as input and, therefore, the number and class of parameters to write in the function heading. Th e sample problems also provide the expected output of the test driver. Four doubles are required to compute the distance between two points (*x*1, *y*1) and (*x*2, *y*2) using this formula:

*distance =* ( *x*1 *– x*2)2+ ( *y*1 *– y*2)2

Here are some predicted outputs for a few sets of values for *x*1, *y*1, *x*2, and *y*2.

##### Sample Problems

***x***

**1**

***y***

**1**

***x***

**2**

***y***

**2**

**Distance**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1.0 | 1.0 | 2.0 | 2.0 | 1.414 |
| 0.0 | 0.0 | 3.0 | 4.0 | 5 |
| -5.7 | 2.5 | 3.3 | –4.7 | 11.5256 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

Th e IPO pattern is now applied to functions as follows:

Input: Input two points at the function call (*x*1, *y*1) and (*x*2, *y*2)

Process: Evaluate ( *x*1 *– x*2)2+ ( *y*1 *– y*2)2

Output: Return the result

#### 4.2.2 DESIGN

Th e designer must decide how many and what class of parameters are required for a function. In this example, four values are needed to represent the two input points (x1, y1, x2, and y2). Th e best class of parameters is double to allow points such as 5.62 and -9.864. Th e best return type is double. With the square root function involved, double helps return precise answers. A good function name is distance—it describes what the function does. Th is leads to a function heading with a return type of double, a function name of distance, and four descriptively named double parameters.

double distance(double x1, double y1, double x2, double y2) // post: Return distance between two points (x1, y1) and (x2, y2)

Now, within the body of the function (the block), the parameters x1, y1, x2, and y2 can be used in the distance formula to compute the result.

result = sqrt(pow((x1 - x2), 2) + pow((y1 - y2), 2));

#### 4.2.3 IMPLEMENTATION

Th e following program puts this all together with a main function written exclusively to test the function (a test driver):

// Call distance four times

#include <iostream> // For cout

#include <cmath> // For sqrt and pow using namespace std;

double distance(double x1, double y1, double x2, double y2) {

// post: Return the distance between any two points double result;

result = sqrt(pow((x1 - x2), 2) + pow((y1 - y2), 2));

return result; }

int main() {

// Test drive the distance function cout << "(1.0, 1.0) (2.0, 2.0): "

<< distance(1.0, 1.0, 2.0, 2.0) << endl; cout << "(0.0, 0.0) (3.0, 4.0): "

<< distance(0.0, 0.0, 3.0, 4.0) << endl; cout << "(-5.7,2.5) (3.3,-4.7): "

<< distance(-5.7,2.5, 3.3,-4.7) << endl; cout << "(0.0, 0.0) (0.0, 0.0): "

<< distance(0.0, 0.0, 0.0, 0.0) << endl; return 0; }

##### Output

(1.0, 1.0) (2.0, 2.0): 1.41421

(0.0, 0.0) (3.0, 4.0): 5

(-5.7,2.5) (3.3,-4.7): 11.5256

(0.0, 0.0) (0.0, 0.0): 0

Argument/parameter associations are analogous to program input. For example, in the second call to distance, the four values are fi rst copied as input to the function distance.

double distance(x1, y1, x2, y2)

distance(0.0, 0.0, 3.0, 4.0)

Control then transfers to the function where the parameters are used to compute the distance between the two points represented by those arguments. Here is the step-by-step computation:

sqrt(pow((x1 - x2), 2) + pow((y1 - y2), 2)) sqrt(pow((0.0-3.0), 2) + pow((0.0-4.0), 2)) sqrt(pow(( -3.0 ), 2) + pow(( -4.0 ), 2)) sqrt( 9.0 + 16.0 ) sqrt( 25.0 ) 5.0

Th e four arguments become input to the function as the system copies the value of each argument to its associated parameter. Th is particular mode of argument parameter association is known as *pass by value* because the values are passed to the function. When a function requires input of small objects such as double or int, write the function heading with value parameters of this form:

*class-name identi er*

#### 4.2.4 TESTING

It is a good idea to test functions individually. Th e previous program did just that. It didn’t do anything else. Th e only purpose for this particular program was to call distance with diff erent sets of arguments and display the return results. Notice the similarity of the four calls to distance and the sample problems. Th e arguments are the input to the function. Th e return result should match the expected results.

It is recommended that you test new functions with a test driver.

#### 4.2.5 SCOPE OF IDENTIFIERS

Th e *scope* of an identifi er is the part of a program from which an identifi er can be referenced. Th e scope of an identifi er extends from the point of the identifi er’s declaration to the end of the block in which it is declared. Recall that a block is delimited by the left and right braces, { and }. For example, the scope of local in the following program is the function one. Th is local, declared in one, cannot be referenced from outside the block in which it was declared, including main.

// Illustrate the scope of an object

#include <iostream> using namespace std; const int maxValue = 9999;

void one() { int local = -1;

// The scope of local is this function cout << local << endl;

// maxValue is known after its declaration including here: cout << maxValue << endl; }

int main() {

// The scope of local is limited to one() so this is an error: local = 5;

// Function one() is known everywhere after its declaration one();

// maxValue is known everywhere after its declaration cout << maxValue << endl; return 0; }

When a variable is declared outside of a block—as in the case of maxValue—its scope begins at the point of declaration and extends to the end of the fi le. Identifi ers declared in a block can be referenced only from within that block. Th ese are *local* identifi ers. Identifi ers declared outside of a block (such as maxValue) are said to be global. *Global identifi ers* may be referenced from any subsequent part of the fi le after its declaration, unless that identifi er is declared again (redeclared) within another block. In this case, the identifi er that was declared fi rst becomes hidden from the block in which it is redeclared. Since many blocks often exist within one program, determining the scope of an object can be somewhat complicated. For example, try to predict the output of the following program, which includes three diff erent declarations of the int variable identi er:

// This program is a tedious test of your ability to

// determine which of the three int variables named // identi er are being referenced at any given point.

#include <iostream> using namespace std; const int identi er = 1; // Global variable

void one() {

// This is a reference to the global identi er cout << "identi er in one(): " << identi er << endl;

}

void two() {

int identi er = 2; // local to two()

cout << "identi er in two(): " << identi er << endl;

}

int main() { int identi er = 3; // local to main() one(); two(); cout << "identi er in main(): " << identi er << endl; return 0; }

##### Output

identi er in one(): 1 identi er in two(): 2 identi er in main(): 3

When the function one is called, the global const int identi er = 1 is referenced. Th is global identi er can be referenced from within any function that does not declare another identifi er named identi er. Th erefore, the identi er that was declared fi rst and initialized to 1 is known (can be referenced) from one even though it was not declared inside one. But when a reference is made to identi er in function two, the global identi er is hidden because of the local identi er. To this point in program execution, one has caused the output 0, and two has caused 2 to be displayed. Th e fi nal statement in main references the identi er local to main—this identi er is initialized as 3.

Typically, a function will have one or more variables declared at the beginning of the block. Th ese variables are said to be local to the function because they may be referenced only from within the function. Th e same protection applies to the parameters of a function. Parameters are local variable declarations declared inside ( and ), rather than inside the function block. Parameters are assigned values passed to the function. Parameters can only be used inside the func-

tion block. Th e restriction provides safekeeping for the local objects so they are not accidentally altered from some other portion of a program.

void f1(double x) { int local = 0;

str = "A"; // Error attempting to reference main's local str }

int main() {

string str; // str is local to main

x = 5.0; // Error attempting to reference f1's parameter x local = 1; // Error attempting to reference f1's local return 0; }

#### SELF-CHECK

4-6 Use the partial program shown below to determine the functions from which each of the following identifi ers may be referenced. cin and cout are initialized in iostream so they are known after #include <iostream>.

// cout b cin MAX c f1 a d f2 main e

#include <iostream> using namespace std; const int MAX = 999;

void f1(int a) {

int b;

}

void f2(double c) {

double d;

}

int main() { int e;

return 0;

}

4-7 Name two things that may be declared local to a function.

4-8 If a variable is declared outside of a function, from where may it be referenced?

##### 4.2.6 SCOPE OF FUNCTION NAMES

Now what about function names? After all, they too are identifi ers. What is their scope? Like cin and cout, the scope of functions in an included fi le like cmath also extends to the end of their own fi le and any fi le with #include<cmath>. So functions such as sqrt, pow, ceil, and fabs may be called from within any block unless the function name is re-declared to be something else.

##### 4.2.7 GLOBAL IDENTIFIERS

Th e problems presented so far are not relatively complex. Th ey are certainly not large. You have probably been working pretty much on your own. However, when programs get large with a team, practice caution with scope.

Global identifi ers are known everywhere after they are declared. Th is opens them up for accidental alteration from anywhere in a very large program. It is diffi cult to ensure that no one will accidentally modify an object at the wrong time. So try to get in the habit of using local objects everywhere possible. Th is means you use parameters between ( and ) and objects between { and }. For example, main declares localX and localY locally.

int main() { double localX, localY;

// . . .

}

If you need to move data between functions, pass them as arguments. Th is means you must declare parameters rather than having some global x.

double f(double x) { // x is local to f double result; // result is local to f

// Do something with x . . .

}

If you need a value in many places throughout a program, make it const.

#include <iostream> using namespace std; const int MAXIMUM\_ENTRIES = 100; // ... a large program with many functions may follow

On the other hand, C++ often uses global identifi ers. Consider the fact that after including

<iostream>, cout is known everywhere, assuming using namespace std; is written before cout is referenced (left column following) or cout is qualifi ed with std:: (right column).

#include <iostream> using namespace std;

void f() { cout << "In f\n"; }

void g() { cout << "In g\n"; }

int main() { f(); g(); cout << "In main\n"; return 0; }

#include <iostream>

// Equivalent code with std:: void f() { std::cout << "In f\n"; }

void g() { std::cout << "In g\n"; }

int main() { f(); g(); std::cout << "In main\n"; return 0;

}

In eff ect, using namespace std; makes cout a global identifi er. Is this okay? Well, a lot of computer scientists believe so. Th ere is usually only one console, so any output to cout will go to the same console, no matter which function sends output to it.

### 4.3 void FUNCTIONS & REFERENCE PARAMETERS

Th e keyword void is used as the return type of functions that do not return anything. Instead of returning values back to the client, void functions are often employed to modify the state of the object(s) passed to them. Th is section shows a void function called swap, which modifi es two arguments. A function must use a reference parameter—with & added—to modify the state of the object(s) in the function call. Here is the general form.

#### General Form 4.4 *Reference Parameter*

*class-name* & *identi er*

#### Examples of reference parameters in function headings

void swap(double & parameterOne, double & parameterTwo) void changeFormat(ostream & cout)

A change to a reference parameter (with &) also modifi es the associated argument. Th e parameter name is a reference to—memory location of—the associated argument.

Although parameters typically obtain input from the caller, they can sometimes establish a stronger connection between argument and parameter. In this fi rst example of reference parameter usage, the swap function must modify two objects. Since only one value can be returned

4.3: void Functions & Reference Parameters

through a return statement, the function requires something besides the return statement to communicate more than one value back to the caller. Th is is accomplished when the special symbol & is placed before the parameter name in the function heading. Instead of receiving a copy of the argument, the function receives the memory location or reference to that argument.

When a change is made to a reference parameter, it will change the same object referenced by the argument. Th is is because the parameter and the argument are pointing to the same object in memory. For example, in the following program, when the swap function alters the parameters parmOne and parmTwo, the arguments argOne and argTwo are also pointing to that same modifi ed object:

// Notice the reference symbol & is in front of parmOne

// and parmTwo. Now a change to parmOne or parmTwo alters // the associated object that is the argument's value.

#include <iostream> using namespace std;

// Swap the values of any two int arguments.

// The & lets any change to the parameter alter it argument void swap(int & parmOne, int & parmTwo) { int temp = parmOne;

parmOne = parmTwo; // Change argument argOne in main parmTwo = temp; // Change argument argTwo in main }

int main() {

int argOne = 89; // argOne argTwo int argTwo = 76;

cout << argOne << " " << argTwo << endl; // 89 76 swap(argOne, argTwo); cout << argOne << " " << argTwo << endl; // 76 89

return 0; }

|  |  |
| --- | --- |
| **Output** |  |
| 89 | 76 |
| 76 | 89 |

If the ampersands (&) are removed from the program above, no change is made to the arguments in main. In this case, the values of argOne and argTwo would be passed by value, not by reference. Without the reference symbol &, the values of parmOne and parmTwo are changed locally, within swap only. Th e values of the associated arguments in main are unaff ected because they are diff erent objects.

Th e following fi gures illustrate the diff erence between reference and value parameters.

**Reference parameters:** *argument and parameter reference the same object*  parmOne = address of argOne and parmTwo = address of argTwo

void swap(int & parmOne, int & parmTwo) { parmOne

parmTwo ~~89~~ 76 parmOne points to the memory location of argOne and

} then changes contents from 89 to 76 in swap. Th is af-

int main() { 76 89 fects the same object pointed to by argOne.

argOne argTwo

} parmTwo points to the memory location of argTwo and

then changes contents from 76 to 89 in swap. Th is affects the same object pointed to by argTwo.

**Value parameters:** *a change to the parameter does not change the associated argument*  parmOne = 89 (value of argOne) and parmTwo = 76 (value of argTwo)

void swap(int parmOne, int parmTwo) {

parmOne ~~89~~ 76

parmTwo Since values are passed to the swap function, when the

76 89

}

swap occurs locally, it does not aff ect the variables in a

int main() { 89 diff erent function.

argOne

76

argTwo

} Values in main are not aff ected when their values are

“passed by value”.

Because a change to a reference parameter changes the argument, the argument must be a variable. Using a literal value or larger expression results in a compile time error.

swap(89, 76); // Error: Argument must be a variable

#### SELF-CHECK

4-9 Write the values of arg1 and arg2 at the moment when return 0; executes.

1. #include <iostream> using namespace std; void changeOr(int a, int b) { a = a \* 2 + 1;
   1. = 123;

}

int main() { int arg1 = 5;

4.4: const Reference Parameters

int arg2 = 5; changeOr(arg1, arg2); // arg1 \_\_\_\_ arg2 \_\_\_\_\_ return 0;

}

1. #include <iostream> using namespace std; void changeOr(int & a, int & b) { a = a \* 2 + 1;
   1. = 123;

}

int main() { int arg1 = 5; int arg2 = 5; changeOr(arg1, arg2); // arg1 \_\_\_\_ arg2 \_\_\_\_

return 0;

}

### 4.4 CONST REFERENCE PARAMETERS

You have now seen two of the three parameter passing modes in C++:

1. value parameters—for passing the values of small objects such as int
2. reference parameters—to allow a function to modify the state of one or more arguments
3. const & (reference) parameters—for safety and effi ciency

A const reference parameter is typically used to pass a “big” object that is not to be modifi ed by the function. A big object is one that requires a lot of memory, a very large string for example. To understand why programmers pass large objects by const reference, consider what happens when arguments are passed into functions.

When passed by value, the entire object is copied into another variable of the same size in the function, which requires twice the memory. For reference parameters, the address of the object is copied to the function, which is only four bytes of memory. In this case, the argument and the parameter reference the same object. For const & parameters, the address of the object is copied to the function, again only four extra bytes of memory needed. With const, any attempt to modify the parameter in the function will be fl agged as an error by the compiler. Th e const prevents accidental changes to the argument. Th e programmer adds const to avoid the bug of unknown modifi cation of an object in a diff erent scope.

|  |  |  |
| --- | --- | --- |
| **Pass by Value int f1(int j)** | **Pass by Reference int f2(string & b)** | **Pass by const Reference void f3(const int & n)** |
| Grab enough memory  to store the entire object and copy all bytes to the function. A change to the parameter has no eff ect on the argument. | Use four bytes of memory to store the address of the object and copy that address to the function. Use this when you need to modify the argument. It’s effi cient too. | const means the argument cannot be changed. Any attempt to change n results in a compile time error. Use this for effi ciency and safety. |
| f1 ***cannot*** modify the argument’s state. | f2 ***can*** modify the argument’s state. | f3 function ***cannot*** modify the argument’s state. Th is is effi cient. |

Th ere are two reasons to use const reference parameters. Th e fi rst is effi ciency—the program executes more quickly. Th e other consideration is better memory utilization—less memory is required to store the large object in the function. For example, passing a small object such as int by value only requires the function to allocate and then copy four bytes of memory. By contrast, one large object passed by value could require thousands of bytes. Th e program might exhaust available memory.

Additionally, every single byte of an argument passed by value will be copied to the function. Th e computer has to do a lot of unnecessary work. Th e program might run noticeably slower. Here are two alternatives to make any program more effi cient in terms of space (saves memory) and time (runs faster):

1. Pass big objects by reference—effi cient but somewhat dangerous.
2. Pass big objects by const reference—effi cient *and* safe.

Th e second option is recommended. Th e program now has much less work to do. When passed by values, the program must then wait until every single byte is copied from the caller to the function. If passed by const & reference, only four bytes are required while the safety of value parameters (cannot change the state of the argument) remains intact. Of course, if you are passing an argument to a function in order to modify the state, you must pass it by reference with &.

Attempts to modify objects passed by const reference result in compile time errors. Using const is an antibugging technique that will let the compiler tell us about accidental attempts to modify the const parameter. Any function that does not modify the object may still be called—string’s length function, discussed in the next chapter, for example. However, the compiler will fl ag any attempt to call functions such as string’s insert function because as the name implies, insert adds things to a string to modify the state of the string object. You cannot assign a new value to a const parameter either.

Chapter Summary

void addSomeStuff(**const** string **&** str) { cout << str.length() << endl; // Okay

str.insert(5, "xtra"); // ERROR: can not modify a const parameter str = "new string"; // ERROR: Can not assign to a const parameter }

However, when using value parameters only, you get no such error message. Th e argument’s object simply does not change.

|  |  |
| --- | --- |
| **Changing** **x** **in** **f** **does not change** **y** **in the main function** | **Th is code results in a compile time error such as** **“cannot modify a const object”** |
| #include <iostream> using namespace std;  double f(double x) { double result;  // This does not modify y x = x – 1.5; result = 2 \* x; return result; }  int main() { // Output:  // 8 double y = 5.0; // y: 5 cout << f(y) << endl; cout << "y: " << y << endl; return 0;  } | #include <iostream> using namespace std;  double f(const double & x) { double result;  // An error. Good!  x = x – 1.5; result = 2 \* x; return result; }  int main() { double y = 5.0; cout << f(y) << endl; cout << "y: " << y << endl; return 0;  } |

It should be noted that only a few objects will be passed by const reference until later. So you will only occasionally see a big object passed by const reference in the next several chapters. Also, value parameters will be more common than reference parameters.

### CHAPTER SUMMARY

* Functions perform some well-defi ned services and can have two-way communication through argument/parameter associations and the return statement. Th e client code supplies input values to the function as arguments. Th e result is returned via the return statement.
* Th ere are several new implementation issues related to functions such as the scope of identi-fi ers:
* All identifi ers must be declared before they can be referenced.
* Th e scope of an object is limited to the block where it is declared.
* Some identifi ers are not declared within a block. In this case, they are global identifi -ers. Examples of global identifi ers include function sqrt after #include <cmath>, and the global object std::cout after #include <iostream>.
* Th e scope of a parameter is limited to its function.
* Th e scope of a function begins at the function heading and continues to the end of its fi le, or the end of the fi le that included the function.
* Th ere are many details to remember when using argument/parameter associations.
* Th e number of arguments used in a function call must match the number of param-eters declared in the function heading.
* Th e void return type precedes the function name when no value is to be returned. You cannot return anything from a void function.
* When one value is to be returned from a function, a non-void return type must begin the function heading. Th e return statement must also be included in the function block. Th e expression in the return statement should be the same class as the return type.
* Sometimes a function needs input—that is what parameters are for. Sometimes a function must return something—that is what the return statement is for. Sometimes a function needs to modify objects in the client code—that’s what reference parameters are for.
* Th e argument used in a function call should usually be the same class as its associated parameter. Th ere are exceptions; for example, an int argument may be assigned to a double parameter with type conversion.
* Parameters intended only to receive copies of the argument values (input parameters) should be declared as value parameters without &.
* Reference parameters (with &) must be used if the intention is to modify the associated argument—a change to a reference parameter alters the object reference by the argument. A change to a value parameter does not.
* const reference parameters are used to pass large objects. Instead of consuming extra bytes of memory and copying that memory, the address is copied—because of &. However, the safety of value parameters is ensured by making the parameter const.

### EXERCISES

1. How many statements may be written in a block delimited by { }?
2. Which function is called fi rst when a C++ program executes?
3. May a function be called more than once?
4. Write the output generated by the following program:

#include <iostream> using namespace std;

4.1: Implementing Your Own Functions

double f2(double x, double y) { return 2 \* x - y; }

int main() {

cout << f2(1, 2.5) << endl; cout << f2(-4.5, -3) << endl; cout << f2(5, -2) << endl; return 0; }

5. Write the output generated by the following function:

#include <iostream>

#include <cmath> using namespace std;

double mystery(double p) { return pow(p, 3) - 1; }

int main() { double a = 3.0; cout << mystery( a) << endl; cout << mystery(4.0) << endl; cout << mystery( -2) << endl; return 0; }

1. Write a function double sumOf3 that returns the sum of any three doubles. For example, sumOf3(1.5, 2.2, 3.7) should return 7.4.
2. Write a function int maxOf4 that returns the largest of the four integer arguments. For example, maxOf4(99, 2, 99, -4) should return 99.
3. What is the scope of these identifi ers being referenced in the following code?

|  |  |  |  |
| --- | --- | --- | --- |
| a. std |  | f. | f |
| b. cin |  | g. | result |
| c. MAX |  | h. | s |
| d. aaa |  | i. | cout |

e. string

#include <iostream>

#include <cmath> using namespace std; const double MAX = 2.0;

double f(double aaa) { double result; result = pow(3.0, aaa); return result; }

int main() { string s = "a string"; cout << f(MAX); return 0; }

1. Will a change to a value parameter modify the associated argument?
2. Will a change to a reference parameter modify the associated argument?
3. Write the output generated by this program:

#include <iostream> using namespace std;

void changeArgs(double & x, double & y) { x = x - 1.1; y = y + 2.2; }

int main() { double a = 3.3; double b = 4.4;

cout << a << " " << b << endl; changeArgs(a, b); cout << a << " " << b << endl; changeArgs(a, b); cout << a << " " << b << endl; return 0; }

### PROGRAMMING TIPS

1. Here are some common mistakes made when writing functions:
   * Placing the semicolon at the end of a function heading:

string move(int n) ; // ERROR

{ // many errors agged here. Remove ; from line above }

Programming Tips

* + Assigning a value to the function name:

double f(double x) {

f = 2 \* x; // ERROR: Can not assign value to function return f; // ERROR: Can not return a function name }

Th e solution: Declare a local object, assign it the value, and return it. Or, in the case of simple functions, simply return the expression:

double f(double x) { return 2 \* x; } or do this when there is more going on inside the function:

double f(double x) { double result; result = 2 \* x; return result; }

* + Failing to return a value from a non-void function:

double f2(double x) { double result; result = 2 \* x;

// ERROR: f2 must return a number

}

* + Returning a value from a void function:

void foo(double x) { return 2 \* x; // ERROR }

1. Th ere are several ways that functions communicate with each other:
   * Th e caller can send values and objects to a function by value.
   * Th e caller sends objects as arguments to a function by reference when the function is de-signed to change the arguments.
   * Th e caller sends objects as arguments to a function by const reference to save time and memory when the function is not supposed to change the arguments.
   * Th e caller gets values back from a function via the return statement.
   * Th e caller gets values back from a function by having the function change arguments associ-ated with reference parameters changed in the function.
2. If you want two or more values back from a function, use reference parameters. Th e return statement returns only one thing. If you need more than one thing back from a function, use one or more reference parameters in addition to a return statement.

**PROGRAMMING PROJECTS**

## 4A SUM THREE

Write the function sumThree that returns the sum of three double arguments.

// Test drive sumThree int main() {

cout << sumThree(1.1, 2.2, 3.3) << endl; // 6.6 cout << sumThree(-1, -2, 3) << endl; // 0 return 0; }

## 4B ROUNDING TO n DECIMAL PLACES

Write a function named round that returns the value of its double argument rounded to the number of decimal places specifi ed as the second argument.

// Test drive round int main() {

// Arguments: number to round (-2.9), decimal places (0) cout << round(-2.9, 0) << endl; // -3 cout << round(-2.59, 1) << endl; // -2.6 cout << round(0.0059, 2) << endl; // 0.01 cout << round(1.23467, 3) << endl; // 1.235 cout << round(9.999999, 4) << endl; // 10 return 0; }

## 4C PAYMENT

Th e payment on a loan is a function of the interest rate, the number of payments (periods), and the amount borrowed. Pass these three values as arguments to a function payment that returns the loan payment. Th e function heading is provided for you along with a test driver. Round your answer to two decimal places (see Section 3.2). See if your answers match an online mortgage calculator.

#include <iostream> // For the cout object

#include <cmath> // For pow, which you de nitely need here using namespace std;

double payment(double amtBorrowed, double interestRate, int numPeriods) {

// TODO: Complete this function

} int main() { // Test drive payment

// 6.0 needs to be divided by 100 and then by 12 to become a monthly

// interest rate, The number of years (30) also needs to multiplied

Programming Projects

// by 12. The following test cases represent a monthly payment.

cout << payment(185000.00, 6.0/100.0/12, 30\*12) << endl; cout << payment(185000.00, 5.0/100.0/12, 30\*12) << endl; cout << payment(185000.00, 4.0/100.0/12, 30\*12) << endl;

return 0; }

Here is the formula used to calculate payments on a loan given the amount borrowed, interest rate for one period, and the number of periods:

(*Rate* 1)*Months*

*Payment = Amount* × *Rate* ×

(*Rate* 1)

## 4D POPULATION GROWTH PREDICTION

According to **http://www.census.gov/popclock/**, at the time of this writing the population growth in the United States can be predicted as follows:

One birth every *8 seconds*

One death every *13 seconds*

One international migrant (net) every *40 seconds*

Write a function that predicts the population for any number of days into the future when also given the current population. Th e following test driver must compile and generate the output shown in comments:

int main() {

cout << populationPrediction (320000000, 0) << endl; // 320000000

// One and two day growth:

cout << populationPrediction(320000000, 1) << endl; // 320006314 cout << populationPrediction(320000000, 2) << endl; // 320012628

// One and two year growth

cout << populationPrediction(320000000, 365) << endl; // 322304554 cout << populationPrediction(320000000, 2\*365) << endl; // 324609108

return 0; }

Once the function has been tested, write a complete program that prompts the user for the current population and the days into the future using a dialog that looks just like this:

Predict population growth given the current population and days into the future

Current population? 320000000

Day into the future? ***365***

In 365 days, the population should grow by 2304554 to become 322304554

## 4E QUADRATIC FORMULA

Th e quadratic formula (below) uses the *a*, *b*, and *c* from quadratic equations of the form *ax*2 + *bx* + *c* to fi nd both roots.

*x* = *-b b2 –* 4*ac*

2*a*

For example, the two real roots of *x*2 + 3*x* − 4 are 1 and −4 as indicated by this dialog and the plotting of this function:

Enter a b and c coef cients of a quadratic equation: ***1 3 -4*** roots: 1 and -4

1x^2 + 3x + -4 when x is 1 should be 0

This should be 0 or very close? 0

1x^2 + 3x + -4 when x is -4 should be 0

This should be 0 or very close? 0

You are asked to complete and test these three functions as described in the comments:

-1

0

1

2

-5

-4

-3

4

3

2

1

0

-1

-2

-3

-4

-5

-6

// Given the 3 coef cients, compute the two roots that

// are made accessible as reference parameters. Assignment to

// root1 and root2 also change the associated arguments. void ndBothRoots(double a, double b, double c, double & root1, double & root2)

// Evaluate any quadratic equation given the 3 coef cients

// and the root in question. This function should return 0.0, // but something close to 0.0 like -6.66134e-16 is okay.

// This function could return nan if b^2 – 4ac < 0 or a is 0. double evaluate(double a, double b, double c, double root)

// Generate the requested dialog using the two functions above. int main()

To avoid the roots being not a number (nan) when there is a negative square root, complex numbers could be used. However, if the roots are not real, the return result is allowed to be nan

Programming Projects

(or -1.#IND in Visual Studio as of this writing). Th e quadratic equation 3*x*2 + 4*x* + 2 has no real roots. Th is program run indicates there are no real roots since nan is returned with 3, 4, and 2 input for a, b, and c respectively:

Enter a b and c coef cients of a quadratic equation: ***3 4 2***

roots: nan and nan

3x^2 + 4x + 2 when x is nan should be 0

This should be 0 or very close? nan

3x^2 + 4x + 2 when x is nan should be 0

This should be 0 or very close? nan

© 2018 Franklin, Beedle & Associates All rights reserved

**C H A P T E R F I V E**

# Sending Messages

## SUMMING UP

You have now used and implemented free (non-member) functions. Th ese functions represent only a small portion of available non-member functions. Th ese free functions—those that are not part of a specifi c class—continue to play a role in the C++ language. You can use new functions by reading the function headings and associated documentation.

## COMING UP

Th is chapter introduces sending messages to existing objects with a syntax that diff ers from calling free functions. Th is chapter explores some of the standard member functions of the classes string, ostream, and istream and two author-supplied classes BankAccount and Grid. Th is will help you develop problem-solving skills while encouraging you to contemplate the increasing importance of classes that encapsulate data members and the member functions that use that state.

After studying this chapter, you will be able to

* send messages to objects
* use string and ostream messages and understand their eff ects
* problem solve with string, Grid and BankAccount objects
* appreciate why programmers partition software into classes, which are collections of member functions combined with their related data members.

### 5.1 MODELING THE REAL WORLD

Th e C++ programming language has primitive types to store Booleans, characters, and numbers. C++ also has many types implemented with the C++ class construct. For example, string (which is implemented as a C++ class) stores a collection of characters along with other information such as the number of characters in that string. string type objects represent names, addresses—all sorts of data. Other classes allow programmers to store large collections of data. Even then, these hundreds of C++ classes do not supply everything that every programmer will ever need. Th ere are many times when programmers discover they need their own types to model things in their applications. Consider the following system from the domain of banking:

105

#### Th e Bank Teller Specifi cation

Implement a bank teller application to allow bank customers to access bank accounts through unique identifi cation. A customer, with the help of the teller, may complete any of the following transactions: withdraw money, deposit money, query account balances, and see the ten most recent transactions. Th e system must maintain the correct balances for all accounts. Th e system must be able to process one or more transactions for any number of customers.

You are not asked to implement this system now. However, you should be able to pick out some types that are relevant to this system. One simple tool for fi nding the types of objects that model a solution is to write down the nouns in the problem statement. Th en consider each as a candidate type that might eventually represent part of the system. Th e types used to build the system come from sources such as

* the problem specifi cation
* an understanding of the problem domain
* the types that come with the programming language

Th e types should model the real world if possible. Here are some candidates:

|  |  |
| --- | --- |
| **Candidate Objects to Model a Solution** |  |
| bank teller | transaction |
| customers | 10 most recent transactions |
| bank account | balance |

Here is a picture to give an impression of the major types in the bank teller system. Th e BankTeller will accomplish this by getting help from many other objects.

BankTeller

Coordinates activities

TransactionList

A log of all transactions

AccountList

A collection of

bankAccount

s

Transaction

One withdrawal or deposit

BankAccount

One customer’s account

5.1: Modeling the Real World

Th is problem specifi cation shows that programs usually consist of many diff erent types. Instead of implementing the entire system and adding a user interface, only one type of object will be used here as an example — BankAccount.

#### 5.1.1 BankAccount OBJECTS

Implementing a BankAccount type as a C++ class (in the chapter that follows) provides the ability to have many BankAccount objects. Each instance of BankAccount represents an account at a bank. Using your knowledge of the concept of a bank account, you might recognize that each BankAccount object should have its own account number and its own account balance. Other values could be part of every BankAccount object: a transaction list, a personal identifi cation number (PIN), and a mother’s maiden name, for example. You might visualize other banking operations, such as creating a new account, making deposits, making withdrawals, and accessing the current balance. Th ere could also be many other banking messages—generateMonthlyStatementAsPDF, for example.

As a preview to a class as a collection of operations and values, here is a defi nition for the

BankAccount type in the fi le BankAccount.h (h stands for header). Th e details for implementing new types as C++ classes will be presented in the next chapter.

#include <string>

class BankAccount { public: BankAccount(std::string initName, double initBalance); // post: Construct call with two arguments:

// BankAccount anAcct("Hall", 100.00);

void deposit(double depositAmount); // post: Credit depositAmount to the balance

void withdraw(double withdrawalAmount); // post: Debit withdrawalAmount from the balance

double getBalance() const; // post: Return this account's current balance

std::string getName() const; // post: Return this account's name

private:

std::string name; double balance; };

Consider this class BankAccount defi nition as a blueprint used by C++ to construct many BankAccount objects. Each BankAccount object will have its own data member's name and currentBalance. Each BankAccount object will understand the same fi ve member functions: BankAccount, deposit, withdraw, getName, and getBalance. While the defi nition is in a fi le named BankAccount.h, the actual implementation of this C++ class will be in another fi le named

BankAccount.cpp.

It should be noted here that the C++ community uses the term *data member* for a function that is part of a class. Th ey also use the term *data member* for the variables that store the state of the objects. Other object-oriented programming languages use the term *method* rather than *member function* and the term *instance variable* rather than *data member*. Th is textbook will use the C++ terms.

BankAccount objects are constructed with two arguments to initialize the object’s state:

* A string to represent the account identifi er, a name for example
* A fl oating-point number to represent the initial account balance

**General Form 5.1** *Constructing objects (initial values are optional)*

*class-name object-name*(*initial-value(s)*);

Example object constructions:

BankAccount anAccount("Chris", 125.50); string str("A string") string str2() // default value is an empty string ""

Every object has

* a name: a variable that references the entire object
* state: the values that the object currently has
* messages: the operations each object can do

Every object will have a variable to provide access to the state of that object. Every object has its own unique state. Every object will understand the same set of messages. For example, given this object construction,

BankAccount anotherAccount("Dakota", 60.00); we can derive the following information:

* a name to access the object: anotherAccount
* state: an account name of “Dakota” and a balance of 60.00
* messages: understands messages like withdraw, deposit, getBalance

Other instances of BankAccount will understand the same set of messages. However, they will have their own separate state. For example, after another BankAccount construction,

BankAccount newAccount("Kim", 1000.00); newAccount has its own name “Kim” and its own balance of 1000.00.

5.2: Sending Messages

#### 5.1.2 CLASS AND OBJECT DIAGRAMS

Th e three characteristics of objects can be summarized with a class diagram:

|  |
| --- |
| BankAccount |
| string name double balance |
| BankAccount(string initName, double initBalance) void deposit(double depositAmount) void withdraw(double withdrawalAmount) double getBalance() const string getName() const |

A class diagram lists the class name in the topmost compartment. Th e instance variables appear in the compartment below it. Th e bottom compartment captures the methods. Objects can also be viewed as instance diagrams where the name of the object is underlined and values are shown:

|  |  |  |
| --- | --- | --- |
| anAccount  name = "Chris" balance = 125.50 | anotherAccount  name = "Dakota" balance = 60.00 | newAccount  name = "Kim" balance = 1000.00 |

Th ese three object diagrams describe the current state of three diff erent BankAccount objects. One class can be used to construct many objects, each with its own separate state (set of values).

### 5.2 SENDING MESSAGES

Objects such as cin, cout and any string object have class member functions. Using them is a bit diff erent from using the free functions such as those declared in cmath. A diff erent syntax is required. Th is diff erent type of function call is even distinguished with a diff erent name— *message*—when using a member function. Some messages return the object’s state. Other messages tell an object to do something.

* A message that asks the object to return its state: anAccount.getBalance();
* A message that tells the object to do something: anAccount.withdraw(25.00);

Th e state of objects is made accessible through certain operations such as getName and getBalance. Other class member functions exist so programmers can modify the state of the object: withdraw and deposit, for example. Here is the general form used to send messages to objects:

**General Form 5.2** *Sending a message to an object*

*object-name.function-name*(*argument-list*) Example BankAccount messages:

anAccount.deposit(237.42); anAccount.withdraw(5); anAccount.getBalance();

Th e following are incorrect:

|  |  |
| --- | --- |
| anAccount.deposit(); | // Missing the amount to deposit |
| deposit(); | // missing the object-name and . |
| anAccount.getBalance; | // missing () |
| anAccount.withdraw("10"); | // wrong class of argument |
| anAccount; | // missing member function name |
| anAccount.withdrawal(10); | // BankAccount has no function withdrawal |

Fortunately, failure to supply the object name, the dot, and the operation name in the proper order usually generates an error message at compile time. Also, as with any function, the compiler complains if the client code does not supply the proper arguments between parentheses. Th e BankAccount class (and therefore all BankAccount objects) has two member functions to access the state of the object: getName and getBalance. Th e BankAccount class has two member functions that modify the state—withdraw and deposit. Th ese operations are exemplifi ed in the following program that constructs two BankAccount objects and sends messages to both of those objects. Th ose messages result in the following actions:

* deposit 133.33 to the object named ba1
* withdraw 250.00 from the object named ba2
* display the names and modifi ed balances of both objects

// Initialize two BankAccount objects and send some messages

#include <iostream> // for cout using namespace std; #include "BankAccount.h" // for class BankAccount

int main() {

BankAccount ba1("Miller", 100.00); BankAccount ba2("Barber", 987.65);

ba1.deposit(133.33); ba2.withdraw(250.00);

cout << ba1.getName() << ": " << ba1.getBalance() << endl; cout << ba2.getName() << ": " << ba2.getBalance() << endl;

return 0;

}

5.2: Sending Messages

#### Output

Miller: 233.33

Barber: 737.65

Objects store varying amounts of data depending on the class to which they belong. Th e state of an object may require many values—and these values also may be of diff erent classes. For example, a BankAccount object stores a string object to represent the account name, and at the same time stores a number to represent the balance. A weeklyEmployee object might store several strings such as name, address, social security number, and several numbers such as pay rate and hours worked. A robot object may store a current position, a map, and the state of its arm mechanism.

#### SELF-CHECK

5-1 Each of the lettered lines has an error. Explain why.

#include <iostream> // For cout

#include "BankAccount.h" // For class BankAccount using namespace std;

int main() {

BankAccount b1("Sam"); // -a

BankAccount b2(500.00); // -b BankAccount b3("Jo", 200.00); // -c b1.deposit(); // -d b1.deposit; // -e b1.deposit("100.00"); // -f B1.deposit(100.00); // -g b1.Deposit(100.00); // -h withdraw(100); // -i cout << b4.getName() << endl; // -j cout << b1.getName << endl; // -k cout << b1.getName(100.00) << endl; // -l return 0; }

5-2 Write the output generated by the following program:

#include <iostream> // For cout using namespace std;

#include "BankAccount.h" // For the BankAccount class int main() {

BankAccount b1("Chris", 0.00); BankAccount b2("Kim", 500.00); b1.deposit(222.22); b1.withdraw(20.00); b2.deposit(55.55); b2.withdraw(10.00);

cout << b1.getName() << ": " << b1.getBalance() << endl; cout << b2.getName() << ": " << b2.getBalance() << endl; return 0; }

### 5.3 string OBJECTS

Like bankAcount, the string type is implemented as a C++ class. Although each string object stores a collection of characters, a programmer may sometimes be interested in one single character. At other times the programmer may require several characters or the current length of a string (number of characters stored). It is sometimes necessary to discover if a certain substring exists in a string. For example, is the substring ", " included in the string "Last, First" and if so, at what index does ", " begin? Th e C++ string type provides a large number of member functions to help with problems requiring knowledge of string values. You will use string objects in many programs.

Each string object stores a collection of zero or more characters. string objects can be constructed in two ways.

**General Form 5.3** *Constructing string objects in two diff erent ways*

string *identi er*(*string-literal*); string *identi er* = *string-literal*;

#### Examples

string stringReference("A String Object"); string anotherStringReference = "Another";

As with most classes, string has member functions that modify the state of string objects— insert, replace, erase—and member functions that return something about the state—length, nd, and substr. Th e string class has operations that allow access to the elements, or individual characters at, [], front, and back. Th ere are also a number of operators that can be applied to string objects such as +, [], <<, and >>.

#### 5.3.1 ACCESSING METHODS

**string::length()**

A length message sent to a string object returns the number of characters currently in the string.

string stringReference("A String Object"); string anotherStringReference = "Another"; stringReference.length(); // returns 15 anotherStringReference.length(); // returns 7

5.3: string Objects

**string::at**

An at message returns the character located at the index passed as an int argument. Notice that string objects have zero-based indexing. Th e fi rst character is located at index 0, and the second character is located at index 1, or as the message at(1).

string str("A string object"); str.at(0); // returns 'A' str.at(1); // returns ' ' str.at(2); // returns 's'

str.at(str.length()-1); // returns 't', the last character **string::nd and string::rnd**

A nd message returns the index of the fi rst character where the entire string argument is found. If the string argument does not exist, nd returns string::npos (no position), which is a very large integer that may be diff erent from the integer shown below. r nd returns the starting index of the *last* occurrence of the string argument.

string str("there is the other the"); str. nd("the"); // returns 0, the rst "the" cout << str.r nd("the"); // returns 19, the last "the" cout << str. nd(" is "); // returns 5 cout << str. nd("not here"); // returns string::npos which // may be 18446744073709551615 **string::substr**

A substr message returns the part of a string starting at the index specifi ed as the fi rst argument. Th e second argument represents the total number of characters to the end of the string.

string str("Smiles a Lot"); str.substr(1, 4); // returns "mile" str.substr(9, 1); // returns "L" str.substr(9, 2); // returns "Lo" str.substr(9, 55); // returns "Lot" **str::front and str::back**

Th e front and back member functions provide access to the fi rst and last characters in the string object.

string str("abc");

// front and back are part of C++11. With some C++ compilers,

// this code may generate compile time errors because their

// string class may does not yet have these member functions.

str.front(); // returns 'a' str.back(); // returns 'c'

#### 5.3.2 MODIFYING METHODS

**str::insert**

An insert message adds additional characters into the string object right before the character indexed by the fi rst argument. Th e second argument can be a string literal or another string object.

string quick("quick"); string all("the brown jumped dog");

all.insert(4, quick); // all.length() increased all.insert(23, "over the lazy"); cout << all; // prints: the quick brown jumped over the lazy dog **str::replace**

Th e replace member function changes the portion of the string that begins at the index of the fi rst argument and spans the number of characters specifi ed as the second argument.

string quick("quick"); string all("the brown jumped dog"); all.replace(4, 14, quick); cout << all; // prints: the quick dog **str::erase**

An erase message erases the part of the string indicated by the indexes specifi ed in the arguments.

string all("the quick brown fox"); all.erase(4, 12);

cout << all << endl; // prints: the fox cout << all.length(); // prints 7

#### 5.3.3 OPERATORS DEFINED FOR string OBJECTS

##### + OPERATOR

Programmers often make one string object from two separate strings with the + operator that concatenates (connects) two or more strings into one string.

string rstName("Kim"); string lastName("Potter");

string fullName = lastName + ", " + rstName; cout << fullName; // prints Potter, Kim Characters can also be concatenated with strings.

5.3: string Objects

fullName = '>' + fullName + '<'; cout << fullName; // prints >Potter, Kim<

##### << AND >> OPERATORS

Th e << and >> operators are overloaded for the string class to allow input and output of strings, just like numbers.

string rstName; cout << "Enter rst name: "; cin >> rstName; // If the user enters Kim cout << "Hello " + rstName; // output would be: Hello Kim

##### [] OPERATOR

Th e [] is like the at member function. Using square brackets, individual characters can be accessed or changed.

string str("abcde");

str[0]; // returns 'a' str[1]; // returns 'b' str[4]; // returns 'e'

str[2] = 'X'; str[3] = 'O'; cout << str; // prints abXOe

Other operators for comparing strings, such as <= and ==, will be presented in a later chapter.

#### SELF-CHECK

5-3 What is the output from the following program?

#include <iostream>

#include <string>

using namespace std; // Allows string instead of std::string

int main() {

string str("Social Network"); cout << str.length() << endl; cout << str.at(0) << endl; cout << str.at(str.length() - 1) << endl; cout << str. nd("Net") << endl; cout << str. nd("net") << endl; cout << str.substr(7, 3) << endl; cout << str.substr(7, 1) << endl; cout << str.substr(7, 99) << endl; cout << str[1] << endl return 0;

}

5-4 What is the modifi ed value of each string object?

|  |  |  |  |
| --- | --- | --- | --- |
| a. | string str1("Social"); str1.replace(0, 1, "UnS"); | c. | string str2("Social"); str2.erase(3, 2); |
| b. | string str3("Social"); str3.insert(3, "iet"); str3.erase(6, 1); | d. | string str4("Social"); str4[0] = 'N'; str4[5] = 'X'; |

str4[2] = 'T';

5-5 Write the code to store the middle character of a string object into a char variable named mid. If there is an even number of characters, store the char to the right of the middle. For example, the middle character of “abcd” is ‘c’.

5-6 For each of the following messages, if there is something wrong, write “error”; otherwise, write the value of the expression.

string str("Any String");

|  |  |
| --- | --- |
| a. length(str) | d. str. nd(" ") |
| b. str.length | e. str.substr(2, 5) |
| c. str(length) | f. str.substr("tri") |

### 5.4 ostream AND istream MEMBER FUNCTIONS

Th e istream and ostream classes provide input and output.

**ostream::width**

Th e member function width modifi es the state of the ostream object named cout.

#include <iostream> using namespace std; int main() { cout << 1; cout.width(5); cout << 2; cout << 3; return 0; }

#### Output

1 23

5.4: ostream and istream Member Functions

Normally, the state of cout is set to display the next output in the minimum number of columns—with no leading spaces—which is the default state of cout. Th e cout.width(5) message temporarily alters the state of cout such that the very next output will be output in a minimum of fi ve columns. After that, the default situation is back in force so the 2 is printed in one column, immediately following the 3.

**ostream::precision**

To gain control over the appearance of fl oating-point output, use the ostream member function precision. A precision message tells the ostream object cout to show a specifi c number of digits in fl oating-point numbers. Unlike width, the precision remains the same until another precision message is sent to cout.

// Send two precision messages to the ostream object named cout

#include <iostream> using namespace std;

int main() { double x = 1.23456;

cout << x << endl; // Default (1.23456) cout.precision(1); // Modify the state of cout cout << x << endl; // Show only one signi cant digit (1) cout.precision(4); // Modify the state of cout cout << x << endl; // Show four digits rounded (1.235) cout << x << endl; // Precision of 4 still in effect

return 0; }

#### Output

1.23456 1

1.235

1.235

**istream::good**

Th e member function good of the istream class returns the state of an input object (usually cin). Normally, cin.good() returns 1, which means “true” if cin is still capable of reading. However, if someone enters an improper value, such as input of BAD instead of a number as shown below, the good message returns 0, which means “false.”

cout << cin.good(); // Returns 1 for good, 0 for bad

Whenever cin.good() is false, no more input is allowed from cin unless other steps are taken. So if you enter an invalid number—an easy input mistake to make—strange things may occur.

// Demonstrate what happens with bad input

#include <iostream> // For the cout and cin objects using namespace std;

int main() { int x = 0.0;

cout << "Is cin good? " << cin.good() << endl; cout << "Enter an int: "; cin >> x;

cout << "Is cin still good? " << cin.good() << endl;

return 0; }

|  |  |
| --- | --- |
| **Dialogue: 1 means true** | **A 2nd Dialogue: 0 means false** |
| Is cin good? 1  Enter an int: 123  Is cin still good? 1 | Is cin good? 1  Enter an int: NotAnInt  Is cin still good? 0 |

#### 5.4.1 CLASS MEMBER FUNCTION HEADINGS

When a function is a member of a class, the function heading is qualifi ed with the class name followed by the scope resolution operator ::. Using :: will be necessary to successfully build a C++ class in the next chapter. It also helps the reader determine when the dot notation is required to send a message. Any function heading of the following form identifi es the function as a class member function:

**General Form 5.4** *Class member function headings*

*class-name* :: *function-name*(*parameters*)

So for example, int string::length() indicates that length is a member of the string class. It is diff erent from the nonmember functions sqrt and pow. Here is the list of some of class member function headings that have been revealed so far (many more exist):

##### EXAMPLES OF CLASS MEMBER FUNCTION HEADINGS

|  |  |
| --- | --- |
| **Class** | **Member function heading** |
| string | int string::length() const; |
|  | // Return the number of characters in this string |

5.4: ostream and istream Member Functions

int string:: nd(string subString); // Return position of rst substring

string string::substr(int pos, int n) const;

// Return the n characters to the right of // string[pos] or up to this string's length

string insert (int pos, const string& str);

// Inserts additional characters into the string right // before the character indicated by pos.

|  |  |
| --- | --- |
| ostream | int ostream::width(int nCols); |
|  | // Next output to this ostream object will be |
|  | // displayed in nCols. Returns the current value |
|  | // of the date member width. |
|  | int ostream::precision(int nDigits); |
|  | // Show oating-point output with nDigits of digits. |
|  | // Also returns the current precision. |
| istream | int istream::good(); |
|  | // post: Return 1 if istream can read or 0 if corrupt |

BankAccount BankAccount::BankAccount(string aName, double initBalance); // post: Construct a BankAccount with two arguments

void BankAccount::deposit(double amount);

// pre: amount >= 0

// post: amount is credited to this object's balance

void BankAccount::withdraw(double amount); // pre: amount >= 0 and <= this object's balance // post: amount is debited from this object's balance

double BankAccount::getBalance() const; // post: Return this object's current balance

string BankAccount::getName() const;

// post: Return this object's name

Th e class name and :: should help you determine whether you must call a non-member (free) function without the function name fi rst or send a message with the object name followed by a dot.

|  |  |
| --- | --- |
| **Free Function Heading** | **Function Call** |
| double pow(double base, double power) | double answer = 0.0; |
| // post: Return base to the power power | double x = 1.023102; |
|  | answer = pow(x, 360.0); |
| **Member Function Heading** | **Message** |
| string string::substr(int pos, int n) | string name("Doe, Jo"); |
| // post: Return n characters of this | int n = name. nd(","); |
| // string beginning at index pos | string last = name.substr(0, n); |

Additionally, to document a function name as a class member function requiring the dot notation, you will often see member functions referred to without the parameter list and return type such as string::length. Th is is true in the context of this textbook and with most online and book documentation.

#### SELF-CHECK

5-7 Write the output generated by the following program. Make sure you line up all output in the correct column.

#include <iostream> using namespace std; int main() {

cout << "123456789012345" << endl; cout.width(3); cout << 1; cout.width(5); cout << 2.3; cout.width(6); cout << "who" << endl; return 0; }

5-8 Write the exact output generated by the following program:

#include <iostream> using namespace std; int main() { cout.precision(3); cout << 9.876543 << endl; cout.precision(1); cout << 1.2 << endl; cout.precision(8); cout << 1.2 << endl; return 0; }

5-9 Write the complete dialogue generated by the following program when:

1. the user enters ***123***
2. the user enters ***XYZ***

#include <iostream> using namespace std; int main() { int anInt(0);

cout << "Enter an integer: "; cin >> anInt;

cout << "Good? " << cin.good() << endl; return 0; }

5-10 What class does each member function belong to?

|  |  |
| --- | --- |
| a. istream::clear | d. string::replace |
| b. Grid::move | e. BankAccount::withdraw |
| c. ostream::width | f. istream::good |

### 5.5 ANOTHER NONSTANDARD CLASS: Grid

Th is section presents another nonstandard class that will be used occasionally over the next several chapters to help you think in terms of objects while providing opportunities to improve problemsolving skills.

Th is section presents a Grid type implemented as a C++ class. Before you study this section, please realize that the Grid class is meant to be used for teaching and learning purposes only. It will be used occasionally in later chapters to demonstrate new concepts in a visual manner. However, Grid objects are not meant to predominate any of those new concepts. Th e graphical state of Grid is meant to help you more readily grasp the access and modifi cation of object state through messages. You will be able to complete a few programming projects comprised only of messages to this object.

Th e Grid class presented here is based on the work of Rich Pattis’ *Karel the Robot: A Gentle Introduction to the Art of Programming* and a game seen at Disney World’s Epcot Center. Th e game asked the question, “Could you be a programmer?” Th e player was invited to guide a pirate ship to a treasure while avoiding obstacles.

A Grid object stores a little rectangular map of rows and columns with an object to move. A Grid object is initialized with fi ve arguments:

Grid *Grid-name* (*rows*, *cols*, *mover-row*, *mover-col*, *direction*); where the fi rst two arguments represent the size of the Grid in rows and columns, the next two arguments are the mover’s starting row and column, and the last argument is the mover’s starting direction. Th e direction must be listed either as north, south, east, or west. Th e following program provides an example initialization with an output message

(Grid::display) that allows the programmer to inspect the state of the Grid. To maintain consistency with C++, which begins counting at 0, the fi rst row is referenced as the 0 row. Th e fi rst column is also referenced as the 0 column and the intersection of the fi rst row, fi rst column location is referenced as 0, 0.

// Initialize and display a Grid object

#include "Grid.h" // For the Grid class

int main() {

// Arguments used to initialize a Grid object go like this:

// #rows, #columns, StartRow, StartColumn, StartDirection

Grid aGrid(8, 16, 4, 8, east); // 4 is the fth and // 8 is the ninth column aGrid.display(); return 0; }

#### Output

The Grid:

. . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . .

. . . . . . . . > . . . . . . .

. . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . .

A Grid object’s state is accessed with class member functions such as these:

* int Grid::row() const
* int Grid::nRows() const
* int Grid::nColumns() const
* void Grid::display() const
* bool Grid::frontIsClear() const

Although you may not see the need for these operations at this point, they will come in handy if you do any problem solving associated with Grid objects.

// Access the state of a Grid object with messages

#include <iostream> // For the cout object using namespace std; #include "Grid.h" // For class Grid

int main() {

Grid aGrid(7, 14, 5, 8, east); // Column 8 is the ninth column cout << "Current row : " << aGrid.row() << endl; cout << "Current column : " << aGrid.column() << endl; cout << "Number of rows : " << aGrid.nRows() << endl; cout << "Number of columns: " << aGrid.nColumns() << endl; cout << "Front is clear? : " << aGrid.frontIsClear() << endl; return 0; }

#### Output

Current row : 5

Current column : 8

Number of rows : 7

Number of columns: 14

Front is clear? : 1

Th e state of any Grid object is modifi ed with the messages Grid::move(), Grid::turnLeft(), and Grid::turnRight().

#include "Grid.h" // For the Grid class

int main() { Grid aGrid(7, 9, 1, 3, east);

aGrid.move(); aGrid.move(); aGrid.turnRight(); aGrid.move(); aGrid.move(); aGrid.turnRight(); aGrid.move(); aGrid.move(); aGrid.turnLeft(); aGrid.move(); aGrid.move(); aGrid.display(); return 0; }

#### Output

The Grid:

. . . . . . . . .

. . . . . .

. . . . . . . .

. . . . . .

. . . . . . . .

. . . v . . . . .

. . . . . . . . .

#### SELF-CHECK

5-11 Write the output of the following program:

#include <iostream> // For cout using namespace std; #include "Grid.h" // For the Grid class

int main() {

Grid aGrid(6, 6, 4, 2, east); aGrid.move(2); aGrid.turnLeft(); aGrid.move(3); aGrid.turnLeft(); aGrid.move(2); aGrid.display();

cout << "row: " << aGrid.row() << endl; cout << "col: " << aGrid.column() << endl; return 0; }

##### 5.5.1 OTHER Grid OPERATIONS

Th ere are several other Grid operations, some of which will be needed in this chapter’s programming projects. Completing those projects provides practice at sending messages to objects—calling member functions—and developing algorithms resulting in a more graphical result. Th e following class diagram lists all Grid member functions. It is not necessary to know the data members to use objects, so the state is omitted here.

###### Grid MEMBER FUNCTIONS

// -- Modi ers void move(); void move(int spaces); void turnLeft(); void turnRight(); void putDown();

void putDown(int putDownRow, int putDownCol); void toggleShowPath(); void pickUp(); void block(int blockRow, int blockCol);

// -- Accessors bool frontIsClear() const; bool rightIsClear() const; int row() const; int column() const; int nRows() const; int nColumns() const; void display() const;

Although this class diagram provides a summary of legal messages, it does not explain the number and class of arguments to use when sending messages to a Grid object. For that, the following subset of the member function headings is provided (all the ones you need to do the programming projects in this chapter) with pre- and postconditions.

###### SUBSET OF Grid MEMBER FUNCTIONS

Th ese help us understand what each function does. A precondition tells us what must be true before a message is sent. A postcondition tells us what will happen if the precondition is met.

Grid::Grid(int Rows, int Cols, int startRow, int startCol, int direction)

// post: Construct a 10-by-10 Grid object with 5 arguments

// Grid aGrid(10, 10, 0, 0, east);

void Grid::move()

// pre: The mover has no obstructions in the next space

// post: The mover is 1 space forward

void Grid::move(int spaces)

// pre: The mover has no obstructions in the next spaces

// post: The mover is spaces forward

void Grid::putDown(int putDownRow, int putDownCol)

// pre: The intersection (putDownRow, putDownCol) has nothing at

// it except, perhaps, the mover

// post: There is one thing at the intersection

void Grid::pickUp()

// pre: There is something to pick up at the mover's location

// post: There is nothing to pick up from the current intersection

void Grid::turnLeft() // post: The mover is facing 90 degrees counterclockwise

void Grid::block(int blockRow, int blockCol)

// pre: There is nothing at the intersection (blockRow, blockCol)

// post: The intersection can no longer be visited

void Grid::display() const // post: The current state of the Grid is displayed on the screen

For example, think about a program that blocks three intersections (represented by #), instructs a kid to eat two cookies, and moves the kid back to the starting point. A few messages to Grid::putDown will place a few “cookies” (or whatever you would like the capital letter “O” to represent) on the Grid. Th en the challenge is sending the proper messages to move the kid to eat the cookies using the Grid member functions such as Grid::move. If the kid is facing south you will see a v, if the kid is facing north you will see ^, if the kid is facing east you will see > and facing west <. To "eat" the cookies send messages to Grid::pickUp. Here is the program:

// This program sets two cookies on the table and instructs a kid

// on how to locate them, "eat" them, and return home

#include "Grid.h" // For the Grid class

int main() {

Grid kid(8, 12, 0, 0, south); kid.putDown(4, 0); kid.putDown(4, 3);

kid.block(3, 2); // Can't move through a block # kid.block(4, 2); kid.block(5, 2);

// Show the state of kid kid.display();

// "Eat" two cookies kid.move(4); kid.pickUp(); kid.move(2); kid.turnLeft(); kid.move(3); kid.turnLeft(); kid.move(2); kid.pickUp();

// Get the kid back home kid.move(4); kid.turnLeft(); kid.move(3);

// Show the ending state kid.display(); return 0; }

###### Output

The Grid v . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . .

. . # . . . . . . . . .

O . # O . . . . . . . .

. . # . . . . . . . . .

. . . . . . . . . . . .

. . . . . . . . . . . .

The Grid < . . . . . . . .

. . . . . . . . . .

. . . . . . . . . .

. # . . . . . . . .

. # . . . . . . . .

. # . . . . . . . .

. . . . . . . .

. . . . . . . . . . . .

##### 5.5.2 FAILURE TO MEET THE PRECONDITIONS

Th ere are many “illegal” messages you can send to a Grid object. For example, you could try sending a move message that asks the mover to move through a block (#) or off the edge of the Grid. All it takes is one incorrect message—moving four rows instead of three, for example.

#### SELF-CHECK

5-12 If you were designing the operations for a Grid object, what would you want to prevent from occurring?

So what should a Grid object do when sent a message that makes no sense? Quite frankly, it’s a bit awkward. Th e object could respond by doing nothing. In this case, the state of the object would remain unaltered. Or the object could travel off the end of the Grid or move through blocks—but this sounds more like a Superman object. Here’s yet another snippy answer: the behavior is *undefi ned*.

Th is awkwardness is circumvented by the notion of preconditions. A function’s *precondition* is what the function presumes to be true when a function is called or the message is sent. For example, the void move(int spaces) operation has the precondition that there is no block or Grid edge in the path of the mover. Also, the Grid::pickUp() message presumes there is something to pick up.

void Grid::move(int spaces)

// pre: The mover has no obstructions in the next spaces

// post: The mover is spaces forward

void Grid::pickUp()

// pre: There is something to pick up at the mover's location

// post: There is nothing to pick up from the current intersection

So what does happen when you violate one of these preconditions? You’ll likely fi nd out if you work on certain Grid-related programming projects.

##### 5.5.3 FUNCTIONS WITH NO ARGUMENTS STILL NEED ( )

You have now seen several messages that require no arguments. If a function has no parameters, it requires no arguments. Here are two examples:

cout << aString.length() << endl; cout << aGrid.row() << endl;

It should be noted, before you do any of this chapter’s programming projects, that even though no values need to be passed as arguments to either string::length or Grid::row, the parentheses must still be included in the message. Th e following code will not do what you might expect:

cout << aString.length << endl; // ERROR: Missing () after length cout << aGrid.row << endl; // ERROR: Missing () after row

Th e parentheses represent the function call operator. Without ( and ), there is no function call— even when zero arguments are needed by the function.

**5.6 WHY FUNCTIONS AND CLASSES?**

*Abstraction* is the process of pulling out and highlighting the relevant features of a complex system. One aspect of abstraction is understanding the computer from the programming-language level without full knowledge of the details at the lower levels. Abstraction is our weapon against complexity.

You can use operations such as sqrt, pow, Grid::move, and any other new function without knowing the implementation details coded by other programmers. Abstraction allows programmers to quickly and easily use int, double, string, BankAccount, and Grid objects. Th e characteristics of int data (a specifi c range of integer values) and int operations (such as addition, multiplication, assignment, input, and output) can be understood without knowing the details of those operations, or even how those values are stored, or how these operations are implemented in the hardware and software. Abstraction is friendly. Abstraction makes life easier. Abstraction helps keep us sane. Abstraction is that little “black box” programmers are always talking about. When you can’t see how a function is implemented, a programmer calls that a “black box.” Th at is abstraction.

Even though C++ is delivered with a large set of abstractions known as functions and classes, additional functions and classes will still be required. New abstractions are built from existing objects, operations, and algorithms. As you begin to create function and class abstractions, set a goal to build these abstractions so they are easy to use and perform a well-defi ned operation. When the details of implementation are long forgotten, you will still be able to use the abstraction because you know *what* it does. You won’t have to remember *how* it does it.

Instead of encapsulating a group of related code statements in a function, you could write all code statements directly in the main function. Th ose statements would then replace the function call. However, as the table below shows, that detailed way is quite extensive in the number of lines required.

|  |  |
| --- | --- |
| **Th e Actions Represented by One Message** |  |
| **Operation Th e Object-Oriented Way** | **Th e Detailed Way** |
| Construct one Grid object Grid g(15,15,9,4,east); | 35 lines |
| Move in current direction g.move(2); | 112 lines |
| Output the Grid g.display(); | 6 lines |
| Change direction g.turnLeft(); | 10 lines |

5.6: Why Functions and Classes?

Th e four messages in the middle column represent the abstract equivalent of coding the 163 lines the non-function way, as listed in the right column. Now imagine a six-message program that moves and turns three times. Th e equivalent non-function way would require approximately 366 detailed lines of code rather than the six messages!

By placing the many lines of detailed code into functions, the programmer may execute that operation with just one message or function call. Th e same message may be sent over and over again. So whenever you have code that can be used more than once in a program, it is preferable to implement that behavior within the confi nes of a free (nonmember) function or as one of many member functions available to the objects. Function calls and messages represent many hidden instructions and details. Th e programmer need not see, nor understand, all implementation details. Encapsulating code in functions also helps avoid code duplication, a sign of poorlywritten programs. *Abstraction*, *encapsulation*, and *black box* are all terms used for hiding information.

#### SELF-CHECK

5-13 Using the previous table, how many lines of code are required to initialize the state of one Grid object using the object-oriented way?

5-14 Using the previous table, how many lines of code are required to initialize the state of one Grid object when the detailed way is used (right column)?

5-15 Write a paper and pencil program that constructs a Grid object and moves it one space in all four directions: north east south west.

By partitioning low-level details into one function, the implementation need only be written once. Another advantage of functions is that the same operation can be used over and over again with a one-line message. Rather than one huge int main() { }, programs are composed of more manageable calls to nonmember (free) functions (sqrt and pow) and messages to class member functions (string::substr and Grid::move). Here are some reasons why C++ programmers use existing functions and objects to better manage the complexity of software development:

* to reuse existing code rather than write it from scratch
* to concentrate on the bigger issues at hand
* to reduce errors by writing the function only once and testing it thoroughly

In the early days of programming, programs were written as one big main program. As programs became bigger, *structured programming* techniques became popular. One major feature of structured programming was to partition programs into functions for more manageable code. Programmers found this helped people understand the program better. It is easier to maintain programs that place related processing details in an independent function. It is easier to fi x a 20-line function in a program with 100 functions than it is to fi x a 2,000-line program. Other reasons for dividing a program into smaller functions include:

* placing details into a function or class makes the code easier to comprehend
* the same actions need to be achieved more than once in a program
* the function or class can be reused in other applications

With free functions, the data are passed around from one nonmember function to another. When the data are available everywhere throughout a large program, they become susceptible to accidental changes.

Now as software has become even more complex, object technology encapsulates collections of functions with the data manipulated by those functions. Developers don’t throw the data around between disparate groups of nonmember functions which would leave them open for accidental attack. As you will see in the next chapter, with object-oriented programming, data are encapsulated with the functions—nice and safe.

##### Historical Progression of How Programs Are Organized into Modules

**Th e Early Days**

**Structured**

**Object-Oriented**

main() {

// 1

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

//500

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

//1000

}

one() {

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

}

two() {

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

}

//...

ninety9() {

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

}

hundred() {

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

}

main() {

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_

}

class ONE {

one()

two()

// ...

ten()

}

// ...

class NINE {

eighty1()

eighty2()

// ...

ninety()

}

//...

class TEN {

ninety1()

ninety2()

//...

hundred()

}

main() {

}

Chapter Summary

#### SELF-CHECK

5-16 What reason for using functions makes the most sense to you?

5-17 Describe one example of how abstraction helps you get through the day.

### CHAPTER SUMMARY

* Th e string class has a large number of operations for manipulating all or part of a string.

Th ese include substr, nd, at, replace, and length.

* Some messages require the object name and a dot ( . ) before the member function name and arguments. Use aString.substr(2, 5) rather than substr(aString, 2, 5).
* Consider using cout.width(10) to right-justify numeric output in 10 columns (or cout.width(9) for 9 columns, and so on). Th e new column width starts after the output of the previous value. It does not start at the left margin.
* Class member functions are often written with the class name and the scope resolution op-erator :: to indicate the class of objects that would understand the message, so you’ll see ostream::width rather than simply width.
* Class member function headings supply the same usage information as their nonmember cousins (sqrt, pow, fmod). Th e return type is given, as is the function name and the number and class of arguments that must be used.
* Class member functions additionally are qualifi ed with their class names, for example, void Grid::move().
* Most classes in this textbook are part of the C++ standard. Th e BankAccount and Grid classes are available at this textbook’s website.
* A class diagram summarizes the names of the messages understood by any instance of a class (object). Th e programmer needs more information to correctly send a message such as number and class of arguments. Th at is why some of the class member functions were shown with pre- and postconditions.
* In the 1960s, programs were written as collections of statements. By the 1970s, programs were usually collections of free functions. Starting in the 1990s, more and more programs have been collections of interacting objects, where each object is an instance of a class containing a collection of member functions. Each improvement allows more complex software to be built.
* Abstraction means the programmer can call a function or send a message without knowing the implementation details. Th e programmer does need to know the function name, the return type, or the number and class of arguments.

### EXERCISES

1. Write the output generated by the following program:

#include <iostream> using namespace std; #include "BankAccount.h" // For class Grid

int main() {

BankAccount b1("One", 100.00); BankAccount b2("Two", 200.00); b1.deposit(50.00); b2.deposit(30.00); b1.withdraw(20.00); cout << b1.getBalance() << endl; cout << b2.getBalance() << endl; return 0; }

1. Write the complete dialogue of this program when the user enters this input in the order requested: MyName 100 22.22 44.44

#include <iostream> // For cout and endl using namespace std;

#include "BankAccount.h" // For the BankAccount class int main() { string name; double start, amount;

// Input: cout << "name: "; // MyName cin >> name;

cout << "initial balance: "; // 100 cin >> start;

// Construct a BankAccount

BankAccount one(name, start);

cout << "deposit? "; // 22.22 cin >> amount; one.deposit(amount);

cout << "withdraw? "; // 44.44 cin >> amount; one.withdraw(amount);

cout << "balance for " << one.getName() << " is " << one.getBalance() << endl; return 0; }

3. Write the output generated by the following program:

#include <iostream> // For the object cout using namespace std;

Exercises

#include "Grid.h" // For the Grid class int main() {

Grid aGrid(6, 6, 1, 1, south);

aGrid.putDown(2, 3); // Place thing at a speci c intersection aGrid.block(0, 0); aGrid.block(5, 5); aGrid.move(2); aGrid.turnLeft();

aGrid.putDown(); // Place thing where the mover is aGrid.move(3); // located, which appears as & aGrid.turnLeft();

aGrid.putDown(); // Place object where the mover is located aGrid.move(1); aGrid.turnLeft(); aGrid.move(1); aGrid.display();

cout << "Mover: row#" << aGrid.row() << " col#" << aGrid.column()

<< endl; return 0; }

1. What is the value of position?

string s("012345678");

// Initialize position to the rst occurrence of "3" in s int position = s. nd("3");

1. What is the value of s2?

string s1("012345678"); string s2(s1.substr(3, 2)); // assert: s2 is a substring of s1

1. What is the value of lengthOfString?

string s3("012345678"); int lengthOfString = s3.length(); // assert: lengthOfString stores the number of characters in s3

1. Choose the most appropriate classes for each of the following from this set of classes: double, int, ostream, istream, string, BankAccount, or Grid. a. Represent the number of students in a section.
   1. Represent a student’s grade point average.
   2. Represent a student’s name.
   3. Represent the number of questions on a test.
   4. Represent a person’s savings account.
   5. Simulate a very limited version of the arcade game Pac-Man.
   6. Read input from a user.
   7. Display output.
2. Name two reasons why programmers use or implement functions.
3. Must a programmer understand the implementation of Grid::move to use it?
4. Answer the following questions given the member function heading:

void Grid::block(int blockRow, int blockCol)

// pre: The intersection at (blockRow, blockCol) has nothing

// at all on it, not even the mover

// post: The intersection at (blockRow, blockCol) is blocked. The

// mover cannot move into this intersection.

* 1. What is the member function name?
  2. What type does it return?
  3. What class does it belong to?
  4. Write a valid message assuming a Grid object named aGrid exists.

1. Write a complete C++ program that will initialize a BankAccount object with an initial balance of $500.00 and your own name. Make a deposit of $125 and a withdrawal of $20.00. Th en show the name and balance. Th e output should look like this:

name: Your Name balance: 605

### PROGRAMMING TIPS

1. You will need author-supplied fi les to complete some programming projects. Th ese are the fi les included with " and " rather than < and > ("Grid.h" and "BankAccount.h", for example). Both fi les need to be located in the same directory (folder) as the .cpp fi le with the main function that you are writing. You can download the proper fi les from this textbook’s website.
2. Distinguish standard #include fi les from nonstandard (user defi ned) fi les. #include standard libraries (classes and objects) with < > and nonstandard classes with " ". Here are some examples:

#include <string> // For the standard string class

#include <iostream> // For cout and cin

using namespace std; // Required to avoid writing std::cout

#include "BankAccount.h" // For class BankAccount

#include "Grid.h" // For class Grid

1. Even if no arguments are required, end messages with (). Do not forget parentheses in messages that do not require arguments.

cout << myAcct.balance; // Error: This references a memory location cout << myAcct.balance(); // Good

Programming Tips

1. C++ begins counting at 0, not 1. Th e fi rst character in a string is referenced with subscript 0, not 1.

cout << aString[0]; // Return the rst character cout << aString[1]; // Return the second character

1. Don’t reference aString[aString.length()]. Th is is an attempt to reference a single value that is not in the range of 0 to aString.length()-1. In general, do not reference characters in a string that do not exist.

string aString;

aString = "This string has 29 characters";

cout << aString[-1]; // ERROR: -1 is out of range, only use 0..28 cout << aString[aString.length()]; // ERROR: 29 is also out of range

1. Two diff erent kinds of constructions are allowed when only one argument is required (C++11 defi nes another, but it is not used until later). One with parenthesis and one with the assignment operator:

string state1 = "Arizona"; string state2("Minnesota");

int n2 = 0;

int n1(0);

double x2 = 0.0; double x1(0.0);

However, when two or more values are needed to initialize an object, use parentheses like this:

BankAccount anAcct("Skyler", 23.41);

Grid aGrid(12, 12, 0, 0, east);

1. Th e :: operator indicates the class to which a function belongs. Th e :: operator is called the “scope resolution operator.” Th e class name followed by :: documents a function as a member function. Any instance of that class will understand the message. Th erefore, string::length documents that any string object will understand the length message. However, the class name and :: are not to be used in the message.

BankAccount anAcct("Milan", 345.67);

// Need 'object-name.functionName' not 'class-name::functionName' cout << BankAccount::balance(); // Invalid cout << anAcct.balance(); // A valid message

**PROGRAMMING PROJECTS**

## 5A A LITTLE CRYPTOGRAPHY

Write a C++ program that hides a message in fi ve words. Use one of the characters in the fi ve input strings to spell out one new word. Make up at least one other message besides these two that requires running the same program twice:

Enter ve words: ***cheap energy can cause problems***

Enter ve integers: ***4 2 1 0 5***

Secret message: peace

Enter ve words: ***programming is very complex work***

Enter ve integers: ***3 0 0 5 2*** Secret message: giver

## 5B LETTER I

Write the code that would go in a main function that constructs a 13-by-7 Grid object and then instructs the mover to “draw” the letter I exactly as shown (the mover could be left anywhere next to the I).

. . . . . . .

. . . . . . .

. . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . > .

. . . . . . .

. . . . . . .

## 5C HURDLES

Write a function void jumpOneHurdle(Grid & g) that instructs the mover to jump one “hurdle” (the block #). Th e main function must make fi ve calls to this function and display the current state after each function call to jumpOneHurdle.

g.display(); // Show initial state, just after construction jumpOneHurdle(g); g.display(); jumpOneHurdle(g); g.display(); jumpOneHurdle(g); g.display(); jumpOneHurdle(g); g.display(); jumpOneHurdle(g); g.display();

Th e fi rst display message should show this state of the Grid object:

Programming Projects

The Grid:

. . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . .

> . . # . . . # . . . # . . . # . . . # . . .

. . . . . . . . . . . . . . . . . . . . . . .

Th e sixth display message should show the mover has jumped fi ve hurdles:

The Grid:

. . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . .

# # # # # > . .

. . . . . . . . . . . . . . . . . . . . . . .

## 5D STAIR CLIMB

Write a function void climbStair(Grid & g) that instructs the mover to climb one step and call it enough times to climb to the top of the stairs. You will need six block messages to simulate the stairs below.

*Before*  *After*

The Grid: The Grid:

. . . . . . . . . . . . . . . .

. . . . . . . . . . . . >

. . . . . # # # . . . # # # . . . . # . . . . . # . . .

. . . # . . . . . # . . . .

> . # . . . . . # . . . . . . . . . . . . . . . . . . . . .

## 5E TEN String PROCESSING FUNCTIONS

Write one C++ program that uses this test driver as a main function to generate the output shown by calling ten new free functions, which are specifi ed below.

// Test drive 10 String processing functions int main() {

cout << " matterAntiMatter(\"LOL\"): " << matterAntiMatter("LOL") << endl; cout << " removeEnds(\"MarkeR\"): " << removeEnds("Marker") << endl; cout << " tripleUp(\"on\"): " << tripleUp("on") << endl; cout << " splitString(\"IU\", \"owe\"): " << splitString("IU", "owe") << endl; cout << " reverse7Chars(\"1234567\"): " << reverse7Chars("1234567") << endl; cout << " halfAndHalf(\"ABcde\"): " << halfAndHalf("ABcde") << endl; cout << "nameRearranged(\"Li, Kim R\"): " << nameRearranged("Li, Kim R") << endl; cout << " middleThree(\"123456\"): " << middleThree("123456") << endl;

// Use reference parameters instead of returning a string string str1("abacada"); remove3(str1, "a");

cout << " remove3(\"abacada\", \"a\"): " << str1 << endl; string str2("ornoon");

replace(str2, 'o', 'X');

cout << "replace(\"ornoon\", 'o', 'X'): " << str2 << endl;

return 0; }

### Expected Output

matterAntiMatter("LOL"): Anti-LOL removeEnds("MarkeR"): arke tripleUp("on"): 1)on 2)on 3)on splitString("IU", "owe"): IoweU reverse7Chars("1234567"): 7654321 halfAndHalf("ABcde"): cdeAB nameRearranged("Li, Kim R"): Kim R. Li middleThree("123456"): 345 remove3("abacada", "a"): bcda replace("ornoon", 'o', 'X'): XrnXXn

1. string antiMatter(string matter)

Everyone knows that interplanetary space travel is fueled by letting matter and antimatter mix. With this in mind, write a function antiMatter that takes a string with the name of some thing or idea. Return a string with “Anti-” prepended to it. Don’t forget the hyphen.

matterAntiMatter("Shoes") returns "Anti-shoes" matterAntiMatter("noisy trucks") returns ""Anti-noisy trucks" matterAntiMatter("LOL") returns "Anti-LOL"

1. string removeEnds(string str)

Complete method removeEnds to return a substring of the supplied string that does not have the characters at either end. Precondition: str always has at least two characters.

removeEnds ("MarkeR") returns "arke" removeEnds ("mom") returns "o" removeEnds ("to") returns ""

1. string tripleUp(string str)

Complete method tripleUp to return a string that has the argument repeated three times with 1), 2), and 3) as shown. Precondition: str.length() 1 tripleUp("top") returns "1)top 2)top 3)top"

1. string splitString(string str, string mid)

Th is function takes in a string of length 2 or greater, and returns a string with a space added into the middle of the string. If the string’s length is an odd number, the second

Programming Projects

half of the string will be the longer half.

splitString("IU", "owe") returns "IoweU" splitString("ab", "\_ \_") returns "a\_ \_b"

1. string halfAndHalf(string str)

Complete method halfAndHalf to return a new string that has the fi rst half of the argument at the end and the last half of the argument at the beginning. If there are an odd number of letters, consider the last half to have one more character than the fi rst half before the split. Precondition:

str.length() 2.

halfAndHalf("1234abcd") returns "abcd1234" halfAndHalf("ABcde") returns "cdeAB" halfAndHalf("Hello") returns "lloHe"

1. string nameRearranged(string name)

Implement nameRearranged that takes a name in the form lastName, ", " rstName, and an initial and returns a string in the form of rstName, initial, ". " and lastName.

nameRearranged("Jones, Kim R") returns "Kim R. Jones"

1. string middleThree(string str)

Implement middleThree so it returns the middle three characters of any string that has three or more characters. If the length of name is even, favor the right. Precondition: str.length() 3.

middleThree("Rob") returns "Rob" middleThree("Roby") returns "oby" middleThree("Robie") returns "obi" middleThree("123456") returns "345"

1. string reverse7Chars(string str)

Implement reverse7Chars so it returns a string that is the reverse of the argument. Precondition: Th e argument str is seven characters long.

reverse7Chars("1234567") returns "7654321" reverse7Chars("morning") returns "gninrom"

1. void remove3(string & str, string sub)

Implement remove3 so it modifi es the string argument str such that the fi rst three occurrences of sub are removed. Precondition: Th e argument sub exists at least three times in str.

string str("there is the other the");

removeThree(str, "he"); // str changes to " tre is t otr the"

string str2("to be or to be or to be");

removeThree(str2, "to "); // str2 changes to " be or be or be"

1. void replace(string & str, char oldC, char newC)

Implement replace to modify the string argument str so that the fi rst three occurrences of oldC are changed to newC. Precondition: Th e argument oldC exists at least three times in str.

string str3("ornoono");

replace(str3, 'o', 'X'); // str3 changes to Xr nXXno

**C H A P T E R S I X**

# Class Defi nitions and Member Functions

## SUMMING UP

Functions hide details, can be called many times, can be reused in other programs, and help in the design of larger programs. Each function performs a well-defi ned service.

## COMING UP

When a function belongs to a class, it becomes a class member function. Class member functions have a lot in common with their nonmember cousins. Chapter 6 presents an introduction to C++ class defi nition and member function implementations. You will learn to read and understand classes by their *defi nitions*—the collection of member function headings (the interface) and data members (the state). In the second part of this chapter, you will learn to implement class member functions. You will also see a few appropriate object-oriented design guidelines that help explain why classes are designed the way they are. After studying this chapter, you will be able to

* read and understand class defi nitions (interface and state)
* implement class member functions using existing class defi nitions
* apply some object-oriented design guidelines

### 6.1 DEFINING A CLASS IN A HEADER FILE

Abstraction refers to the practice of using and understanding something without full knowledge of its implementation. Abstraction allows the programmer using a class to concentrate on the data characteristics and the messages that manipulate the state. For example, a programmer using the string class need not know the details of the internal data representation or how those operations are implemented in the hardware and software. Th e programmer can concentrate on the set of allowable messages—*the interface*.

Th is chapter presents some implementation issues that so far have been hidden. In the fi rst part of this chapter, the BankAccount class will be studied at the implementation-detail level. However, before examining the physical side of class design, let’s consider some of the design decisions that were made for this textbook’s BankAccount class.

141

All BankAccount objects have four allowable operations: deposit, withdraw, getBalance, and getName. Th ere could have been more, or there could have been fewer. Th e member functions for BankAccount were chosen to keep the class simple and to provide a collection of operations that are relatively easy to relate to. A compromise was made. Th e design decisions were infl uenced by the context—a fi rst example of a C++ class used in a particular domain, the area of banking. Th e BankAccount member functions that make up the interface are only a subset of the operations named by students who were asked this question: what should we be able to do with bank accounts? Th e data members are also a subset of the operations named by students who were asked this question: what should bank accounts know about themselves?

Many additional operations that were recognized by students (transfer, applyInterest, printMonthlyStatement) and many additional data members (type of account, record of transactions, address, Social Security number, and mother’s maiden name) were not included. Th e design of these classes was aff ected by the intention of keeping these objects as simple as possible while retaining some realism. However, a group of object-oriented designers developing large-scale applications in the banking domain would likely retain many of the operations and attributes recognized by students. Th ere is rarely one single design that is correct for all circumstances.

Designing anything requires making decisions in an eff ort to make the thing “good.” Good might mean having a software component that is easily maintainable; it might mean classes that can be reused in other applications; or it might mean a system that is very robust—one that can recover from almost any disastrous event. Good might mean a design that results in something that is easier to use, prettier, etc. Th ere is rarely ever a single perfect design. Th ere are usually trade-off s. Design is an iterative process that evolves with time.

Design is infl uenced by personal opinion, evolving research, and the domain, which could be banking, information systems, process control, engineering, and so on. Fortunately, there are design guidelines to show the way, a few of which are presented later. Let’s now turn to the construct that captures many of these design decisions in object-oriented software development—the *class defi nition*.

Th e classes of objects under study—ostream, istream, string, BankAccount, and Grid—are building blocks of larger programs. However, programs typically require many other classes. Th ey may be standard classes, classes that are bought off the shelf, or other classes that must be designed and implemented by a programming team.

Because it is diffi cult to have mastery of all classes in a large project, this section provides some general techniques for understanding unfamiliar classes. Th e knowledge attained here also provides experience with the major component of object-oriented software development—the class.

6.1: Defi ning a Class in a Header File

Th is process begins with learning to read class defi nitions. You will also implement member functions and add new operations to existing classes. Th is approach has the added benefi t of making it easier to design and implement new classes of your own.

A class defi nition lists member functions after the keyword public:. Th is set of operations represents the class interface. Th e class defi nition also lists the *data members*—the object declarations after private:. Th is set of data members represents the state of the objects.

A class defi nition provides a lot of information. A class defi nition stresses the *what*, not the *how*. It lists the messages understood by the objects. It specifi es the number, type, and order of arguments required when sending a message to one of the objects. When documented with preconditions, postconditions, and example messages, a class defi nition also explains how to use instances of the class. Th e documentation may provide other pertinent information. All of these things allow the programmer to use objects of the class without knowing the details of the implementation.

#### General Form 6.1 *Class defi nition*

class *class-name* {

public: // MEMBER FUNCTIONS (the interface)

//--constructor

*class-name*(*parameter-list*);

//--modi ers

*function-heading*; // Member functions that *function-heading*; // modi es the state

//--accessors

*function-heading* const; // Member function that access *function-heading* const; // but can't change state . . .

private: // DATA MEMBERS (the state) *object-declaration* // Data member *object-declaration* // Data member . . .

}; // Class de nitions must end with a semicolon

#### 6.1.1 DEFINING classBankAccount

Now let’s get down to a concrete, familiar example. Recall that the data members in the private section represent the state. Every BankAccount object stores its own private name and balance data. Th e public section has the member functions representing the messages each BankAccount understands: withdraw, deposit, getBalance, and getName. Th ese are combined in the header fi le BankAccount.h as a class defi nition.

##### Class Defi nition: *BankAccount*

File BankAccount.h

#include <string>

// Do not place using statements in header les. Use std::

class BankAccount { public: BankAccount(std::string initName, double initBalance); // post: Construct with two arguments, example:

// BankAccount anAcct("Hall", 100.00);

void deposit(double depositAmount); // post: Credit depositAmount to the balance

void withdraw(double withdrawalAmount); // post: Debit withdrawalAmount from the balance

double getBalance() const; // post: Return this account's current balance

std::string getName() const; // post: Return this account's name

private:

std::string name; double balance; }; // Don't forget the semicolon

Most BankAccount member function headings in this BankAccount class defi nition are similar to the nonmember function headings—they usually have return types and parameters. However, one of the member function headings above does not fall into this category. Can you spot something diff erent about the function heading with the name BankAccount?

First of all, the BankAccount::BankAccount member function has no return type. It also has the same name as the class! Th ese special member functions are referred to as *constructors* because they are used to “build” objects. Specifi cally, constructors associate the object name with a portion of memory and initialize the data members of the object as in a BankAccount object construction. For example, this code constructs a BankAccount object with an initial name of "Pat Barker" and an initial balance of 507.34, which can be referenced with the variable named anAccount.

BankAccount anAccount("Pat Barker", 507.34); When another object is constructed like this:

BankAccount another("Skyler Boatwright", 437.05);

6.1: Defi ning a Class in a Header File

there exists a separate BankAccount object with its own balance of 437.05 and its own name of "Skyler Boatwright". So the return values of these two messages would be 507.34 followed by 437.05.

cout << anAccount.balance() << endl; // 507.34 cout << another.balance() << endl; // 437.05

#### SELF-CHECK

Use this class defi nition to answer the self-check questions that follow:

/\*

* Class de nition for LibraryBook
* le: LibraryBook.h

\*/

#include<string>

class LibraryBook { public:

//--constructor

LibraryBook(std::string initTitle, std::string initAuthor);

// post: Initialize a LibraryBook object

//--modi ers void borrowBook(std::string borrowersName);

// post: Records the borrower's name // and makes this book not available

void returnBook(); // post: The book becomes available

//--accessors

bool isAvailable() const; // post: returns true if this book is not borrowed

std::string getBorrower() const; // post: Return borrower's name if this book is not available

std::string getBookInfo() const; // post: Returns this book's title and author

private:

std::string author; std::string title; std::string borrower; bool available; };

6-1 What is the name of the class defi ned above?

6-2 Name all the member functions that modify the state of the objects.

6-3 Name all the member functions that access the state of the objects and cannot change that state.

6-4 Name all data members.

6-5 What type of value is returned by LibraryBook::getBorrower? 6-6 What type of value is returned by LibraryBook::isAvailable?

6-7 Initialize one LibraryBook object using your favorite book and author.

6-8 Send the message that borrows your favorite book. Use your own name as the argument.

6-9 Write the message that returns the borrower’s name of the book.

### 6.2 IMPLEMENTING CLASS MEMBER FUNCTIONS

Class member function implementations are similar to those of their nonmember relatives—with these diff erences:

1. Class member functions implemented outside of the class defi nition must be qualifi ed with the class name and the scope resolution operator ::. Th is tells the compiler they are member functions of a particular class and as such, they are allowed to directly reference the private data members.
2. Th e constructors are class member functions with the same name as the class and they do not have a return type. Th e return type is not needed because constructors return a new object of the type specifi ed in the constructor and class name.

Th e relatively familiar BankAccount class will be used to demonstrate member function implementations. For each .h fi le there will be a .cpp fi le that #includes the .h (header) fi le with the class defi nition. Th is .cpp fi le implements the member functions.

#### 6.2.1 IMPLEMENTING CONSTRUCTORS

A constructor is a special member function that always has the same name as the class. It never has a return type. Although member functions can be defi ned within a class defi nition, this textbook uses the software engineering principle of separating interface from implementation by implementing the member functions in a separate fi le. In this case, the member functions must begin with class-name ::.

Th e following code implements the two-parameter constructor:

// File name: BankAccount.cpp

#include "BankAccount.h" // Allows for separate compilation

BankAccount::BankAccount(string initName, double initBalance) { name = initName;

6.2: Implementing Class Member Functions

balance = initBalance; } // . . . more member functions need to be implemented . . .

Th is is the function that executes whenever a BankAccount is initialized with two arguments: a string followed by a number.

In the following code, the account name "Corker" is passed to the parameter initName, which in turn is assigned to the private data member name. Th e starting balance of 250.55 is also passed to the parameter named initBalance, which in turn is assigned to the private data member balance. After an object is constructed, the state of the object is initialized.

// Call the two-parameter constructor

BankAccount anInitializedAccount("Corker", 250.55);

// Output: cout << anInitializedAccount.getName() << endl; // Corker cout << anInitializedAccount.getBalance() << endl; // 250.55

Th ere is a major diff erence between implementing class member functions and their nonmember cousins. Class member function implementations must be preceded with the class name and the :: operator. For example, the BankAccount constructor is preceded with BankAccount:: to inform the compiler that it is a member function and as such, has access to the object’s private data members. Failure to add BankAccount:: results in a nonmember function that cannot reference the data members. For example, the compiler will generate error messages at any attempt to access private data members (name and balance). BankAccount:: is missing.

BankAccount(string initName, double initBalance) { // <-- WHOOPS name = initName; // ERROR: name is not known balance = initBalance; // ERROR: balance is not known }

##### Scope Rule for C++ Classes

Th e scope of private members is limited to the class member functions.

So remember to precede a class member function implementation with the class to which it belongs and the special symbol ::. Th is defi nes the function as a class member function that can access the private data members. A member function can do whatever it has to do with the state.

#### 6.2.2 IMPLEMENTING MODIFYING MEMBER FUNCTIONS

A member function may either modify the state or access the state of an instance of the class. For example, consider BankAccount::deposit, which modifi es the private data member balance.

void BankAccount::deposit(double depositAmount) { balance = balance + depositAmount;

}

When the following deposit message is sent, the argument 157.42 is copied by value to the parameter depositAmount, which is then added to this object’s balance:

anAcct.deposit(157.42);

Notice that the function headings match the class defi nition. Specifi cally, the return type of BankAccount::deposit is void and there is one double argument.

// function headings in BankAccount.h//--modi ers void deposit(double depositAmount); void withdraw(double withdrawalAmount); // . . .

Th e BankAccount::withdraw function is another modifying member function that changes the state of a BankAccount object. Specifi cally, a withdraw message deducts withdrawalAmount from balance:

void BankAccount::withdraw(double withdrawalAmount) { balance = balance - withdrawalAmount; }

When the following withdraw message is sent, the argument 50.00 is copied by value to the parameter withdrawalAmount, which is then subtracted from balance:

anAcct.withdraw(50.00);

As you are implementing class member functions, make sure all function headings match the appropriate function heading in the class defi nition. Your member function implementations, stored in a diff erent fi le, must have the same exact return type, function name, number, type, and order of parameters as exist in the class defi nition. A good idea is to copy all the member functions to your implementation fi le. Th at will ensure you keep member function headings the same and you won’t miss implementing a member function. Th en replace the semicolon at the end of each member function heading with a function body and add the class name:: to the start of each member function name.

It should be noted here that there could be much more processing within the body of a class member function. Th e member function implementations in this chapter have been kept intentionally simple during this introduction to member function implementations.

#### 6.2.3 IMPLEMENTING ACCESSING MEMBER FUNCTIONS

It is good design to make the data members private and have functions that allow access to that state. Some of these accessing functions simply return the value of a data member.

string BankAccount::getName() const { return name; }

double BankAccount::getBalance() const { return balance;

}

6.2: Implementing Class Member Functions

Because these accessing functions in the class defi nition have the keyword const, the implementation of the member function must also include const after the function heading and before the block start at {. Th e keyword const denotes a member function that does not modify state. If you examine the accessor implementations above, you’ll notice no data members get changed in the block. getName and getBalance simply return the values of those data members. If you pass an object by const & reference, these const methods can be used in that other function. On the other hand, the modifying functions withdraw and deposit change the state of the object, balance specifi cally. If you pass an object by const & reference, an attempt to use these modifying (non const) methods will be the source of a compile time error. Th e const function can be used when passed by const reference.

Remember to make sure all member function headings exactly match the headings in the class defi nitions (without ;). And remember to type the class name and :: before the class member function name in the .cpp fi les. To summarize, here is the complete implementation of all of BankAccount member functions in the fi le BankAccount.cpp.

##### Member Function Implementation: *BankAccount*

File BankAccount.cpp

/\*

* Implement the member functions de ned in BankAccount.h

\*

* File name: BankAccount.cpp

\*/

#include "BankAccount.h" using namespace std;

//--constructor

BankAccount::BankAccount(string initName, double initBalance) { name = initName; balance = initBalance; }

//--modi ers void BankAccount::deposit(double depositAmount) { balance = balance + depositAmount; }

void BankAccount::withdraw(double withdrawalAmount) { balance = balance - withdrawalAmount; }

//--accessors

double BankAccount::getBalance() const { return balance; } string BankAccount::getName() const { return name; }

#### SELF-CHECK

6-10 How does a function implementation become a member of a class?

6-11 Can class member functions reference the private data members?

6-12 Can nonmember functions reference private data members?

6-13 Use this implementation of the LibraryBook member functions to write the output generated by the program below:

/\*

* Implement the member functions de ned in LibraryBook.h

\*

* File name: LibraryBook.cpp

\*/

#include <string> using namespace std; #include "LibraryBook.h" const std::string AVAILABLE\_MESSAGE = "CAN BORROW";

//--two argument constructor

LibraryBook::LibraryBook(std::string bookTitle, std::string bookAuthor) { title = bookTitle; author = bookAuthor; available = true;

borrower = AVAILABLE\_MESSAGE; }

// -- modi ers -void LibraryBook::borrowBook(std::string borrowersName) { borrower = borrowersName; available = false; }

void LibraryBook::returnBook() { borrower = AVAILABLE\_MESSAGE; available = true; }

//--accessors

bool LibraryBook::isAvailable() const { return available; }

std::string LibraryBook::getBorrower() const { return borrower;

}

6.3: Default Constructors

std::string LibraryBook::getBookInfo() const { return "'" + title + "' by " + author; }

Here is the program that uses this new type LibraryBook now implemented as a C++ class.

// Send every possible message to a LibraryBook object

#include <iostream> using namespace std;

#include "LibraryBook.h" // For class LibraryBook de nition

int main() {

LibraryBook aBook("Tinker Tailor Soldier Spy", "John le Carre"); cout << aBook.getBookInfo() << endl; cout << aBook.getBorrower() << endl;

cout << aBook.isAvailable() << endl; // 1 if true, 0 if false aBook.borrowBook("Charlie Archer"); cout << aBook.getBorrower() << endl; cout << aBook.isAvailable() << endl; aBook.returnBook();

cout << aBook.isAvailable() << endl; cout << aBook.getBorrower() << endl; return 0; }

##### Output

'Tinker Tailor Soldier Spy' by John le Carre

CAN BORROW

1

Charlie Archer

0

1

CAN BORROW

### 6.3 DEFAULT CONSTRUCTORS

Every class requires at least one constructor. A class can have more than one, as long as they have a diff erent number, type and/or order of parameters. Consider this simple class Adder that has two constructors. Th e constructor with no parameters is known as the default constructor. Th e programmer can specify whatever default state seems appropriate in the default constructor, which in this case sets the data member sum to 0.0.

// File: Adder.h

#include <string> class Adder { public:

// Default constructors have no parameters.

// Construct an Adder with sum staring at 0.0

Adder();

// Construct an Adder with sum starting at start

Adder(double start);

void add(double number); // post: add number to sum

double getSum() const; // post: Return the sum of all added numbers

private:

double sum; // total of all scores added };

Because class Adder has two constructors, Adder objects can be constructed two diff erent ways.

Adder adder1(123.45); // Call one argument constructor

Adder adder2; // New: Call the default constructor, no ()

Th e object referenced by adder1 gets built using the one argument constructor

Adder::Adder(double start). Th e object referenced by adder2 gets built using the default constructor Adder::Adder(), which initializes sum to 0.0 as shown in the class implementation fi le Adder.cpp.

#include "Adder.h" using namespace std;

Adder::Adder() { sum = 0.0; }

Adder::Adder(double start) { sum = start; }

void Adder::add(double number) { sum = sum + number; }

double Adder::getSum() const { return sum; }

Th e following program uses both constructors to show the diff erence.

#include <iostream> using namespace std; #include "Adder.h"

int main() {

Adder adder1(123.45);

cout << " Initial sum: " << adder1.getSum() << endl;

6.3: Default Constructors

Adder adder2;

cout << " Default sum: " << adder2.getSum() << endl;

adder2.add(1.1); adder2.add(2.2); adder2.add(3.3);

cout << "After 3 adds: " << adder2.getSum() << endl;

return 0; }

#### Output

Initial sum: 123.45

Default sum: 0

After 3 adds: 6.6

Here are the reasons to have a default constructor in addition to other constructors:

* Th ey are required to have collections of objects (see Chapter 10, “Vectors”).
* Th ey guarantee initialization to a specifi c state. Programmers always know what to expect (more vivid examples are yet to come).
* Th ey defi ne the default values used when another default constructor is called. For example, the default state for string is the empty string "".

#### 6.3.1 FUNCTION OVERLOADING

You may be wondering how there could be two constructors since they have the same name. Th rough a technique known as *function overloading*, more than one function with the same name is allowed to exist. However, there has to be something that distinguishes two functions with the same name. One of these distinguishing characteristics is having a diff erent number of parameters. Function overloading allows the programmer to have a default constructor with zero parameters in the same scope as a constructor with one or more parameters. In other words, C++ distinguishes between the two constructor function headings inside the class defi nition. Function overloading also occurs when the type of parameters diff ers, even if there are the same number of parameters. Th ese three functions may exist in the same scope because the types of the one parameter are diff erent.

void aFunction(int n); void aFunction(long n); void aFunction(string str);

Th ese functions may exist in the same scope because the order of parameters is diff erent.

void aFunction(int n, string s); void aFunction(string s, int n);

However, functions that diff er only in their return type cannot be overloaded.

void aFunction(int n); string aFunction(int n); // <- Error

### 6.4 THE STATE OBJECT PATTERN

Even though quite diff erent in specifi c operations and state, string, BankAccount, and LibraryBook objects have the following common characteristics:

* private data members store the state of the object
* constructors initialize the state
* some messages modify the state
* other messages allow access to the current state of the object

Th ese commonalities guide the eff ective use of these and similar classes of objects. Th ese patterns also help programmers understand how to use new objects. Th e constructors, modifi ers, and accessors in the public: section of a class defi nition are the operations available to all instances of the class.

#### 6.4.1 CONSTRUCTORS

Constructors are present for many reasons, including initializing the state of any instance of the class. As shown earlier, objects are initialized like this:

string aString("initial string"); // State is "initial string" BankAccount anAcct("Xi Grey", 215); // name and balance are set

LibraryBook aBook("Tale of Two Cities", "Charles Dickens");

// Title and author are set and this book is available to borrow

#### 6.4.2 MODIFIERS

Modifying methods change the state of an object. Modifi ers are part of the State Object pattern for a variety of reasons. Perhaps it’s best to simply show some example messages that modify the state of an object:

aString.replace(1, 3, "NEW"); // assert: s2 is "iNEWial string"

g.move(5);

// assert: The mover is ve spaces forward

anAcct.withdraw(50.00); // assert: The balance of anAcct is 50.00 less

aBook.borrowBook("Fred Featherstone");

// assert: aBook's borrower has become Fred Featherstone

6.4: The State Object Pattern

Sending a modifi er message results in a change of state. Modifi ers are not declared with const after the function heading—accessors are.

#### 6.4.3 ACCESSORS

Accessors are part of the state object pattern simply because programmers often need to access the state of an object. An accessor message returns information related to the state of an object. An accessor may simply return the value of a data member as with LibraryBook::borrower and BankAccount::balance. Accessors may also need to do some internal processing using the state of an object to return the information (employee::incomeTax, for instance). Here are some example messages that access the state of objects:

s2.length() // Return the number of characters in s2

g.row() // Return the mover's current row anAcct.getBalance() // Return the current balance of anAcct aBook.getBorrower() // Return the borrower's name of aBook

#### 6.4.4 NAMING CONVENTIONS

Modifying operations are typically given a name that indicates the message will change the state of the object. Th is is easily accomplished if the designer of the class simply gives a descriptive name to the operation. Th e name should describe—as best as possible—what the operation actually does. Another way to help programmers who use a class to distinguish modifi ers from accessors is to give the modifi ers names that can be used as verbs such as withdraw, deposit, borrowBook, and returnBook, for example. Th e accessors are given names that often begin with “get,” such as getBorrower and getBalance. Considering that the constructor has the same name as the class, some guidelines are established for designing and reading class defi nitions. Th ese three categories of operations—typical of state objects—can be distinguished by using the following naming conventions.

|  |  |
| --- | --- |
| **Operation** | **Name** |
| Constructor | Same name as the class |
| Modifi er | Identifi er name that could be used as a verb |
| Accessor | Identifi er name that begins with “get” |

Above all, always try to use identifi ers that describe what the object is. For example, don’t use x as the name of the operation to withdraw money from a BankAccount or turnRight to make the mover turn left.

**6.4.5 public: OR private:**

One of the considerations in the design of a class is the placement of member functions and data members under the most appropriate access mode, either public: or private:. Whereas public members of a class can be called from another function outside of the class, the scope of private members is limited to the class member functions. For example, the BankAccount data member balance is only known to the member functions of the BankAccount class. On the other hand, any member declared in the public: section of a class is known everywhere in the class and also in the block of source code where the object is declared (or globally, if defi ned outside of a block).

|  |  |
| --- | --- |
| **Access Mode** | **Where Is the Member Known?** |
| public: | In all class member functions and in the block of the client code where the object has been declared (in main, for instance). |
| private: | Only inside the class member functions. Because these are known everywhere in the class, you do not have to pass or return those values among the class member functions. |

Although the data members representing state could have been declared under public:, it is highly recommended that all data members be declared under the private: access mode. Th ere are several reasons for this.

Th e consistency helps simplify some design decisions. More importantly, when data members are made private:, the state can be modifi ed only through a member function. Th is prevents client code from indiscriminately changing the state of objects. For example, it’s impossible to accidentally make a credit like this from anywhere outside of the class:

// An error occurs: attempting to modify private data myAcct.balance = myAcct.balance + 100000.00; // <- ERROR

or a debit like this:

// An error occurs: attempting to modify private data myAcct.balance = myAcct.balance - 100.00;

#### 6.4.6 SEPARATING INTERFACE FROM IMPLEMENTATION

Th e practice of studying a class through its interface represents a principle in software engineering. It allows one to separate the interface from the implementation—the details of how the operations actually work. In C++, the completed member function implementations are often separated from the class defi nition by placing them in separate fi les. Historically, class defi nitions have been kept in .h (header) fi les with member function implementations in .cpp fi les (fi le extensions vary). Some programmers implement the member functions directly in the same fi le as the class defi nitions.

Th e convention used in this textbook is to separate the class defi nition from the implementation. Th is is done by storing the class defi nition in a .h fi le and the member function implementations in a .cpp fi le. It is often the case that several fi les are combined together to make an executable program. Th ere are several ways to do this. Th e following fi gure illustrates one way to do this with these commands using the GNU compiler:

6.4: The State Object Pattern

g++ -c BankAccount.cpp g++ -c main.cpp g++ -o main main.o BankAccount.o

./main

/\*

\* Implement the member functions

\* defined in BankAccount.h

\*

\* File name: BankAccount.cpp

\*/

#include "BankAccount.h"

//--constructor

BankAccount::BankAccount(std::string initName,

double initBalance){

name = initName;

balance = initBalance;

}

…

/\*

\* Define class BankAccount

\*

\* File name: BankAccount.h

\*/

#ifndef BANKACCOUNT\_H\_

#define BANKACCOUNT\_H\_

#include <string>

class BankAccount {

public:

BankAccount(std::string initName,

double initBalance);

#include <iostream>

using namespace std;

#include "BankAccount.h"

int main() {

BankAccount ba1("Miller", 100.00);

BankAccount ba2("Barber", 987.65);

ba1.deposit(133.33);

ba2.withdraw(250.00);

cout << ba1.getName() << ": " << ba1.getBalance() << endl;

cout << ba2.getName() << ": " << ba2.getBalance() << endl;

return 0;

}

g++ -c BankAccount.cpp

g++ -o main main.o BankAccount.o

g++ -c main.cpp

./main

Miller: 233.33

Barber: 737.65



Ba

nk

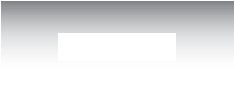
Ac

co

un

t.

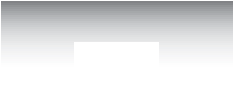
o



ma

in

.o



ma

in

Compiler

Linker

#### SELF-CHECK

6-14 What is meant when the const keyword is part of the function heading in a class defi nition?

6-15 Which member functions have the same name as the class?

6-16 What do accessors do?

6-17 What do modifi ers do?

6-18 What do constructors do?

6-19 What are the data members for?

### 6.5 OBJECT-ORIENTED DESIGN GUIDELINES

One particular object-oriented design decision involves determining where to place the data members that store object state. More specifi cally, since this text uses C++ as the implementation language, the designer has to decide if data member functions go in the public: or the private: section of a C++ class. Th e following design guideline states that a good design protects object state from the outside world:

#### Object-Oriented Design Guideline

All data should be hidden within its class.

Although data members could be public:, the convention used in this text—and in any well-designed class—is to hide the data members. C++ data members are easily hidden when declared in the private: section of the class defi nition. Th is simplifi es some design decisions. A private: data member can then only be modifi ed or accessed through messages.

Th is prevents users of the class from indiscriminately changing certain data such as an account balance. Th e state of an object can be protected from accidental or improper alteration. With data members declared in the private: section, the state of any object can only be altered through a message. It becomes impossible to accidentally make a false debit like this:

// Compile time error: attempt to modify private data

// If balance is public:, what is the new balance? myAcct.balance = myAcct.balance - myAcct.balance;

However, if balance had been declared in the public: section, the compiler would not protest. Th e resulting program would allow you to destroy the state of any object. Th e hidden balance is more properly modifi ed only when the transaction is allowed according to some policy. What happens, for instance, if a withdrawal amount exceeds the account balance in a withdraw

6.5: Object-Oriented Design Guidelines

message? Some accounts allow this by transferring money from a savings account. Other bank accounts may generate loans in increments of $100.00.

With balance declared in the private: access section, users of the class must instead send a withdraw message. Th e client code relies on the BankAccount to determine if the withdrawal is to be allowed. Perhaps the BankAccount object will ask some other object if the withdrawal is to be allowed. Perhaps it delegates authority to some unseen bankManager object. Perhaps the BankAccount object itself can decide what to do. Although this text’s implementation of BankAccount doesn’t do much, real-world withdrawals do.

By hiding data and other details, all credits and debits must “go through the proper channels.” Th is might be quite complex. For example, each withdrawal or deposit may be recorded in a transaction fi le to help prepare monthly statements for each BankAccount. Th e withdraw and deposit operations may have additional processing to prevent unauthorized credits and debits. Part of the hidden red tape might include manual verifi cation of a deposit or a check-clearing operation at the host bank; there may be some sort of human or computer intervention before any credit is actually made. Such additional processing and protection within the deposit and withdraw operations help give BankAccount a “safer” design. Because all hidden processing and protection is easily circumvented when data members are exposed in the public: section, the object designer must enforce proper object use and protection by hiding the data members.

#### 6.5.1 COHESION WITHIN A CLASS

Th e set of messages described in the class interface should be strongly related. A class stores data, and that data should be strongly related. In fact, all elements of a class should have a persuasive affi liation with each other. Th ese ideas relate to the preference for tight cohesion (solidarity, hanging together, adherence, unity) within a class. For example, don’t expect a BankAccount object to understand the message isPreheated. Th is may be an appropriate message for an oven object, but certainly not for a BankAccount object. Here is one guideline related to the desirable attribute of cohesion:

##### Object-Oriented Design Guideline

Keep related data and behavior in one place.

Th e BankAccount class should hide certain policies such as handling withdrawal requests greater than the balance. Th e system’s design improves when behavior and data combine to accomplish the withdrawal algorithm. Th is makes for nice clean messages from the client code, like this:

anAccount.withdraw(withdrawalAmount);

Th is client code relies on the BankAccount object to determine what should happen. Th e behavior should be built into the object that has the necessary data. Perhaps the algorithm allows a withdrawal amount greater than the balance—with the extra cash coming as a loan or as a transfer from a savings account. Even though the BankAccount class of this textbook does very little, a real bank account class might have eight diff erent actions that are triggered for every withdrawal—all behind the scenes.

**6.5.2 WHY ARE ACCESSORS const AND MODIFIERS NOT?**

You may be wondering why const is added to function headings intended to access, rather than modify, the object’s state. Th e answer has to do with the three diff erent parameter modes. When an object is passed by value or by reference to a function, that function can send any and all possible messages to that object inside the other function. However, when the const reference parameter mode is utilized, the function promises not to change that object. In fact, it cannot. To illustrate, consider the following function that will not compile—there is a compile time error at the attempt to withdraw from the const reference parameter ba. Th is is actually a good thing. Th e reason for using const reference parameters is to avoid accidental modifi cation of the associated argument.

// Illustrate connection between member functions tagged as const

// functions and passing objects of that class as const parameter

#include <iostream> // For cout and endl using namespace std; #include "BankAccount.h" // For the BankAccount class

void display(const BankAccount & ba) {

// Can send accessing messages--they are declared const cout << "{ BankAccount: " << ba.getName() << ", $" << ba.getBalance() << " }" << endl;

// This modi er was not tagged with const. A compile time // error will be generated since ba is a const parameter.

ba.withdraw(234.56); // <-- ERROR at compile time }

int main() { BankAccount anAcct("Angel Draper", 1234.56);

display(anAcct); return 0; }

Th is protection works fi ne for standard classes such as string. Th e same protection will only work with your new classes if care is taken to tag the accessors as const and leave the modifi ers as non-const.

A consistent use of const accessors allows the accessing messages to be sent to the const parameters. At the same time, by not using const with modifi ers, a const parameter prevents use of a message that will change the object.

6.5: Object-Oriented Design Guidelines

##### Object-Oriented Design Guideline

Only const messages are allowed on const parameters.

On the other hand, it is okay to send messages that do not modify the object. Th is safety net is possible only when the programmer diligently tags accessing class member functions with const and always remembers not to tag a modifi er that way.

class BankAccount {

public: //--modi ers void deposit(double depositAmount); // No const for modi ers void withdraw(double withdrawalAmount);

//--accessors

double getBalance() const; // Use const on accessors string getName() const; // . . .

Th is leads to another design guideline:

##### Object-Oriented Design Guideline

Always declare accessor member functions as const.

Perhaps the biggest problem with this guideline is in remembering the guideline. It is easily violated. You’ll never know the ramifi cations until an instance of your class is passed as a const reference parameter. As another example, consider the Grid class modifi ers, which are non-const, and some accessors, which are declared as const functions.

class Grid { public: . . . //--modi ers void move(int spaces);

. . .

//--accessors int row() const; int column() const;

. . .

};

Th e presence of const tells the compiler to allow the message to be sent even for objects passed by const reference (g here):

void doSomething(const Grid & g) { cout << g.row() << endl; // OKAY cout << g.nColumns() << endl; // OKAY

g.display(); // OKAY

g.move(); // Compile time ERROR

g.pickUp(); // Compile time ERROR }

On the other hand, the attempt to send non-const messages such as Grid::move results in a compile time error like these (more cryptic error messages exist) depending on the compiler used:

non-const member function 'Grid::move()' called for const object

* or -

attempt to modify a const object

* or -

member function 'pickUp' not viable: 'this' argument has type 'const Grid', but function is not marked const

Declaring accessors as const functions allows existing objects to be safely passed to a const parameter. However, it takes diligence to maintain the same safety net for the new classes that you write. Remember these two class design guidelines:

* 1. Modifi ers should *not* be declared const so the compiler can catch attempts to modify const objects.
  2. Accessors should be declared const so objects can be safely passed to const parameters and still allow non-modifying messages.

It would be easier to completely ignore these rules, but the only way to get away with it would be to never pass objects to const parameters. Th is textbook uses const in a member function because it says something about whether or not a function modifi es the state of an object. And this is something object-oriented programmers must know about. Th e designer of the class must still decide if the message will modify an instance of the class or not.

#### SELF-CHECK

6-20 Using the class defi nition of the BankAccount class, list the lines that cause errors in a standard C++ compiler (1, 2, 3, and/or 4).

#include <iostream> using namespace std; #include "BankAccount.h" // For the BankAccount class

void check(const BankAccount & b, double amount) { cout << b.getName() << endl; // 1 b.deposit(amount); // 2

b.withdraw(amount); // 3 cout << b.getBalance() << endl; // 4

}

Chapter Summary

int main() {

BankAccount myAcct("Me", 12345.00); check(myAcct, 50.00); return 0; }

### CHAPTER SUMMARY

* Th is chapter showed class defi nitions with a collection of function headings that represent the class interface. Th ese are the message names that any object of the class will understand.
* A class defi nition lists:
* the class member functions with parameters and return types, collectively known as the interface
* the data members, known collectively as the state
* Each object of a class may store many values, which may be of diff erent classes. For example, each BankAccount object stores string data for the name and numeric data for the balance.
* Th e state object pattern guides class design when the primary need for the object is to store state and provide adequate access to it. Th e state object pattern in C++ recommends that the following items be included in a class defi nition:
* a constructor to initialize objects with programmer-supplied state
* modifying functions
* accessor functions
* private data members to store the state of every object Modifying class member functions changes the state of the object.
* Accessor functions provide access to the state of an object.
* Accessors have the keyword const attached at the end of the function heading.
* Ramifi cations of adhering to Object-Oriented Design Guideline “All data should be hidden within its class” include:
* Good: can’t mess up the state (compiler complains)
* Bad: need to implement additional accessors (getBalance, for example)
* Th e ramifi cations of adhering to Design Guideline “Keep related data and behavior in one place” include:
* Good: results in a more intuitive design
* Good: easier to maintain
* Th e ramifi cations of adhering to Design Guideline “Always declare accessor member func-tions as const” include:
* Good: helps the user distinguish between modifi ers and accessors
* Good: adheres to the principle that objects passed as const reference parameters cannot be accidentally modifi ed by the function while allowing the function to send const messages
* Bad: it is easy to forget to use const and the error will not show up until the object is passed in the three diff erent modes—the result is more extensive testing to ensure the safety of const and the effi ciency of const reference parameters
* Class member functions are implemented in a manner similar to nonmember functions. However, class member functions must be qualifi ed with the class name and :: (the scope resolution operator). Th is gives the function access to the private data members.
* Class defi nitions have historically been stored in .h fi les.
* Member function implementations have historically been stored in .cpp fi les. A class should be designed to exhibit high cohesion: the data should be related to the operations
* the messages should be related to each other

### EXERCISES

1. Does the interface of a class refer to its member functions or its data members?
2. Does the client code need to know the names of data members to use objects of the class?
3. Describe the scope of the public members of a class.
4. Describe the scope of the private members of a class.
5. Give one justifi cation for making the data members of a class private.
6. If the designer of BankAccount class changed the name balance to my\_Balance, would programs using BankAccount need to be changed?
7. If a designer changed the name of the withdraw message to withdrawThisAmount after the class was already in use by dozens of programs, would these dozens of programs need to be changed?
8. What is responsible for deciding if a particular LibraryBook is available for lending, the LibraryBook or the program using LibraryBook?
9. Should a BankAccount object understand the message isThisBrakeLockingUp?
10. If an object is passed by value, which messages can be sent: modifi ers, accessors, or both?
11. If an object is passed by reference (with &), which set of messages can be sent: modifi ers, accessors, or both?
12. If an object is passed by const reference as in (const Grid & aGrid), which set of messages can be sent: modifi ers, accessors, or both?
13. Given this defi nition for a class Counter class, predict the output from the test driver below:

Exercises

/\*

\* Filename: Counter.h

\*/ class Counter { public:

//-- constructor

Counter(int maxValue);

// post: Initialize count to 1 and set the maximum count

// modi ers void click(); // post: If count is at maximum, set count to 1, otherwise add 1

// to the count. This uses the % operator when adding to count.

void reset(); // post: Resets the counter to 1

// accessor

int getCount() const; // post: Return the current count

private:

int count; // Current count, always start at 1 int max; // The largest value count can reach };

**TEST DRIVER**

#include <iostream> using namespace std;

#include "Counter.h" // For the counter class de nition

int main() { // Test drive counter class

Counter aCounter(3);

cout << "a: " << aCounter.getCount() << endl; aCounter.click();

cout << "b: " << aCounter.getCount() << endl; aCounter.click();

cout << "c: " << aCounter.getCount() << endl; aCounter.click();

cout << "d: " << aCounter.getCount() << endl; aCounter.click();

cout << "e: " << aCounter.getCount() << endl; aCounter.reset();

cout << "f: " << aCounter.getCount() << endl; return 0; }

14. Write all code that would go into Counter.cpp that completely implements all member functions defi ned in Counter.h so that the program above generates the correct output.

### PROGRAMMING TIPS

1. Working with three fi les is more diffi cult than working with one, but some programming projects will now require that you work with three fi les, not just one. Th is takes a little patience as you grow accustomed to working with multiple fi les. Remember, the .h fi le contains the class defi nition; the .cpp fi le contains the member function implementations. Th e third fi le has the main function.
2. Th ere is a variety of ways to make classes available. Even though the convention of having one fi le include the .h and .cpp fi les is atypical, it makes things easier and matches the standard (many #include fi les do not have .h anymore). However, someday you may be asked to create object fi les or project fi les to compile and link programs using author-supplied classes. Th en your program may just include the .h fi le so it can compile. Linking comes later.

#include " BankAccount.h " // Other steps required to link int main() { // . . .

}

1. Th e nonmember function syntax applies to member function headings also. Th e function heading in the implementation must match the function heading in the class defi nition in terms of
   * return type (none for constructors)
   * function name
   * number of parameters
   * type of parameters
   * order of parameters
   * use of const in both places (or neither)
2. Don’t write using namespace std; in header fi les. Once you use a namespace you can’t unuse it. While it may not cause any problems in the programs in this textbook, you should get in the habit now to avoid future programming problems.
3. Function headings in the implementation fi le (.h) diff er from the function headings in the implementation fi le (.cpp).
   * functions need className:: to precede the function name
   * the function body { } replaces the semicolon

/\* \* File name: CD.h

\*/

#include <string> class CD {

public:

CD(std::string initArtist, std::string initTitle); std::string getArtist() const;

private:

std::string artist, title; };

/\*

\* File name: CD.cpp

\*/

#include "CD.h" using namespace std;

CD::CD(string initArtist, string initTitle) { // . . .

}

string CD::getArtist() const {

// . . .

}

### PROGRAMMING PROJECTS

#### 6A ADD int getTransactionCount TO BankAccount

Allow BankAccount objects to keep track of and report the number of transactions, deposits, and withdrawals made since the initialization of any BankAccount object. Name this new function int getTransactionCount(). Use this test driver and ensure your output matches and compiles:

#include <iostream> using namespace std; #include "BankAccount.h"

int main() {

BankAccount anAcct("Do 3", 3.00);

cout << "0? " << anAcct.getTransactionCount() << endl; anAcct.deposit(10.00); anAcct.withdraw(20.00); anAcct.deposit(30.00); cout << "3? " << anAcct.getTransactionCount() << endl;

BankAccount another("Do 1", 1.00); another.withdraw(25.00);

cout << "1? " << another.getTransactionCount() << endl;

return 0;

}

##### Output

0? 0

3? 3

1? 1

#### 6B ADD turnAround AND turnRight TO class Grid

Add the following operations to the defi nition of the Grid class in the fi le named Grid.h:

void turnAround(); // post: The mover is facing the opposite direction

void turnRight(); // post: The mover is facing 90 degrees clockwise

Also add both class member functions at the top of the fi le named Grid.cpp. Please try to ignore all the other stuff in that rather large fi le. You will fi nd it easier to use the existing member function turnLeft to implement these two new functions.

|  |  |
| --- | --- |
| #include "Grid.h"  int main() {  Grid g(6, 12, 1, 9, east);  g.display();  g.turnAround();  g.move(5);  g.turnLeft();  g.move(2);  g.turnRight();  g.move(3);  g.display(); return 0;  } | The grid:  . . . . . . . . . . . .  . . . . . . . . . > . .  . . . . . . . . . . . .  . . . . . . . . . . . .  . . . . . . . . . . . .  . . . . . . . . . . . .  The grid:  . . . . . . . . . . . .  . . . . . .  . . . . . . . . . . .  . < . . . . . . .  . . . . . . . . . . . .  . . . . . . . . . . . . |

## 6C CLASS AVERAGER

Given the following defi nition of class Averager in the fi le Averager.h, implement all member functions in a new fi le Averager.cpp. You should be able to add any number of test or quiz scores and fi nd the average and number of scores added at any time.

/\*

* De ne class Averager that maintains the average for \* any number of quiz or test scores.
* \* File name: Averager.h (available on this book's website)

\*/ class Averager { public:

// Construct an Averager with no scores added.

Averager();

//-- modi ers void addScore(double score); // post: Add a score so the count and average are correct.

//--accessors double getAverage() const; // post: Return the average of all scores entered.

int getScoresAdded() const; // post: Return how many scores were added

private:

int n; // number of scores added so far, initially 0 double sum; // total of all scores added, initially 0.0 };

Th is test driver should generate the expected output below:

#include <iostream> using namespace std; #include "Averager.h"

int main() {

Averager averager;

cout << " 0? " << averager.getScoresAdded() << endl;

averager.addScore(90.0);

cout << " 90? " << averager.getAverage() << endl; cout << " 1? " << averager.getScoresAdded() << endl;

cout << endl; averager.addScore(100.0); averager.addScore(80.0); averager.addScore(70.0); averager.addScore(60.0); averager.addScore(53.0);

cout << "Scores Added 6? " << averager.getScoresAdded() << endl; cout << " Average 75.5? " << averager.getAverage() << endl; return 0; }

### Expected Output

0? 0

90? 90

1? 1

Scores Added 6? 6

Average 75.5? 75.5

### 6D class PiggyBank

A PiggyBank object encapsulates the contents of a piggy bank with messages associated with real world actions. It knows how many of each coin—pennies, nickels, dimes, and quarters—are in it along with total cash value. A PiggyBank object can also be emptied with a drainTheBank message, which also returns the amount of money at that moment. Here is the class defi nition:

/\*

* This class models a piggy bank to which pennies, nickels, dimes,
* and quarters can be added. A PiggyBank object maintains how many
* of each coin it holds and can tell you the total amount of money \* in it.

\*

* File name: PiggyBank.h (available on this book's website)

\*/ class PiggyBank { public:

PiggyBank();

// post: An PiggyBank is built with no coins

void addPennies(int penniesAdded);

// pre: penniesAdded > 0

// post: This PiggyBank has penniesAdded more pennies

void addNickels(int nickelsAdded);

// pre: nickelsAdded > 0

// post: This PiggyBank has nickelsAdded more nickels

void addDimes(int dimesAdded);

// pre: dimesAdded > 0

// post: This PiggyBank has dimesAdded more dimes

void addQuarters(int quartersAdded);

// pre: quartersAdded > 0

// post: This PiggyBank has quartersAdded more quarters

double drainTheBank();

// post: Remove all of the coins from this PiggyBank

// and returns how much there was before it was emptied

//-- Accessors int getPennies(); // post: Return the total number of pennies in this bank

int getNickels(); // post: Return the total number of nickels in this bank

int getDimes(); // post: Return the total number of dimes in this bank

int getQuarters();

// post: Return the total number of quarters in this bank

double getTotalCashInBank();

// post: return the total cash in the bank. Pennies are

// $0.01, nickels are $0.05, dimes are $0.10, and quarters

// are $0.25 (no half or one dollar coins).

private:

int pennies, nickels, dimes, quarters; };

Th is test driver should generate the expected output below:

#include <iostream> using namespace std; #include "PiggyBank.h"

int main() { PiggyBank pb;

cout << " 0? " << pb.getTotalCashInBank() << endl; pb.addPennies(4); pb.addNickels(3); pb.addDimes(2); pb.addQuarters(1);

cout << " 4? " << pb.getPennies() << endl; cout << " 3? " << pb.getNickels() << endl; cout << " 2? " << pb.getDimes() << endl; cout << " 1? " << pb.getQuarters() << endl; cout << "0.64? " << pb.getTotalCashInBank() << endl; cout << "0.64? " << pb.drainTheBank() << endl; cout << " 0? " << pb.getTotalCashInBank() << endl; return 0; }

#### Expected Output

0? 0

4? 4

3? 3

2? 2

1? 1

0.64? 0.64

0.64? 0.64

0? 0

### 6E class Employee

*Note: Th is type asks for a new member function in Chapter 7: Selection*

While programmers at Chrystal Bends, Inc., were designing the payroll system, they realized they needed an Employee type. An Employee object is responsible for maintaining the information necessary to complete an employee’s paycheck for employees who get paid on an hourly basis. Th is Employee object is responsible for computing its own gross and net pay and computing how much to withhold for Social Security tax (6.2% of the gross pay) and Medicare tax (1.45% of the gross pay) for the week. Th e Chrystal Bends, Inc. programming team has designed this C++ class defi nition, which you are asked to use to implement and test the member functions.

/\* \* Model a weekly employee who gets paid on an hourly basis.

* Only two taxes are included so far: Medicare and Social
* Security. You may be asked to add Federal Income tax later

\*

* File name: Employee.h (available on this book's website)

\*/

#include <string>

#include <iostream>

#include <cmath>

class Employee { public:

// Constants for two taxes. C++11 needed for initialization.

const double SOCIAL\_SECURITY\_TAX\_RATE = 0.062; const double MEDICARE\_TAX\_RATE = 0.0145;

// Contructor

Employee(std::string initName, double hourlyRate);

// post: A Employee is built with 0.00 hours worked.

void giveRaise(double raise);

// pre: raise > 0. The argument 3.5 means a 3.50% raise.

// post: The hourly rate of pay has changed

void setHoursWorked(double hoursWorked); // pre: hoursWorked >= 0.0 // post: hours worked for the current week is set.

// Gross pay, net pay, and taxes can now be computed.

//--accessors std::string getName(); double getHoursWorked(); double getHourlyRate(); double getSocSecurityTax(); double getMedicareTax(); double getGrossPay(); double getNetPay();

private: // data members std::string name; double rate; double hours; };

Th e following test driver sends all possible messages to one Employee. Th e hours worked per week must be set with a setHoursWorked message in order for the gross pay taxes and net pay to be computed. Th is test driver should generate the expected output below:

#include <iostream>

#include "Employee.h" using namespace std;

// Test Driver int main() {

Employee emp1("Ali", 10.00);

cout << " Ali? " << emp1.getName() << endl; cout << " 10? " << emp1.getHourlyRate() << endl; cout << " 0? " << emp1.getHoursWorked() << endl; cout << " 0? " << emp1.getGrossPay() << endl;

// Record the hours worked in the current week emp1.setHoursWorked(40.00);

cout << " 400? " << emp1.getGrossPay() << endl; cout << " 24.8? " << emp1.getSocSecurityTax() << endl; cout << " 5.8? " << emp1.getMedicareTax() << endl; cout << " 369.4? " << emp1.getNetPay() << endl; cout << endl; emp1.giveRaise(10); // 10% raise

cout << " 11? " << emp1.getHourlyRate() << endl; cout << " 440? " << emp1.getGrossPay() << endl; cout << "406.34? " << emp1.getNetPay() << endl; }

#### Expected Output

Ali? Ali

10? 10

0? 0

0? 0

400? 400

24.8? 24.8

5.8? 5.8 369.4? 369.4

11? 11

440? 440

406.34? 406.34

© 2018 Franklin, Beedle & Associates All rights reserved

**C H A P T E R S E V E N**

# Selection

## SUMMING UP

Until this point, all programs in this textbook executed all statements in a sequential fashion, in order from the fi rst statement of each block to the last. Th e function calls and messages executed unseen code that involved other forms of statement control.

## COMING UP

Chapter 7 examines statements that select which actions execute. Depending on the current circumstances, an action may execute one time but not the next. Th e alternatives are made possible with the C++ if, if...else, and switch statements. After studying this chapter you will be able to

* recognize when to use the Guarded Action pattern (do something onlyunder certain conditions)
* implement the Guarded Action pattern with the C++ if statement
* use relational operators such as < and >
* create and evaluate expressions with the logical operators
* use bool objects
* understand the Alternative Action pattern
* implement the Alternative Action pattern with the C++ if...else statement
* implement the Multiple Selection pattern with if...else and switch
* solve problems using the Multiple Selection pattern

### 7.1 SELECTIVE CONTROL

Programs must often anticipate a variety of situations. For example, an automated teller machine (ATM) must serve valid bank customers—but it must also reject invalid access. Once validated, a customer may wish to perform a balance query, a cash withdrawal, or a deposit transaction. Th e code that controls an ATM must permit these diff erent requests. Without selective forms of control—the new statements of this chapter—all bank customers could only perform one particular transaction. Worse yet, invalid PINs could not be rejected!

175

Before any ATM becomes operational, programmers must implement code that anticipates all possible transactions. Th e code must turn away customers with invalid PINs. Th e code must prevent invalid transactions such as cash withdrawal amounts that are not in the proper increment, typically $20.00. Th e code must be able to deal with customers who attempt to withdraw more than they have. To accomplish these tasks, a new form of control is needed—a statement to permit or prevent execution of certain statements depending on certain inputs.

#### 7.1.1 THE GUARDED ACTION PATTERN

Programs often need actions that do not always execute. At one moment, a particular action must occur. At some other time—the next day or the next millisecond perhaps—the same action must be skipped. For example, one student may have made the dean’s list because the student’s grade point average (GPA) was 3.5 or higher. Th at student becomes part of the dean’s list. Th e next student may have a GPA lower than 3.5 and should not become part of the dean’s list. Th e action—adding a student to the dean’s list—is guarded. Th e *Guarded Action pattern* and the C++ means of implementing it are shown next.

**Algorithmic Pattern:** *Th e Guarded Action pattern*

|  |  |
| --- | --- |
| **Pattern:** | Guarded Action |
| **Problem:** | Do something only if certain conditions are true |
| **Outline:** | if ( *true-or-false-condition is true* ) *execute this action (s)* |
| **Code Example:** | if (GPA >= 3.5) |

cout << "Made the deans list" << endl;

#### 7.1.2 THE if STATEMENT

Th is Guarded Action pattern is often implemented with the C++ if statement.

##### General Form 7.1 if *statement*

if (*logical-expression*) *true-part*;

Th e *logical-expression* is any expression that evaluates to either true or false. Th e *true-part* may be any valid C++ statement, including a block that uses curly braces { } to treat two or more statements as if they were one statement.

7.1: Selective Control

##### Example: if *statements*

cin >> hoursStudied; if (hoursStudied > 4.5)

cout << "You are ready for the test" << endl;

if (hours > 40.0) { regularHours = 40.0; overtimeHours = hours - 40.0; }

When an if statement is encountered, the logical expression is evaluated to a false (zero) or true (nonzero) value. Th e true part executes only if the logical expression is true. So in the fi rst example above, the output "You are ready for the test" appears only when the user enters something greater than 4.5 hours. When the input is 4.5 or less, the true part is skipped—the action is guarded. Here is a fl owchart view of the Guarded Action pattern:

**Flowchart view of the**

if

**statement**

*logical-expression*

**true (nonzero)**

**false (zero)**

*Any C++ statement*

Th e next program illustrates how selection alters the fl ow of control. Each of the sample dialogues below illustrates that the code performs diff erent actions due to the variety of conditions. More specifi cally, the musicAward function returns a diff erent string due to the diff erent arguments in the three function calls from main.

// Show that the same code can return three different results.

// showAward has three instances of the Guarded Action pattern.

#include <iostream> // For cout and endl

#include <string> // For the string class using namespace std;

string musicAward(long int recordSales) {

// pre: Argument < maximum long int (usually 2,147,483,647)

// post: Return a message appropriate to record sales string result; if (recordSales < 500000)

result = "--Sorry, no certi cation yet. Try more concerts.";

if (recordSales >= 500000)

result = "--Congrats, your music is certi ed gold.";

if (recordSales >= 1000000)

result = result + " It's also gone platinum!";

return result; }

int main() {

// Test drive showAwards three times with different results cout << 123456 << musicAward( 123456) << endl; cout << 504123 << musicAward( 504123) << endl; cout << 3402394 << musicAward(3402394) << endl; return 0; }

##### Output

123456--Sorry, no certi cation yet. Try more concerts.

504123--Congrats, your music is certi ed gold.

3402394--Congrats, your music is certi ed gold. It's also gone platinum!

Th rough the power of the if statement, the same exact code results in three diff erent versions of statement execution. Th e if statement controls execution because the true part executes only when the logical expression is true. Th e if statement also controls statement execution by disregarding statements when the logical expression is false. For example, the platinum message is disregarded when recordSales is less than one million.

### 7.2 RELATIONAL OPERATORS

Two new operators, < and >=, test the relationship between the value of recordSales and the numeric values 500,000 and 1,000,000. Th ey are part of the set of relational operators that create logical expressions—an important part of if statements (see table below):

|  |  |
| --- | --- |
| **Relational Operator** | **Meaning** |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| == | Equal to |
| != | Not equal to |

7.2: Relational Operators

When a relational operator is applied to two operands that can be compared, the result is one of two values: true or false. Th e next table shows some examples of simple logical expressions and their resulting values. Notice that objects such as double and string can be compared to other objects of the same class. string objects are related alphabetically—"A" is less than "B" and "D" is greater than "C", for example.

#### Logical Expression Result Logical Expression Result

|  |  |  |  |
| --- | --- | --- | --- |
| double x = 4.0;  x < 5.0 x > 5.0 x <= 5.0 5.0 == x x != 5.0 | true false true false true | string name = "Bill";  name == "Sue" name != "Sue" name < "Chris"  "Bobbie" > Bobby"  "Bob" < "Bobbie" | false true true false true |

Th is is a good time to point out an all-too-common and diffi cult-to-track-down error that can create havoc. All math courses you have ever taken use = for algebraic equality. When you try to do that in C++, you will actually be using the assignment operator = rather than ==, the C++ equality operator. Th e problem is that the compiler does not detect an error. Consider this if statement:

int x = 0; if (x = 3) cout << x << " equals 3" << endl;

##### Output

3 equals 3

First x was 0, then it became 3 while testing the logical expression x = 3, which is an assignment. It turns out that C++ assignment operations evaluate to the value being stored. Th e expression x = 3 not only assigns 3 to x, it also evaluates to the value actually assigned, which in this case is 3, or nonzero, or true. If you want to compare x to 3, use == like this:

int x = 0; if (x == 3) cout << x << " equals 3" << endl;

##### Output

There is no output.

#### SELF-CHECK

7-1 Which expressions evaluate to true assuming j and k are initialized like this:

int j = 4; int k = 8;

1. (j+4) == k d. j != k g. j = 0 *careful*
2. 0 == j e. j < k h. j = 165 *careful*
3. j >= k f. 4 == j

7-2 Write the output generated by the following code:

1. string option = "A"; d. int grade = 45; if (option == "A") if (grade >= 70) cout << "addRecord"; cout << "passing" << endl; if (option == "D") { if (grade < 70) cout << "deleteRecord"; cout << "dubious" << endl;

} if (grade < 60) cout << "failing" << endl;

1. string option = "D"; e. int grade = 65; if (option == "A") if (grade >= 70) cout << "addRecord"; cout << "passing" << endl; if (option == "D") if (grade < 70)

cout << "deleteRecord"; cout << "dubious" << endl;

if (grade < 60)

cout << "failing" << endl;

1. string option = "a"; f. int g = 45; if (option == "A") { // Careful!

cout << "addRecord"; cout << "g: " << g << endl;

} if (g = 70)

if (option == "D") { cout << "at cutoff" << endl;

cout << "deleteRecord"; cout << "g: " << g << endl;

} if (g = 1)

cout << "you get one" << endl; cout << "g: " << g << endl;

7.3: The Alternative Action Pattern

### 7.3 THE ALTERNATIVE ACTION PATTERN

Programs must often select from a variety of actions. For example, one student passes with a fi nal grade of >= 60.0 and the next student fails with a fi nal grade of < 60.0. Th is is an example of the Alternative Action algorithmic pattern. Th e program must choose one course of action or an alternative.

**Algorithmic Pattern:** *Th e Alternative Action pattern*

|  |  |
| --- | --- |
| **Pattern:** | Alternative Action |
| **Problem:** | Need to choose one action from two alternatives |
| **Outline:** | if ( *true-or-false-condition is true* ) *execute action-1* else  *execute action-2* |
| **Code**  **Example:** | if ( nalGrade >= 60.0) cout << "passing" << endl; else  cout << "failing" << endl; |

#### 7.3.1 THE if...else STATEMENT

Th e Alternative Action pattern is implemented in C++ with the if...else statement. Th is control structure can be used to choose between two diff erent courses of action (and as shown later, to choose between more than two alternatives).

##### General Form 7.2 if...else *statement*

if (*boolean-expression*)

*true-part*; else

*false-part*;

Th e if...else statement is an if statement followed by the alternate path after an else. Th e *true-part* and the *false-part* may be any valid C++ statement, including a block.

if (sales <= 20000.00)

cout << "No bonus this month" << endl; else cout << "Bonus coming" << endl;

When an if...else statement is encountered, the *logical expression* evaluates to either false or true. When true, the true part executes—the false part does not. When the logical expression is false, only the false part executes.

Th e next example illustrates how if...else works. When x has a value less than or equal to zero, the output is FALSE. When x is positive, the true part executes and TRUE is output.

double x; cout << "Enter x: "; cin >> x; if (x > 0.0)

cout << "TRUE" << endl; else cout << "FALSE" << endl;

**Flowchart view of the Alternative Action pattern**

*logical-expression*

**true (nonzero)**

**false (zero)**

*Any C++ statement*

Here is another example of if...else demonstrating alternative action that depends on the logical expression (miles > 900000). Sometimes the true part executes—when miles is greater than 90000. Otherwise, the false part executes—when miles is not greater than 90000.

int miles; cout << "Enter miles: "; cin >> miles; if (miles > 90000) {

cout << "Tune-up " << (miles-90000) << " miles overdue" << endl;

} else {

cout << "Tune-up due in " << (90000-miles) << " miles" << endl; }

When miles is input as 96230, the output is Tune-up 6230 miles overdue, but when miles is input as 89200, the false part executes; the output is Tune-up due in 800 miles.

#### SELF-CHECK

7-3 What output occurs when miles is 90000?

Th e ability to choose is a powerful feature of any programming language. Th e if...else statement provides the means to make a program general enough to generate useful information

7.3: The Alternative Action Pattern

appropriate to a variety of data. For example, an employee’s gross pay may be calculated as hours times the rate when hours is less than or equal to 40. However, certain employers must pay timeand-a-half to employees who work more than 40 hours per week. Gross pay with overtime can be computed as follows:

pay = (40 \* rate) + (hours - 40) \* 1.5 \* rate;

With alternative actions, a program can correctly compute gross pay for a variety of values including those less than 40, equal to 40, and more than 40. Th is instance of the Alternative Action pattern is now placed in the context of a complete program.

// Illustrate the exibility offered by Alternative Action

#include <iostream> using namespace std;

int main() { double pay = 0.0; double rate = 0.0; double hours = 0.0;

cout << "Enter hours worked and rate of pay: "; cin >> hours >> rate;

if (hours <= 40.0)

pay = hours \* rate; // True part else

pay = (40 \* rate) + (hours - 40) \* 1.5 \* rate; // False part cout << "pay: " << pay << endl;

return 0; }

**Dialogue 1:**

Enter hours worked and rate of pay: ***38.0 10.0*** pay: 380

**Dialogue 2:**

Enter hours worked and rate of pay: ***42.0 10.0*** pay: 430

It should be noted that semicolon (;) placement in if...else statements is somewhat confusing at fi rst. If you observe a compile time error near an if...else statement, look closely at the placement or lack of semicolons. Also be careful that you don’t place a semicolon immediately after the logical expression. Th is is a common mistake. In this case, the true part is an empty statement where nothing happens and what follows ; is not part of the if statement.

#### SELF-CHECK

7-4 Given the following code:

if (hours >= 40.0) hours = 40 + 1.5 \* (hours - 40);

Determine the fi nal value of hours when hours starts as:

1. 38 c. 42
2. 40 d. 43.5

7-5 Write the output generated by each of the following programs given these initializations of n and x:

int n = 8; double x = -1.5;

1. if (x < -1.0) c. if (x >= n)

cout << "true" << endl; cout << "x is high";

else else

cout << "false" << endl; cout << "x is low"; cout << "after if...else";

1. if (n >= 0) { d. // true part is another if...else cout << "zero or pos"; if (x <= 0.0) {

} if (x < 0.0) else { cout << "neg"; cout << "neg"; else

} cout << "zero";

} else cout << "pos";

7-6 Write an if...else statement that displays your name if option has the value 1, and displays your school if option has the value of anything else.

### 7.4 BLOCKS WITH SELECTION STRUCTURES

Th e special symbols { and } have been used to gather a set of statements that are treated as one inside the body of a function. Th ese two special symbols delimit (mark the boundaries of) a block. Th e block groups together many actions, which can then be treated as one. Th e block is also useful for combining more than one action as the true or false part of an if...else statement.

// This program uses blocks for both the true and false parts. The // block makes it possible to treat many statements as one.

#include <iostream> using namespace std;

7.4: Blocks with Selection Structures

int main() { double GPA = 0.0;

double margin = 0.0; // How far from dean's list cut-off

cout << "Enter GPA: "; cin >> GPA; if (GPA >= 3.5) {

// True part contains more than one statement in this block cout << "Congratulations, you are on the dean's list." << endl; margin = GPA - 3.5;

cout << "You made it by " << margin << " points." << endl;

} else {

// False part contains more than one statement in this block cout << "Sorry, you are not on the dean's list." << endl; margin = 3.5 - GPA;

cout << "You missed it by " << margin << " points." << endl;

} return 0; }

Th e block makes it possible to treat several statements as one. When GPA is input as 3.7, GPA >= 3.5 becomes true and the following dialog is generated:

Enter GPA: ***3.7***

Congratulations, you are on the dean's list.

You made it by 0.2 points.

When GPA is 2.9, GPA >= 3.5 becomes false and this output occurs:

Enter GPA: ***2.9***

Sorry, you are not on the dean's list.

You missed it by 0.6 points.

Th is alternative execution is provided by the two possible evaluations of the logical expression GPA >= 3.5. If true, the true part executes; if false, the false part executes.

#### 7.4.1 THE TROUBLE IN FORGETTING AND

Neglecting to use the block can cause a variety of errors. Modifying the previous example illustrates what can go wrong if the block is not used when attempting to execute both cout statements.

if (GPA >= 3.5) margin = GPA - 3.5;

cout << "Congratulations, you are on the dean's list." << endl; cout << "You made it by " << margin << " points." << endl; else // <- ERROR: Unexpected else

With { and } removed there is no block; the two highlighted statements no longer belong to the preceding if...else—even though the indentation might make it appear as such. Th is previous code represents an if statement, followed by two cout statements, followed by the reserved word else. When else is encountered, the C++ compiler complains because there is no statement that begins with an else.

Here is another example of what can go wrong when a block is omitted. Th is time, { and } are omitted after else.

else

margin = 3.5 - GPA;

cout << "Sorry, you are not on the dean's list." << endl; cout << "You missed it by " << margin << " points." << endl;

Th ere are no compile time errors here, but the code does contain an intent error. Th e fi nal two statements always execute! Th ey do not belong to if...else. Whenever GPA >= 3.5 is false, the code does execute as one would expect, but when this logical expression is true, the output is not what is intended. Instead, this rather confusing output shows up:

Congratulations, you are on the dean's list.

You made it by 0.152 points.

Sorry, you are not on the dean's list.

You missed it by -0.152 points.

Although not necessary, it could help if you always use blocks as the true and false part of if and if...else statements. Th e practice can make for more readable code and at the same time prevent intent errors such as the one above. One of the drawbacks is that there are more lines of code and more sets of curly braces to line up. Also, as you’ll see in the second part of this chapter with the Multiple Selection pattern, the action is most often only one statement. Th e block is not required.

### 7.5 bool OBJECTS

C++ has bool objects to store either one of these constants: true or false. Named after the mathematician George Boole, bool objects simplify logical expressions as demonstrated in the following program:

// Demonstrates bool initialization and assignment. A standard C++ // compiler has bool, true, and false built in.

#include <iostream> using namespace std;

int main() {

// Initialize three bool objects to false bool ready, willing, able; double credits = 28.5; double hours = 9.5;

// Assign true or false to all three bool objects ready = hours >= 8.0; willing = credits > 20.0; able = credits <= 32.0;

// If all three bools are true, the logical expression is true if (ready && willing && able) cout << "YES" << endl; else

cout << "NO" << endl;

return 0; }

#### Output

YES

Like other objects, bool objects can be declared, initialized, and assigned a value. Th e assigned expression should be a logical expression—one that evaluates to true or false. Two new constants are also added: true and false. Th is is shown in the initializations of the three bool objects in the previous program.

Th e bool class is often used as the return type in both nonmember and class member functions. For example, the LibraryBook class has a member function that returns true when a book is available or false if it was checked out.

bool LibraryBook::isAvailable() // post: Return true if this book is available, or false if not

Here is an example free function that returns true if the integer argument is odd:

// Demonstrate a simple bool function

#include <iostream> using namespace std;

bool isOdd(int n) {

// post: Return true if n is an odd integer return (n % 2) != 0; }

int main() { int j = 3;

// Ensure j is an even number if (isOdd(j)) { j = j + 1;

}

cout << j << endl;

return 0;

}

#### Output

4

#### 7.5.1 BOOLEAN OPERATORS

C++ has three Boolean operators, ! (not), ¦¦ (or), and && (and), to create more complex logical expressions. For example, this logical expression:

(test >= 0) && (test <= 100) shows the logical “and” operator (&&) applied to two logical operands. Since there are only two logical values, true and false, the following table shows every possible combination of logical values and the logical operators !, ¦¦, and &&:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **! (not)** | | **¦¦ (or)** | | **&& (and)** | |
| **Expression** | **Result** | **Expression** | **Result** | **Expression** | **Result** |
| ! false  ! true | true false | true ¦¦ true true ¦¦ false false ¦¦ true false ¦¦ false | true true true false | true && true true && false false && true false && false | true false false false |

Th e next example logical expression uses the Boolean operator && (logical “and”) to ensure a test is in the range of 0 through 100 inclusive. Th e logical expression is true when test has a value greater than or equal to 0 (test >= 0) and at the same time is less than or equal to 100

(test <= 100).

if ((test >= 0) && (test <= 100)) cout << "Test is in range"; else cout << "\*\*Warning--Test is out of range";

Here is how the if statement evaluates its logical expressions when test has the value 97 and then 977 (to simulate an attempt to enter 97 when the user accidentally presses 7 twice):

**When test is 97**

**When test is 977**

(test >= 0) && (test <= 100) (test >= 0) && (test <= 100) ( 97 >= 0) && ( 97 <= 100) (977 >= 0) && (977 <= 100)

true && true true && false true false

#### 7.5.2 OPERATOR PRECEDENCE RULES

Programming languages have precedence rules governing the order in which operators are applied to the operand(s). For example, in the absence of parentheses, the relational operators >= and <= are evaluated before the && operator. Most operators are grouped (evaluated) in a left-to-right order: a / b / c / d is equivalent to (((a / b) / c) / d).

However, there is one notable exception. Th e assignment operator groups in a rightto-left order to allow multiple assignments such as this: x = y = z = 0.0 is equivalent to

(x = (y = (z = 0.0))). Th e expression z = 0.0 returns 0.0, which is then transferred to y, which is transferred to x.

Th e following table lists some (though not all) of the C++ operators in order of precedence. Th e :: and () operators are evaluated fi rst (have the highest precedence), and the assignment operator = is evaluated last. Although there are more operators in C++, this table represents all the operators used in this textbook, and they have all been discussed already.

##### Precedence rules of C++ operators (partial list)

|  |  |  |  |
| --- | --- | --- | --- |
| **Category** | **Operators** | **Descriptions** | **Grouping** |
| Highest | :: , () | Scope resolution, Function call | Left to right |
| Unary | ! , + , - | Not, Unary plus, Unary minus | Right to left |
| Multiplicative | \* , / , % | Multiplication, Division, Remainder | Left to right |
| Additive | + , - | Binary plus, Binary minus | Left to right |
| Input/Output | >> , << | Stream extraction, Stream insertion | Left to right |
| Relational | < , >  <= , >= | Less than, Greater than  Less or equal, Greater or equal | Left to right |
| Equality | == , != | Equal, Not equal | Left to right |
| and | && | Logical and | Left to right |
| or | ¦¦ | Logical or | Left to right |
| Assignment | = | Assign right value to left value | Right to left |

One of the problems with these elaborate precedence rules is simply trying to remember them. When unsure, use parentheses to clarify these precedence rules. Using parentheses makes the code more readable and therefore more understandable.

#### SELF-CHECK

7-7 Evaluate the following expressions to true or false:

1. (false ¦¦ true)
2. (true && false)
3. (1 \* 3 == 4 - 1)
4. (false ¦¦ (true && false))
5. (3 < 4 && 3 != 4)
6. (! false && ! true)
7. ((5 + 2) > 3 && (11 < 12))
8. ! ((false && true) ¦¦ false)
   1. Write an expression that is true only when the int object named score is in the range of 1 through 10 inclusive.
   2. Write an expression that is true if test is outside the range of 0 through 100 inclusive.
   3. Write the output generated by the following code (be careful):

double GPA = 1.03; if (GPA = 4.0) cout << "President's list";

##### 7.5.3 THE BOOLEAN “OR” ¦¦ WITH A grid OBJECT

Th e next sample logical expression uses the operator ¦¦ (logical “or”) to determine if the mover in a grid object is on one of the four edges. Th e logical expression is true when the mover is in row number 0, column number 0, the last row, or the last column.

(g.row() == 0)

¦¦ (g.row() == g.nRows()-1)

¦¦ (g.column() == 0)

¦¦ (g.column() == g.nColumns()-1)

Th is logical expression evaluates like this when the mover is in row 1, column 5 of a 6-by-6 grid object (the ¦¦ operator evaluates in a left-to-right order):

The grid:

. . . . . .

. . . . . > . . . . . .

. . . . . .

. . . . . . . . . . . .

g.row()==0 ¦¦ g.row()==g.nRows()-1 ¦¦ g.column()==0 ¦¦ g.column()==g.nColumns()-1

1==0 ¦¦ 1==5 ¦¦ 5==0 ¦¦ 5==5 false ¦¦ false ¦¦ 5==0 ¦¦ 5==5 false ¦¦ false ¦¦ 5==5 false ¦¦ true true

Th e only time this expression is false is when all four subexpressions are false. If any one of them is true, the expression evaluates to true, in fact, more quickly than you might think (see short circuit Boolean evaluation below). Now here is the same expression put into the context of a function that determines if the mover is on the edge of *any* grid object:

// Show a more complex logical expression inside a bool function

#include <iostream> // For cout using namespace std; #include "Grid.h" // For class Grid bool moverOnEdge(const Grid & g) {

// post: Return true if the mover is on an edge or false if not bool result;

result = (g.row() == 0) // On north edge? ¦¦ (g.row() == g.nRows()-1) // On south edge?

¦¦ (g.column() == 0) // On west edge? ¦¦ (g.column() == g.nColumns()-1); // On east edge?

return result; }

int main() {

// Test drive moverOnEdge

Grid tarpit(6, 6, 2, 5, east);

if (moverOnEdge(tarpit)) { cout << "On edge" << endl;

} else {

cout << "Not on edge" << endl; }

return 0; }

###### Output

On edge

#### SELF-CHECK

7-11 Many tests are necessary for the moverOnEdge function. Write the output from the preceding program when the mover is at each of the following intersections:

|  |  |  |
| --- | --- | --- |
| **Row** | **Column** | **Output—on edge or not?** |
| 3 | 4 |  |
| 4 | 3 |  |
| 2 | 2 |  |
| 0 | 2 |  |
| 2 | 0 |  |

##### 7.5.4 SHORT CIRCUIT BOOLEAN EVALUATION

In the logical expression (E1 && E2), E1 is evaluated fi rst and if it is false, E2 is not evaluated. Th is is called *short circuit evaluation*. It is satisfactory because false && false is false. So is false && true. Evaluating the second expression E2 is not necessary. Th is is the way C++ evaluates logical expressions—stopping as soon as possible. Short circuit evaluation is also possible with the “or” operator ¦¦. In the expression (E1 ¦¦ E2), E1 is evaluated fi rst and if E1 is true, E2 is not evaluated. A programmer can actually get away with code like this:

if ((x >= 0.0) && (sqrt(x) <= 4.0))

When x is negative, the second expression with sqrt(x) is never evaluated. By checking for x >= 0.0 fi rst, the square root of a negative number never occurs. Switch the order of statements and a runtime error occurs when x < 0.0.

Also consider the previous example of Boolean evaluation in the bool function moverOnEdge.

When the mover was in row 2 and column 5, the fi rst three Boolean subexpressions were false. Th erefore, the evaluation had to carry on until the fourth (and fi nal) subexpression. Th is entire expression would also be evaluated whenever the mover was not on an edge—all four subexpressions would be false. However, consider the same expression if the mover had been in row 0:

The grid:

. . . . . > . . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

|  |  |  |
| --- | --- | --- |
| g.row()==0 |  | true |
| ¦¦ g.row()==g.nRows()-1 |  | not evaluated |
| ¦¦ g.column()==0 |  | not evaluated |

¦¦ g.column()==g.nColumns()-1 not evaluated

As soon as g.row()==0 evaluates to true, the following three subexpressions do not need to be evaluated. true ¦¦ anything is true. Short circuit Boolean evaluation is part of C++ because it improves runtime effi ciency. Imagine evaluating one or two fewer subexpressions millions of times.

#### SELF-CHECK

7-12 Evaluate the following expressions after each set of assignments to x and y:

((fabs(x - y) >= 0.001) && (x >= 0.0) && (sqrt(x) < 6.5))

|  |  |
| --- | --- |
| a. x = 1.0 | c. x = -1.0 |
| y = 2.0 | y = 2.0 |
| b. x = 56.77779 | d. x = -1.0 |
| y = 56.77777 | y = 1.0 |

7-13 How many subexpressions evaluate when the mover is in row 5, column 3, assuming a 6-by-6 grid?

g.row()==0 ¦¦ g.row()==g.nRows()-1

¦¦ g.column()==0

¦¦ g.column()==g.nColumns()-1

7.6: A bool Member Function

### 7.6 A bool MEMBER FUNCTION

Th e class defi nition of BankAccount has a void withdraw member function with the precondition that the withdrawal amount must not be greater than the balance. Th is implementation currently allows the balance to go negative when the preconditions are not met by the client code (the following member function is from BankAccount.cpp).

void BankAccount::withdraw(double withdrawalAmount) {

// pre: withdrawalAmount <= balance balance = balance - withdrawalAmount; }

A better design would be to disallow negative balances. Th en the client code would not have to worry about satisfying the precondition. Th e withdraw message could avoid negative balances. Th e return type could also become bool so the client code has the chance to determine if the withdraw message was successful or not. First, the class defi nition would have to be changed. void is changed to bool in the fi le BankAccount.h:

bool withdraw(double withdrawalAmount); // post: If withdrawalAmount <= balance && withdrawalAmount > 0.0, // debit withdrawalAmount from this balance and return true.

// Otherwise don't change anything--just return false.

Th en the implementation is changed in BankAccount. Th e Alternative Action pattern chooses between debiting the account and returning true or returning false when the balance is not large enough.

bool BankAccount::withdraw(double withdrawalAmount) { bool result = true;

if ((withdrawalAmount > balance) ¦¦ (withdrawalAmount <= 0.00)) result = false; else

balance = balance - withdrawalAmount;

return result; }

Th e following program test drives this new behavior. Because withdraw returns either true or false, the message can be used as a test expression.

// Test drive the "safe" BankAccount::withdraw

#include <iostream> using namespace std;

#include "BankAccount.h" // A modi ed "safe" BankAccount

int main() {

BankAccount aSafeAccount("Charlie", 50.00); double withdrawalAmount; cout << "Enter amount to withdraw: ";

cin >> withdrawalAmount;

if (aSafeAccount.withdraw(withdrawalAmount)) {

cout << "Balance = $" << aSafeAccount.getBalance() << endl;

} else {

cout << "Could not withdraw " << withdrawalAmount << endl; }

// Can ignore return result aSafeAccount.withdraw(10000); return 0; }

#### Dialogue 1

Enter amount to withdraw: ***75.00***

Could not withdraw 75.00

#### Dialogue 2

Enter amount to withdraw: ***20.00***

Balance = $30

In C++, any function return result can be ignored. Th is new version of withdraw could be used as it was before—as a stand-alone statement rather than as part of an if statement. Th is is done at the attempt to withdraw $1,000.00 as the last statement in main, just before return 0.

#### SELF-CHECK

7-14 Using this new safe version of the BankAccount::withdraw function, write the output generated by the program given below for each value of wAmount:

1. double wAmount = 100.00; c. double wAmount = 112.50;
2. double wAmount = -100.00; d. double wAmount = 200.00;

#include <iostream> // For cout using namespace std; #include "BankAccount.h" // For the BankAccount class

int main() {

BankAccount b("Kilroy", 112.50);

double wAmount = -100.00; // Substitute new values here if (b.withdraw(wAmount)) { cout << "okay" << endl;

}

7.7: Multiple Selection

else {

cout << "failed" << endl;

} return 0; }

### 7.7 MULTIPLE SELECTION

*Multiple Selection* refers to the times when the programmer needs to select one action from many possible actions. Th is is something that occurs quite often when programming. Th e pattern that solves this problem can be implemented as an if...else statement that has other if...elses nested inside their false parts. Th e more actions there are to choose from, the more nesting occurs. Th is pattern is summarized as follows:

**Algorithmic Pattern:** *Th e Multiple Selection pattern*

|  |  |
| --- | --- |
| **Pattern:** | Multiple Selection |
| **Problem:** | Must execute one set of actions from three or more alternatives |
| **Outline:** | if ( *condition 1 is true* ) *execute action 1*  else if ( *condition 2 is true* )  *execute action 2*  // ... else if ( *condition n-1 is true* )  *execute action n-1* else  *execute action n* |
| **Code**  **Example:** | if (grade < 60.0) result = "F"; else if (grade < 70) result = "D"; else if (grade < 80) result = "C"; else if (grade < 90) result = "B"; else result = "A"; |

Th e following program contains an instance of the Multiple Selection pattern to select from one of the three possible actions:

// Multiple selection where exactly one cout statement executes.

// The output is dependent on the input value for GPA.

#include <iostream> using namespace std;

int main() { double GPA;

cout << "Enter your GPA: "; cin >> GPA;

if (GPA < 3.5) {

cout << "Try harder" << endl;

}

else {

// Execute this multiple selection statement if (GPA < 4.0)

cout << "You made the dean's list" << endl; else

cout << "You made the president's list" << endl;

} return 0; }

Notice that the false part of the fi rst if...else statement is another if...else statement. If GPA is less than 3.5, Try harder is output and the program skips over the nested if...else. However, if the logical expression is false (when GPA is greater than or equal to 3.5), the second if...else statement determines if GPA is high enough to qualify for either the dean’s list or the president’s list. Here one alternative selection is nested inside another alternative selection. When implementing the Multiple Selection pattern, it is important to use proper indentation so the code will execute as its written appearance suggests. Th e readability realized by good indentation habits can save you time during program implementation, which includes testing. To illustrate the fl exibility in formatting, the previous multiple selection may be rewritten in the following preferred manner to line up the three paths through this control structure:

if (GPA < 3.5)

cout << "Try harder" << endl; else if (GPA < 4.0)

cout << "You made the dean's list" << endl; else cout << "You made the president's list" << endl;

Th e previous formatting represents the preferred method of this textbook. However, you could also use blocks to make multiple selection look like this:

if (GPA < 3.5) {

cout << "Try harder" << endl;

}

7.7: Multiple Selection

else if (GPA < 4.0) {

cout << "You made the dean's list" << endl;

} else {

cout << "You made the president's list" << endl; }

#### 7.7.1 ANOTHER EXAMPLE: DETERMINING LETTER GRADES

Some instructors use a scale like the following to determine the proper letter grade to assign to a student. Th e letter grade is based on a percentage representing a weighted average of all work for the term.

##### Value of Percentage (should be in the range

**Assigned Grade**

A

B

C

D

F

A function could be implemented with if statements that begin like this:

if ( percentage >= 90.0) result = "A";

if ( percentage >= 80.0 && percentage < 90 ) // Not necessary result = "B";

if ( percentage >= 70.0 && percentage < 80 ) // Not necessary // . . .

However, when given the problem of choosing from among fi ve diff erent actions, try to remember that the choice is Multiple Selection, not Guarded Action. Th e preferred Multiple Selection is also more effi cient at runtime. Th e Multiple Selection pattern is also less prone to intent errors.

string letterGrade(double percentage) {

// pre: percentage >= 0.0 && percentage <= 100.0

// post: Return letter grade according to external documentation string result; // Determine the proper result . . .

if (percentage >= 90.0) result = "A"; else if (percentage >= 80.0) result = "B"; else if (percentage >= 70.0) result = "C";

else if (percentage >= 60.0) result = "D"; else result = "F";

return result; }

Here, the output depends on the value of percentage. If percentage is greater than or equal to 90.0, then the statement result = "A"; executes. Th e program skips over all other statements after the fi rst else. If percentage == 50.0, then all logical expressions are false and the program executes the action after the fi nal else: result = "F";.

When percentage has a value between 60.0 and 90.0, logical expressions evaluate until the fi rst one that is true. When percentage >= 90.0 is false, the opposite logical expression, percentage < 90.0, must be true. Th e second logical expression, percentage >= 80.0, evaluates when the fi rst expression is false. When the fi rst true logical expression is fi nally encountered, the very next true part executes and the program skips over the remaining alternative(s). Th is function could be improved by ensuring that letter grades are returned only when percentage is within the range of 0.0 through 100.0 inclusive. Th ere is a possibility, for example, that an argument will be passed as 777 instead of an intended input of 77. Since 777 >= 90.0 is true, the function improperly returns "A" when "C" would have been the correct result. letterGrade could be modifi ed to contain a test for out-of-range input. Th is fi rst logical expression now checks to see if percentage is either less than 0.0 or greater than 100.0.

if ((percentage < 0.0) ¦¦ (percentage > 100.0)) result = "\*\*Error--Percentage is not in range [0...100]"; else if (percentage >= 90) result = "A";

If percentage is out of range, the result becomes an error message and the program skips over the remainder of the nested if...else structure. Rather than returning an incorrect letter grade for percentages less than 0 or greater than 100, this string is returned instead for the argument 777:

\*\*Error--Percentage is not in range [0...100]

#### 7.7.2 MULTIPLE RETURNS

Th e previous implementation of letterGrade shows that the proper letter grade is fi rst assigned to the local object named result. Another implementation option uses multiple return statements. Th e fi rst time any return statement executes, the function terminates. Th erefore, a function could be written with many return statements:

string letterGrade(double percentage) { if (percentage >= 90) return "A";

7.8: Testing Multiple Selection

if (percentage >= 80) return "B"; if (percentage >= 70) return "C"; if (percentage >= 60) return "D"; if (percentage >= 0)

return "F"; // ERROR: runtime error when percentage < 0 }

If you do use the technique of multiple returns, ensure that something is always returned. For example, the previous code will not return anything for arguments that are less than 0.0. Th is code might cause a warning or a compile time error. Worse, some systems will wait until it is too late and generate a runtime error. Th e problem goes away when the block ends like this instead:

// . . .

if (percentage >= 0) return "F";

return "Error: argument to letterGrade < 0"; }

And while you’re at it, consider returning an error message when percentage > 100 if that would in fact be an error.

### 7.8 TESTING MULTIPLE SELECTION

Consider how many function calls should be made to test the letterGrade function with Multiple Selection—or for that matter, any function or segment of code containing Multiple Selection. To test this particular example to ensure that Multiple Selection is correct for all possible percentage arguments, the function could be called with all numbers in the range from -1.0 through 101.0. However, this would require a virtually infi nite number of function calls. Th is is unnecessary!

First consider a set of test data that executes every possible branch through the nested if...else. Branch coverage testing occurs by observing what happens when each and every statement (the true or false part) of a nested if...else executes once. Th ese three things are necessary to correctly perform branch coverage testing:

* Establish a set of data that executes all branches of the Multiple Selection.
* Execute the portion of the program containing the Multiple Selection for all selected data values. Th is can be done with a test driver.
* Observe that the program segment behaves correctly for all data values.
* For example, the following data set executes all branches of letterGrade:

-1.0 55.0 65.0 75.0 85.0 95.0 101.0

A test driver could start like this:

int main() {

cout << "-1.0 = " << letterGrade(-1.0) << endl; cout << "55.0 = " << letterGrade(55.0) << endl; cout << "65.0 = " << letterGrade(65.0) << endl; // . . .

and then the program output must be examined to indicate that every function call returned the proper value:

-1.0 = \*\*Error--Percentage is not in range [0...100]

55.0 = F

65.0 = D

. . .

#### 7.8.1 BOUNDARY TESTING

Boundary testing occurs by observing what happens for each cut-off (boundary) value. Th is extra eff ort could go a long way. For example, boundary testing avoids situations where students with 90 are accidentally shown to have a letter grade of B rather than A. Th is would occur when the logical expression (percentage >= 90) is accidentally coded as (percentage > 90). Th e arguments of 60, 70, 80, and 90 complete boundary testing of the code above.

Perhaps the best testing strategy is to select test values that combine branch and boundary testing at the same time. For example, a percentage of 90.0 should return A. Th e value of 90 not only checks the path for all; it also tests the boundary—90.0 is the cut-off . Counting down by tens to 60 checks all boundaries. But it misses one path: the one that sets result to F. Adding 59.9 completes the test driver.

int main() {

// A test driver for string letterGrade(double percentage) cout << "90.0? " << letterGrade(90.0) << endl; // 90.0? A cout << "80.0? " << letterGrade(80.0) << endl; // 80.0? B cout << "70.0? " << letterGrade(70.0) << endl; // 70.0? C cout << "60.0? " << letterGrade(60.0) << endl; // 60.0? D cout << "59.9? " << letterGrade(59.9) << endl; // 59.9? F return 0; }

### 7.9 THE ASSERT FUNCTION

So far testing has been done by printing things with cout. Th is requires a careful inspection of the cout statements and the associated output. Placing the expected output on the same line, as is done above with letterGrade, can help. Th e expected and actual values are right next to each other. If they do not match, then either the expected value is incorrect, the return value is incorrect, or perhaps both. Th e C++ assert function can also be used to test where the expected value is next to the function call or message.

7.9: The Assert Function

Th e assert function takes a bool argument. If the argument is false, C++ will inform you with a line of output that begins with Assertion failed. In this case, assert will terminate the program.

If all expressions in all calls to the function are true, there is *no* output. So if you write your tests with assert functions, you only have to look at output when something has gone wrong. Th e following main function represents a test of letterGrade that is equivalent to the version above with fi ve cout statements. Getting no output means the assertions all passed—there were no detected errors when comparing the expected values like "A" with the actual return values like letterGrade(90.0).

/\*

* Test letterGrade using assert

\*

* File name: main.cpp

\*/

#include <cassert> // Required for the assert function

#include <string> using namespace std;

string letterGrade(double percentage) {

// pre: percentage >= 0.0 && percentage <= 100.0

// post: Return letter grade according to external documentation string result; if (percentage >= 90.0) result = "A"; else if (percentage >= 80.0) result = "B"; else if (percentage >= 70.0) result = "C"; else if (percentage >= 60.0) result = "D"; else result = "F"; return result; }

int main() {

assert("A" == letterGrade(90.0)); assert("B" == letterGrade(80.0)); assert("C" == letterGrade(70.0)); assert("D" == letterGrade(60.0)); assert("F" == letterGrade(59.9)); }

Th e preceding code will have no output. Changing the "C" to a "D" will make the expression false in this call to the assert function:

assert("D" == letterGrade(70.0));

Now we get this output from assert indicating the assertion failed. You even get the line number and the fi le name where the assertion failed:

#### Output

Assertion failed: ("D" == letterGrade(70.0)), function main, le main.cpp, line 32.

Th is method of testing is suggested for the programming projects at the end of this chapter. Th ere are several functions that have many branches. Th e assert function makes testing easier.

#### SELF-CHECK

7-15 Which value of percentage would detect the intent error in the following code?

if (percentage >= 90) result = "A"; else if (percentage >= 80) result = "B"; else if (percentage > 70) result = "C"; else if (percentage >= 60) result = "D"; else result = "F";

7-16 What string is incorrectly assigned to letterGrade for the value of percentage you answered above?

7-17 Would you be happy with the result if your grade were computed with this argument?

7-18 Using the nested structure below, write the return value for each of these six diff erent arguments for weather: -40 20 -1 42 15 31

string weather(int temp) { if (temp <= -40) return "extremely frigid"; else if (temp < 0) return "below freezing"; else if (temp < 20) return "freezing to mild"; else if (temp < 30) return "warm"; else if (temp < 40) return "very hot"; else

return "toast"; }

7-19 List the range of integers that would cause the previous program to display warm.

7.10: The switch Statement

7-20 List the range of integers that would cause the previous program to display below freezing.

7-21 Write a test for weather that uses the assert function to completely test this free function.

### 7.10 THE switch STATEMENT

Th e C++ switch statement also implements the Multiple Selection pattern. Although nested if...else statements can do anything the switch statement does, it is included here because you will see it in other C++ programs and because some programmers prefer this implementation of Multiple Selection.

#### General Form 7.3 *Th e C++* switch *statement*

switch (*switch-expression*) { case *constant-value-1*: *statement(s)-1*; break ; case *constant-value-2*: *statement(s)-2*; break ; ... case *constant-value-n*: *statement(s)-n*; break ; default :

*default-statement(s)*;

}

When a switch statement is encountered, the switch-expression is compared to *constant-value-1*, *constant-value-2*, through *constant-value-n* until a match is found. When the switch expression matches one of these values, the statements following the colon execute. If no match is made, the statement(s) after default execute.

Th e keyword default needs to be present only if some processing is desired whenever the switch expression cannot match any of the case values. With no default, it is possible that no statements will execute inside the switch statement. Sometimes that is the appropriate design. Th e following switch statement chooses one of three paths based on the input value of option. If the user enters 1, the fi rst case section of code is executed. Th e fi rst break terminates the switch statement.

int option = 0;

cout << "Enter option 1, 2, or 3: "; cin >> option;

switch(option) { case 1:

cout << "option 1 selected" << endl; break; case 2: cout << "option 2 selected" << endl; break; case 3: cout << "option 3 selected" << endl; break; default:

cout << "option < 1 or option > 3" << endl; } // End switch

If neither 1, 2, nor 3 are entered, the statement(s) after default: execute.

Th e switch expression (option above) and each constant value (1, 2, and 3 above) after case must be compatible. In fact, the constants must be one of the C++ integral types, which consist of the integer types (int, long, and so on) or char—the class discussed in the next section. Th e break statement—a new reserved word—causes an exit from the control structure the program is executing. Th e break statement at the end of each case section causes a jump out of the switch statement. In fact, the switch statement typically requires many break statements. Th ey avoid unintentional execution of the remaining portions of the switch statement.

#### 7.10.1 char OBJECTS

Th e char class of objects is an integral type often used as the constant value in switch statements.

A char object stores one character constant—a character between single quotes (apostrophes):

'A' 'b' '?' '8' ' ' ','

Th ere are several special escape sequences—a backward slash (\) followed by one of a select few characters that have special meaning (see the following table).

|  |  |
| --- | --- |
| **Escape Sequence** | **Meaning** |
| '\n' | new line |
| '\"' | double quote in a char |
| '\'' | single quote in a char |
| '\\' | one backward slash |
| '\t' | tab |

char objects are declared, initialized, assigned values, and displayed in the same way as the other fundamental types like int.

// Use some char objects

#include <iostream> using namespace std;

7.10: The switch Statement

int main() {

// Declare and initialize some char objects char one, two; char letterGrade = 'A'; char newLine = '\n';

// Assignment is possible with character expressions one = 'T'; two = 'o';

// Output some char objects, char constants, and escape sequences cout << "letterGrade is " << letterGrade << endl; cout << one << two << newLine << one << '\t' << two << endl; cout << '\"' << 'A' << ' ' << '\\' << ' ' << 'S' << 't' << 'r' << 'i' << 'n' << 'g' << '?' << '\'' << endl;

return 0; }

##### Output

letterGrade is A

To

T o

"A \ String?'

Th e char type has its own set of nonmember functions included in cctype. For example, the toupper function returns 68, which is the ASCII (numeric) code for the uppercase equivalent of its argument. If you want to see the actual character, typecast with char() to see the ‘D’:

cout << toupper('d') << endl; // Output: 68 cout << char(toupper('d')) << endl; // Output: D

Here is a switch statement that uses characters as the constant expressions. It chooses one of fi ve paths based on the value of the char object option:

// Illustrate another switch statement

#include <iostream> // For cout << using namespace std; #include <cctype> // For toupper(char) returns uppercase char

int main() { char option;

cout << "B)alance W)ithdraw D)eposit Q)uit: "; cin >> option; switch(toupper(option)) { case 'B': cout << "Balance selected" << endl; break; case 'W': cout << "Withdraw selected" << endl; break; case 'D':

cout << "Deposit selected" << endl; break; case 'Q': cout << "Quit selected" << endl; break; default:

cout << "Invalid choice" << endl; } // End switch

return 0; }

##### One Possible Dialogue

B)alance W)ithdraw D)eposit Q)uit: ***D***

Deposit selected

If the value extracted for option is B, the message Balance selected is output and break is executed to exit the switch control structure. If Q or q is input, Quit selected is output and another break is executed. In this example, each case is evaluated until option is matched to one of the four char values following case. If option is any other value, the message Invalid choice is displayed.

One fi nal comment on the switch statement: don’t forget to include the optional break statements in the case portions of switch. Failure to break out of the switch causes all remaining statements to execute. Although this may be what you want in some unusual circumstance, it is usually not a good idea to forget the breaks. For example, imagine the preceding switch with all break statements removed:

switch(toupper(option)) { case 'B':

cout << "Balance selected" << endl; case 'W':

cout << "Withdraw selected" << endl; case 'D':

cout << "Deposit selected" << endl; case 'Q':

cout << "Quit selected" << endl; default:

cout << "Invalid choice" << endl; } // End switch

Now when B is input, every statement executes—including the default!

B)alance W)ithdraw D)eposit Q)uit: ***B***

Balance selected

Withdraw selected

Deposit selected

Quit selected Invalid choice

Chapter Summary

#### SELF-CHECK

7-22 Write the output produced by this switch statement:

char option = 'A';

switch(option) { case 'A': cout << "AAA";

break; case 'B':

cout << "BBB";

break; default:

cout << "Invalid";

}

7-23 What is the output from the code above when option is B? 7-24 What is the output from the code above when option is C?

7-25 What is the output from the code above when option is D?

7-26 Write a switch statement that displays your favorite music if the int object choice is 1, your favorite food if choice is 2, and your favorite instructor if choice is 3. If the option is anything else, display Error. Don’t forget the break statements.

### CHAPTER SUMMARY

* Selection requires logical expressions that evaluate to true or false. Th e logical expressions usually have one or more of the following relational, equality, or logical operators:

< > <= >= != == ! ¦¦ &&

* Th e Guarded Action pattern is implemented with the if statement that either executes a collection of statements or skips them depending on the circumstances.
* Th e Alternative Action pattern, implemented with the C++ if...else statement, is used to choose one action or its alternative—two choices.
* Multiple Selection can be implemented with nested if...else statements or with the switch statement. Multiple Selection should be used whenever there are three or more actions to select from.
* Selection control allows the program to respond to a variety of situations in an appropriate manner.
* Th e bool class and the bool constants true and false are sometimes used as the return type of a function to conveniently return information about the state of an object. Is the book available? Was the withdraw message successful or not? Are there real roots to this equation?
* Several examples of Multiple Selection showed the need for thorough testing.
* When implementing the Multiple Selection pattern, be sure to thoroughly test the code with the Multiple Selection. Establish a set of data that executes all branches and tests all cut-off (boundary) values.
* Without thorough testing, a program may only appear to work when in fact there is perhaps one value among thousands that does not work.

### EXERCISES

1. True or False: When an if statement is encountered, the *true-part* always executes.
2. True or False: When an if or if...else statement is encountered, valid logical expressions are evaluated to either true, false, or maybe.
3. Proper indentation and spacing improve readability. Th e next code segment is an example of poor indentation; try to predict the output.

int j=123;if (j>=0)if (0==j)cout<<"one";else cout<<"two";else cout<<"three";

1. Write the output from the following code fragments:

|  |  |  |  |
| --- | --- | --- | --- |
| a. | double x = 4.0; if (10.0 == x) cout << "is 10"; else  cout << "not 10"; | c. | int j = 0, k = 1; if (j != k) cout << "abc"; if (j == k) cout << "def";  (j <= k) cout << "ghi"; if (j >= k) cout << "klm"; |
| b. | string s1 = "Ab"; string s2 = "Bc"; if (s1 == s2) cout << "equal"; if (s1 != s2) cout << "not"; | d. | double x = -123.4, y = 999.9; if (x < y) cout << "less "; if (x > y) cout << "greater "; if (x == y) cout << "equal "; if (x != y) cout << "not eq. "; |

1. Write the output from the following code fragments:
   1. string name = "Parker"; if (name >= "A" && name <= "F") cout << "A..F"; if (name >= "G" && name <= "N") cout << "G..N"; if (name >= "O" && name <= "T") cout << "O..T"; if (name >= "U" && name <= "Z") cout << "U..Z";
   2. int t1 = 87, t2 = 76, larger = 0;

if (t1 > t2) larger = t1; else larger = t2;

cout << "larger: " << larger;

Exercises

* 1. double x1 = 2.89; double x2 = 3.12;

if (fabs(x1 - x2) < 1)

cout << "true";

else

cout << "false";

1. Write the output generated from the following program fragments, assuming j and k are int objects with the values 25 and 50, respectively.

int j = 25; int k = 50;

|  |  |  |  |
| --- | --- | --- | --- |
| a. | if (j == k) cout << j; cout << k; | c. | if (j > k || k < 100) cout << "THREE"; else  cout << "FOUR"; |
| b. | if (j <= k && j >= 0) cout << "ONE" << else  cout << "TWO"; | d. | if (j >= 0 && j <=100) cout << "FIVE"; else  cout << "SIX"; |

1. Write a statement that displays YES if intObject is positive, NO if intObject is negative, or NEUTRAL if intObject is zero.
2. Write a statement that will add 1 to the int object j only when the int object counter has a value less than the int object n.
3. Write a statement that displays Hello if the int object hours has a value less than 8, or Goodbye if hours has any other value.
4. Write a program fragment that guarantees that the int object amount is even. If amount is odd, increment amount by 1.
5. Write a program segment that adds 1 to the int object amount if amount is less than 10. In this case, also display Less than 10. If amount is greater than 10, subtract 1 from amount and display Greater than 10. If amount is 10, just display Equal to 10.
6. Write an expression that is true if and only if the mover in the Grid object myGrid is on one of the four corners of the Grid.
7. Write function inc3 that increments all three arguments associated with the parameters by 1.0. Th e following function call must change the objects as shown.

double x = 0.0, y = 0.0, z = 0.0; inc3(x, y, z);

// assert x, y, and z all equal 1.0.

1. Implement function bool turnTillClear(Grid & grid) that faces the mover in the fi rst direction that has a clear front (like the mover in the left column below). In this case, return true. Return false when the mover is surrounded as shown with the Grid on the right.

The grid:The grid:

. . . . . . . . . .. . . # # # . . . .

. . . # < . . . . .. . . # < # . . . . . # # # # # # # # .. # # # # # # # # . mover turns left twicemover is trapped

The grid:The grid:

. . . . . . . . . .. . . # # # . . . .

. . . # > . . . . .. . . # ^ # . . . . . # # # # # . . . . . # # # # # # # # .

1. Write the output from the following program when:

|  |  |
| --- | --- |
| a. choice = 3 | c. choice = 2 |
| b. choice = 1 | d. choice = 0 |

#include <iostream> using namespace std; int main() {

int choice = 3; // Change 3 to 1, 2, and then 0 switch(choice) { case 1: cout << "1 selected" << endl; break; case 2: cout << "2 selected" << endl; break; case 3: cout << "3 selected" << endl; break; default:

cout << "Invalid choice" << endl;

} // End switch return 0; }

### PROGRAMMING TIPS

1. Take notice of the diff erence between = and ==. = assigns and == compares. It is very easy, even natural, to write = instead of ==. Th e following code will always execute the true part because grade = 100 returns 100, which is nonzero, which is true.

Programming Tips

if (grade = 100) cout << "another perfect score" << endl; else

cout << "this never ever executes" << endl;

Perhaps this is the most famous C++ “gotcha”: using = instead of == in an if statement.

1. Test drivers help protect against errors. Use test drivers with many calls to the function containing the Multiple Selection. Send arguments that check all boundary values. Send arguments that ensure that each branch executes at least once.
2. Th e compound statement may be used even if it is not required. Consider always using curly braces to mark the beginning and end of the true part and the false part of an if...else. You *must* use the block to treat several statements as one. You *may* use the block for readability and to help avoid bugs.
3. Th e way a mathematician writes an expression does not always work in C++. It is easy, even natural, to write the following code that checks to see if a value is in a certain range:

int x = 2222; if ( 0 <= x <= 100 ) { cout << x << " is in the range of 0 through 100" << endl;

}

If you’re lucky, you will get a warning on a compiler. However, in either case, the code compiles and runs. Th en when x is 2222, you get this output indicating an intent error:

2222 is in the range of 0 through 100 // Wrong

Th at’s because the logical expression evaluates like this:

if ( 0 <= x <= 100 ) 0 <= 222 <= 100

true <= 100 // True is like 1 and 1 <= 100 is true true

1. Short circuit evaluation makes programs more effi cient and comes in handy sometimes. Short circuit Boolean evaluation is always in eff ect to make programs run more quickly, especially when millions of comparisons are made. You might fi nd that fact useful occasionally. One particular algorithm in a later chapter uses short circuit evaluation to avoid runtime errors.
2. Don’t forget to use the break statement to terminate switch statements. C++ executes all code to the bottom of the switch or until the fi rst break is encountered. Th e following code works correctly when option == 'W'. On the other hand, when option == 'B', both options execute as shown in the accompanying output.

switch(option) { case 'B':

cout << "Balance selected" << endl; case 'W':

cout << "Withdraw selected" << endl;

}

**Output** (whenoption == 'B'**)**

Balance selected

Withdraw selected

### PROGRAMMING PROJECTS

#### 7A A HALF-DOZEN SELECTION METHODS

Write one C++ program where your main method is a test driver of your own design that tests six new free functions implemented in the same fi le. You may write your own tests with assert functions.

/\*

* A test driver like this may be used to test the functions

\*

* File name: TestSelectionFunctions (on the book's website)

\*/ int main() {

// Test isEven assert(isEven(-2)); assert(isEven(0)); assert(isEven(2)); assert( ! isEven(-1)); assert( ! isEven(1)); // . . . many more asserts are available

1. bool isEven(int number)

Complete the free function isEven to return true if the integer argument is an even number.

isEven(-2) returns true isEven(0) returns true isEven(2) returns true isEven(-1) returns false isEven(1) returns false

1. int largest(int a, int b, int c)

Complete the free function largest to return the largest of three integers.

largest(2, 4, 6) returns 6 largest(1, 2, 2) returns 2 largest(-5, -2, -7) returns -2

1. string rstOf3Strings(string a, string b, string c)

Complete the free function rstOf3Strings to return a reference to the string that is not “greater than” the other two. Th is is the string that alphabetically precedes, or is equal to, the other two arguments. Use the relational operator to compare strings. Note: "abc" < "abc " and "A" < "a".

rstOf3Strings("c", "b", "a") returns "a"

rstOf3Strings("B", "B", "a") returns "B"

rstOf3Strings("ma", "Ma", "ma") returns "Ma"

rstOf3Strings("x ", "x ", "x ") returns "x "

1. string letterGrade(double numericGrade)

Complete the free function letterGrade that returns the proper letter grade as a string for a plus/minus system with the following scale:

#### Percentage

**Grade**

|  |  |
| --- | --- |
| 93.0 ≤ percentage  90.0 ≤ percentage < 93.0  87.0 ≤ percentage < 90.0 83.0 ≤ percentage < 87.0 80.0 ≤ percentage < 83.0 77.0 ≤ percentage < 80.0 70.0 ≤ percentage < 77.0 60.0 ≤ percentage < 70.0 percentage < 60.0 | A   1. B+   B   1. C+   C  D  F |

After implementing the function, perform branch and boundary testing. If the argument is a value outside the range of 0.0 through 100.0, return "Unkown" as the string letter grade.

5. double salary(double sales)

Complete the free function salary that returns a salesperson’s salary for the month according to the following policy:

#### Sales Over But Not Over Monthly Salary

0 $10,000 Base salary

$10,000 $20,000 Base salary plus 5% of sales over $10,000

$20,000 $30,000 Base salary plus $500.00 plus 8% of sales over $20,000

$30,000 Base salary plus $1300.00 plus 12% of sales over $30,000

Th e base salary is $1,500.00, which means salary returns a value that is never less than 1500.00.

When sales are over $10,000, commission is added to the base salary. For example, when sales equals 10001, the monthly salary is $1,500.00 + 5% of $1.00 for a total of $1,500.05, and when sales is 20001, the monthly salary is $1,500.00 + $500.00 + 8% of $1.00 for a total of $2,000.08.

6. int romanNumeral(char numeral)

Complete the free function romanNumeral that returns the numeric equivalent of an upper- or lowercase Roman numeral, which is actually a char. Roman numerals and their decimal equiva-

lents are ‘I’ (or ‘i’) = 1, ‘V’ (or ‘v’) = 5, ‘X’ (or ‘x’) = 10, ‘L’ (or ‘l’) = 50, ‘C’ (or ‘c’) = 100, ‘D’ (or ‘d’) = 500, and ‘M’ (or ‘m’) = 1,000. If the input is not a valid Roman numeral, return -1.

romanNumeral('i') returns 1 romanNumeral('I') returns 1 romanNumeral('v') returns 5 romanNumeral('X') returns 10 romanNumeral('L') returns 50 romanNumeral('c') returns 100 romanNumeral('D') returns 500 romanNumeral('m') returns 1000

##### 7B A HALF DOZEN CALENDAR FUNCTIONS

Write one C++ program where your main method is a test driver of your own design that tests six new free functions implemented in the same fi le. You may write your own tests using output statements or asserts.

/\*

* A test driver like this may be used to test the functions

\*

* File name: TestCalendarFunctions (on the book's website)

\*/ int main() {

// Test isLeapYear assert(isLeapYear(2016)); assert(isLeapYear(2020)); assert( ! isLeapYear(2019)); assert( ! isLeapYear(2100));

1. bool isLeapYear(int year)

Complete the free function isLeapYear that returns true if the integer argument represents a leap year in which February has 29 days instead of 28 days. Th is is done because there are actually close to 365.25 days in a year. A leap year is a year after 1582 that is evenly divisible (no remainder after division) by four unless it is the end of a century. In this case—where the year is also evenly divisible by 100—year must also be divisible by 400. For example, 2000 and 2400 are leap years but 1900 and 2100 are not. Let leapYear return true if the argument represents a leap year or false if it does not.

isLeapYear(1580) returns false isLeapYear(1584) returns true isLeapYear(2020) returns true isLeapYear(-2020) returns false isLeapYear(2100) returns false

1. string day(int dayOfWeek)

Complete the free function dayOfWeek that returns the string "Monday" if the int argument passed to the parameter dayOfWeek is 1, returns "Tuesday" for the argument 2, and so on up through returning "Sunday" if the argument is 7. Return "Unknown" if the argument is not in the range of 1 through 7.

dayOfWeek(0) returns "Unknown" dayOfWeek(3) returns "Wednesday" dayOfWeek(4) returns "Thursday" dayOfWeek(6) returns "Saturday" dayOfWeek(8) returns "Unknown"

1. int daysInMonth(int month, int year)

Complete the free function daysInMonth that returns the number of days in a month for the given year. Th ere are 30 days in the months September, April, June, and November, or months 9, 4, 6, and 11. February has 28 days unless it is a leap year, when it has 29. All other months—1, 3, 5, 7, 8, 10, and 12 (December)—have 31 days. Assume the year is always >= 1582. You may use your own existing method isLeapYear. Return -1 if month is not in the range of 1 through 12.

daysInMonth(1, 2020) returns 31 daysInMonth(2, 2020) returns 29 daysInMonth(2, 2019) returns 28 daysInMonth(0, 2019) returns -1 daysInMonth(13, 2019) returns -1

1. int thanksDate(int rstDay)

In the U.S., Th anksgiving falls on the fourth Th ursday of each November. Complete method thanksDate that determines the day of the month upon which Th anksgiving falls, no matter which day November begins on. November can begin on any day where 1 represents Monday, through 7, which represents Sunday. A valid call would be thanksDate(2) to indicate the fi rst day of November is Tuesday. thanksDate should then return the day of the month upon which Th anksgiving falls, which is 24 (as shown in the calendar below). Arguments can only be 1 (for Monday) through 7 (for Sunday). If the argument is out of the range of 1 through 7, return -1.

thanksDate(2) returns 24 // 1-Nov is Tue thanksDate(5) returns 28 // 1-Nov is Fri thanksDate(7) returns 26 // 1-Nov is Sun

|  |
| --- |
| **November** |
| **Su Mo Tu We Th Fr Sa** |
| **1 2 3 4 5**  **6 7 8 9 10 11 12**  **13 14 15 16 17 18 19 20 21 22 23 24 25 26**  **27 28 29 30** |

1. bool validDate(string date)

Write free function validDate to return true if the string argument is a valid calendar date. Th e arguments always take the form of month, day, and year as positive integers separated by / as in "mm/dd/yyyy". If the string argument does not express a proper date, return false. You will need the free function std::**stoi**(string possibleInt) (**s**tring **to i**nteger) that returns the integer value of the string argument with the precondition that the string argument is a valid integer. For example, str("08") returns 8 and str("2021") returns 2021.

validDate("01/31/2016") returns true validDate("12/31/2017") returns true validDate("06/15/2018") returns true validDate("02/28/2019") returns true validDate("02/29/2019") returns false validDate("2019/06/06") returns false

1. int dayNumber(string date)

Write free function dayNumber to return how many days a valid date is into the year. If the string argument is not a valid date, return -1.

dayNumber("01/03/2016") returns 3 dayNumber("12/31/2017") returns 365 dayNumber("12/31/2020") returns 366 dayNumber("13/11/2020") returns -1

##### 7C CLASS STUDENT

Given header fi le Student.h below, implement the member functions in a new fi le named Student.cpp for class Student such that they satisfy all postconditions in the class defi nition (shown below). Use this table to satisfy the postconditions of Student::standing:

|  |  |
| --- | --- |
| **Credits Completed** | string **Return Value** |
| Less than 30 credits  30 credits to less than 60 credits  60 credits to less than 90 credits  90 credits or more | "Freshman"  "Sophomore"  "Junior"  "Senior" |

/\*

* De ne a Student type that knows its GPA and class standing.

\*

* File name: Student.h

\*/

#include <string>

class Student { public:

Student(std::string initName, double initCredits, double initQualityPoints); // post: Initialize a student with a 3 argument constructor // Student s("Ryan", 30.0, 120.0); // Straight A sophomore.

void completedCourse(double credits, double numericGrade); // post: Record a completed course by adding credits to credits // and incrementing the qualityPoints by credits\*numericGrade.

// aStudent.completedCourse(4.0, 3.67) is a 4 credit A-

double getGPA() const; // post: return the current grade point average.

std::string getStanding() const; // post: use selection to return the current standing as either

// Freshman, Sophomore, Junior, or Senior.

std::string getName() const; // post: return the student's name

private:

std::string name;

double credits; // Total credits completed

double qualityPoints; // sum of credits multiplied by grades };

Implement the methods in Student.cpp along with your own test driver. Construct at least four students at the cutoff s for the standings. Make sure you send every possible message to one or more objects.

##### 7D EMPLOYEE OVERTIME PAY AND FEDERAL INCOME TAX

To complete this project, fi rst complete class Employee from the Chapter 6 programming projects. Th is project asks you to make these three changes to the specifi cation in the previous chapter.

1. Now that Chrystal Bends, Inc. has interstate commerce, employees are entitled to 1.5 times their rate of pay for any hours worked over 40.00 in the week. For example, someone working 42 hours a week at $10.00 per hour would have a gross pay of

40 \* 10.00 + 2 \* 15.00 = 430.00. Change getGrossPay method to allow overtime pay.

1. Add method getIncomeTax to compute how much to withhold for this tax on the paycheck.
2. Add two data members, one to store marital status as either "S" for single or "M" for married and another data member to store the number of withholding allowances such as 1 through 99. Modify the constructor to take in these two new arguments. Th is code must compile:

Employee we("Peyton", 9.70, "S", 2); cout << we.getIncomeTax() << endl;

Fully test your two new methods, especially getIncomeTax. No one should be withholding too much from an employee’s paycheck. Perhaps more importantly, no one should withhold too little federal income tax per week. Employees might have to pay fi nes or even go to jail for withholding too little each week. You will need many getIncomeTax messages with a lot of categories for both single and married employees. Th ere are 14 total income tax withholding categories for a weekly payroll period, and you are required to use the Percentage Method Tables of IRS Publication 15 (Circular E) Employer's Tax Guide to determine these categories. Th e percentage method from the IRS Employer's Tax Guide is summarized in the following section:

##### THE PERCENTAGE METHOD FROM THE IRS EMPLOYER'S TAX GUIDE

Under the percentage method, you will use “TABLE 1—Weekly Payroll Period” for the weekly employee payroll period based on the number of withholding allowances claimed on the Form W-4 and the amount of wages; fi nd the amount of tax to withhold. Use these steps to deterimine the income tax to withhold under the percentage method:

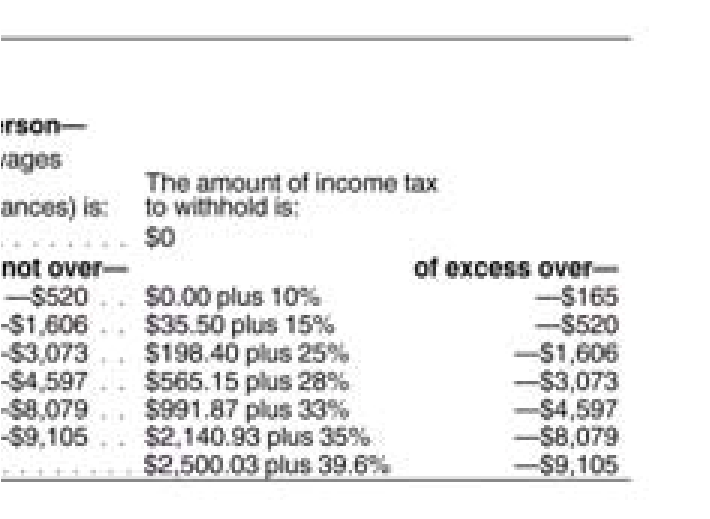
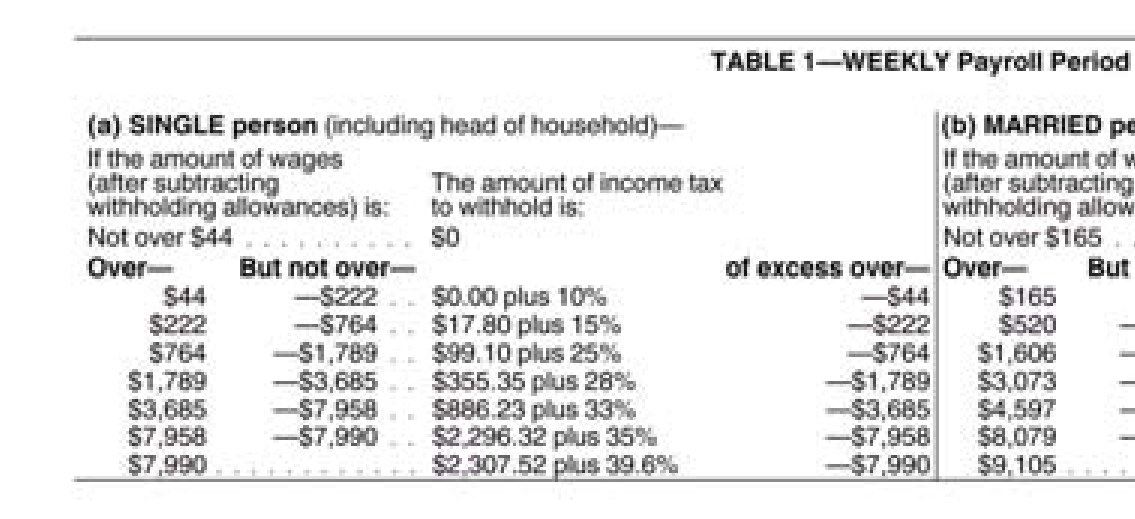
1. Multiply one withholding allowance for your payroll period by the number of allowances the employee claims. One Withholding Allowance = $76.00.
2. Subtract that amount from the employee’s gross pay.
3. Determine the amount to withhold from the appropriate table (see table below).

Example: An unmarried employee is paid $600 this week. Th is employee has a Form W-4 claiming two withholding allowances. Using the percentage method, fi gure the income tax to withhold as follows:

|  |  |
| --- | --- |
| 1. Total wage payment | $600.00 |
| 2. One allowance | $76.90 (changes year to year) |
| 3. Allowances claimed on Form W-4 | 2 |
| 4. Multiply line 2 by line 3 | $153.80 |
| 5. Amount subject to withholding (subtract line 4 from line 1) | $446.20 |
| 6. Tax to be withheld on from Table: | $17.80 + 0.15 × ($446.20 − $222) = $51.43 |

(2nd row under (a) SINGLE person)

*Note:* Th e table below has the details for amounts to withhold for 2015. For other years, fi nd the IRS Publication 15 for the year in question, (Circular E), Employer’s Tax Guide.



© 2018 Franklin, Beedle & Associates All rights reserved

**C H A P T E R E I G H T**

# Repetition

## SUMMING UP

Two important control structures have now been discussed—sequence and selection. Sequential control refers to the time when every statement executes—one after another. One of those statements could be a selection statement; in that case, one or more statements may be skipped. Selection executes diff erent actions under diff erent circumstances. Of course, it is your responsibility as a programmer to ensure that the proper actions always occur under the proper circumstances.

## COMING UP

Chapter 8 begins a study of repetitive control, the third major control structure. Repetition is discussed within the context of two major algorithmic patterns—the Determinate Loop pattern and the Indeterminate Loop pattern. Th ese two patterns may be implemented with the C++ for and while statements, respectively. A repetitive control structure executes some actions a specifi ed, predetermined number of times or until some event terminates the loop. After studying this chapter, you will be able to

* recognize and use the Determinate Loop pattern to execute a set of statements a predeter-mined number of times
* implement determinate loops with the C++ for statement
* recognize and use the Indeterminate Loop pattern to execute a set of statements until some event occurs to stop it (no more data, for example)
* implement indeterminate loops with the C++ while statement

### 8.1 REPETITIVE CONTROL

*Repetition* refers to the repeated execution of a set of statements. Repetition occurs naturally in non-computer algorithms such as these:

* For every name on the attendance roster, call the name. Mark “0” if absent or a checkmark if present.

221

* Practice the fundamentals of a sport.
* Add the fl our ¼ cup at a time, whipping until smooth.

Repetition is also used to express algorithms intended for computer implementation. If something can be done once, it can be done repeatedly. Th ese examples have computer-based applications:

* Process any number of customers at an automated teller machine (ATM).
* Continuously accept reservations.
* While there are more fast food items, sum each item.
* Compute the course grade for every student in a class.
* Microwave the food until either the timer reaches 0, the Cancel button is pressed, or the oven door is opened.

Th is chapter examines repetitive algorithmic patterns and the C++ statements that implement them. It begins with a statement that executes a collection of actions a fi xed, predetermined number of times.

**8.1.1 WHY IS REPETITION NEEDED?**

Many jobs once performed by hand are now accomplished by computers at a much faster rate. Th ink of a payroll department with the job of producing employee paychecks. With only a few employees, this task could certainly be done by hand. However, with several thousand employees, a very large payroll department would be necessary to hand compute and generate that many paychecks in a timely fashion. Other situations requiring repetition include, but are certainly not limited to: fi nding an average, searching through a collection of objects for a particular item, alphabetizing a list of names, and processing all the data in a fi le. Let’s start with the following code that fi nds the average of exactly three numbers. No repetitive control is present yet.

double sum = 0, average, number; cout << "Enter number: "; // <- Repeat cin >> number; // <- these sum = sum + number; // <- statements

cout << "Enter number: "; cin >> number; sum = sum + number;

cout << "Enter number: "; cin >> number; sum = sum + number;

average = sum / 3.0; cout << "average = " << average;

Th ere is a drawback to this brute-force approach to repetition. Any time a larger or smaller set of numbers needs averaging, the program itself must be modifi ed. It is not general enough to handle input sets of various sizes. Using the previous approach, the three statements would need

©

to be repeated for every number. Th is means averaging 100 numbers would require an additional

97 copies of these three statements. Also, the constant 3.0 in average = sum / 3.0; would have to be changed to 100.0. A situation like this is improved with a structure that can execute these three statements over and over again.

**8.2 ALGORITHMIC PATTERN:**

### THE DETERMINATE LOOP

Without the selection control structures of the preceding chapter, computers are little more than nonprogrammable calculators. Selection control makes computers more adaptable to varying situations. However, what makes computers even more powerful is their ability to repeat the same actions accurately and very quickly. Two algorithmic patterns emerge. Th e fi rst involves performing some action a specifi c, predetermined (known in advance) number of times. For example, to fi nd the average of 142 test grades, repeat some process exactly 142 times. To pay 89 employees, repeat some process 89 times. To produce grade reports for 32,675 students, repeat some process 32,675 times. Th ere is a pattern here.

In each of these examples, the program requires that somehow the exact number of repetitions be predetermined. In these situations, the number of times to repeat the process must be pre-established and constant. One too many or one too few repetitions results in an incorrect algorithm. Th is pattern of predetermining the number of repetitions and then executing a set of statements precisely that number of times is called the Determinate Loop pattern.

#### Th e Determinate Loop Pattern

|  |  |
| --- | --- |
| **Pattern:** | Determinate Loop |
| **Problem:** | Do something exactly *n* times, where *n* is known in advance. |
| **Algorithm:** | *determine n*  *repeat the following n times* { *perform these actions* } |
| **Code**  **Example:** | double sum = 0.0; int n; double number; |

cout << "Enter n: "; cin >> n;

// do something n times

for (int count = 1; count <= n; count = count + 1) { cout << "Enter number: "; // <- Repeat these cin >> number; // <- statements sum = sum + number; // <- n times }

Th e Determinate Loop pattern uses an integer—named n here—to represent the number of times the process must repeat. However, other appropriately-named objects certainly are allowed, such as numberOfEmployees. So the fi rst thing to do in the Determinate Loop pattern is to determine n somehow.

n = *number of repetitions*

Th e number of repetitions may come from keyboard input as in cin >> n;. Or n may be defi ned at compile time, int n = 124;. Or n may be passed as an argument to a function as in pow(x, n). Once n is defi ned, another object—named count here—controls the loop iterations. Other appropriately-named objects could be used, counter for example. Th e Determinate Loop pattern is shown next in the context of a small program. It is implemented with the C++ for statement.

// Determine the average of n inputs. The user must supply n.

#include <iostream> // For cout, cin, and endl using namespace std;

int main() {

int n = 0; // The number of inputs--supplied by user double sum = 0.0; // Keep running sum double number; // Temporarily store each input double average; // Holds the average for potential future use

cout << "How many numbers do you need to average? "; cin >> n;

for (int count = 1; count <= n; count = count + 1) { cout << "Enter number: "; cin >> number; sum = sum + number; }

average = sum / n;

cout << "Average of " << n << " numbers is " << average;

return 0; }

#### Dialogue

How many numbers do you need to average? ***4***

Enter number: ***70***

Enter number: ***80***

Enter number: ***90***

Enter number: ***100***

Average of 4 numbers is 85

C++ has several structures for implementing the Determinate Loop pattern. Th e for statement is most frequently used because it combines everything needed after n is determined.

#### 8.2.1 THE for STATEMENT

Th e following for statement shows the three components that maintain the Determinate Loop pattern with the C++ for statement.

int n = 5; // Predetermined number of iterations for (int count = 1; count <= n; count = count + 1) {

// Execute this block n times

}

In the preceding for loop, count is declared and initialized with the value of 1. Next, count <= n (1 <= 5) evaluates to true and so the block executes. When the statements inside the block are done, count increments by 1 (count = count + 1). Th ese three components ensure that the block executes n times:

count = 1 // Declare and initialize counter count <= n // Loop test count = count + 1 // Update counter

##### General Form 8.1 for *statement*

for (*initial-statement*; *loop-test*; *update-step*) { *repeated-statements(s)*;

}

When a for loop is encountered, the *initial-statement* is executed fi rst—and only once. Th e *loop-test* is then checked before each execution of the *repeated-statement(s)*. Th e *update-step* executes after each iteration of the repeated part. Th is process continues until the loop test is false.

Th is generalized behavior of a for loop is summarized in this fl owchart view:

*initial-statement*

*loop-test*

*update-step*

**true**

**false**

go on to the next statement

*repeated-part*

for

count 15

int n = 5;

for (int count = 1; count <= n; count = count + 1) { cout << count << " "; }

##### Output

1 2 3 4 5

Although a block is not necessary to repeat one statement, consider always using a block

({ }) with the for loop. Th is practice helps avoid diffi cult-to-detect intent errors later if you get into the habit of using { and }.

#### 8.2.2 OTHER INCREMENT AND ASSIGNMENT OPERATORS

Assignment operations alter computer memory even when the object on the left of = is also involved in the expression to the right of =. For example, the int object count is updated by +1 with this assignment operation:

count = count + 1;

Th is type of update—incrementing an object—is used so frequently that C++ off ers other, additional incrementing operators. Th e ++ and -- operators increment and decrement an object by 1, respectively. For example, the expression count++; adds 1 to the value of count. Th e expression x-- reduces x by 1. Th e ++ and -- unary operators alter the numeric object that they follow (see the table below).

|  |  |
| --- | --- |
| **Statement** | **State of** count |
| int count = 0; | 0 |
| count++; | 1 |
| count++; | 2 |
| count--; 1 | |

So, within the context of the counting for loop, the update step can be written as count++ rather than count = count + 1. Th e for loop

for (int count = 1; count <= n; count = count + 1) // . . .

may now be written as this equivalent loop with the ++ operator used in the update step:

for (int count = 1; count <= n; count++) // . . .

Th ese new assignment operators are shown because they provide a convenient method for accomplishing incrementing and decrementing operations in the for loop. Another reason has to do with the fact that most C and C++ programs use the ++ operator in for loops.

C++ also has several assignment operators in addition to =. Two of them are used to add and subtract value from an object.

|  |  |
| --- | --- |
| **Operator** | **Equivalent Meaning** |
| += | Increment object on left by value on right. |
| -= | Decrement object on left by value on right. |

Th ese two new operators alter the numeric object that they follow (see the table below).

|  |  |
| --- | --- |
| **Statement** | **State of** count |
| int count = 0; | 0 |
| count += 3; | 3 |
| count += 4; | 7 |
| count -= 2; | 5 |

Whereas the operators ++ and -- increment and decrement the object by one, respectively, the operators += and -= increment and decrement the object by any amount. Th e += operator is most often used to accumulate values inside a loop. For example, the following code sums all input by the user:

// Demonstrate the summing pattern

#include <iostream> using namespace std;

int main() { int n; double aNum;

double sum = 0.0; // Maintains running sum, so start at 0.0

cout << "How many numbers are there to sum? "; cin >> n;

cout << "Enter " << n << " numbers now: "; for (int count = 1; count <= n; count++) { cin >> aNum;

sum += aNum; // Equivalent to sum = sum + aNum;

}

cout << "Sum: " << sum << endl;

return 0;

}

|  |  |  |
| --- | --- | --- |
| **Dialogue** | |  |
| How many numbers are there to sum? ***4***  Enter 4 numbers now: ***7.5 3.0 1.5 2.0*** | |  |
| for **Loop Iteration State of** count **State of** aNum | | **State of** sum |
| 0 0 | 0.0 | 0.0 |
| 1 1 | 7.5 | 7.5 |
| 2 2 | 3.0 | 10.5 |
| 3 3 | 1.5 | 12.0 |
| 4 4 | 2.0 | 14.0 |

Th e += and -= operators also increment and decrement for the loop counters by values other than 1:

for (int count = 0; count <= 10; count += 2) { // Count by twos cout << count << " ";

}

// Output: 0 2 4 6 8 10

#### SELF-CHECK

8-1 Does a for loop evaluate the loop test fi rst?

8-2 Must a for loop update step increment the loop counter by + 1?

8-3 Do all for loops always execute the repeated part at least once?

8-4 Describe a situation when the loop test count <= n of a for loop never becomes false.

8-5 Write the output from the following program segments.

a. for (int c = 1; c < 5; c = c + 1) { cout << c << " "; }

b. int n = 5; for (int c = 1; c <= n; c++) { cout << c << " ";

}

1. int n = 3; for (int c = -3; c <= n; c += 2) { cout << c << " "; }
2. for (int c = 0; c < 5; c++) { cout << c << " "; }
3. for (int c = 5; c >= 1; c --) { cout << c << " "; }
4. cout << "before" << endl; int n = 0; for (int c = 1; c <= n; c++) { cout << c << " ";

} cout << "after" << endl;

* 1. Write a for loop that displays all the integers from 1 to 100 on separate lines.
  2. Write a for loop that displays all the integers from 10 down to 1.

##### 8.2.3 DETERMINATE LOOPS WITH Grid OBJECTS

A Grid object has row numbers that range from 0 to aGrid.nRows()-1 inclusive. Th e column numbers range from 0 to aGrid.nColumns()-1 inclusive. Using these facts and the Determinate Loop pattern allows us to manipulate Grid objects more compactly. For example, the blockBorder function (below) has two for loops that block all intersections on all four edges of any Grid object.

// Use for loops to set blocks around a Grid of any size

#include <iostream> using namespace std; #include "Grid.h" // For the Grid class

void setBorder(Grid & g) { // Changing g changes the argument

// pre: The mover is not on an edge

// post: The entire outside border is blocked int r, c;

// It is useful that objects know things about themselves--number

// of rows and columns for example, which vary from Grid to Grid for (r = 0; r < g.nRows(); r++) {

g.block(r, 0); // Block west edge

g.block(r, g.nColumns()-1); // Block east edge }

// The rst and last columns are blocked already so block // column #1 up to 1 less than the last column for (c = 1; c < g.nColumns() - 1; c++) {

g.block(0, c); // Block most of the north edge

g.block(g.nRows()-1, c); // Block most of the south edge

}

}

int main() {

Grid aGrid(8, 10, 1, 1, east);

Grid anotherGrid(3, 30, 1, 28, west);

setBorder(aGrid); aGrid.display(); cout << endl;

setBorder(anotherGrid); anotherGrid.display(); return 0;

}

###### Output

The Grid

# # # # # # # # # #

# > . . . . . . . #

# . . . . . . . . #

# . . . . . . . . #

# . . . . . . . . #

# . . . . . . . . #

# . . . . . . . . #

# # # # # # # # # #

The Grid

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# . . . . . . . . . . . . . . . . . . . . . . . . . . . < #

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

Th e for loops—applied in yet another instance of the Determinate Loop pattern—reduce the number of instructions. For example, a 20-by-20 Grid would require exactly 76 block messages. More importantly, such a brute-force approach would allow the function to work only on a 20-by-20 Grid. Th e accessor functions Grid::nColumns and Grid::nRows and the determinate for loop pattern allow the function to work properly for any sized Grid. Th is is because every Grid object knows its own size.

#### SELF-CHECK

8-8 What diff erence occurs when the fi rst for loop in setBorder is changed to for (r = 0; r <= g.nRows(); r++)?

8-9 What diff erence occurs when the second for loop in setBorder is changed to for (c = 1; c < g.nColumns(); c++)?

8-10 What diff erence would occur when the function heading is changed to void setBorder(Grid g)?

### 8.3 APPLICATION OF THE DETERMINATE LOOP PATTERN

*Problem:* Write a program that determines a range of temperature readings. Range is defi ned as the diff erence between the highest and lowest. Th e user must supply the number of temperature readings fi rst.

In this application, the user is required to enter the total number of temperature readings before entering the actual temperatures. Th e output must be labeled as Range followed by the range of temperatures (23–11 or 12 with this dialogue). Th e dialogue must look like this:

#### Dialogue

Enter number of temperature readings: ***6***

Enter temperatures:

***11***

***15***

***19***

***23***

***20***

***16***

Range: 12

#### 8.3.1 ANALYSIS

Th e number of temperature readings will fi rst be obtained from the user. An integer named n will serve nicely. Another numeric object is required to hold the individual temperature readings as they are processed. Th is object could be appropriately named aTemp. Th e range of temperature readings is the diff erence between the highest and lowest temperature readings in the list, so two more objects are needed to store the highest and lowest.

To fi nd the range without the aid of a computer (easier with a small number of temperature readings), one could glance at the list of numbers and simply keep track of the highest and lowest while scanning the list from top to bottom:

|  |  |  |
| --- | --- | --- |
| aTemp | highest | lowest |
| -5 | -5 | -5 |
| 8 | 8 | -5 |
| 22 | 22 | -5 |
| -7 | 22 | -7 |
| 15 | 22 | -7 |

For this set of data, the range = highest - lowest = 22 - (-7) = 29 as shown in this sample problem.

|  |  |  |  |
| --- | --- | --- | --- |
| **Problem Description** | **Object Name** | **Sample Values** | **Input/Output** |
| Compute the range of | n | 5 | Input |
| temperature readings | aTemp | -5, 8, 22, -7, 15 | Input |
|  | highest | 22 | Process |
|  | lowest | -7 | Process |
|  | range | 29 | Output |

#### 8.3.2 DESIGN

For a large list—an approach more suited to a computer—the algorithm mimics the repetition of the hand-operated version just suggested. It uses a determinate loop to compare every temperature reading in the list to the highest and lowest—updating them if necessary.

##### Determining the Range Algorithm

*input the number of temperature readings (n)* for *each temperature reading* { *input aTemp from user*

if *aTemp is greater than highest so far,*  *store it as the highest*  if *aTemp is less than lowest so far,*  *store it as the lowest*

} *range = highest – lowest*

As usual, it is a good idea to walk through the algorithm to verify its soundness.

1. Input the number of temperature readings (n == 5)
2. Input aTemp from user (aTemp == -5)
3. If aTemp > highest so far (-5 > Whoops!), store it as highest

Th ere is no value for highest or lowest. Let us now assume the program will initialize highest and lowest to 0:

lowest = 0; highest = 0;

1. Input the number of temperature readings (n == 5)
2. Input aTemp from user (aTemp == -5)
3. If aTemp is greater than highest so far (-5 > 0), store it as highest (highest stays 0)
4. If aTemp is less than lowest so far (-5 < 0), store it as lowest (lowest becomes -5) Well, this seems to work. How about one more iteration?
5. Input aTemp from user (aTemp == 8)
6. If aTemp is greater than highest so far (8 > 0), store it as highest (highest becomes 8)
7. If aTemp is less than lowest so far (8 < 0), store it as lowest (lowest stays -5)

Seems okay. Try three more inputs to verify that highest and lowest are correct. Finally, the last step in the algorithm (after the repetition) produces the range: range = highest - lowest.

#### SELF-CHECK

8-11 What is the range when n is 4 and the temperature readings are 1 2 3 4?

If you did the previous self-check question correctly, you will notice that lowest stays 0. Th e initial value of lowest is less than all subsequent inputs. So what might have seemed to work does not. Th e fi rst test set works only because a negative temperature was input. Th e same algorithm will not work on a warmer day when all temperatures are positive. So instead of initializing both highest and lowest to 0, consider setting highest to something ridiculously low, say -9999, so low that any input will have to be higher. Set highest to something ridiculously high like 9999, so high that any input will have to be lower. Better yet, set lowest to the largest integer that is defi ned in C++ as INT\_MAX. Also set lowest to the minimum integer that is defi ned in C++ as INT\_MIN. You may need to use #include <climits>.

#include <iostream>

#include <climits> // for INT\_MIN and INT\_MAX

int main() {

std::cout << INT\_MIN << std::endl; std::cout << INT\_MAX << std::endl; return 0; }

##### Output (may vary with diff erent compilers)

-2147483648

2147483647

##### 8.3.3 IMPLEMENTATION

Since the problem stated that the user must fi rst supply the number of inputs, the exact number of repetitions is determined. Th is is an instance of the Determinate Loop pattern.

for (int count = 1; count <= n; count++) {

// Process one input

}

Th e following program implements a corrected algorithm (written now in C++ rather than pseudocode). Notice that the user need not input the same set of temperature readings twice— the checks are made for both the highest and the lowest within the same loop.

// Determine the range of temperatures in a set of known size

#include <iostream>

#include <climits> // For INT\_MIN and INT\_MAX using namespace std;

int main() { int aTemp, n, range;

int highest = INT\_MIN; // All ints will be >= INT\_MIN int lowest = INT\_MAX; // All ints will be <= INT\_MAX

cout << "Enter number of temperature readings: "; cin >> n;

// Input rst temperature to record it as highest and lowest cout << "Enter readings 1 per line" << endl;

// Use a determinate loop to process n temperatures for (int count = 1; count <= n; count++) {

// Get the next input cin >> aTemp;

// Update the highest so far, if necessary if (aTemp > highest) highest = aTemp;

// Update the lowest so far, if necessary if (aTemp < lowest) lowest = aTemp; }

range = highest - lowest; cout << "Range: " << range << endl; return 0; }

###### Dialogue

Enter number of temperature readings: ***5***

Enter readings 1 per line

***-5***

***8***

***22***

***-7***

***15***

Range: 29

##### 8.3.4 TESTING

Programmers gain confi dence that a program works by picking an arbitrary number of test cases. Th e fi rst test case was letting n be 5 with readings of ***-5***, ***8***, ***22***, ***-7***, and ***15***. Th is set of inputs shows the diff erence between the highest and lowest is (22 − (−7)) or 29. Looking at the dialogue and seeing the range is 29 could lead us to believe that the algorithm and implementation are correct. However, the only thing that is sure is this: when those particular fi ve temperatures were entered, the correct range was displayed. Th e data used in this previous test would indicate that everything is okay. Other test cases include letting n be 1 for a range of 0, two numbers that are the same,

a sequence of all negative integers, a sequence of all positive integers, a sequence in ascending order, and another sequence in descending order.

Testing only reveals the presence of errors, not the absence of errors. If the range were shown as an obviously incorrect answer (-11, for example), hopefully, you would detect the presence of the error. Now consider this slightly diff erent implementation sometimes seen in introductory courses:

for (int count = 1; count <= n; count++) { cin >> aTemp; if (aTemp > highest) highest = aTemp; else if (aTemp < lowest) lowest = aTemp; }

#### SELF-CHECK

8-12 Trace the code above using the same input (assume n is 5):

-5 8 22 -7 15

and predict the value stored in range. Is it correct?

8-13 Trace through the same code with these inputs (assume n is 5):

5 4 3 2 1

Predict the value stored in range. Is it correct?

8-14 Trace through the same code with these inputs (assume n is 3):

1 2 3 4

Predict the value stored in range. Is it correct?

8-15 *Multiple Choice:* When is range incorrectly computed?

1. When the input is entered in descending order.
2. When the input is entered in ascending order.
3. When the input is entered in neither ascending or descending order.

8-16 What must be done to correct the error?

##### 8.3.5 WHAT TO DO WHEN AN INTENT ERROR IS DETECTED

When you detect an intent error and a loop is involved, it is recommended that you display important values, such as highest and lowest, for each iteration of the loop. Th is simple debugging tool reveals what is happening and, in so doing, helps the debugging process. A few well-placed output statements can be very revealing. For example, a debugging output statement could be included in the loop that contained the intent error:

for (int count = 1; count <= n; count++) { cin >> aTemp;

// Add an output statement in the loop to aid debugging cout << highest << " " << lowest << endl; . . .

Now the dialogue, while testing the incorrect algorithm just before the preceding self-checks, would look like this:

Enter number of temperature readings: 3

Enter readings 1 per line

5

-2147483648 2147483647

7

5 2147483647

12

7 2147483647

Range: -2147483635

**8.4 ALGORITHMIC PATTERN:**

### THE INDETERMINATE LOOP

Although the Determinate Loop pattern occurs frequently in many algorithms, it has a serious limitation—someone, somehow, must determine the number of repetitions in advance. Quite often this is impossible, or at least very inconvenient and diffi cult. For example, an instructor may have a diff erent number of tests to average as attendance varies between terms. A company may not have a constant number of employees as there are hires, fi res, layoff s, transfers, and retirements. Th e schools where the software is distributed may have a diff erent number of students each day.

It is often necessary to execute a set of statements an undetermined number of times, for example to process report cards for *every* student in a school—not precisely 310 every term. Programs cannot always depend on prior knowledge to determine the exact number of repetitions. It is often more convenient to think in terms of “process a report card for all students” rather than “process precisely 310 report cards.” Th is leads to another recurring pattern in algorithm design that captures the essence of repeating a process an unknown number of times. It is a pattern to help design a process that iterates until some event occurs to indicate the looping is fi nished.

Here are some events used in this textbook to terminate loops:

* Th e loop counter becomes greater than the desired number of iterations
* Th e mover on a Grid can no longer move forward
* Th e user enters a special value to indicate there is no more input
* Th e end of the fi le is reached (see Chapter 9: File Streams)

Whereas the number of repetitions for determinate loops is known in advance, the Indeterminate Loop pattern uses other techniques to stop. With indeterminate loops, the number of repetitions need not be known in advance.

#### Th e Indeterminate Loop Pattern

|  |  |
| --- | --- |
| **Pattern:** | Indeterminate Loop |
| **Problem:** | Some process must repeat an unknown number of times so some event is needed to terminate the loop. |
| **Algorithm:** | while (*the termination event has not occurred*) { *perform these actions*  *do something to bring loop closer to termination* } |
| **Code**  **Example:** | // Place things until the mover is blocked while (aGrid.frontIsClear()) { aGrid.putDown(); |

aGrid.move(); }

Th e C++ while statement is often used to implement the Indeterminate Loop pattern:

#### General Form 8.2 while *statement*

while (*loop-test*) { *repeated-statements*

}

Th e *loop-test* is a logical expression that evaluates to either true or false. Th e *repeated-statement(s)* may be any C++ statement, but it is usually a set of statements enclosed in { and }.

When a while loop is encountered, the loop test evaluates to either true or false. If true, the repeated part executes. Th is process continues while (as long as) the loop test is true.

*loop-test*

*statement-2*

**true**

**false**

go on to the next statement

*statement-1*

*statement-n*

## 8.4.1 THE USE OF while TO IMPLEMENT THE DETERMINATE LOOP PATTERN

Th e while loop could also implement the Determinate Loop pattern. It is simply a matter of moving the initialization before the while loop and the update step to the bottom of the repeated part.

// initialization while ( *loop-test* ) { // Activities to be repeated *update-step* }

Th e following code represents an alternate implementation of the determinate loop pattern:

// Sum the rst n integers int accumulator = 0;

int count = 1; // Initialization int n = 5; // Initialization while (count <= n) { // Loop test accumulator = accumulator + count ; // Action count++; // Update step } cout << "Sum of the rst " << n << " integers is " << accumulator;

Although the while loop can also be used for determinate loops, the for loop is more concise and convenient. It is recommended that you use the for loop when the number of iterations is known in advance. When this cannot be determined, as with indeterminate loop problems, use the while statement instead.

### 8.4.2 INDETERMINATE LOOP PATTERN WITH Grid OBJECTS

Th ere are many events used to terminate the loop in an Indeterminate Loop pattern. Consider moving the mover up until the edge of a Grid, or perhaps a block, prevents the mover from continuing—a subproblem that will come in very handy for one of the Grid-related programming projects.

// The event loop terminates when the front is no longer clear

#include <iostream> // For cout

using namespace std; // allow cout instead of std::cout #include "Grid.h" // For the Grid class

void moveTillStopped(Grid & g) {

// post: The mover is facing a block or edge in front while (g.frontIsClear()) {

g.move();

}

}

int main() {

Grid tarpit(5, 10);

cout << "When initialized with only the number of rows\n" << "and columns, a Grid object gets a random opening\n"

<< "with the mover at a random location and direction\n"

<< endl;

moveTillStopped(tarpit); tarpit.display();

return 0; }

#### Dialogue

When initialized with only the number of rows and columns, a Grid object gets a random opening with the mover at a random location and direction

The Grid

# # # # # # # # # #

. . . . . . . . . #

# . . . > #

# . . . . . . . . #

# # # # # # # # # #

Because of the randomness of the Grid object tarpit, moveTillStopped uses an indeterminate loop to advance the mover. Th e g.move() message may repeat the move message once, twice, or as many times as necessary to get the mover up against the “wall.”

### SELF-CHECK

8-17 Why must moveTillStopped use an Indeterminate Loop rather than a Determinate Loop pattern?

#### 8.4.3 THE INDETERMINATE LOOP USING A SENTINEL

A *sentinel* is a specifi c input used to terminate an indeterminate loop. A sentinel value should be the same class of data as the other input. However, the sentinel is not meant to be processed as a valid part of the input. For example, the following set of inputs hints that the input of -1.0 is the event that terminates the loop and that -1.0 is not to be counted as a valid test score. Otherwise, the average would not be 80.0.

##### Dialogue

Enter tests scores [0.0 through 100.0] or -1.0 to quit

***80.0 90.0***

***70.0***

***-1.0***

Average of 3 tests = 80

Th is dialogue asks the user either to enter data in the range of 0.0 through 100.0 inclusive or to enter -1.0 to signal the end of data.

With sentinel indeterminate loops, a message is usually displayed to indicate how the user must end the input. It is important to inform the user that a sentinel is being used. Th e user must be told what that sentinel value is. It could have been -999 or any other negative number, for example.

#### 8.4.4 USING cin >> AS A LOOP TEST

Up to this point, cin >> has often been used for input. What hasn’t been revealed is this: a cin statement returns true when it is successful. If the input operation fails to get a number, cin >> is replaced with false. Th is means that a cin >> statement can be, and often is, used as the logical expression in an if...else or while statement:

if (cin >> intObject) - or - while (cin >> intObject)

Both of the above logical expressions return true when cin successfully extracts a valid integer from the input stream. However, the same logical expressions return false if an invalid integer is encountered in the input stream or, as shown later, the end of fi le is found. With this new information, implementation of sentinel loops is simplifi ed when the cin extraction is part of the loop test.

// The priming extraction is now part of the loop test while ((cin >> testScore) && (testScore != sentinel)) { accumulator = accumulator + testScore; // Update accumulator n++; // Update total inputs }

Th e actual return value of the input statement cin >> testScore isn’t all that important.

However, because cin >> testScore is guaranteed to execute fi rst, testScore is guaranteed to have obtained a valid numeric value through keyboard input before it is compared to the sentinel.

Th e second part (testScore != sentinel) evaluates to true *only* if the input was the sentinel (-1.0 in this case). So for any valid data, this loop test is true && true, which evaluates to true. In this case, the loop executes the repeated part. For example, the loop test is evaluated like this when 95.0 is entered:

##### Loop Test Evaluation When Input Is 95.0

while ((cin >> testScore) && (testScore != sentinel)) ( true && ( 95.0 != -1.0 ))

( true && true ) true

Th e loop test is true only if a valid number is input and that number is not -1.0. With 95.0, the loop test is true (true && true is true). When the sentinel of -1.0 is entered, the loop test is false (true && false is false)—the termination condition is reached.

##### Loop Test Evaluation When Input Is -1.0 (or -1)

while ((cin >> testScore) && (testScore != sentinel)) ( true && ( -1.0 != -1.0 )) true && false false

Th is loop test is now placed into the context of a program that computes the average of any number of inputs:

// Use a sentinel of -1 to terminate a loop

#include <iostream> using namespace std;

int main() {

const double sentinel = -1.0; // User enters this to terminate double accumulator = 0.0; // Maintain running sum of inputs int n = 0; // Maintain total number of inputs double testScore, average;

// Prompt

cout << "Enter test scores [0.0 through 100.0] or " << sentinel << " to quit" << endl;

// Input and process at the same time

while ((cin >> testScore) && (testScore != sentinel)) { accumulator += testScore; // Update accumulator n++; // Update total inputs }

if (n > 0) {

average = accumulator / n;

cout << "Average of " << n << " tests = " << average << endl;

} else

cout << "Can't average 0 numbers" << endl; return 0; }

##### Dialogue

Enter test scores [0.0 through 100.0] or -1.0 to quit

***80.0 90.0 70.0 -1.0***

Average of 3 tests = 80

Th e following table traces the changing state of the important objects to simulate execution of the previous program. In addition to keeping the running sum of the test scores in accumulator, n must also be incremented by 1 for each valid testScore.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Iteration** | testScore | accumulator | n | testScore != sentinel Number |
| Before the loop | NA | 0.0 | 0 | NA |
| Loop 1 | 70.0 | 70.0 | 1 | true |
| Loop 2 | 90.0 | 160.0 | 2 | true |
| Loop 3 | 80.0 | 240.0 | 3 | true |
| After the loop | NA | 240.0 | 3 | NA |

### SELF-CHECK

8-18 Determine the average for each of the following code fragments by simulating execution when the user inputs ***70.0***, ***90.0***, ***80.0***, ***-1***.

1. cin >> testScore; while (testScore != sentinel) { cin >> testScore; accumulator += testScore; // Update accumulator

n++; // Update total inputs

}

average = accumulator / n; // Division by 0 possible

1. cin >> testScore; while (testScore != sentinel) { accumulator += testScore; // Update accumulator n++; // Update total inputs

cin >> testScore;

}

average = accumulator / n; // Division by 0 possible

* 1. If you answered 80.0 for both a and b above, redo both until you get diff erent answers.
  2. Which code (a or b) is equivalent to the previous complete program where the loop test is while ((cin >> testScore) && (testScore != sentinel))?

#### 8.4.5 INFINITE LOOPS

It is possible that a loop may never execute, not even once. It is also possible that a while loop will never terminate. Consider the following while loop that potentially continues to execute until external forces are applied (turning off the computer or a hardware failure, for example). Th is is potentially an *infi nite loop*, something that is usually undesirable.

cin >> testScore; while (testScore != sentinel) {

accumulator += testScore; // Update accumulator n++; // Update total inputs }

Th e loop repeats virtually forever. Th e termination condition can never be reached. Th e loop test is always true because there is no statement in the repeated part that brings the loop closer to the termination condition of testScore == sentinel. When writing while loops, remember to ensure that the loop test will eventually become false.

When you do get a program that executes a loop over and over again—and you will—use the system-dependent method necessary to terminate the program that is executing the infi nite loop (ask your instructor).

### SELF-CHECK

8-21 How many iterations of the previous loop occur when the user enters the sentinel (-1.0) as the very fi rst input?

8-22 What activity should be added to the previous while statement so each loop iteration brings the loop one step closer to termination?

8-23 Th e following code represents another example of an infi nite loop. What must be done to make this loop terminate as intended?

while (g.frontIsClear());

{

g.move(); }

8-24 Write the output from the following C++ program fragments:

1. int n = 3;

int counter = 1; while (counter <= n) { cout << counter << " "; counter++;

}

1. int last = 10; int count = 2; while (count <= last) { cout << count << " "; count += 2;

}

* 1. Write the number of times count is output. Assume count, sum, and n have been declared as int objects. “Zero,” “infi nite,” and “unknown” are valid answers.

|  |  |  |  |
| --- | --- | --- | --- |
| a. | while (count <= n) { cout << count<< endl;  } | d. | count = 1; sum = 0;  while (count <= 5) { sum += count; sum++;  } |
| b. | n = 5; count = 1; while (count <= n) cout << count << endl; count++; | e. | count = 1; n = 5;  while (count <= n) { count++;  } |
| c. | count = 1; n = 0;  while (count <= n) cout << " " << count; | f. | count = 10;  while (count >= 0);  {  count = count - 2;  } |

* 1. Write a code fragment that sums all integers until the user enters 999.

### 8.5 THE do whileSTATEMENT

Th e do while statement is similar to the while loop. It allows a collection of statements to be repeated while an expression is true. Th e primary diff erence is the time at which the loop test is evaluated. Th e while loop test is evaluated at the beginning of each iteration. Th e do while statement evaluates the loop test at the end of each iteration. Th is means that the do while loop always executes its repeated part at least once.

#### General Form 8.3 do while *loop*

do {

*repeated-statement(s)*

} while ( *loop-test* ) ;

When a do while statement is encountered, all statements within the block ({ }) execute. Th e *loop-test* is evaluated at the *end* of the loop—not at the beginning. If true, the *repeatedstatement(s)* executes again. If the test expression is false, the loop terminates. Although the block is not absolutely required with the while and for statements, braces must always exist between the do and the while in this statement. Here is an example of the do while loop that displays the increasing value of counter to simulate its execution.

8.5: The do while Statement

int counter = 1; int n = 4;

cout << endl << "Before loop..." << endl; do {

cout << "Loop #" << counter << endl; counter++;

} while (counter <= n); cout << "...After loop" << endl;

#### Output

Before loop...

Loop #1

Loop #2

Loop #3

Loop #4

...After loop

Th e do while loop is a good choice for repetition whenever a set of statements must be executed at least once to initialize objects used later in the loop test. For example, the do while loop is the preferred statement when asking the user to enter one of several options. For example, the do while loop in the char type function nextOption repeatedly requests the user to enter one of three choices. Th e loop does not terminate until the user enters a valid option. main also uses a do while loop to process as many deposits and withdrawals as the user wants.

// Use a do while loop that repeatedly asks for a valid option

#include <cctype> // For toupper

#include <iostream> // For cout, cin, and endl

#include <string> using namespace std;

char nextOption() {

// post: Return an uppercase W, D, or Q string option; char rstChar; do {

cout << "W)ithdraw, D)eposit, or Q)uit: "; cin >> option;

rstChar = toupper(option.at(0));

} while (( rstChar != 'W') && ( rstChar != 'D') && ( rstChar != 'Q')); return rstChar;

}

int main() { char choice = 'Q';

do {

choice = nextOption(); // assert: choice is either 'Q', 'W', or 'D'

if ('W' == choice)

cout << "\nValid entry--process W\n" << endl;

if ('D' == choice)

cout << "\nValid entry--process D\n" << endl;

if ('Q' == choice)

cout << "\nHave a nice day :)" << endl; } while (choice != 'Q');

return 0; }

#### Dialogue (the user enters one valid entry, three invalid choices, and quits with Q)

W)ithdraw, D)eposit, or Q)uit: W

Valid entry--process W

W)ithdraw, D)eposit, or Q)uit: make

W)ithdraw, D)eposit, or Q)uit: 3

W)ithdraw, D)eposit, or Q)uit: invalid entries

W)ithdraw, D)eposit, or Q)uit: Q

Have a nice day :)

Because at least one character must be obtained from the keyboard before the test expression evaluates, the do while loop in nextOption is used instead of a while loop—the loop must iterate at least once. Also, a do while loop is used in main to get an option because it needs at least one user input to evaluate whether or not the user wants to quit. Although do while is not necessary, it is a bit easier to implement than using a while loop.

cout << "W)ithdraw, D)eposit, or Q)uit: "; cin >> option;

rstChar = toupper(option.at(0)); while (( rstChar != 'W') && ( rstChar != 'D') && ( rstChar != 'Q')) { cout << "W)ithdraw, D)eposit, or Q)uit: "; cin >> option;

rstChar = toupper(option.at(0));

}

#### SELF-CHECK

8-27 Write the output produced by the following code:

1. int count = 1;

do { cout << count << endl; count++;

} while (count <= 3);

1. double x = -1.0; do { cout << x << endl; x = x + 0.5; } while (x <= 1.0);

8.6: Loop Selection and Design

* 1. Write a do while loop that prompts for and reads numbers until the number is in the range of 1 through 10 inclusive.
  2. Write a do while loop that prompts for and then reads characters until the user enters w, W, d, D, q, or Q at the prompt Enter W)ithdraw D)eposit Q)uit:.

### 8.6 LOOP SELECTION AND DESIGN

For some people, loops are easy to implement, even at fi rst. For others, infi nite loops and intent errors are more common. In either case, the following outline is off ered to help you choose and design loops in a variety of situations:

1. Determine which type of loop to use
2. Determine the loop test
3. Write the statements to be repeated
4. Bring the loop one step closer to termination
5. Initialize objects if necessary

#### 8.6.1 DETERMINE WHICH TYPE OF LOOP TO USE

If the number of repetitions is known in advance or read as input, use the Determinate Loop pattern, which has a statement specifi cally designed for this—the for loop. Although you can use the while loop to implement the Determinate Loop pattern, consider using the for loop instead. Th e while implementation allows you to omit one of the key counting parts, making any intent errors more diffi cult to detect and correct. If you omit one of the counting parts of a for loop, you’ll get an easy-to-detect-and-correct compile time error.

If you need to wait until some event occurs during execution of the loop, the Indeterminate Loop pattern is more appropriate. If so, use the while loop. If the loop must always execute once, for example when input data must be checked for validity (an integer value that must be in the range of 0 through 100), use the do while loop. A do while loop is also a good choice for menudriven programs that repeatedly request options until the menu choice for *quit* is entered.

#### 8.6.2 DETERMINE THE LOOP TEST

If the loop test is not obvious, try writing the conditions that must be true for the loop to terminate. For example, if you want the user to enter STOP to stop entering input, the termination condition is:

inputName == "STOP"; // Termination condition

Th e logical negation inputName != "STOP" can be used directly as the loop test of a while or do while loop.

while (inputName != "STOP") { do {

// . . . // . . .

} } while (inputName != "STOP")

#### 8.6.3 WRITE THE STATEMENTS TO BE REPEATED

Th is is why the loop is being written in the fi rst place. Some common tasks include keeping a running sum, keeping track of a high or low value, or counting the number of occurrences of some value. Other tasks that will be seen later include searching for a name in a list or repeatedly comparing all string elements of a list in order to alphabetize it.

#### 8.6.4 BRING THE LOOP ONE STEP CLOSER TO TERMINATION

To avoid an infi nite loop, at least one action in the loop body must bring it closer to termination. In a determinate loop, this might mean incrementing or decrementing a counter by some specifi c value. Inputting the next value is another way to bring loops closer to termination—for example, when the user inputs data until the sentinel is extracted from the input stream. In a for loop, the repeated statement should be designed to bring the loop closer to termination, usually by incrementing the counter. In general, the loop test should contain at least one object that is altered during each iteration of the loop.

#### 8.6.5 INITIALIZE OBJECTS IF NECESSARY

Check to see if any objects used in either the body of the loop or the loop test need to be initialized. Doing this usually ensures that the objects of the loop and the objects used in the iterative part have been initialized. For example, consider this loop:

double sum, x, average; int n;

cout << "Enter numbers or -1 to stop: "; while (x != -1) { sum = sum + x; n++; cin >> x;

} average = sum / n;

With this code, the values of sum, average, x, and n are garbage. What is the initial value of sum, perhaps -1,234.5 or 99,999.9? Th e fi rst value of x is unknown, as is that of n. Consider each object in the loop test and the iterative part as potential candidates for initialization. Th is may require setting n to 0, but it may also require that some object becomes initialized through the input statement. Here is the corrected code:

Chapter Summary

double sum = 0.0; int n = 0;

double x, average; // x and average don't require initialization cout << "Enter numbers or -1 to stop: "; cin >> x; while (x != -1) { sum += x; n++; cin >> x;

} average = sum / n;

#### SELF-CHECK

8-30 Which loop best accomplishes these tasks?

1. Sum the fi rst fi ve integers (1 + 2 + 3 + 4 + 5).
2. Find the average value for a set of numbers when the size of the set is known.
3. Find the average value for a set of numbers when the size of the set cannot be determined by the program or the user until the data has been completely entered.
4. Obtain a character from the user that must be an uppercase I, S, or Q.

8-31 For a loop to process inputs called value until -1 is entered:

1. Write a termination condition.
2. Write the loop test for a while or do while loop.

8-32 For each loop, which objects are not initialized but should be?

1. while (count <= n) {

// . . . ;

}

1. for (int count = 1; count <= n; count = count + inc) {

// . . . ;

}

### CHAPTER SUMMARY

* Repetition is an important method of control for all programming languages. Typically, the body of a loop has statements that may change the state of one or more objects during each loop iteration.
* Th e for loop is often used to implement the Determinate Loop pattern, which requires that the number of repetitions be determined *before* the loop is encountered.
* Determinate loops rely on this value (n perhaps) and a properly initialized and incremented loop counter (count perhaps) to track the number of repetitions. Th e counter is compared to the known number of iterations at the start of each loop. Th e counter is automatically updated at the end of each for loop iteration.
* Th ere are a number of ways to determine the number of loop iterations before the loop executes. Th e number of iterations may be input from the user, passed as an argument to a function, initialized in advance, or n may be part of the state of some object. For example, every Grid object knows its number of rows and number of columns. Every string object knows how many characters it has at any given moment.
* Th is Determinate Loop pattern is so common that a specifi c statement—the for loop—is built into almost all languages.
* Indeterminate loops rely on some external event for their termination. Th e terminating event may occur at any time.
* Indeterminate loops are used when the program is unable to determine, in advance, the number of times a loop must iterate. Th e terminating events include sentinels extracted from the input stream (-1 as a test or “Q” in a menu selection). Th ese types of loops allow any number of bank customers to make any number of transactions or repeatedly prompt a user for input until valid input is entered.
* Although the while loop is the only repetitive statement needed to solve any computer problem, the for loop is more convenient under certain circumstances. Th e for loop requires the program to take care of the initialization, loop test, and repeated statement all at once. Th e compiler protests if one of these important steps is missing. Th e for loop provides a more compact and less error-prone determinate loop.
* Remember these steps if you are having trouble designing loops:

 Determine which type of loop to use

 Determine the loop test

 Write the statements to be repeated

 Bring the loop one step closer to termination

 Initialize objects if necessary

### EXERCISES

1. How many times will the following loops execute cout << "Hello ";? “Zero,” “unknown,” and “infi nite” are perfectly legitimate answers.
   * 1. int n = 5;

for (int count = 1; count <= n; count++) {

cout << "Hello ";

}

* + 1. int n = 0;

for (int count = 5; count >= n; count --) {

cout << "Hello ";

}

Exercises

* + 1. int n = 5;

for (int count = 1; count <= n; count --) {

cout << "Hello ";

count++;

}

* + 1. int n = 0;

for (int count = 1; count <= n; count++) {

cout << "Hello ";

}

1. Write the output produced by these for loops:

for (int counter = 1; counter <= 5; counter++) cout << " " << counter;

cout << "Loop One"; // Incorrectly indented to confuse for (int counter = 10; counter >= 1; counter--) cout << " " << counter;

cout << "Blast Off"; // Correctly indented to avoid confusion

1. Write loops to produce the outputs shown:
   1. 10 9 8 7 6 5 4 3 2 1 0
   2. 0 5 10 15 20 25 30 35 40 45 50
   3. -1000 -900 -800 -700 -600 -500 -400 -300 -200 -100 0
2. Write the output generated by the following code:

int count = 0; while (count < 5) { cout << " " << count ; count = count + 1; }

1. Write a loop that sums all the integers between start and stop inclusive that are input from the keyboard. You may assume start is always less than or equal to stop. If the input were 5 for start and 10 for stop, the sum would be 5 + 6 + 7 + 8 + 9 + 10 (45).
2. How many times will Hello be displayed using the following program segments? “Zero,” “undetermined,” and “infi nite” are perfectly legitimate answers.

a. while (count <= 10) cout << "Hello"; b. count = 1; while (count <= 7) { cout << "Hello";

count++;

}

* + 1. count = 7; while (count <= 1) { cout << "Hello";

}

* + 1. count = 1; while (count <= 5) cout << "Hello"; count++;

1. Write a while loop that produces this output:

-4 -3 -2 -1 0 1 2 3 4 5 6

1. Write a while loop that displays 100, 95, . . ., 5, 0 on separate lines.
2. Write a loop that counts how many perfect scores (the number 100) are entered from the keyboard.
3. Convert the following code to its for loop counterpart:

int counter = 1; double sum = 0; int n, number;

cout << "Enter number of ints to be summed: "; cin >> n; while (counter <= n) { cin >> number; sum = sum + number; counter++;

} cout << sum;

1. Write a loop that counts the number of words input by a user until the user enters the string ENDOFDATA (must be uppercase letters, no spaces).
2. Write the complete output generated by the following program when the user enters 1, 2, 3, 4, and -1 on separate lines.

#include <iostream> using namespace std;

int main() { double test; double sum = 0.0;

cout << "Enter tests or a negative number to stop: " << endl; while ((cin >> test) && (test >= 0.0)) { sum = sum + test;

}

cout << "Sum: " << sum << endl; }

1. Write the output generated by the following code:

string choice("BDWBQDW"); int count = 0;

while (choice[count ] != 'Q') {

Exercises

cout << "Opt: " << choice[count ] << endl; count++; }

1. How many times will the following loops execute cout << "Hello ";? “Zero,” “unknown,” and “infi nite” are perfectly legitimate answers.

|  |  |  |  |
| --- | --- | --- | --- |
| a. | count = 1; n = 10; do {  cout << "Hello "; } while (count > n); | c. | count = -1; do {  cout << "Hello "; count++;  } while (count != -3); |
| b. | n = 10; count = 1; do {  cout << "Hello "; count = count - 2; } while (count <= n); | d. | count = 1; do {  cout << "Hello "; count++;  } while (count <= 100); |

1. Write a do while loop that generates this output:

10 9 8 7 6 5 4 3 2 1 0

1. Write the output generated by the following program:

#include <iostream> using namespace std;

int main() { int count = -2; do {

cout << " " << count; count--;

} while (count > -6);

}

1. Convert the following code to its do while counterpart:

char option;

cout << "Enter option A, B, or Q: "; cin >> option; option = toupper(option);

while ((option != 'A') && (option != 'B') && (option != 'Q')) { cout << "Enter option A, B, or Q: "; cin >> option; option = toupper(option); }

1. Write a function named option that prompts for and returns an uppercase S, A, M, or Q only. Th e return type of the option function must be char. Return S, A, M, or Q through the function name, not as a reference. Th e following code must only assign one of four allowed letters to choice:

char choice = option(); cout << choice; // Output must be either S, A, M, or Q only!

### PROGRAMMING TIPS

1. Pick the type of loop you want to use. After recognizing the need for repetition, decide if the number of repetitions can be determined in advance. If so, this is a determinate loop that is best implemented as a for loop. If the number of iterations cannot be determined in advance, fi rst determine the event that will terminate the loop. Use its logical negation as the loop test. For example, the loop will terminate when someone enters the word

STOP. Th e termination condition is (word == "STOP"). Th e loop test is the logical negation while (word != "STOP").

1. Beware of infi nite loops. Th ey are easy to create and sometimes very diffi cult to fi nd. Can you spot why these are infi nite loops?

|  |  |
| --- | --- |
| 1. int count = 1; int sum = 0;   while (count <= 100);  { // Sum the rst 100 integers sum += count ; count++; }   1. for (int count = 0; count <=   100; count++);  { // Sum the rst 100 integers sum += count ;  } | 1. int count = 1; int sum = 0;   while (count <= 100)  // Sum the rst 100 integers sum += count ; count++;   1. for (int count = 0; count <= 100; count++) {   // Sum the rst 100 integers sum += count ; count --;  } |

1. Always write a compound statement for the iterative part of a while loop even if it is not necessary. Th is provides a better chance of including any increment statement as part of the loop rather than accidentally leaving it outside the loop.
2. Use debugging prints to fi nd out what is going on in a loop. When in doubt, place a debugging cout statement inside the loop to display some important object that should be changing. Th is can be very revealing. Sometimes you’ll spot an infi nite loop. Other times you might spot that the loop test was never true.

while ( . . . ) {

// . . .

mid = (lo + hi) / 2.0;

cout << "In loop, mid == " << mid << endl; }

1. Loops may not always execute the iterative part. It is possible that a loop will execute zero times, or fewer than you might have thought.
2. Th ings become much simpler when the input statement is part of an indeterminate loop test. If you are used to a priming read for sentinel loops—especially you Pascal programmers—try to forget it. C++ allows input as part of the loop test, so it is easier to write sentinel loop tests like this:

while ((cin >> aNumber) && (aNumber != sentinel)) {

// Process aNumber but not the sentinel

}

1. Sometimes a "quasi-infi nite" loop with a break is the easiest way to implement a loop. If you are having trouble with loop tests, consider using a loop with a guarded break (this code is equivalent to the previous sentinel loop):

while ( true ) { cin >> aNumber;

if (aNumber == sentinel) // The termination condition break; // Exit this loop

// Otherwise process aNumber

}

**PROGRAMMING PROJECTS**

## 8A WIND SPEED RECORDING

Write a program that determines the lowest, highest, average, and range of a set of wind speed readings which are all positive or zero. Terminate the loop with any negative input. Be sure you notify the user how to terminate data entry.

### Dialogue

Enter wind speed readings or a number < 0 to quit:

5.0 6.0 2.0

8.0 -999 n: 4 High: 8

Low: 2

Range: 6

Ave: 5.25

## 8B BANK TELLER

Write a C++ program that allows the user to create exactly one BankAccount object and then make as many withdrawals and deposits as desired. Th e fi nal line of output should be the balance.

Your code should not allow a withdrawal greater than the balance. Use the following dialogue as a guide to this problem’s specifi cation:

Enter customer name: Jackson

Enter initial balance: 0.00

W)ithdraw, D)eposit, or Q)uit: ***D*** Enter deposit amount: 250.00

W)ithdraw, D)eposit, or Q)uit: ***W***

Enter withdrawal amount: 300.00

\*\*Amount requested exceeds account balance\*\*

W)ithdraw, D)eposit, or Q)uit: ***w*** Enter withdrawal amount: 200.00

W)ithdraw, D)eposit, or Q)uit: ***q***

Ending balance: 50

## 8C FIND THE Grid EXIT

Write a C++ function named ndExit that instructs the mover to fi nd the lone exit in any Grid. Make sure you initialize the Grid object with only two arguments—number of rows and number of columns. Th is ensures that you will get a Grid that has only one exit. Also, the mover will be at a random location facing a random direction every time you run the program. Th is will help you test your solution. Use the following test driver:

#include "Grid.h" // For the Grid class

void ndExit(Grid & g) {

// pre: The Grid has exactly one exit, but not at a corner

// post: The mover is located at the lone exit // You complete the function here }

int main() {

// Test drive ndExit

Grid tarpit(10, 16);

// assert: The 10-by-16 Grid has the mover in a random location

tarpit.display(); ndExit(tarpit); tarpit.display();

return 0;

}

### Output

The Grid:

# # # # # # # # # # # # # # # #

# #

# . . . . . . . . . . . . #

# . . . . . . . . . . . . #

# . . . . . . . . . #

# . . . . . . . . . . . . . #

< . . . . . . . . . . . . . #

# . . . . . . . . . . . . . . #

# . . . . . . . . . . . . . . #

# # # # # # # # # # # # # # # #

## 8D A HALF-DOZEN FUNCTIONS WITH LOOPS

Write one C++ program where your main method is a test driver of your own design that tests six new free functions implemented in the same fi le. You may write your own tests with assert functions.

/\* \* A test driver like this may be used to test the functions.

\*

\* File name: TestRepetitionFunctions (on the book's website)

\*/ int main() { // Test rstNints assert(15 == rstNints(5)); assert(21 == rstNints(6));

1. int rstNints(int n)

Given an integer argument that represents the number of integers to sum, return the sum of the fi rst n integers. Use a for loop. Do not use a formula. Assume the argument is always positive.

rstNints(1) returns 1

rstNints(2) returns 3, which is 1+2

rstNints(5) returns 15, which is 1+2+3+4+5

1. int factorial(int n)

Return n factorial, which is written as n!. 5! = 5\*4\*3\*2\*1 or in general, n! = n\*(n-1)\*(n-2) …\*2\*1. Use a for loop.

factorial(1) returns 1 factorial(2) returns 2, which is 2 \* 1 factorial(4) returns 24, which is 4 \* 3 \* 2 \* 1

1. string reverseString(string arg)

Return a new string that has arg’s characters in reverse order.

reverseString("") returns "" reverseString("1") returns "1" reverseString("1234") returns "4321"

1. int charPairs(string str)

Return the number of times two consecutive characters occur in the given string.

charPairs("") returns 0 charPairs("H") returns 0 charPairs("aabbcc") returns 3 charPairs("!!!") returns 2 charPairs("mmmm") returns 3 charPairs("mmOmm") returns 2

1. int bonacci(int n)

Return the correct Fibonacci number for the given argument. precondition: n >= 0. (*Hint:* Keep track of two consecutive Fibonacci numbers.)

n bonacci(n) n bonacci(n)

* 1. 0 5 5
  2. 1 6 8
  3. 1 7 13
  4. 2 8 21
  5. 3 9 34

1. void replace(string & str, char oldC, char newC)

Modify the string argument associated with the parameter str so that all occurences of oldC are replaced with newC.

string arg = "bookkeeper"; replace(arg, 'e', 'X'); assert("bookkXXpXr" == arg);

## 8E MASTERMIND

In this project, you are going to implement a number guessing game known as Mastermind. Th is assignment will give you more experience with:

* Strings
* User input
* if statements while statements
* Testing
* Problem solving

To fi rst give you an overall feeling for the fi nished game, we fi rst present a dialog to show how the game will be played. Assuming your program has generated a “secret” number that has fi ve unique digits, the game should prompt the player to guess the number. Th e input is errorchecked only to ensure the user enters a string of length 5. Entering "what?" will not help at all, but it should be allowed and should count as a try at the secret number. If the user enters "123456" or "1234", notify the user they must enter 5 digits but do not count this as an attempt at the secret number.

Th e game rules insist that you give the player some feedback on guesses. Based on that feedback, the player makes more guesses. Guessing continues until the secret number is guessed or until the maximum number of tries (32) is reached. Here is one sample dialog that plays one game where the user determines the secret number is ***12345***.

Enter your 5 digit guess: ***11111***

Try number: 1

Digits found: 1

Correct position: 1

Enter your 5 digit guess: ***22222***

Try number: 2

Digits found: 1

Correct position: 1

Enter your 5 digit guess: ***99999***

Try number: 3

Digits found: 0

Correct position: 0

Enter your 5 digit guess: ***12333***

Try number: 4

Digits found: 3

Correct position: 3

Enter your 5 digit guess: ***12344***

Try number: 5 Digits found: 4

Correct position: 4

Enter your 5 digit guess: ***what?***

Try number: 6

Digits found: 0

Correct position: 0

Enter your 5 digit guess: ***123456*** '123456' must have a length of 5

Enter your 5 digit guess: ***54321***

Try number: 7

Digits found: 5

Correct position: 1

Enter your 5 digit guess: ***12345***

Try number: 8

Digits found: 5

Correct position: 5 You won in 8 tries!

Before implementing the main function, you are asked to fi rst write four well-tested examples that will make writing the actual game itself much easier with far fewer bugs. Th ese three methods listed next should be tested well and have the same exact function headings. Sample asserts are included below each function heading to help explain the method’s use and behavior.

// Generates a 5-digit, valid secret "number" as a string.

// A secret number is valid if it contains no duplicates and

// all ve characters are digits '0'..'9' string generateSecretNumber()

// Return the number of digits that are contained in both the

// secret number and the guess. For example when secretNumber

// is 12345 and guess is 67821, the two numbers, actually strings,

// share two digits: 1 and 2. int uniqueDigitsFound(string secretNumber, String guess)

// Sample assertions from MasterMindTest.java (not a complete test) assert(5 == uniqueDigitsFound("12345", "21435")); assert(0 == m.uniqueDigitsFound("12345", "67890"));

// Returns the number of matching digits between the guess

// and the secret number. For example when secretNumber is // "12345" guess is "12675" returns 3 as the 1, 2, and 5 all

// have the same value at the same location.

int foundInPosition(String secretNumber, String guess)

// Sample assertions from MasterMindTest.java (not a complete test) assert(1, m.foundInPosition("12345", "99399")); assert(3, m.foundInPosition("12345", "19395"));

## 8F CLASS Elevator

Write the class defi nition and implement the member functions for class elevator with a constructor that places an elevator at a selected fl oor. Include a void select(int goToFloor) member function that allows fl oors to be selected. For every fl oor, the message going up or going down should be displayed before the current fl oor of the elevator. At that point, select should print "open at" the destination fl oor. Th e precondition is that the fl oor is selectable, or an int in the range of 1–100. Here is one sample output to give you an idea of what a simulated elevator will look like on your screen:

#include "Elevator.h" // For class Elevator int main() {

Elevator aLift(1); // Construct an Elevator object aLift aLift.select(5); aLift.select(3); return 0; }

### Output

start on oor 1 going up to 2 going up to 3 going up to 4 going up to 5 open at 5 going down to 4 going down to 3 open at 3

© 2018 Franklin, Beedle & Associates All rights reserved

**C H A P T E R N I N E**

# File Streams

## SUMMING UP

Th e major control structures have now been presented—sequence, selection, and repetition. In the previous chapter, two major repetitive patterns emerged. Th e Determinate Loop pattern is used when the number of repetitions can be determined in advance. Th e Indeterminate Loop pattern occurs so frequently that the while loop is part of almost every programming language.

## COMING UP

Chapter 9 presents two standard C++ classes—ifstream for obtaining input from an external fi le on a disk, and ofstream for saving program output onto a disk fi le. Processing input from a disk fi le is a classic instance of the Indeterminate Loop pattern. After studying this chapter, you will be able to

use ifstream objects for disk fi le input use ofstream objects for disk fi le output

### 9.1 ifstream OBJECTS

Because keyboard input is fairly common, inclusion of <iostream> was designed to make cin immediately available. Th e cin object is automatically initialized and associated with the keyboard. However, input may also be obtained from many other sources, such as a mouse, a disk fi le, or a graphics tablet. An ifstream object is needed to read data from a disk fi le.

Th e ifstream (input fi le stream) class is declared in fstream. Th erefore, this compiler directive must be added to programs intended to extract input from a disk fi le:

#include <fstream> // For the ifstream class

Th e ifstream class is similar to the istream class. For example, the familiar extraction operator >> is also used to input data from a fi le stored on a disk. Th e same rules that apply to keyboard input for ints, doubles, and strings also apply to input from a fi le.

An ifstream object is often constructed with the fi le name it will be associated with.

263

**General Form 9.1** *Initializing* ifstream *objects for existing fi les*

ifstream *object-name* (" *le-name*");

Th e le-name is the name of an existing disk fi le. If the fi le is not found, the ifstream object initialization fails and an attempt to use the object name for input will fail. Th e state of the ifstream object can be tested immediately to determine if the fi le was found.

In the next example, inFile is the object name and "input.data" is the associated operating system fi le name.

ifstream inFile("input.data"); // Construct an ifstream object

Now this code will read input from the fi le input.data rather than from the keyboard.

inFile >> intObject;

Th e following program uses an ifstream object to read three integers from a disk fi le. Notice that there are a few diff erences between programs that extract keyboard input and the one below:

* Before, programs used cin—an istream object—for keyboard input. Now inFile—an ifstream object—is used for fi le input.
* Whereas cin is automatically constructed, your program must construct an ifstream object with an existing disk fi le name associated with it.
* Prompts aren’t needed anymore. Th e same >> operator reads an integer and stores it into the int object, but there is no need to prompt the fi le for the next input.

// Include fstream for I/O streams dealing with disk les

#include <fstream> // For the ifstream class

#include <iostream> // For cout using namespace std;

int main() { int n1, n2, n3;

// Initialize an ifstream object so inFile is an input stream // associated with the operating system le named input.data ifstream inFile("input.data");

// Extract three integers from the le input.data inFile >> n1 >> n2 >> n3; cout << "n1: " << n1 << endl; cout << "n2: " << n2 << endl; cout << "n3: " << n3 << endl;

return 0;

}

9.1: ifstream Objects

Assuming the fi le input.data stores these three integers:

70 80 90 this output is generated:

#### Output

n1: 70 n2: 80 n3: 90

If the fi le input.data stores these three integers:

-45 77 23 this output is generated:

#### Output

n1: -45 n2: 77 n3: 23

File input works just like keyboard input—spaces and new lines separate the input data. Th is applies to all data seen so far: string, int, and double. If an integer is encountered in the fi le during an attempt to read a double, the int is promoted to a double. Th e one input diff erence is this: with an ifstream object, keyboard input is not necessary for stream extraction. Once the program begins to run, data can be read from the disk fi le without user input.

#### SELF-CHECK

9-1 Write a complete program that reads the fi rst 30 strings from an input fi le named student.data and displays each string to the screen. Remember to #include <fstream>.

##### 9.1.1 GETTING THE PATH RIGHT

If your input fi le is not stored in the current working directory, you may need to use an operating system path to locate it. For Windows, which uses \ to separate directory names, you need the escape sequence \\ (two backslashes) to specify full path names. So the fi le name may appear like this:

ifstream inFile( "c:\\mystuff\\input.data" );

“\\” represents only one backslash. Omitting one \ from \\ is virtually guaranteed to result in not fi nding the fi le:

ifstream inFile( "c:\mystuff\input.data" ); // Need \\, not \

Th is problem doesn’t exist in Unix because the / character is used to separate directories, and so / can be used “as is”:

ifstream inFile("myC++Stuff/input.data");

Also consider what happens if the fi le is not found. Input operations such as inFile >> will not execute. If you don’t seem to be extracting input from the fi le or the values appear to be garbage, chances are the fi le does not exist as specifi ed, it has a diff erent name, it’s in a diff erent directory, you used \ rather than \\ in DOS or Windows, or your disk is bad, or. . . .

You can use the following alternate selection action to ensure that the user is notifi ed that the fi le has not been found:

if( ! inFile ) {

// If true, the input le was not found. cout << "Failed to nd the le." << endl;

} else {

// Process le input data // . . .

}

### 9.2 THE INDETERMINATE LOOP PATTERN APPLIED TO DISK FILES

Th e previous chapter on repetition showed how sentinel loops process an undetermined number of keyboard inputs. Th e same type of logic works with the *end-of-fi le event*, which requires some knowledge of the operating system you are using. Th e end-of-fi le event is entered from the keyboard using the key sequence Ctrl-Z (^Z) with Windows or Ctrl-D (^D) in Unix.

When the end-of-fi le event is encountered on an input stream, the input statement (cin >>, for example) returns false (0, actually). So once again, the cin statement can be used as a loop test for processing an undetermined number of inputs.

while(cin >> x) { // Input value at start of each iteration

// Process value

}

Each time the cin statement returns true, the valid input is processed. When the user enters the end-of-fi le key sequence (Ctrl-C in DOS or Ctrl-D in Unix), the state of cin is altered to return false and the loop terminates.

Th e loop in the following program terminates when the user enters end of fi le. Th e loop test (cin >> x) returns false when end-of-fi le is detected.

9.2: The Indeterminate Loop Pattern Applied to Disk Files

#### Dialogue

Enter doubles, Ctrl-D, Ctrl-Z, or Command-Period to quit

1 3 4 ^D

Average: 2.66667

A word of warning: End-of-fi le sets the state of the input stream such that subsequent keyboard input is ignored unless some extra work is performed.

#### SELF-CHECK

9-2 What is the output of the preceding program if the user enters end-of-fi le fi rst?

##### 9.2.1 PROCESSING UNTIL END-OF-FILE

You can use the end-of-fi le event to process all data in a fi le without determining the amount of data in that fi le beforehand. Th is is shown in the next program where an indeterminate loop breaks the loop when there is no more data in inFile—the end of the input fi le was detected.

// Count how many numbers are in a disk le. The ifstream object // named is used as the input stream, not cin.

#include <fstream> // For the ifstream class

#include <iostream> using namespace std;

int main() {

ifstream inFile("numbers.data"); double x = 0.0; // Store le inputs here temporarily int n = 0;

if( ! inFile ) {

// If true, the input le was not found

cout << "Failed to nd the le numbers.data" << endl;

} else { cout << "The le was successfully constructed" << endl; while( inFile >> x ) { n++; // Track the number of loops

cout << "iteration #" << n << ": " << x << endl;

} cout << "End of le reached. " << n << " numbers found." << endl;

} return 0; }

To visualize this loop action, the repeated part simply displays each successfully extracted number. Th e output shown below appears when the fi le named input.data contains the following four numbers:

0.001 9

8.0

1.5

###### Output

The le was successfully constructed iteration #1: 0.001 iteration #2: 9 iteration #3: 8 iteration #4: 1.5

End of le reached. 4 numbers found.

#### SELF-CHECK

9-3 What is the output of the preceding program if:

1. the fi le numbers.data does not exist?
2. the fi le numbers.data contains one number?
3. the fi le numbers.data contains zero numbers (the fi le is empty)?

9-4 Write the output of the following program with the various data stored in the fi le input.data. (*Note:* inFile >> intObject will fail if an invalid number is encountered in the input fi le stream; input need not be on separate lines.)

1. 1 2 3 c. 1 2 3 BAD
2. 1 2 3 4 5 d. 1.5 2.6 3.7

#include <fstream> // For the ifstream class

#include <iostream> // For cout using namespace std; int main() {

ifstream inFile("input.data"); int sum = 0; int intObject; while(inFile >> intObject) { sum += intObject;

}

cout << sum << endl; return 0; }

##### 9.2.2 LETTING THE USER SELECT THE FILE NAME

It is sometimes appropriate to allow the user to enter the fi le name while the program is running. In this situation, it is appropriate to read the fi le name as a string. However, the string object itself cannot be used to initialize an ifstream object.

string leName; cout << "Enter le name: "; cin >> leName;

ifstream inFile( leName);

// ERROR: ifstream::ifstream(string) not found

Th e ifstream constructor needs the character portion of a string, which is returned with string::c\_str. Th is message returns the characters of the string object.

ifstream inFile( leName.c\_str() );

### 9.3 INDETERMINATE LOOP WITH MORE COMPLEX DISK FILE INPUT

Th e Indeterminate Loop pattern is often used to process data stored in a fi le—and that data can be quite complex. To accomplish this, the programmer must know the format of that data or must be able to specify their format. Th is is possible even if there is a collection of input data of diff erent types and those data are spread out over two or more lines.

Th e example of this section uses an input fi le where all data concerning one employee is stored on one line in the fi le. Th e algorithm works like this: input one line of data and process it until there is no more data. Th e termination condition is end of fi le. So the loop test would be:

while (there is data in the input stream) process the newly read data

An indeterminate loop is capable of processing an unspecifi ed number of inputs with data that need not be entered from the keyboard. With the end-of-fi le event as the termination condition, the number of iterations depends on the size of the fi le. Th e loop is easily written to eff ectively process all the employee data in a fi le whether there are zero, one, two, or many employees. For example, if the fi le employee.data contains the following data,

12.00 1 S Milan Archer

12.44 2 M Lennon Arrowsmith

11.11 3 M Oakley Baxter

10.00 0 S Charlie Bond a properly constructed loop should process exactly four employees. Th e same code should also work with fi les of diff erent sizes (diff erent numbers of employees). Th is is an advantage over determinate loops that require the number of iterations to be determined before the loop begins to execute.

Th e next program implements a loop that uses the end-of-fi le event as the termination condition. During the loop test, all items needed to construct one Employee object (class Employee) are read from the ifstream object referenced by inFile.

while(inFile >> hourlyRate >> exemptions >> maritalStatus >> rstName >> lastName) {

// process the data

}

If there are enough data (of the proper format), the while loop executes the repeated part.

Once inside this block, a new Employee object is constructed with the fi le input data just read in.

Employee anEmp(name, hourlyRate, maritalStatus, exemptions);

For each employee on fi le, the getGrossPay() message is sent to each Employee after setting the hours worked for the week.

// This program reads data from an input le to construct Employee objects, // set the hours worked for the week, and show the gross pay for each.

#include <iostream>

#include <fstream> // For the ifstream class using namespace std; #include "Employee.h" // For the Employee class

int main() { string rstName, lastName; double hourlyRate, hoursThisWeek; int exemptions; string maritalStatus;

// Initialize an input stream with a disk le as the source ifstream inFile("employee.data"); if (!inFile) {

// Show error if the le "payroll.data" is not found on the disk cout << "\*\*Error opening le 'employee.data'" << endl;

} else {

// Process data until end of le while (inFile >> hourlyRate >> exemptions >> maritalStatus

>> rstName >> lastName) { string name (lastName + ", " + rstName); cout << "Hours worked for " << name << "? "; cin >> hoursThisWeek;

Employee anEmp(name, hourlyRate, maritalStatus, exemptions); anEmp.setHoursWorked(hoursThisWeek); // Print the gross pay in a minimum of 3 spaces with 2 decimals places

// with a preceding $ and a new line '\n' after the gross pay.

printf("$%3.2f \n", anEmp.getGrossPay());

}

} return 0; }

#### Output

Hours worked for Archer, Milan? ***40***

$480.00

Hours worked for Arrowsmith, Lennon? ***30***

373.20

Hours worked for Baxter, Oakley? ***0*** $0.00

Hours worked for Bond, Charlie? ***42***

$430.00

Notice that the output shows exactly four employees. Had the disk fi le contained a diff erent number of employees, a diff erent-sized report would have been generated without any change to the program or the need to determine the number of employees beforehand. Th is is a good time to use an indeterminate loop.

#### SELF-CHECK

9-5 Describe what would happen if the S were omitted from the last line in the fi le used for input in the preceding program:

12.00 1 S Milan Archer

12.44 2 M Lennon Arrowsmith

11.11 3 M Oakley Baxter

10.00 0 Charlie Bond

9-6 Describe what would happen if the 0 were omitted from the last line in the fi le used for input in the preceding program:

12.00 1 S Milan Archer

12.44 2 M Lennon Arrowsmith

11.11 3 M Oakley Baxter

10.00 S Charlie Bond

##### 9.3.1 MIXING NUMBERS AND STRINGS

Th e preceding self-checks point to a problem that occurs when input contains numbers, characters, and strings. If one line of input is incorrect, the program will likely fail or produce incorrect output. Consider the following incorrect input:

12.00 S 1 Milan Archer

When that input is read and executes:

while(inFile >> hourlyRate >> exemptions >> maritalStatus >> rstName >> lastName)

Th e fi rst time through the loop, the S is encountered while attempting to read an integer for exemptions. Th e input stream fails. Th e loop terminates. No objects are constructed inside the loop. Th is results from a fi le with just one out-of-place piece of data. If you are having problems reading data from a fi le, make sure the input statement has the proper objects and that the input fi le has the correct data.

##### 9.3.2 THE getline FUNCTION

Th e previous example works because the program assumed there were two strings at the end of each line in the fi le. And just as importantly, the fi le had exactly two strings at the end of every line. But consider what would happen if the program could not assume there were going to be exactly two strings. For example, what if some employees had a middle initial, some had none, and others had two middle names for a total of four distinct strings in their names?

12.00 1 S Milan J. Archer

12.44 2 M Lennon Arrowsmith

11.11 3 M Oakley S. T. Baxter

10.00 0 S Charlie Bond

Th e previous program read a fi le in which each line ended with a fi rst name followed by a last name. An alternative approach would now be required to read the string input at the end of each line in the fi le above. Th is can be accomplished with a function named getline from the string library.

Here is a simplifi ed function heading for the getline function. Notice that two parameters have &, so they modify the arguments in the caller.

istream & getline(istream & is, string & str, char sentinel = '\n') // post: Extracts string input from is (with blanks) until the end

// of line has been encountered

Th is comes in handy for reading things like names and addresses. Th e nonmember getline function extracts all the characters from the input fi le stream until the end of fi le is encountered or the new-line character '\n' is found. Th is means that blank spaces normally used to separate strings become part of one larger string value.

Th e fi rst argument to getline is any input fi le stream—cin or inFile, for example. Th e second argument is any string object that will be modifi ed by getline. Th e string object will store all the characters from the current input stream until end of line. Th e third argument is optional.

If omitted, the end-of-string marker is the new-line character '\n'.

Th is is the fi rst example of a *default argument*. With the assignment of '\n' to sentinel in the parameter list, the getline function can be called with only two arguments. In this case, the third parameter is automatically assigned the value of the expression to the right of =. Th is is called a default argument. Th erefore, the following two calls to getline are equivalent:

string fullName;

getline(inFile, fullName, '\n'); getline(inFile, fullName);

On the other hand, you can specify the third argument to be any sentinel character you wish. So to read an entire sentence from the keyboard, use this:

string sentence;

cout << "Enter a sentence ended with a period <'.'>: " << endl; getline(cin, sentence, '.');

// assert: sentence has all characters up to, but not including // '.'. The '.' is pulled out of input stream (discarded).

Th e getline function also returns a reference to the input stream. Th e return value is true unless the end of fi le or the sentinel is found. Th is means getline can be used as a loop test. Th e following program demonstrates how getline can be used to read all the lines in any input fi le. Th e input is the program itself, so the number of lines should be 17.

#include <iostream> // 1 File name: getline.cpp

#include <fstream> // 2 #include <string> // 3 using namespace std; // 4

// 5 int main() { // 6 string aLine; // 7 ifstream inFile("getline.cpp"); int lineCount = 0; // 9 // 10 while(getline(inFile, aLine)) { lineCount++; // 12 } // 13 // 14

cout << "Lines in getline.cpp: " << lineCount << endl; return 0; // 16 } // 17

###### Output

Lines in getline.cpp: 17

#### SELF-CHECK

9-7 What is the value of street when the user enters each line at the prompt?

a. 1313 Mockingbird Lane. b. 1214 Chestnut Drive.

#include <iostream> // For cout

#include <string> // For getline and string using namespace std; int main() { string street;

cout << "Enter street address, end with a period <.> " << endl; getline(cin, street, '.'); cout << street; return 0; }

Getting back to the problem of reading names that may have one, two, three, or any number of spaces, the while loop for the payroll problem could now be replaced by this to allow for any number of names.

string fullName;

// Process data until end of le

while (inFile >> hourlyRate >> exemptions >> maritalStatus

&& (getline(inFile, fullName))) { // Extract rst blank character in fullName fullName = fullName.substr(1, fullName.length() - 1); cout << "Hours worked for " << fullName << "? "; cin >> hoursThisWeek;

Employee anEmp(fullName, hourlyRate, maritalStatus, exemptions); anEmp.setHoursWorked(hoursThisWeek);

// Print the gross pay in a minimum of 3 spaces with 2 decimals places

// with a $ and a new line '\n' after the gross pay.

printf("$%3.2f \n", anEmp.getGrossPay()); }

### 9.4 OFSTREAM OBJECTS

Th is section introduces class ofstream (output fi le stream) for storing program output to more permanent disk fi les. Th e ofstream class is a specialization of the ostream class, just as ifstream is a specialization of the istream class. Th erefore, the operations and messages that could be sent to cout can also be sent to ofstream objects.

#include <iostream> // For cout

#include <fstream> // For the ofstream class using namespace std; int main() {

ofstream outFile("out.data");

outFile << "This string goes to a disk, not the screen" << endl; double x = 1.23; outFile << x << endl; outFile.width(30); outFile << x << endl;

cout << "This string goes to the screen" << endl; return 0; }

#### Output (to the fi le associated with the object named outFile)

This string goes to a disk le, not the screen

1.23

1.23

1.23

Chapter Summary

#### Output (to the screen)

This string goes to the screen

#### SELF-CHECK

9-8 What output goes to the disk fi le named out.data?

ofstream out("out.data"); for(int j = 1; j <= 5; j++) cout << j << " ";

### CHAPTER SUMMARY

* An ifstream object may be associated with a disk fi le so large that amounts of data may be input quickly—with no human intervention.
* Th e ! operator is overloaded to determine if a fi le has not been properly opened for input.
* Use the input operator >> in the loop test to read input until the end of any fi le of any size.
* You can use ofstream objects like cout. Th e only diff erence is that the output goes to a disk fi le rather than the computer screen.

### EXERCISES

1. What does ifstream stand for?
2. Write the code that declares an input stream named inFile associated with the fi le called numbers.data located in the current working folder (directory).
3. Which #include is needed to construct ifstream objects?
4. Write a complete C++ program with the correct #includes with a loop that counts the number of words contained in a fi le. A word is any collection of characters separated by spaces, tabs, or new lines. For example, there are 14 words in the following sentence (recall that string constants are separated by blanks, tabs, and new lines):

Here's one word, another, and another. There are a total of 14 words here.

1. Write a sentinel event-controlled loop that counts the number of perfect test scores (the number 100) in a fi le named tests.data.

### PROGRAMMING TIPS

1. Use getline to read strings with blank spaces. Sometimes several strings represent one string input. When asking for someone’s name or address and you don’t know how many values will be input, use the getline function.

string address;

cout << "Enter your address: "; getline(cin, address); cout << "Address: " << address << endl;

#### Dialogue

Enter your address: ***1313 Mockingbird Lane, Washington D.C.***

Address: 1313 Mockingbird Lane, Washington D.C.

1. Be careful when using getline and >> together. Be careful when mixing getline with the >> operator on the same input stream. Th e >> operator skips whitespace; getline does not. Worse yet, cin >> will stop at the new line. A subsequent getline will go up to the new line, eff ectively reading nothing. In this case, you will need an extra getline to get beyond the end of the line.
2. Use test drivers for reading complex data. Seemingly bizarre things can occur when you try to input complex data in an end-of-fi le loop. Consider fi rst writing a test driver with code that inputs the fi rst line from the fi le and then displays it.
3. Input is messy. Using istream >> and getline on the same input stream can cause diffi cultto-detect errors. Additionally, when there is a mix of integer, fl oating-point, character, and string input, it is not always easy to get the input statements correct. Th e number of objects in an input statement must always be correct. Th e input fi le must always be correct.

**PROGRAMMING PROJECTS**

## 9A WIND SPEEDS ON FILE

Write a program that determines the lowest, highest, and average of a set of wind speed readings from a fi le. Th e number of readings is not known in advance. First create a fi le in your working

(current) directory as wind.data and use the ifstream constructor to open the fi le for input as follows:

ifstream inFile("wind.data");

Programming Projects

Th e program should work for all fi les containing only ints so any number of inputs should produce correct results. Run your program with the following fi le named wind.dat. Verify that the output is correct by producing results by hand and comparing your output.

2 6 1 2 5

5 4 3 12 16

10 11 12 13 14

## 9B WORDS IN A FILE

Write a C++ program that approximates the number of words in a fi le that has the fi le name input by the user. Remember to use string::c\_str to initialize the ifstream object.

cin >> leName;

ifstream inFile( leName.c\_str());

## 9C PAYROLL REPORT (PREREQUISITE 7D: CLASS EMPLOYEE)

In this project you are asked to use your Employee class as the basis for a payroll program that processes many employees. Th e input data to be processed are stored in an external fi le with the following format:

Sam Barker 40.0 15.00 2 S

Casey Baker 42.0 12.00 3 M

Joey Cook 30.5 9.99 1 S

Chris Glazer 40.0 11.57 1 M

Create a report in a new fi le named payroll.report that looks like the following (with ? replaced by the correct answers, of course). Also show all totals for every category except the pay rate (*Note:* Income tax is based in the 2015 Employer Tax Guide; this changes yearly).

### Output fi le named payroll.report

Pay Hours Gross Income SocSec Medi Net Employee

Rate Worked Pay Tax Tax care Pay Name

===== ==== ======= ====== ====== ====== ======== =======

15.00 40.0 600.00 51.43 37.20 8.70 502.67 Barker, Sam

12.00 42.0 ? ? ? ? ? Baker, Casey

9.99 30.5 ? ? ? ? ? Cook, Joey

11.57 40.0 ? ? ? ? ? Glazer, Chris ------ --------- ------ ------ ------- -------- Totals 152.5 ??????.?? ????.?? ????.?? ????.?? ????.??

© 2018 Franklin, Beedle & Associates All rights reserved

**C H A P T E R T E N**

# Vectors

## SUMMING UP

Almost all objects studied so far either store one element of a specifi c value such as double and int or are made up of two or more possibly dissimilar elements such as Employee and

BankAccount.

## COMING UP

To get interesting things done, we often use a collection of data. For example, we might need a list of students, a list of phone contacts, a list of text threads, a list of prices at diff erent online stores, and so on. Th e ability to store many elements as one object is used to solve a wide variety of programming problems. In this chapter, we begin with one of the simplest, and arguably the most useful, way of storing data: the C++ vector type. After studying this chapter you will be able to

* construct and use vector objects that store collections of any type
* implement algorithms to process a collection of objects
* use the sequential search algorithm to locate a specifi c element in a vector
* pass vector objects to functions
* sort vector elements into ascending or descending order understand how to search with the classic binary search algorithm

### 10.1 THE STANDARD C++ vector CLASS

Th e vector class constructs objects that store *collections* of objects. All vector objects are considered *homogeneous* because the objects in the collection are of the same type—a collection of numbers or a collection of string objects, for example. Th e objects in the collection may be any one of the standard types such as int, double, or string. Additionally, any programmer-defi ned class that has a default constructor can be contained in a vector. You can have a collection of any objects that you can dream up. Here are two general forms for initializing vector objects:

279

#### General Form 10.1 vector *initialization*

vector <*type*> *vector-name*(*capacity*);

- or vector <*type*> *vector-name*(*capacity*, *initial-value*);

* *type* specifi es the class of objects stored in the vector.
* *vector-name* is any valid C++ identifi er.
* *capacity* is an integer expression representing the maximum number of elements that can be stored into the vector.
* Th e optional *initial-value* is the value that will be assigned to every element in the vector. If there is only one argument (*capacity*), then the default constructor for that class sets the initial values (recall that with double and int, the default values will be garbage).

Examples of vector Initializations

vector <int> garbage(1000000); // A million integers of unknown value vector <double> x(100, 99.9); // Store 100 numbers, all equal to 99.9 vector <string> names(20, "TBA"); // Store 20 strings, all equal to "TBA"

To use vector, include <vector>, and when using namespace std; you can write vector instead of std::vector.

#include <vector> // For the vector<type> class using namespace std;

It should be noted that the vector syntax and algorithms that follow apply to primitive C++ arrays declared, such as int garbage[100] and string names[20]. Th e main benefi ts of using vectors include:

* vector objects check for invalid indexes such as accessing the element at index -1.
* vector has several useful member functions like resize(200).
* vector objects can have all elements initialized when constructed, while primitive C++ arrays require an additional for loop to do the same thing.

#### 10.1.1 ACCESSING INDIVIDUAL ELEMENTS IN A COLLECTION

Any vector object supports access to any element by using an index into the vector. An individual vector element can be referenced directly through subscripts that are written with square brackets [ and ].

**General Form 10.2** *Accessing one* vector *element*

*vector-name* [*integer-expression*]

Th e subscript range of a C++ vector is an integer value in the range of 0 through its capacity - 1. Th erefore, the individual objects of x declared as vector <double> x(8, 0.0);

may be referenced using the integer subscripts 0, 1, 2, 3, . . . 7, but not 8. Values can be stored into the fi rst two vector elements of x with these two assignment statements:

// Assign new values to the rst two elements of vector named x x[0] = 2.6; x[1] = 5.7;

Because C++ has zero-based indexing, the fi rst vector element is referenced with subscript 0 or as x[0], and the fi fth element with subscript 4 or x[4]. Th is subscript notation allows individual vector elements to be displayed, used in expressions, and modifi ed with assignment and input operations. In fact, you can do anything to an individual vector element that can be done to an object of the same class.

Th e familiar assignment rules apply to vector elements. For example, a string constant cannot be assigned to a double, and a string constant cannot be stored in a vector element declared to store int values.

x[2] = "Wrong type of constant"; // ERROR: x stores numbers

Since any two doubles can be added with +, subscripted vector elements can also be used in arithmetic expressions like this:

x[2] = x[0] + x[1]; // Store 8.3

Keyboard input can also be used to set the state of vector elements like this:

cout << "Enter two numbers: "; cin >> x[3] >> x[4];

##### Dialogue

Enter two numbers: ***9.9 5.1***

After this user input of ***9.9*** and ***5.1*** into the fourth and fi fth vector elements and the previous assignments to the fi rst three vector elements, the state of x now looks like the following.

capacity ()

x[0] x[1] x[2] x[3] x[4] x[5] x[6] x[7]

x:

8

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2.6 | 5.7 | 8.3 | 9.9 | 5.1 | 0.0 | 0.0 | 0.0 |

elements of x:

#### 10.1.2 vectorPROCESSING WITH DETERMINATE forLOOPS

Programmers must frequently reference many consecutive vector elements. Th e simplest case might be to display all the meaningful elements of a vector. Th e C++ for loop provides a convenient way to do this. Th is program includes the same vector assignments as above, with a for loop added at the end to display the fi rst n = 5 elements. Notice that x[5], x[6], and x[7] still have the initial value of 0.0.

#include <iostream>

#include <vector> // For the vector<type> class using namespace std;

int main() { vector<double> x(8, 0.0);

// Assign new values to the rst two elements of vector named x x[0] = 2.6; x[1] = 5.7;

x[2] = x[0] + x[1]; // Store 8.3

cout << "Enter two numbers: "; cin >> x[3] >> x[4];

int n = 5; // assert: n represents the number of meaningful elements

// Display the meaningful elements of x--the rst n elements cout << "\nThe rst " << n << " elements of x: " << endl; for (int index = 0; index < n; index++) { cout << "x[" << index << "]: "; cout << x[index] << endl; }

return 0; }

##### Dialogue

Enter two numbers: ***9.9 5.1***

The rst 5 elements of x:

x[0]: 2.6 x[1]: 5.7 x[2]: 8.3 x[3]: 9.9 x[4]: 5.1

Th e fi rst n elements of x are easily referenced by altering the int named index, which acts both as the counter in the for loop and as the subscript inside the for loop (x[index]). With index serving both roles—as shown in the code above—the specifi c vector element referenced as x[index] will depend on the value of index. For example, when index is 0, x[index] is a reference to the fi rst element in x; when index is 4, x[index] is a reference to the fi fth element of x.

#### 10.1.3 PROCESSING THE FIRST nELEMENTS IN A vector

Here is another example of a for loop that compares the fi rst n vector elements to fi nd the largest fl oating-point value using the vector x in the preceding program.

// First set the largest as the rst element . . . double largest = x[0];

// . . . then compare all other vector elements x[1] through x[n-1] for(int i = 1; i < n; i++) { if (x[i] > largest) largest = x[i]; }

// Display the largest cout << "The largest element in vector x = " << largest;

##### Output

The largest element in vector x = 9.9

A vector often stores fewer meaningful elements than its capacity. Th erefore, you usually need to have an object that stores the number of elements in the vector that are currently under consideration. In the previous code, n was used to limit the elements being referenced. Only the fi rst fi ve elements were searched to fi nd the largest. Imagine trying to fi nd the largest number in x without limiting the search to the fi rst n elements. Th e largest could incorrectly be some garbage value at index 5, 6, or 7.

Th e Determinate Loop pattern with for loops conveniently performs vector processing, which is the inspection of, reference to, or modifi cation of a selected number of vector elements. Th e number of elements (n here) is the predetermined number of vector elements that must be processed. Algorithms that include vector processing in this chapter include

* displaying some or all elements of a vector
* fi nding the sum, average, or largest of all vector elements searching for a particular object in the vector
* arranging elements in a certain order (ordering elements from largest to smallest or alphabet-izing a vector of string objects from smallest to largest)

#### 10.1.4 OUT-OF-RANGE SUBSCRIPT CHECKING

Th e standard vector class does not check subscripts to ensure that they are within the proper range of 0 through its capacity 1. Th erefore, the programmer must be careful to avoid subscripts that are not in the range specifi ed at initialization. If you are using the standard vector class without subscript range checking, the following assignments may destroy some other portion of memory, such as another object’s state:

x[-2] = 4.5; // Careful! These out-of-range subscripts are not x[8] = 7.8; // guarded against and could crash your computer.

Th e result could be seemingly unrelated errors, bugs, or even a system crash. All subscripts should be in the range of 0 through the vector’s capacity 1.

Without range checking, an out-of-range subscript destroys other areas of memory. Th is creates diffi cult-to-detect bugs. More dramatically, your computer may “hang” or crash. Even worse, with a workstation that runs all the time, you may get a latency error that aff ects computer memory now, but won’t crash the system perhaps for weeks.

As an example of a problem an out-of-range subscript may create, consider what might happen with the following assignment:

result = x[n];

Th e value stored at x[n] is one beyond the vector’s capacity. It is some random garbage value.

On one system, this statement produced the output shown in the comment:

// There is no warning or error with the statement cout << "x[n]: " << x[n] << endl;

##### Output

x[n]: -33686019

Th e standard vector class provides the at member function to avoid out-of-range subscripts. Th e result may look diff erent, but this message will gracefully terminate the program rather than store some random value into result:

result = x.at(n); // Gracefully terminates the program. Good.

To ensure your program will not run with an out-of-range index, use the at(int) message. Your program will then terminate early with an error message indicating the reason. Th is is preferable to fi xing errors that are diffi cult to locate. Here is what will happen with the vector class when using the at member function:

#include <vector> // For the vector<type> class

#include <iostream> using namespace std;

int main() { vector<double> x(8);

// Attempt to assign 100 to all elements of vector named x

for (int i = 1; i <= x.capacity(); i++) {

x.at(i) = 100; } cout << "Program would terminated above with x[8]" << endl;

return 0; }

##### Output for a program that terminated early (this will vary among diff erent systems)

libc++abi.dylib: terminating with uncaught exception of type std::out\_of\_range: vector

Th ere may be a temptation to always use vector::at in subsequent examples. However, programmers have been using subscripts for a very long time. You will see a lot of code with the square brackets ([ ]), so this textbook will rely on subscripts. Feel free to use at messages when developing your own programs.

##### 10.1.5 vector::capacity, vector::resize, =

Many messages can be sent to a standard C++ vector object. Each vector object is responsible for knowing how many objects it can store—its capacity. A vector also knows how to increase or decrease that capacity—a vector can resize itself.

After a vector has been initialized, the vector::capacity message returns the maximum number of elements that the vector can hold. Th e vector::resize message tells the vector to change to the new capacity supplied as the single argument. What is weird, however, is the capacity message returns a larger capacity when the argument is less than the capacity. In the example below, v2’s capacity still shows 100 even though at(55) terminates the program with the error message shown:

// Demonstrate capacity and resize

#include <vector> // For the standard vector<type> class

#include <iostream> using namespace std;

int main() {

vector <int> v1; // v1 cannot store any elements with 0 capacity vector <int> v2(100, -1);

cout << "v1 can store " << v1.capacity() << endl; cout << "v2 can store " << v2.capacity() << endl;

v1.resize(22);

cout << "v1 can now store " << v1.capacity() << endl;

// Odd behavior when the argument is less than the current capacity.

// at(55) shows you can not access past the smaller capacity. v2.at(55) = 123;

cout << "v2.at(55): " << v2.at(55) << endl; v2.resize(55);

cout << "v2 can now store " << v2.capacity() << endl; cout << "v2 has this -1s: " << v2.size() << endl; cout << "v2.at(55): " << v2.at(55) << endl;

return 0; }

###### Output

v1 can store 0 v2 can store 100 v1 can now store 22 v2.at(55): 123 v2 can now store 100 v2 this many meaningless -1s: 55

v2.at(55): libc++abi.dylib: terminating with uncaught exception of type std::out\_of\_range: vector

If you resize a vector to have more capacity, the original elements in the lower subscripts are still there. However, if you resize a vector to be smaller, the elements in the higher locations are lost. Truncation occurs.

One vector can be assigned to another. Th e vector to the left of the = operator becomes an exact copy of the vector to the right of =. Th e vector on the left, like any other object to the left of =, is destroyed.

// Demonstrate capacity and resize

#include <iostream>

#include <vector> // For the vector<type> class using namespace std;

int main() { vector <int> v1(3, -999); vector <int> v2;

v2 = v1;

// assert: v2 now stores 3 elements == -999 for(int index = 0; index < v2.capacity(); index++) { cout.width(5); cout << v2[index]; }

return 0;

}

###### Output

-999 -999 -999

#### SELF-CHECK

Use this initialization to answer the questions that follow:

vector <int> x(100, 0);

10-1 How many integers can be stored in x?

10-2 Which integer subscript references the fi rst element in x?

10-3 Which integer subscript references the last element in x?

10-4 What is the value of x[23]?

10-5 Write the code that stores 78 into the fi rst element of x.

10-6 Write code that stores 1 into x[99], 2 into x[98], 3 into x[97], . . ., 99 into x[1], and 100 into x[0]. Use a for loop.

10-7 Write code that displays all elements of x on separate lines. Use a for loop.

10-8 What happens when this code executes: x[-1] = 100; 10-9 Name two vector member functions.

10-10 Write the output generated by the following program:

#include <vector> // For the standard vector<type> class

#include <iostream> using namespace std;

int main() { int n = 5; vector <int> x(n, 0); for(int i = 0; i < n; i++) { x[i] = i; }

x.resize(2 \* n);

// Show the rst ve elements are still in x for(int i = 0; i < n; i++) { cout.width(5); cout << x[i];

} cout << endl;

for(int i = 0; i < x.capacity(); i++) { cout.width(5);

cout << x[i];

} cout << endl;

return 0; }

### 10.2 SEQUENTIAL SEARCH

One of the major reasons for using vector objects is to have individual elements retained in the computer’s fast memory, where they will be frequently accessed. Th is often means searching for the existence of some element in the collection. So another common vector-processing operation involves searching. Searching examples include, but are certainly not limited to, searching for a student name in the registrar’s database, looking up the price of an item in an inventory, or obtaining information about a bank account. One such algorithm used to look up a vector element is called *sequential search*.

Th e sequential search algorithm attempts to locate a given element by comparing the item being sought with every object in the vector. Th e algorithm searches in a one-after-the-other (sequential) fashion. Sequential search continues as long as the search value has not been found or until there are no more elements left in the vector to compare.

Th is sequential search algorithm is presented here within the context of a vector of string objects. Although the search element here will be a person’s name, the vector being searched could contain other kinds of objects—numbers, students, or employees, for example—as long as the object of the class can be compared with the == operator.

// Initialize and show the rst n elements of vector named name

#include <iostream>

#include <string>

#include <vector> // For the standard vector<type> class using namespace std;

// This free function uses the sequential search algorithm to return

// the index of searchName in the vector or -1 if searchName is not found.

int indexOf(string searchName, const vector<string> & names, int n) {

// Just show the vector elements for now for (int i = 0; i < n; i++) { if (searchName == names[i]) return i;

}

// searchName not found return -1; }

int main() {

vector<string> myFriends(10); int n = 5; // Set the number of meaningful elements to be searched

myFriends[0] = "Sage"; myFriends[1] = "Harley"; myFriends[2] = "Peyton"; myFriends[3] = "Quinn"; myFriends[4] = "Taylor";

cout << "Sage is at index " << indexOf("Sage", myFriends, n) << endl; cout << "Peyton is at index " << indexOf("Peyton", myFriends, n) << endl; cout << "Taylor is at index " << indexOf("Taylor", myFriends, n) << endl;

if(indexOf("Not Here", myFriends, n) == -1) { cout << "Not Here was not found" << endl; }

return 0; }

#### Output

Sage is at index 0

Peyton is at index 2

Taylor is at index 4

Not Here was not found

#### SELF-CHECK

10-11 What value is returned if searchName is not in the vector referenced by names?

10-12 How many comparisons (iterations of the search loop) are necessary when searchName matches myFriends[0]?

10-13 How many comparisons (iterations of the search loop) are necessary when searchName matches myFriends[n-1]?

10-14 How many comparisons are necessary when searchName matches myFriends[3]?

10-15 How many comparisons are necessary when searchName isn’t in myFriends?

10-16 How many sequential search comparisons occur when the vector has no useful data in it—that is, when n == 0?

### 10.3 MESSAGES TO INDIVIDUAL OBJECTS IN A vector

Subscript notation is used to send messages to individual elements. Th e vector name must be accompanied by a *subscript* to specify the particular vector element to which the message is sent. Th e subscript distinguishes the specifi c object to which the operation is to be applied. For example, the length of myFriends[0] "Sage" is referenced with this expression:

myFriends [0].length(); // The length of the rst name in the vector

Th e expression myFriends.length() would be an error because this would represent an attempt to fi nd the length of the entire vector. Th e length function is defi ned for string, but not for the vector class (although vector::resize and vector::capacity are defi ned).

Now consider determining the total assets of all BankAccount objects in a vector of

BankAccounts. Th e following program fi rst sets up a tiny database of four BankAccount objects.

Th erefore, this statement account[0] = BankAccount ("Baker", 0.00);

fi rst constructs a BankAccount object with the name "Baker" and a balance of 0.00. Th e BankAccount object is then assigned to the fi rst vector element account[0].

// Illustrates a vector of programmer-de ned objects

#include <iostream>

#include <vector> // For the vector<type> class using namespace std; #include "BankAccount.h" // For the BankAccount class

int main() {

vector<BankAccount> account(100);

// Initialize the rst n elements of account int n = 4;

account[0] = BankAccount("Baker", 0.00); account[1] = BankAccount("Cook", 100.00); account[2] = BankAccount("Cartright", 200.00); account[3] = BankAccount("FensterMacher", 300.00); // assert: The rst n elements of account are initialized

double assets = 0.0;

// Accumulate balance of n BankAccount objects stored in account for (int i = 0; i < n; i++) { assets += account[i].getBalance(); } cout << "Assets: " << assets << endl;

return 0; }

#### Output

Assets: 600

#### SELF-CHECK

10-17 Write the output generated by the following program:

#include <iostream>

#include <vector> // For the vector<type> class

#include <string> // For the string class using namespace std;

int main() { vector<string> s(10);

// Initialize the rst 4 elements of account s[0] = "First"; s[1] = "Second"; s[2] = "Third"; s[3] = "Fourth"; int n = 4;

for (int i = 0; i < n; i++) {

cout << s[i].substr(1, s[i].length() - 2) << " "; }

return 0; }

##### 10.3.1 INITIALIZING A VECTOR OF OBJECTS WITH FILE INPUT

In some of the preceding programs, the vectors of objects were initialized in several assignment statements. vector objects can also be initialized through disk fi le input. To demonstrate, imagine the following is part of the input data fi le named bank.data with a total of 12 accounts on 12 lines:

Cust0 0.00

AnyName 111.11

Alex 222.22

Andy 333.33

Ash 444.44

Cust5 555.55

. . . fi ve lines are omitted . . .

Cust11 1111.11

If the vector is declared with a maximum capacity of 20 like this, then the fi rst BankAccount object can be stored in account[0]:

vector <BankAccount> account(20); // assert: account could store 20 default BankAccount objects So an object named numberOfAccounts starts at 0:

int numberOfAccounts = 0;

Th en the vector of BankAccount objects can be initialized one account at a time with these steps:

1. Input two items per line—a name and a balance.
2. Construct a BankAccount and store it into the next available vector location.
3. Increase the number of accounts by 1.

Th e vector::capacity function will also be used to safeguard against using subscripts beyond the account’s boundaries of 0 through 19.

Th e following while loop test expression should be true before a BankAccount object can be added at the next available location in the vector. If there are no more data in the fi le, (inFile >> name >> balance) is false and the loop will terminate. Also, if there are more data in the fi le but no more room in the vector, (numberOfAccounts < account.capacity()) is false and the loop terminates for a diff erent reason—there is no room.

while ((inFile >> name >> balance) &&

(numberOfAccounts < account.capacity())) { account[numberOfAccounts] = BankAccount(name, balance); numberOfAccounts++; }

While there is room for another element and there are more data in the fi le, the repeated part executes. Inside the loop, the two objects (name and balance) are passed on to the BankAccount constructor to construct a BankAccount, which is then stored in the next consecutive vector element. Th is initialization and assignment must occur before numberOfAccounts is incremented from 0 to 1 during the fi rst iteration of the loop.

Now numberOfAccounts accurately indicates the number of accounts processed so far, and the fi rst BankAccount object is stored into account[0]. During each loop iteration, numberOfAccounts represents not only the total number of meaningful accounts stored in the vector, but also the next available vector subscript into which the next BankAccount object can be stored. When the end of the fi le is encountered, numberOfAccounts will have the correct value—one greater than the subscript storing the last account.

Th is processing is shown in the context of a complete program which sets up a small database of bank customers:

// Initialize a vector of BankAccount objects through le input

#include <vector> // For the vector<type> class

#include <fstream> // For the ifstream class

#include <iostream> // For cout and endl

#include <string> // For the string class using namespace std; #include "BankAccount.h" // For the BankAccount class

int main() { string leName = "bank.data"; ifstream inFile( leName.c\_str());

if (!inFile) { cout << "\*\*Error\*\* " << leName << " was not found" << endl;

} else {

vector<BankAccount> account(20); string name; double balance = 0.0; int numberOfAccounts = 0;

while ((inFile >> name >> balance)

&& (numberOfAccounts < account.capacity())) { account[numberOfAccounts] = BankAccount(name, balance); numberOfAccounts++; }

cout << "Number of accounts on le: " << numberOfAccounts << endl; cout << endl;

cout << "The accounts" << endl; cout << "===========================" << endl; for (int index = 0; index < numberOfAccounts; index++) { cout.width(2); cout << index << ". "; cout << account[index].getName();

cout.width(20 - account[index].getName().length()); cout << account[index].getBalance() << endl;

}

} // end else

return 0; }

**Input File: bank.data**

**Output**

Number of accounts on le: 12

Cust0 0.00

The accounts

AnyName 111.11

===========================

Alex 222.22

1. Cust0 0

Andy 333.33

1. AnyName 111.11

Ash 444.44

1. Alex 222.22

Cust5 555.55

1. Andy 333.33

Cust6 666.66

1. Ash 444.44

Cust7 777.77

1. Cust5 555.55

Cust8 888.88

1. Cust6 666.66

Cust9 999.99

1. Cust7 777.77

Cust10 1010.10

1. Cust8 888.88

Cust11 1111.11

1. Cust9 999.9910. Cust10 1010.1

11. Cust11 1111.11

#### SELF-CHECK

10-18 Write two assignment statements that initialize two additional BankAccount objects with assignment statements in the next two vector locations. Use any data you desire.

10-19 What would happen if the input fi le bank.data contained 21 lines, each line representing one account? Remember, account.capacity() is 20.

10-20 Write code to initialize a vector of integers from a fi le named int.dat. Assume the fi le never has more than 1,000 integer values.

10-21 Which object in your code represents the number of initialized elements?

10-22 Write code that verifi es proper initialization of the vector of the previous two self-check questions.

### 10.4 vector ARGUMENT/PARAMETER ASSOCIATIONS

Sometimes it may be necessary to pass a vector to either a member function or a nonmember function through argument/parameter association. Th is requires a diff erent syntax in the parameter list. Th ere are three ways to declare a vector parameter, but only these two should ever be used:

#### Pass by Reference (when the function must modify the associated vector argument)

*return-type* *function-name* (vector <*class*> & *vector-name)*

#### Pass by const Reference (runtime effi cient with & and safe with const)

*return-type* *function-name* (const vector <*class*> & *vector-name*)

A vector object should not be passed by value. Th is parameter-passing mode is usually ineffi cient since vector objects can consume a large amount of memory.

void inef cient(vector <BankAccount> accounts, int n) {

// VALUE parameter (should not be used with vectors). All elements // of acct are copied after allocating additional memory.

}

Recall that passing by value causes the function to allocate memory for a copy of the object passed by value. Th is could be thousands or even millions of bytes. Th e program could terminate because of lack of memory. Additionally, every byte of the vector needs to be copied, which could noticeably slow down the program. Passing by const reference has the same meaning, but is more effi cient.

Use pass by reference (with &) when a function is supposed to modify the associated argument:

void initialize(vector <BankAccount> & accounts, int & n){

// REFERENCE parameter (allows changes to argument)

// Only a pointer to acct is copied // A change to acct here changes the argument in the caller

}

10.4: vector Argument/Parameter Associations

When a function requires a vector but should not modify the associated argument, pass the vector by const reference:

void display(const vector <BankAccount> & accounts, int & numberOfAccounts)

{

// CONST REFERENCE parameter (for ef ciency and safety)

// Only a reference to the acct is copied (4 bytes)

// A change to acct does NOT change the argument

}

Th e next program passes a vector by reference to the function initialize in order to communicate the initialized array back to main. Th e main function passes by reference a vector of doubles to a void function named initialize. Because the vector and int parameters x and numberOfAccounts are declared as a reference parameter with &, any change to x or numberOfAccounts inside of initialize also changes the arguments in the main function test and n.

#include <vector> // For the vector<type> class

#include <iostream> using namespace std;

void initialize(vector<int> & x, int & numberOfAccounts) { // Two reference parameters

// post: Initialize the rst n elements of the argument numberOfAccounts = 5;

x.resize(numberOfAccounts); x[0] = 75; x[1] = 88; x[2] = 67; x[3] = 92; x[4] = 51; // The arguments associated with x and n, test and n in main, // will also be modi ed.

}

void display(const vector<int> & x, int numberOfAccounts) { // Const reference

// Display the vector with n meaningful values cout << "The vector: ";

for (int i = 0; i < numberOfAccounts; i++) { cout.width(5); cout << x[i] << " ";

}

cout << endl; }

int main() { vector<int> vec(10, 0); int n;

// Initialize test and n initialize(vec, n); display(vec, n);

return 0; }

#### Output

The vector: 75 88 67 92 51

#### 10.4.1 const REFERENCE & PARAMETERS

Th e preceding program showed that the arguments—test and n—were passed to function initialize by reference. Th is was done to allow the function to modify both arguments and communicate the changes back to main. However, sometimes a vector is passed as input to a function, where no changes should be made. In this case, the const reference form should be used like in the initialize function above. Part of the reason is effi ciency—the program executes more quickly. Th e other consideration is better memory utilization—less memory is required to store the vector in the called function. A vector object passed by value requires as much memory as the argument.

// A vector should not be passed by value like this void display(vector <double> x, int n) { // Value parameter

// This function must obtain the memory necessary to store x when x

// could have a large capacity of large objects

}

So if the vector argument had a capacity of 100,000 elements, void display would need to consume an additional 100,000 elements. Additionally, every single element would need to be copied from the client code (the caller) to the called function. Th is can be time consuming, especially when the vector’s capacity is large and/or the size of each element is large. Th e computer has to do a lot of unnecessary work. Th e program would run noticeably slower and might exhaust available memory.

Here are two alternatives to make any program more effi cient in terms of space (saves memory) and time (runs faster):

1. Pass the vector by reference—effi cient but dangerous.
2. Pass the vector by const reference—effi cient and safe.

Th e second option is highly recommended—the computer program has much less work to do. Using const is also an antibugging technique that will let the compiler catch attempts to modify the constant objects. Any const member function may still be called— vector::capacity, for example. However, the compiler will fl ag any attempt to send a nonconst message:

// precondition: x.capacity() > 0

void display(const vector <int> & x, const int n) {

cout << "\nThe vector's capacity is " << x.capacity() // <- Okay cout << x[0]; // <- OKAY to reference vector element x[0] = 123; // <- ERROR caught during compilation

}

error: cannot assign to return value because function 'operator[]' returns a const value

Pass vector objects or any large object by const reference.

#### SELF-CHECK

10-23 Why should vector and Grid objects be passed by const reference when you have always seen int and double variables passed by value?

10-24 If the average size of the BankAccount objects in a vector of capacity 100,000 is 57 bytes, how many bytes of additional memory would have to be reserved and then copied into each of the following functions? Remember, pass by reference typically requires four bytes of memory:

1. void one(vector<BankAccount> v1)
2. void two(vector<BankAccount> & v1)
3. void one(const vector<BankAccount> & v1)

### 10.5 SORTING

Th e elements of a vector are often arranged into either ascending or descending order through a process known as *sorting*. For example, a vector of test scores is sorted into ascending order by rearranging the numeric values in lowest-to-highest order. A vector of string objects sorted in ascending order establishes an alphabetized list (A’s before B’s, B’s before C’s). To sort a vector, the elements must be compared with the < operator. If one object can be less than another object of the same type, then the vector is *sortable*. For example, 85 < 79 and "A" < "B" are valid expressions.

Th e following code declares and gives meaningful values to a part of the vector named data to demonstrate sorting a vector of integers:

vector<int> data(10, 0); // Store up to 10 integers int n = 5; data[0] = 76; data[1] = 74; data[2] = 100; data[3] = 62; data[4] = 89;

Th ere are many sorting algorithms. Even though others are more effi cient (run faster), the relatively simple selection sort algorithm is presented here. Th e goal here is to arrange a vector of integers into ascending order, the natural ordering of integers.

|  |  |  |
| --- | --- | --- |
| **Object Name** | **Unsorted vector** | **Sorted vector** |
| data[0] | 76.0 | 62.0 |
| data[1] | 91.0 | 76.0 |
| data[2] | 100.0 | 89.0 |
| data[3] | 62.0 | 91.0 |
| data[4] | 89.0 | 100.0 |

With the selection sort algorithm, the largest integer must end up in data[n - 1] (where n is the number of meaningful vector elements). Th e smallest number should end up in data[0]. In general, a vector x of size n is sorted in ascending order if x[i] <= x[i + 1] for i = 0 to n-2. Th e selection sort begins by locating the smallest element in the vector by searching from the fi rst element (data[0]) through the last (data[4]). Th e smallest element, data[3] in this vector, is then swapped with the top element, data[0]. Once this is done, the vector is sorted at least through the fi rst element.

Placing the Smallest Value in

the “Top” Position (

index 0

)

top == 0

**After**

**Sorted**

**Before**

data[0]

**76.0**

**62.0**

data[1]

91.0

91.0

data[2]

100.0

100.0

data[3]

**62.0**

**76.0**

data[4]

89.0

89.0

Th e task of fi nding the smallest element is accomplished by examining all vector elements and keeping track of the index with the smallest integer. After this, the smallest vector element is swapped with data[top] where top will range from 0 to n-1. Here is an algorithm that accomplishes these two tasks:

**Algorithm:** *Finding the smallest element in the vector and switching it with the topmost element*

1. *top* = 0

*// At rst, assume that the rst element is the smallest*

1. *indexOfSmallest* = *top*

*// Check the rest of the vector (data*[*top + 1*] *through data*[*n - 1*]*)*

1. for *index ranging from top* + 1 *through* n - 1 (c1) if *data*[*index*] < *data*[*indexOfSmallest*] *indexOfSmallest* = *index*

*// Place the smallest element into the rst position and place the rst vector // element into the location where the smallest vector element was located.*

1. swap *data*[*indexOfSmallest*] *with data*[*top*]

Th e following algorithm walkthrough shows how the vector is sorted through the fi rst element. Th e smallest integer in the vector will be stored at the "top" of the vectordata[0]. Notice that indexOfSmallest changes only when a vector element is found to be less than the one stored in data[indexOfSmallest]. Th is happens the fi rst and third times step (c1) executes.

**Step top indexOfSmallest index [0] [1] [2] [3] [4] n**

? ? ? 76.0 91.0 100.0 62.0 89.0 5

1. 0 " " " " " " " "
2. " 0 " " " " " " "
3. " " 1 " " " " " "

(c1) " 1 " " " " " " "

(c) " " 2 " " " " " "

(c1) " " " " " " " " "

(c) " " 3 " " " " " "

(c1) " 2 " " " " " " "

(c) " " 4 " " " " " "

(c1) " " " " " " " " "

1. " " 5 " " " " " "
2. " " " 62.0 " " 76.0 " "

Th is algorithm walkthrough shows indexOfSmallest changing twice to represent the index of the smallest integer in the vector. After traversing the entire vector, the smallest element is swapped with the top vector element. Specifi cally, the preceding algorithm swaps the values of the fi rst and fourth vector elements, so 62.0 is stored in data[0] and 76.0 is stored in data[3].

Th e vector is now sorted through the fi rst element!

Th e same algorithm can be used to place the second-smallest element into data[1]. Th e second traversal must begin at the new "top" of the vector—index 1 rather than 0. Th is is accomplished by incrementing top from 0 to 1. Now a second traversal of the vector begins at the second element rather than the fi rst. Th e smallest element in the unsorted portion of the vector is swapped with the second element. A second traversal of the vector ensures that the fi rst two elements are in order. In this example vector, data[3] is swapped with data[1] and the vector is sorted through the fi rst two elements:

#### top == 1 Before After Sorted

data[0]

62.0

62.0

data[1]

**91.0**

**76.0**

data[2]

100.0

100.0

data[3]

**76.0**

**91.0**

|  |  |  |
| --- | --- | --- |
| data[4] 89.0 | 89.0 |  |
| Th is process repeats a total of n-1 times: |  |  |
| top == 2 **Before** | **After** | **Sorted** |

data[2] **100.0** **89.0** data[3] 91.0 91.0 data[4] **89.0** **100.0**

|  |  |  |  |
| --- | --- | --- | --- |
| data[0] | 62.0 | 62.0 |  |
| data[1] | 76.0 | 76.0 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| An element may even be swapped with itself: | | | | |  |
| top == 3 **Before After** | | | | | **Sorted** |
|  | data[0] | 62.0 | 62.0 |  | |
|  | data[1] | 76.0 | 76.0 |  | |
|  | data[2] | 89.0 | 89.0 |  | |

data[3] **91.0** **91.0** data[4] 100.0 100.0

When top goes to data[4], the outer loop stops. Th e last element need not be compared to anything. It is unnecessary to fi nd the smallest element in a vector of size 1. Th is element in data[n - 1] must be the largest (or equal to the largest), since all of the elements preceding the last element are already sorted in ascending order:

|  |  |  |  |
| --- | --- | --- | --- |
| top == 3 and 4 | **Before** | **After** | **Sorted** |
| data[0] | 62.0 | 62.0 |  |
| data[1] | 76.0 | **76.0** |  |
| data[2] | 89.0 | 89.0 |  |
| data[3] | 91.0 | 91.0 |  |
| data[4] | 100.0 | 100.0 |  |

Th erefore, the outer loop changes the index top from 0 through n - 2. Th e loop to fi nd the smallest index in a portion of the vector is nested inside a loop that changes top from 0 through n - 2 inclusive.

**Algorithm:** *Selection Sort*

for *top ranging from 0 through n - 2* { *indexOfSmallest* = *top* for *index ranging from top + 1 through n - 1* {

if *data*[*indexOfSmallest*] < *data*[*index*] then

*indexOfSmallest* = *index*

}

swap *data*[*indexOfSmallest*] with *data*[*top*]

}

Here is the C++ code that uses selection sort to sort the vector of numbers shown. Th e vector is printed before and after the numbers are sorted into ascending order.

#include <vector> #include <iostream> using namespace std;

void sort(vector<int> & data, int n) { int indexOfSmallest = 0;

for (int top = 0; top < n - 1; top++) {

// First assume that the smallest is the rst element in the subvector indexOfSmallest = top;

// Then compare all of the other elements, looking for the smallest for (int index = top + 1; index < data.capacity(); index++) {

// Compare elements in the subvector if (data[index] < data[indexOfSmallest]) indexOfSmallest = index; }

// Then make sure the smallest from data[top] through data.size

// is in data[top]. This message swaps two vector elements. double temp = data[top]; // Hold on to this value temporarily data[top] = data[indexOfSmallest]; data[indexOfSmallest] = temp;

}

}

vector<int> initialize() { vector<int> v(5); v[0] = 76; v[1] = 91; v[2] = 100; v[3] = 62; v[4] = 89; return v; }

void display(vector<int> v) { for (int i = 0; i < v.capacity(); i++) { cout << v[i] << " ";

}

cout << endl; }

int main() {

vector<int> data = initialize();

cout << "Before sorting: "; display(data);

sort(data, data.capacity()); cout << " After sorting: "; display(data);

return 0; }

#### Output

Before sorting: 76 91 100 62 89

After sorting: 62 76 89 91 100

Most sort routines arrange the elements from smallest to largest. However, with just a few simple changes, any type of elements that allow the < and > operators may be arranged into descending order using the > operator.

if (data[index] < data[indexOfSmallest]) indexOfSmallest = index;

becomes

if (data[index] > data[indexOfLargest]) indexOfLargest = index;

#### SELF-CHECK

10-25 Alphabetizing a vector of string objects requires a sort in which order—ascending or descending?

10.6: Binary Search

10-26 If the largest element in a vector already exists as the fi rst, what happens when the swap function is called for the fi rst time (when top = 0)?

10-27 Write code that searches for and stores the largest element of vector x into largest. Assume that all elements from x[0] through x[n-1] have been given meaningful values, so all vector elements should be considered.

### 10.6 BINARY SEARCH

Th is chapter has shown the sequential search algorithm used to locate a string in a vector of string objects. Th is section examines the more effi cient binary search algorithm. Binary search accomplishes the same search task more quickly. It is faster than a sequential search, especially when the vector is large. However, one of its preconditions is that the vector must be sorted. By contrast, the slower sequential search does not require the vector to be sorted and the algorithm is simpler.

In general, binary search works like this. If a vector of objects is sorted, half of the vector’s elements are eliminated from the search each time a comparison is made. Th is is summarized in the following algorithm that searches for any element:

**Algorithm:** *Binary Search*

while *the element is not found and it still may be in the vector* { *determine the position of the element in the middle of the vector*  if *the element in the middle is not the one being searched for: eliminate the half of the vector that cannot contain the element*

}

Each time the search element is compared to one vector element, the binary search eff ectively eliminates half the vector elements from the search fi eld. In contrast, the sequential search only eliminates one element from the search fi eld for each comparison. Assuming a vector of string objects is sorted in alphabetic order, sequentially searching for "Ableson" does not take long since "Ableson" is likely to be located as one of the fi rst vector elements. However, sequentially searching for "Zevon" would take much more time because the sequential search algorithm fi rst searches through all names beginning with A through Y before arriving at the Z’s. Binary search gets to "Zevon" much more quickly.

Th e binary search algorithm has these preconditions:

1. Th e vector must be sorted (in ascending order for now).
2. Th e subscripts that reference the fi rst and last elements must represent the entire range of meaningful elements.

Th e element in the middle of the vector is accessed by computing the vector subscript that is halfway between the fi rst and last positions of the meaningful elements. Th is is the average

of the two subscripts that represent the fi rst and last elements in the vector. Th ese become subscripts in the search and will be referred to as rst, mid, and last. Here is the vector to be searched:

vector <string> str(32); int n = 7;

str[0] = "ABE"; // rst == 0 str[1] = "CLAY"; str[2] = "KIM"; str[3] = "LAU"; // mid == 3 str[4] = "LISA"; str[5] = "PELE"; str[6] = "ROE"; // last == 6

Th e binary search algorithm is preceded with several assignments to get things going:

searchString = the string being searched for rst = subscript of the fi rst meaningful vector element last = subscript of the last meaningful vector element mid = ( rst + last) / 2

At this point, one of three things can happen:

1. Th e element in the middle of the vector matches the search name—the search is complete.
2. Th e search element precedes the middle element. Th e second half of the vector can be eliminated from the search fi eld.
3. Th e search element follows the middle element. Th e fi rst half of the vector can be eliminated from the search fi eld.

Th is is written algorithmically as:

**Algorithm:** *Binary Search (more refi ned while assuming ascending sort)*

if *searchString* == *str*[*mid*] then *searchString is found* else

if *searchString* < *str*[*mid*]  *eliminate mid...last elements from the search*  else *eliminate rst...mid elements from the search*

Th e binary search algorithm is implemented here as a free function assuming the vector named str has been constructed, initialized, and sorted:

10.6: Binary Search

#include <vector>

#include <iostream>

#include <string> using namespace std;

vector<string> initialize() { vector<string> str(7); str[0] = "ABE"; str[1] = "CLAY"; str[2] = "KIM"; str[3] = "LAU"; str[4] = "LISA"; str[5] = "PELE"; str[6] = "ROE"; return str; }

// pre: The vector named str is sorted in ascending order.

// str[0] through str[6] are de ned vector elements.

// string de nes < and ==. int indexOf(string searchString, vector<string> str, int n) { int rst = 0;

int last = n - 1; // last = 6;

while (( rst <= last)) {

int mid = ( rst + last) / 2; // (0 + 6) / 2 = 3 if (searchString == str[mid]) // Check the three possibilities return mid; // 1) searchString is found else if (searchString < str[mid]) // 2) It's in rst half so last = mid - 1; // eliminate second half else

// 3) It's in second half so eliminate rst half

rst = mid + 1;

}

return -1; // searchString not found }

void display(vector<string> v) { for (int i = 0; i < v.capacity(); i++) cout << v[i] << " "; cout << endl; }

int main() {

vector<string> data = initialize(); cout << indexOf("LISA", data, data.capacity()); return 0; }

#### Objects Before Comparing searchString ("LISA") to str[mid] ("LAU")

str[0] "ABE" rst == 0 str[1] "CLAY" str[2] "KIM"

str[3] "LAU" mid == 3 str[4] "LISA" str[5] "PELE" str[6] "ROE" last == 6

After comparing searchString to str[mid], rst is increased and a new mid is computed:

~~str[0] "ABE"~~ // Because "LISA" is greater than str[mid], the ~~str[1] "CLAY"~~ // the objects str[0] through str[3] no longer need ~~str[2] "KIM"~~ // to be searched and can now be eliminated from ~~str[3] "LAU"~~ // subsequent search str[4] "LISA" rst == 4 str[5] "PELE" mid == 5 str[6] "ROE" last == 6

Since searchString < str[mid] or "LISA" < "PELE" is true, last is decreased and a new mid is computed:

~~str[0] "ABE"~~ ~~str[1] "CLAY"~~ ~~str[2] "KIM"~~ ~~str[3] "LAU"~~

str[4] "LISA" rst == 5 last == 5 mid == 5 ~~str[5] "PELE"~~ // Because "LISA" is less than str[mid], eliminate ~~str[6] "ROE"~~ // str[5] through str[6] from the search eld

Now str[mid] does equal searchString, so the algorithm will break out of the loop.

Th e binary search algorithm can be more effi cient than the sequential search that only eliminates one element per comparison. Binary search eliminates half the elements for each comparison. For example, when n == 1,024, a binary search eliminates 512 elements from further search after the fi rst comparison.

Now consider the possibility that the element being searched for is not in the vector. For example, to search for "CARLA", the values of rst, mid, and last progress as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Comparison** | rst | mid | last | **Comment** |
| 1 | 0 | 3 | 6 | Compare "CARLA" to "LAU" |
| 2 | 0 | 1 | 2 | Compare "CARLA" to "CLAY" |
| 3 | 0 | 0 | 0 | Compare "CARLA" to "ABE" |
| 4 | 1 | 0 | 0 | rst <= last is false and the function returns -1 |

Th e loop test ( rst <= last) evaluates to false when searchString ("CARLA") is not stored in the vector. Notice that last is less than rst—the two subscripts have crossed each other.

str[0] "ABE" last == 0 mid == 0 str[1] "CLAY" rst == 1 str[2] "KIM" str[3] "LAU"

Chapter Summary

str[4] "LISA" str[5] "PELE" str[6] "ROE"

After searchString ("CARLA") is compared to str[1] ("ABE"), no further comparisons are necessary. Th is is the second of two conditions that terminate the loop. Since rst is no longer less than or equal to last, searchString cannot be in the vector.

#### SELF-CHECK

10-28 Write at least one precondition for a successful binary search.

10-29 What is the maximum number of comparisons (approximately) performed on a list of 1,024 elements during a binary search? (*Hint:* After one comparison, only 512 vector elements need be searched; after two searches, only 256 elements need be searched and so on.)

10-30 During a binary search, what condition signals that the search element does not exist in a vector?

10-31 What changes must be made to the binary search when the elements are sorted in descending order?

### CHAPTER SUMMARY

* Whereas objects may store data of many diff erent types at the same time (a string, an int, and even a vector, for example), a vector object stores collections of the same class (a vector of char, int, string, or BankAccount objects, for example).
* Individual vector elements are referenced with subscripts. With a C++ vector, the int expression of a subscript reference should be in the range of 0 through the capacity - 1. For example, the valid subscript range of vector <double> x(100) is 0 through 99 inclusive.
* Out-of-range subscripts may not be detected at compile time and may cause system crash-es, destruction of other objects, or some other system-specifi c problems. It depends on the vector class you are using. Programmers must guard against these potential hazards. One of the easiest ways to do this is to use vector::at.
* An integer named n or size is usually an important piece of data that must be maintained in addition to the vector elements themselves. Th e number of meaningful elements is important in any vector-processing algorithm.
* Any vector object can be resized to have a diff erent maximum capacity. If it is resized to be bigger, the meaningful elements remain. However, if a vector is resized to be smaller, truncation of meaningful elements may occur.
* Th e selection sort algorithm was used to arrange vector elements into ascending order. Any object that can be compared with < may be sorted.
* vector objects may also be sorted in ascending order, which is more appropriate sometimes, especially with string elements where ascending order means alphabetical order.
* Th e binary search algorithm is more effi cient than sequential search. However, the vector must fi rst be sorted for binary search to work properly.

### EXERCISES

1. Show the output generated by the following program:

#include <iostream>

#include <vector> using namespace std;

int main() { const int MAX = 10; vector<int> x(MAX);

for (int i = 0; i < 3; i++) x[i] = i \* 2; for (int i = 3; i < MAX; i++) x[i] = x[i - 1] + x[i - 2]; for (int i = 0; i < MAX; i++) cout << i << ". " << x[i] << endl; return 0; }

1. How many elements must be given meaningful values for a vector with 100 elements?
2. Declare a C++ vector called vectorOfInts that stores 10 integers with subscripts 0 through 9.
3. Write code that determines the largest value of a vector named list. Assume all elements from index 0 through list.size()-1 have been assigned meaningful values.
4. Write code to determine the average of integers in a vector named list. Assume all elements from index 0 through list.size()-1 have been assigned meaningful values.
5. Write the output generated by the following program:

#include <iostream>

#include <vector> #include <string> using namespace std;

void init(vector<char> & data, int & n) { // postcondition: Initialize data as a vector of chars.

// Initialize n as the number of meaningful elements.

Exercises

n = 5; data[0] = 'c'; data[1] = 'b'; data[2] = 'e'; data[3] = 'd'; data[4] = 'a'; }

void display(const vector<char> & data, int n) { // post: Show all meaningful elements of data cout << endl;

cout << "Vector of chars: "; for(int i = 0; i < n; i++) cout << data[i] << " "; cout << endl; }

void mystery(vector<char> & data, int n) {

// post: Reverse the order of data int last; char temp;

last = n - 1;

for(int i = 0; i < n / 2 + 1; i++) { temp = data[i]; data[i] = data[last]; data[last] = temp; last--;

}

}

int main() {

vector<char> characters(10, ' '); int n;

init(characters, n); display(characters, n); mystery(characters, n); display(characters, n);

return 0; }

1. Write code to declare and initialize a vector of 10 string objects with keyboard input. Your dialogue should look like this:

Enter 10 strings

#0 First

#1 Second . . .

#9 Tenth

1. Write code that sets found to true if a given string is found in the following vector. If a string is not in the vector, let found be false. Assume only the fi rst n vector elements are initialized and are to be considered.

vector<string> s(200); int n = 127; bool found = false;

1. How many comparisons does a sequential search make when the search element is stored in the fi rst vector element and there are 1,000 meaningful elements in the vector?
2. How many comparisons does a sequential search make when the search element does not match any vector element and there are 1,000 elements in the vector?
3. Assuming a large number of searches are made on a vector, and it is just as likely that an element will be found in the fi rst position as the last position, approximate the average number of comparisons after 1,000 searches when there are 1,000 elements in the vector.
4. Write the output generated by the following program (trick question):

#include <vector> // For the vector<type> class

#include <iostream> using namespace std;

void init(vector<int> x, int n) {

// post: Supposedly modify n and the rst n elements of test in main x[0] = 0; x[1] = 11; x[2] = 22; x[3] = 33; x[4] = 44; n = 5; }

int main() {

vector <int> test(100, 0); int n;

// Initialize test and n init(test, n);

// Display the vector with n meaningful values cout << "The vector: "; for(int i = 0; i < n; i++) cout << test[i] << " ";

return 0; }

1. How would you change the code in exercise 12 so that the output is:

The vector: 0 11 22 33 44

Exercises

1. Write the output generated by the following program:

#include <vector> // For the vector<type> class

#include <iostream> using namespace std;

void f(const vector<int> & x) { cout << x[0] << endl; cout << x.capacity() << endl; }

int main() {

vector <int> test(10000, -1); f(test); return 0; }

1. Which lines contain compile time errors?

void f1(vector<int> x) {

cout << x[0] << endl; // Line 1 cout << x.capacity() << endl; // Line 2 x[0] = 999; // Line 3 }

1. Which lines contain compile time errors?

void f2(const vector<int> & x) { cout << x[0] << endl; // Line 1 cout << x.capacity() << endl; // Line 2 x[0] = 999; // Line 3 }

1. Which of the previous two functions (exercises 15 and 16) is more effi cient in terms of space and time—f1 or f2?
2. Write the output generated by the program segment below using the initialized vector of string objects.

#include <iostream>

#include <string>

#include <vector> // For the vector<type> class using namespace std;

int main() { vector<string> x(10); int j; int top = 0; int n = 5;

x[0] = "Alex"; x[1] = "Andy"; x[2] = "Ari"; x[3] = "Ash"; x[4] = "Aspen";

for (top = 0; top < n - 1; top++) { int subscript = top; for (j = top + 1; j <= n - 1; j++) { if (x[j] < x[subscript]) subscript = j;

}

string temp = x[subscript]; x[subscript] = x[top]; x[top] = temp;

}

for (int index = n - 1; index >= 0; index--) { cout << x[index] << endl;

} return 0; }

19. Write the output of the program segment below using this initialized vector of string objects.

vector <string> str(20); str[0] = "ABE"; str[1] = "CLAY"; str[2] = "KIM"; str[3] = "LAU"; str[4] = "LISA"; str[5] = "PELE"; str[6] = "ROE"; str[7] = "SAM"; str[8] = "TRUDY";

int rst = 0; int last = 8; int mid;

string searchString("CLAY");

cout << "First Mid Last" << endl; while ( rst <= last) { mid = ( rst + last) / 2; cout << rst << " " << mid << " " << last << endl; if (searchString == str[mid]) break; else if (searchString < str[mid]) last = mid - 1; else rst = mid + 1;

}

Programming Tips

if ( rst <= last) cout << searchString << " found" << endl; else

cout << searchString << " was not" << endl;

1. Write the output generated by the program segment in exercise 19 when searchString is assigned each of the following values:
   1. searchString = "LISA" d. searchString = "ABLE"
   2. searchString = "TRUDY" e. searchString = "KIM"
   3. searchString = "ROE" f. searchString = "ZEVON"
2. List at least one condition that must be true before a successful binary search can be implemented.
3. Using a binary search, what is the maximum number of comparisons (approximately) that will be performed on a list of 256 sorted elements? (*Hint*: After one comparison, only 128 vector elements need be searched; after two searches, only 64 elements need be searched; and so on.)

### PROGRAMMING TIPS

1. C++ begins to count at 0. Th e fi rst vector element is referenced with subscript 0, not 1 as is done in some other programming languages.
2. A vector often has a capacity greater than its number of meaningful elements. Sometimes vector objects are initialized to store more elements than are actually needed. In this case, only the fi rst n elements are meaningful.
3. Use a second variable named n,perhaps to maintain the number of meaningful vector elements. When vector objects are used as a data member, consider using another data member to store the number of meaningful elements. You may need to resize to allow additional capacity. However, when using vector objects outside of a class, make sure you have an integer that maintains the number of meaningful elements. Consider the following code that counts the number of elements from a fi le as individual vector elements are initialized. Th e number of meaningful elements is maintained in n, so it was necessary to initialize n to 0 and then increment n by 1 for each number on fi le.

vector <double> x(100, 0.0); double aNumber; int n = 0;

while ((inFile >> aNumber) && (n < x.capacity())) { x[n] = aNumber; n++; }

1. Consider using at(index) instead of [index]. Th e at message is safer. You get a runtime error that is easier to track down than a change to some memory in an unknown place.
2. Th e last meaningful vector element is in x[n-1], not x[n]. Don’t reference x[n]. Th is can be done in the code of the third programming tip by accidentally writing the for loop like this:

int n = 10; vector<int> x(n, 123);

for (int j = 0; j <= n; j++) { // Used <= instead of < cout.width(5);

cout << x[j]; // Will eventually reference garbage }

1. Prevent assignments to a vector with out-of-range subscripts. Th e code of the third programming tip has a loop test that terminates before assignment to x[x.capacity()]. When n equals the capacity, the loop terminates.

while ((inFile >> aNumber) && (n < x.capacity()) ) {

// The loop test prevents assignment to x[x.capacity()] x[n] = aNumber; n++; }

It would also be useful to notify the user that something went wrong in this case. Terminating the program prematurely is an easy (but awkward) way of doing this:

if (n == x.capacity() && inFile) {

cout << "\*\*Error\*\* Vector was too small. Terminating program" << endl; return 0; }

7. Make your programs robust with vector::resize and vector::capacity. Th e code in programming tip 6 can be most irritating to users once they have purchased your software. A sounder way to handle the awkward situation of having too small a vector is to resize it when necessary. With the following code, the vector’s capacity will increase by 10 elements every time the vector fi lls up. Th is code also demonstrates the advantage of keeping track of the number of meaningful values in a separate integer variable because size() and capacity() are 20 when there were only 17 numbers in the input fi le.

int aNumber; ifstream inFile("numbers"); vector<int> x(10); int n = 0;

Programming Tips

while (inFile >> aNumber) { if (n == x.capacity()) {

x.resize(n + 10);

} x[n] = aNumber; n++; }

cout << " n: " << n << endl; cout << " Size: " << x.size() << endl; cout << "Capacity: " << x.capacity() << endl; Output when the input fi le numbers has 17 integers:

n: 17 Size: 20

Capacity: 20

1. Do not pass vector objects by value. Passing any big object by value slows down program execution and requires unnecessary memory runtime allocation. If a function needs the values of a vector but is not supposed to modify the vector, pass the vector by const reference like this:

void constReferenceIsGood(const vector<double> & x, int n) {

// This function can reference any element in x, but cannot change x

}

As usual, if a function is meant to modify the argument (a vector in this case), pass it by reference like this:

void init(vector<double> & x, int & n) { // Reference parameter

// This function can change any element in x

}

Even string objects should be passed by const reference rather than by value because they are sometimes big.

1. Th e standard vector class does not check subscripts with [], but it does with vector::at(int). Consider using x.at(subscript) instead of x[subscript]. Th ey are equivalent expressions with one notable exception—when the subscript is out of range, vector::at reports it as an error. You’ll fi nd out about the error right away during testing. Th is is preferable to using some random value accessed with an out-of-range subscript. Your code would look diff erent than other C++ programs due to the historical use of [] and the newness of at.

#include <vector> // For the vector<type> class

#include <iostream> using namespace std; int main() { int n;

cout << "Enter vector capacity: "; cin >> n; vector <int> x(n);

for(int index = 0; index < n; index++) {

x.at(index) = index; } cout << "First: " << x.at(0) << endl; cout << "Last: " << x.at(x.capacity() - 1) << endl;

return 0; }

#### Dialogue

Enter vector capacity: ***100***

First: 0

Last: 99

Th e previous code once again demonstrates that the fi rst element in the vector is referenced with a subscript of 0, and the last element with index capacity()-1. So the following statement code would generate a runtime error and terminate the program: cout << "Last: " << x.at(x.capacity()) << endl; // Always an error

1. Th ere are many sorting algorithms other than selection sort. Selection sort is only one of the many known sorting algorithms. Several others have approximately the same runtime effi ciencies. Some are even better. For example, quicksort is usually much more effi cient. Th is chapter was not written to cover sorting completely. It is only a very brief introduction to another category of vector-processing algorithms.
2. Th ere are many searching algorithms. Sequential and binary search are only two of the known searching algorithms. For small amounts of data, sequential search works very nicely. For larger amounts of data stored in sorted vectors, binary search works well. Other ways to store very large amounts of data that can be searched rapidly include hash tables and binary trees, for example. Th ese are topics usually covered in a second course.

**PROGRAMMING PROJECTS**

## 10A REVERSE

Write a complete C++ program that reads an undetermined number of integers (maximum of 100) and displays them in reverse order. Th e user may not supply the number of elements, so a sentinel loop must be used. Here is one sample dialogue:

Enter up to 100 ints using -1 to quit:

***70***

***75***

***90***

***60***

***80***

***-1***

Reversed: 80 60 90 75 70

## 10B SHOW THE ABOVE-AVERAGE ONES

Write a complete program that inputs an undetermined number of positive numeric values, determines the average, and displays every value that is greater than or equal to the average. Th e user may not supply the number of elements, so use a sentinel loop. Here is one sample dialogue:

Enter numbers or -1 to quit

***70***

***75***

***90***

***60***

***80***

***-1***

Average: 75

Inputs >= average: 75 90 80

## 10C SEQUENTIAL SEARCH FUNCTION

Write a function named search that returns the subscript of the fi rst found search element in a vector of string objects. If the search element is not found, search should return -1.

## 10D A COLLECTION OF BANKACCOUNT OBJECTS

Write a complete C++ program that creates an undetermined number of BankAccount objects and stores them in a vector. Th e input should come from an external fi le that looks like the following, but may contain 1, 2, 3, or up to exactly 20 lines (each line represents all data necessary to create one BankAccount object):

Hall 100.00

Solly 53.45

Kirstein 999.99 . . .

Pantone 8790.56

Brendle 0.00

Kentish 1234.45

After initializing the vector and determining the number of BankAccount objects, display every BankAccount that has a balance greater than or equal to $1,000.00. Th en display every

BankAccount that has a balance less than or equal to $100.00. Your output should look like this:

Balance >= 1000.00

Pantone: 8790.56

Kentish: 1234.56

Balance < 100.00:

Solly: 53.45

Brendle: 0.00

## 10E PALINDROME 1

A *palindrome* is a collection of characters that read the same backward as forward. Write a program that extracts a string from the keyboard and determines whether or not the resulting string is a palindrome (recall that string objects reference individual characters with the subscript operator [ ]). Some examples of palindromes are YASISAY, racecar, 1234321, ABBA, level, and MADAMIMADAM. Here are two sample dialogues. (*Note:* Do not use any blank characters! If you prefer, complete programming project 10F instead.)

Enter string: ***MADAMIMADAM***

Reversed: MADAMIMADAM

Palindrome: Yes

Enter string: ***RACINGCAR***

Reversed: RACGNICAR

Palindrome: No

## 10F PALINDROME 2

A *palindrome* is a collection of characters that read the same backward as forward. Write a program that extracts a line of characters from the keyboard using

getline(istream& is, string& aString)

and then determines whether or not the resulting string is a palindrome. Th e blank characters should be ignored and it must be case-insensitive. (*Hint:* First convert the individual characters to uppercase with the toupper function from <cctype>, and then create a new string with no space characters.)

Enter a line: A man a plan a canal Panama

AMANAPLANACANALPANAMA is a palindrome

## 10G FIBONACCI NUMBERS

Th e Fibonacci numbers start as 1, 1, 2, 3, 5, 8, 13, 21. Notice that the fi rst two are 1, and any successive Fibonacci number is the sum of the preceding two. Write an entire program that properly initializes a vector named b to represent the fi rst 20 Fibonacci numbers ( b[1] is the second Fibonacci number). Do not use 20 assignment statements to do this. Th ree should suffi ce.

## 10H SALARIES

Write a program that inputs an undetermined number of annual salaries from an input fi le. After this, display all salaries that are above average. Also show the percentage of salaries that are above average. If the input fi le contains this data:

30000.00

24000.00

35000.00

32000.00

25000.00 then your output should look like this:

Average salary = 29200.00 Above average salaries:

30000.00

35000.00

32000.00

60% of reported salaries were above average

## 10I BINARY SEARCH FUNCTION

Write a free function named search that returns the subscript location of the fi rst found search element in a vector. Use the binary search algorithm. If the search element is not found, search should return -1. Test your function, of course.

## 10J FREQUENCY

Write a C++ program that reads integers from a fi le and reports the frequency of each integer. For example, if the input fi le contains the numbers as shown to the left below, your program should generate the output shown to the right below, with the highest numbers fi rst. Create the input fi le so that all numbers are in the range of 0 through 100 inclusive. (*Hint*: Start with a vector of capacity 101 with all elements set to 0.)

|  |  |
| --- | --- |
| **Th e File** test.dat | **Th e Program Dialogue** |
| 75 85 90 100 | Enter le name: ***test.dat*** |
| 60 90 100 85 | 100: 3 |
| 75 35 60 90 | 90: 8 |
| 100 90 90 90 | 85: 3 |
| 60 50 70 85 | 75: 3 |
| 75 90 90 70 | 70: 2  60: 3  50: 1  35: 1 |

## 10K EIGHT VECTOR PROCESSING FUNCTIONS

1. int numberOfPairs(const vector<string> & strs)

Complete numberOfPairs to return the number of times a pair occurs in strs. A pair is any two string elements that are equal (case sensitive) at consecutive vector indexes. Th e vector may be empty or have only one element. In both of these cases, return 0. Here is some testing code that uses push\_back messages to add elements at the end of the vector (you need #include <cassert> for the assert function).

vector<string> strs; strs.push\_back("a"); assert(0 == numberOfPairs(strs)); strs.push\_back("a"); assert(1 == numberOfPairs(strs)); strs.push\_back("a"); assert(2 == numberOfPairs(strs)); strs.push\_back("b"); strs.push\_back("b");

// a a a b b

assert(3 == numberOfPairs(strs));

1. int numberOfVowels(const vector<char> & chars)

Given a fi lled vector of char elements, complete numberOfVowels to return the number of vowels which could be the letters A, E, I, O, or U in either uppercase or lowercase. If the vector is empty, return 0. Here is some testing code to help explain the expected behavior:

vector<char> chars; chars.push\_back('x'); assert(0 == numberOfVowels(chars)); chars.push\_back('A'); chars.push\_back('a'); assert(2 == numberOfVowels(chars)); chars.push\_back('I'); chars.push\_back('o'); chars.push\_back('U'); chars.push\_back('e'); assert(6 == numberOfVowels(chars));

1. bool sumGreaterThan(const vector<double> & doubles, double sum)

Given a fi lled vector of double elements, write function sumGreaterThan to return true if the sum of all vector elements is greater than sum. Return false if sum is less than or equal to the elements in doubles. Also return false if the vector is empty. Here is some testing code to help explain the expected behavior:

vector<double> doubles; doubles.push\_back(4.0);

assert(sumGreaterThan(doubles, 4.0)== false); doubles.push\_back(0.1);

assert(sumGreaterThan(doubles, 4.0)== true);

1. int howMany(const vector<string> & vec, string valueToFind)

Complete function howMany to return the number of elements in a vector of strings that equals valueToFind. Th e vector may be empty.

vector<string> strings; strings.push\_back("A"); strings.push\_back("a"); strings.push\_back("A"); assert(0 == howMany(strings, "x")); assert(1 == howMany(strings, "a")); assert(2 == howMany(strings, "A"));

1. void sortOfSort(vector <int> & nums)

Complete function sortOfSort that modifi es the parameter nums to place the largest integer at index n-1 and the smallest integer at nums[0]. Th e other elements must still be in the vector, but not in any particular order. You must modify the given vector argument by changing nums in method sortOfSort.

|  |  |
| --- | --- |
| **Original vector** | **Modifi ed vector (some elements may diff er in order)** |
| {4, 3, 2, 0, 1, 2} | {0, 3, 2, 1, 2, 4} |
| {4, 3, 2, 1} | {1, 3, 2, 4} |
| {4, 3, 1, 2} | {1, 3, 2, 4} |

vector<int> nums; nums.push\_back(4); nums.push\_back(3); nums.push\_back(1); nums.push\_back(2); sortOfSort(nums); assert(1 == nums[0]); assert(4 == nums[3]);

assert(nums[1] == 2 || nums[1] == 3); // depends on your algorithm assert(nums[2] == 2 || nums[2] == 3);

(*Hint:* Get the smallest value at index 0 before you look for and swap the largest and the last index.)

1. void evensLeft(vector<int> & nums)

Modify the parameter nums so it still contains the exact same numbers as the given vector, but rearranged so that all the even numbers come before all the odd numbers. Other than that, the numbers can be in any order. You must modify the vector argument by changing nums in method evensLeft. Th e vector may be empty or have only one element. In both cases, no change should be made.

|  |  |
| --- | --- |
| **Original vector** | **Modifi ed vector** |
| {1, 0, 1, 0, 0, 1, 1} | {0, 0, 0, 1, 1, 1, 1} |
| {3, 3, 2} | {2, 3, 3} |

vector<int> ints; ints.push\_back(3); ints.push\_back(3); ints.push\_back(2); evensLeft(ints); assert(2 == ints[0]); assert(3 == ints[1]); assert(3 == ints[2]);

1. void shiftNTimes(vector<int> & nums, int numShifts)

Complete shiftNTimes to modify nums so it is “left shifted” n times. So shiftNTimes({6, 2, 5, 3}, 1) changes the vector argument to {2, 5, 3, 6} and shiftNTimes({6, 2, 5, 3}, 2) changes the vector argument to {5, 3, 6, 2}. You must modify the vector argument by changing the parameter nums inside method shiftNTimes. Remember, a change to the parameter inside the method shiftNTimes changes the argument.

shiftNTimes( {1, 2, 3, 4, 5, 6, 7}, 3 ) modifi es the vector to { 4, 5, 6,

7, 1, 2, 3 }

shiftNTimes( {1, 2, 3, 4, 5, 6, 7}, 0 ) does not modify the vector shiftNTimes( {1, 2, 3}, 5) modifi es the vector to { 3, 1, 2 } shiftNTimes( {3}, 5) modifi es the vector to { 3 }

vector<int> nums2; nums2.push\_back(1); nums2.push\_back(2); nums2.push\_back(3);

nums2.push\_back(4); nums2.push\_back(5);

shiftNTimes(nums2, 2); assert(3 == nums2[0]); assert(4 == nums2[1]); assert(5 == nums2[2]); assert(1 == nums2[3]); assert(2 == nums2[4]);

8) void replaced(char[] & vector, char oldChar, char newChar)

Modify the vector arguments so all occurrences of oldChar are replaced by newChar.

replaced ({'A', 'B', 'C', 'D', 'B'}, 'B', '+') must modify the vector argument to be {'A', '+', 'C', 'D', '+'}.

### Original vector Modifi ed vector

replaced({'A', 'B', 'C', 'D', 'B'}, 'C', 'L') { 'A', 'B', 'L', 'D', 'B' } replaced({'n', 'n', 'D', 'N'}, 'n', 'T') { 'T', 'T', 'D', 'N' }

vector<char> chars2; chars2.push\_back('n'); chars2.push\_back('n'); chars2.push\_back('D'); chars2.push\_back('N'); replaced(chars2, 'n', 'T'); assert('T' == chars2[0]); assert('T' == chars2[1]); assert('D' == chars2[2]); assert('N' == chars2[3]);

## 10L CLASS STATS

First create the header fi le Stats.h to store all member function declarations and any instance variables you would need. You will certainly need a vector. Th en create a new fi le named Stats.cpp and implement all of the member functions of the class defi nition. Th e following test method must generate the output shown below when the input fi le numbers has these 10 integers:

5 1 6 2 3 8 9 4 7 10

/\*

* Stats.cpp
* \* A test driver for class Stats

\*/

#include <fstream>

#include <iostream>

#include "Stats.h"

using namespace std;

int main() { ifstream inFile("numbers"); int x = 0; Stats tests;

while (inFile >> x) { tests.add(x); }

cout << "Elements before sort: "; tests.display(); tests.sort();

cout << endl << " Elements after sort: "; tests.display();

cout << endl; cout << endl << "Statistics for the rst 10 integers" << endl; cout << " Size: " << tests.size() << endl; cout << " Mean: " << tests.mean() << endl; cout << " High: " << tests.max() << endl; cout << " Low: " << tests.min() << endl; cout << " Median: " << tests.median() << endl;

return 0; }

### Output

Elements before sort: 5 1 6 2 3 8 9 4 7 10

Elements after sort: 1 2 3 4 5 6 7 8 9 10

Statistics for the rst 10 integers

Size: 10

Mean: 5.5

High: 10

Low: 1

Median: 6

**C H A P T E R E L E V E N**

# Generic Collections

## SUMMING UP

You have now experienced generic vector objects as a way to store a collection of related elements of a specifi c type.

## COMING UP

Th is chapter introduces another collection class named Set to provide a review of vector processing, class defi nitions, and member function implementations. Th is chapter also introduces the C++ template mechanism that allows a collection to store a specifi c type of object. After studying this chapter you will be able to

* build your own collection class to store any type of element
* better understand classes with data members, constructors, and member functions
* better understand how to develop functions that involve vector processing

### 11.1 COLLECTION CLASSES

As you continue your study of computing, you will spend a fair amount of time exploring ways to manage collections of data. Th e vector class presented in the previous chapter is only one of many classes designed for just this purpose. Collection classes have the following characteristics:

* Th eir main responsibility is to store a collection of objects.
* Th ey usually add and remove objects from the collection.
* Th ey allow access to individual elements in a variety of ways.

Th e Set collection class used in this chapter provides a review of class defi nitions and member function implementations. Th is time, however, member functions will employ vectorprocessing algorithms. You will also learn how collection classes can be made to store only a specifi c type. Th e type can be passed as an argument to the class when constructed.

Th e main purpose of the Set class is to store a collection of unique objects. A Set object has the following characteristics:

325

* A Set may store any type of object, as shown with class vector.
* Set elements are unique—duplicate elements are not allowed. Set elements need not be maintained in any particular order.

A Set object will understand the messages isEmpty, insert, remove, size, and contains. Data members include a vector named elements, which stores a collection, and an integer named n to store the current number of elements in the Set object. Before discussing this new type, consider the C++ template mechanism that permits one class to store any type of object.

#### 11.1.1 PASSING TYPES AS ARGUMENTS

Expressions such as x or 1.5 can be passed as arguments to functions. Class names (types) such as int, double, string, and BankAccount can also be passed as arguments using the C++ template mechanism, <int> or <string> for example. Passing type names as arguments allows programmers to use the same collection class to hold any type of object. Th is means we need only one Set class instead of a diff erent Set class for every type to be stored.

Th e C++ Standard Template Library (STL) uses the template mechanism to implement the standard collection types such as vector, list, stack, and queue. Rather than implementing a vector, list, stack, and queue for each new type of element that a programmer might think of, the compiler instead uses the single class template to create them automatically.

For each class name passed as an argument, a new class is automatically created to manage collections of that class. For example, we can have sets of any type:

Set<string> ids; // Store string objects only

Set<double> nums; // Store numbers only

Set<BankAccount> accounts; // Store BankAccount objects only

Another advantage to templates is that we can restrict the type to insert only that one type. For example, these messages compile:

ids.insert("c1w4"); nums.insert(123); accounts.insert(BankAccount("c1w4", 100.00));

Th ese messages will not compile since the argument is not the proper type (which is a good thing, actually).

ids.insert(123); // Argument must be a string nums.insert("c1w4"); // Argument must be a num accounts.insert(100.00); // Argument must be a BankAccount

#### 11.1.2 TEMPLATES

Type parameters allow programmers to pass a data type to a class to communicate the desired type of element to store.

11.1: Collection Classes

##### General Form 11.1 *Class templates*

template<class *template-parameter*> class *class\_name*

Template declarations written before a class give the template parameter scope that extends throughout the entire class defi nition. For example, the Set template class in C++ could begin like this:

template<class Type> class Set { public:

// Allow insertion of only one speci c type void insert (Type element);

Th e Set type makes frequent use of the template parameter named Type. For example, when a Set is constructed like this:

Set <string> names; the parameter Type is replaced by the Type name that was passed as the argument between angle brackets, which is string here. C++ then generates this code:

void insert(string element);

However, if a constructor were invoked to initialize a Set of ints as in

Set <int> x;

C++ generates this code where Type gets replaced with int:

void insert(int element);

Because Set is declared as a template class, the compiler can use it as a model to build any number of other classes to store diff erent types of elements as a Set.

Th e class parameter named Type has scope that extends to the end of the class defi nition. Th is means that Type may be used anywhere in the class defi nition, such as in the public section or in the private data member section. For example, writing the type parameter (named Type here) before the class defi nition is critically important to the templated types:

template<class Type> class Set { public: Set();

Type insert(Type element) private:

Type key;

};

Th e public method insert will accept parameters only of Type passed as the Type argument. Th e type of the private data member key is also the type passed as the Type argument. At declaration, the Type identifi er is replaced with the argument specifi ed at declaration. For example, these object initializations cause s1 and s2 to become templated to string and double, respectively:

Set <string> s1; Set <double> s2;

template<class Type> template<class Type>

class Set { class Set { public: public: Set(); Set();

void insert(string element){ void insert(double index){ }

} private:

private: double key;

string key; };

};

Here is a sample program to summarize the capabilities of the generic Set class:

#include <iostream>

#include <string> using namespace std; #include "Set.h" // For a generic (with templates) Set class

int main() {

Set<string> names;

cout << "After contruction, size is " << names.size() << endl; // 0 cout << "and the Set isEmpty: " << names.isEmpty() << endl; // true

// Add a few elements, duplicates not allowed names.insert("Chris"); names.insert("Chris"); names.insert("Dakota"); names.insert("River");

names.remove("River"); // Succeeds names.remove("Not here"); // No change to the Set

cout << endl << "After 4 insert attempts and 2 remove attempts: " << endl; cout << "isEmpty: " << names.isEmpty() << endl; // false cout << "size: " << names.size() << endl; // 2 cout << "contains(\"Chris\")? " << names.contains("Chris") << endl; cout << "contains(\"Dakota\")? " << names.contains("Dakota") << endl; cout << "contains(\"River\")? " << names.contains("River") << endl; cout << "contains(\"No\")? " << names.contains("No") << endl;

return 0;

}

11.1: Collection Classes

##### Output (1 means true, 0 means false)

After contruction, size is 0 and the Set isEmpty: 1

After 4 insert attempts and 2 remove attempts:

isEmpty: 0 size: 2 contains("Chris")? 1 contains("Dakota")? 1 contains("River")? 0 contains("No")? 0

One of the advantages of this Set class, and an issue that becomes important in many applications, is the number of elements that may be stored with one Set object. Th e maximum number of set elements cannot always be determined in advance. Th e number of objects that may be contained in one Set object depends on the size of the objects and the amount of available memory in the free store. Th e best answer is this: the class Set will store as many objects as memory allows; there is no fi xed maximum size. Th e logic that handles this will be in the insert method.

To remove an object from a Set, the equality operator == must be defi ned for the type pass as a type parameter because it will be used in contains and remove messages. Here is code from

BankAccount.cpp that overloads the == operator to return true if the name of the BankAccount to the left of == is equal to the name of the argument to the right of ==.

// Overload the == operator to compare two BankAccount objects bool BankAccount::operator == (const BankAccount& right) const { return name == right.name; }

Th e binary operator == can now be applied to BankAccount objects:

BankAccount acct1("Ali", 123.44);

BankAccount acct2("Ali", 567.88);

BankAccount acct3("Billie", 567.88);

if(acct1 == acct2 && !(acct1 == acct3)) // true cout << "acct1 == acct, but not acct3: " << endl;

##### Output

acct1 == acct, but not acct3

Also, since Set is a collection using a vector, C++ requires the type of Set element to have a default constructor, a constructor with zero parameters like this one in BankAccount.cpp:

// A default constructor is require if you want a collection of these

BankAccount::BankAccount() { name = "???"; balance = -9.99; }

#### SELF-CHECK

Use this object declaration to answer the following questions:

Set<int> intSet;

11-1 How many integers can be stored in intSet?

11-2 Write code that prints the number of elements in intSet.

11-3 Write a message that attempts to add the integer 89 to the collection of integers in intSet.

11-4 Write a message that will remove 89 from intSet.

#### 11.2 class Set<Type>

Th is section shows how a vector and templates are combined to implement a Set class. Th is Set object:

* is generic because any type of element is allowed with <Type>
* does not have a fi xed maximum size—it allocates memory as long there is some memory available in the free store

For several reasons, this new Set type will have the class defi ned in one fi le, not the usual two fi les. Th e main reason for this is to prevent compile time errors. Some compilers require all code in one fi le to get templates to work. Th ere will not be the usual separation of class defi nition in a header (.h) fi le from the implementation in a separate .cpp fi le.

A second reason for using just one .h fi le is it prevents writing the same line before every method heading. Th e code looks cleaner with a dozen fewer of these repeated lines: template <class Type>. Also there are about a dozen fewer occurrences of Set:: before each method defi nition. So the generic (template) Set class will be built in the same fi le: Set.h. All methods are in the same fi le:

/\*\*

* Set.h
* \* This is a collection class to represent sets of any type.
* Duplicate elements are not allowed.

\*/

11.2: class Set<Type>

#ifndef SET\_H\_ #de ne SET\_H\_ #include <vector>

template<class Type> class Set {

A vector will be used to store the elements in any Set<Type>. Th is Set will also maintain an int variable n to store the number of unique elements:

private: std::vector<Type> elements; int n;

Th e member data n will be initialized to 0 in the constructor, must be increased by 1 for each successful insert, and must be decreased by 1 for each successful remove.

##### 11.2.1 THE CONSTRUCTOR Set()

Th e Set constructor guarantees that a Set starts empty with a capacity of 20, which could be larger or smaller.

// The public constructor public:

//--constructor

Set() { elements.resize(20);

n = 0; // This Set object has zero elements when constructed }

Th e programmer may construct Set objects like this:

Set <double> tests;

Set <string> names;

Set <BankAccount> names;

###### 11.2.2 bool contains(Type const& value) const

When working with Set objects, it is often important to know if a set contains a specifi c element. Th e contains member function uses a loop to sequentially search the vector. Once found, contains immediately returns true.

// Return true if value is in this set bool contains(Type const& value) const { for (int i = 0; i < n; i++) { if (value == elements[i]) return true;

}

return false; }

©

If all n elements are searched for and none were == to element, the loop terminates and contains returns false to indicate element is not in this set.

###### 11.2.3 void insert(Type const& element)

Since this collection class is a Set, insert must fi rst ensure the collection does not contain the element. If so, a check is made to ensure the vector can store more elements. If not, the vector capacity is increased before the new element is stored.

// If element is not == to any element, add element to this Set

// The vector will be resized to hold more elements if needed. void insert(Type const& element) { if (contains(element)) return;

// Otherwise add the new element at the end of the vector

// First make sure there is enough capacity if (n == elements.size()) {

// Add memory for 10 more elements whenever needed elements.resize(n + 10); }

// Insert after the last meaningful element in this set. elements.at(n) = element; n++; }

###### 11.2.4 bool remove(Type const& removalCandidate)

Th e remove method will remove a found element by overwriting it with the last element in the collection. If removalCandidate is not found, the method simply returns false.

// pre: The removalCandidate type must overload the == operator // post: If found, removalCandidate is removed from this Set.

// // Remove removalCandidate if found and return true.

// If removalCandidate is not in this Set, return false.

bool remove(Type const& removalCandidate) { // Find the index of the element to remove int index = 0;

while (index < n && !(removalCandidate == elements[index])) { index++;

}

// When subscript == size() removalCandidate was not found if (index == n) { return false;

} else { // Found it when elements[subscript] == removalCandidate.

// Overwrite removalCandidate with the element at the largest index elements[index] = elements[n - 1];

// decrease size by 1, and n--;

// report success to the client code where the message was sent return true;

}

}

11.3: The Iterator Pattern

##### SELF-CHECK

11-5 How many diff erent classes are built when this code is compiled?

Set<string> ids;

Set<int> studentNumber;

Set<double> points;

Set<double> tests;

11-6 What is the value of a Set’s size() after each of these situations? Assume the Set is constructed before each situation.

1. 10 successful inserts, then 5 successful removals
2. 40 successful inserts
3. 40 successful inserts, then 40 successful removals

### 11.3 THE ITERATOR PATTERN

Because each Set object always knows how many elements it stores (n), a collection object can be given functions designed to sequentially iterate over the entire collection of items. Th is can be made a part of any collection class.

Th is textbook’s Set class uses iterator methods to visit the contained objects. Th e following program shows how the client code could iterate over the entire collection without having to worry about going out of bounds. It is a preview of the four methods that will be added to class Set<Type> to allow access to all elements.

#include <iostream> using namespace std;

#include "Set.h" // For a generic (with templates) Set class

#include "BankAccount.h"

int main() {

Set<BankAccount> set; // Store set of 3 BankAccount objects

BankAccount anAcct("Devon", 100.00); set.insert(anAcct);

set.insert(BankAccount("Chris", 300.00)); set.insert(BankAccount("Kim", 200.00));

set. rst(); // Initialize an iteration over all elements double total = 0.00; while(set.hasMore()) {

cout << set.current().getName() << " has "; cout << set.current().getBalance() << endl; total += set.current().getBalance(); set.next();

}

cout << "Total balance: " << total << endl;

return 0; }

#### Output

Devon has 100

Chris has 300

Kim has 200

Total balance: 600

Th e initial statement in the loop— rst()—sets the Set object’s internal index to refer to the fi rst item in the collection. Th e loop test with hasMore() is true as long as there is at least one more element to visit. At the end of each loop iteration, the repeated statement set.next() updates the internal index either to refer to the next item in the collection or to make sure hasMore() will return false. Inside the loop, current() returns a reference to the element in the collection that is the element that can be accessed.

#### SELF-CHECK

11-7 Write code that determines the maximum BankAccount balance no matter how many elements are in the Set<BankAccount>.

##### 11.3.1 THE ITERATOR MEMBER FUNCTIONS

Th e Set iterator member functions exist for the sole purpose of allowing client code to access any and all of the Set elements in a sequential fashion, from the fi rst element to the last. Th e rst() function must be called to begin the iteration to set the private data member current to refer to the fi rst element in the Set object:

void rst() { currentIndex = 0; }

Th e hasMore() member function returns true if there is at least one more element to access. You will see this message used as the loop test:

while(set.hasMore())

Th e hasMore() member function compares the private data member currentIndex to return true when there is at least one more element to visit:

bool hasMore() const { return currentIndex < n;

}

Chapter Summary

Th e next() member function simply increments the internal index:

void next() { currentIndex++; }

And fi nally, current() returns the element referred to by the internal cursor current. Notice that the return type is whatever the client code specifi ed in constructing the Set<Type>.

Type current() const { return elements[currentIndex]; }

### CHAPTER SUMMARY

* Classes with type parameters allow the user to pass a type name as an argument to a class. Th is allows collection classes such as vector, list, and Set to manage any type of object.
* A class template permits the compiler to create many diff erent classes. Th e compiler does the work. Th e programmer need not implement separate StringVector, IntVector, and BankAccountVector classes, for example.
* Member functions may be implemented in one fi le with no separate header, while some compilers require it. Th is also reduces a large amount of repeated syntax. Set was built that way as will be the projects. Here is an outline of the Set class in one fi le with comments and the code between curly braces removed. Use this as a model for implementing the Stack and PriorityList programming projects.

/\*

* + File name: Set.h
  + /

#ifndef SET\_H\_ #de ne SET\_H\_ #include <vector>

template<class Type> class Set {

private: std::vector<Type> elements; int n; int currentIndex;

public:

Set() { } void insert(Type const& element) { } bool remove(Type const& removalCandidate) { } int size() const { }

bool contains(Type const& value) const { }

bool isEmpty() const { } void rst() { } bool hasMore() const { } void next() { }

Type current() const { }

};

#endif /\* SET\_H\_ \*/

* Th e Set class illustrates how a vector can be utilized as a storage mechanism in a class that provides higher-level messages such as insert and remove—no subscripts required.
* Collection classes such as Set and vector store collections of objects while providing suitable access to the elements.
* Th e Set class introduced the notion of iterator member functions that allow client code to traverse the entire collection without revealing the underlying structure. Sets are not ordered or indexed, so iterators are needed to visit the nodes. Other types such as vector are indexed and have the [] and at operations.

### EXERCISES

1. Use this code to answer each of the questions below:

#include "Set.h" int main() { Set<double> db;

// . . .

* 1. How many doubles can db store?
  2. Write code that adds at least four unique elements to db.
  3. Write code that displays all elements in db on separate lines using the iterator methods.
  4. Write code that determines the range of values in db. Range is defi ned as the largest values minus the smallest.

Programming Tips

1. Code a templated class named plus that shows what happens to two values when + is applied. You may place the class defi nition and method implementation in one fi le named Plus.h. Th e following code should generate the output shown in the comments:

// You only need one template class

Plus<int> a(2, 3);

Plus<double> b(2.2, 3.3);

Plus<string> c("Abe", "Lincoln");

a.show(); // 5

b.show(); // 5.5

c.show(); // AbeLincoln

1. Write code that fi nds the range of integers in Set<int> intSet;. Range is defi ned as the largest integer minus the smallest integer.

### PROGRAMMING TIPS

1. Th ere are many standard C++ collection classes (vector, list, stack, queue) that are more versatile and robust than our Set class. You do not need to use the Set class for any real work. Th e Set class was presented here as a review of class defi nitions and vector processing. C++ even has a generic class set. Th e Set class in this chapter introduced how to build generic collections using templates in one fi le.
2. When implementing generic collections, put all code in one fi le. Th is reduces the amount of repeated code before each member function. Some compilers require one fi le only for template classes.
3. Iteration is prevalent; sets are not. Th e iterator functions were shown to make you aware that this pattern is frequently used, while showing one way to access all elements of a collection. Th e syntax and method names are diff erent in the C++ Standard Template Library. Sets are not used as often as other collections such as lists, stacks, and maps.
4. Templates provide genericity. Th e value of templates can be appreciated if you realize that one only needs one template class to create a new class for any C++ type or any new type you create. As you continue your study of C++, you will see other template classes.
5. Templates provide a lot of extra syntax. Consider the following simple and incomplete collection class that stores elements like a wait line: fi rst in, fi rst out. Th e column on the left shows two fi les and oft-repeated syntax, about 80 words. Th e single .h fi le on the right column is shorter—fewer lines, fewer words, and fewer symbols such as <, >, and ::.

// File Queue.h

#include <vector>

template<class Type>

class Queue {

private:

std::vector<Type> elements;

int rst;

int last;

public:

Queue();

void add(Type const& element);

Type remove();

}

;

// File Queue.cpp

#include "Queue.h"

template<class Type>

Queue<Type>::Queue() {

elements.resize(1000);

rst = -1;

last = -1;

}

template<class Type>

void Queue<Type>::add(Type const& element) {

last = (last+1) % elements.capacity();

elements[last] = element;

}

template<class Type>

Type Queue<Type>::remove() {

rst = ( rst+1) % elements.capacity();

return elements[ rst];

}

// File Queue.h

#include <vector>

template<class Type>

class Queue {

private:

std::vector<Type> elements;

int rst;

int last;

public:

Queue() {

elements.resize(1000);

rst = -1;

last = -1;

}

void add(Type const& element) {

last = (last+1) % elements.capacity();

elements[last] = element;

}

Type remove() {

rst = ( rst+1) % elements.capacity();

return elements[ rst];

}

}

;

### PROGRAMMING PROJECTS

#### 11A class Stack<Type>

Implement a generic (with templates) Stack. A Stack allows elements to be added and removed in a last-in, fi rst-out (LIFO) manner. Stacks have an operation called push to place elements at the “top” of the stack, and another operation called pop to remove and return the element at the top of the stack. Th e only element on the stack that may be referenced is the one on the top. Th is means that if two elements are pushed onto the stack, the topmost element must be “popped”

Programming Projects

(removed) from the stack before the fi rst-pushed element can be referenced. Here is a Stack for storing up to 20 integers. Your program must compile and generate the output.

#include <iostream>

#include "Stack.h" using namespace std;

int main() { Stack<int> intStack(20); // stack of 20 ints

// Use intStack intStack.push(1); intStack.push(2); intStack.push(3); intStack.push(4);

cout << "4? " << intStack.peek() << endl; cout << "4? " << intStack.pop() << endl; cout << "3? " << intStack.peek() << endl;

cout << "isEmpty 0? " << intStack.isEmpty() << endl; cout << "3 2 1? "; while(! intStack.isEmpty()) { cout << intStack.pop() << " ";

} cout << endl;

cout << "isEmpty 1? " << intStack.isEmpty() << endl;

return 0; }

##### Output

4? 4

4? 4 3? 3 isEmpty 0? 0 3 2 1? 3 2 1 isEmpty 1? 1

*Note:* See the beginning of a Queue class in the Programming Tips section of this chapter for a complete class in one .h fi le.

#### 11B PriorityList<Type>

Th is project asks you to implement a collection class PriorityList<Type> using a vector data member. Th is new type will store a collection of elements as a zero-based indexed list where the element at index 0 is considered to have higher priority than the element at index 1. Th e element at index size()-1 has the lowest priority. An instance of this collection class will be able to store just one type of element such as <string>. Remember that the element at index 0 is the top priority; the element at index size()-1 is the lowest priority.

PriorityList<string> todos;

todos.insertElementAt(0, "Study for the CS exam"); todos.insertElementAt(0, "Get groceries"); todos.insertElementAt(0, "Sleep");

for(int priority = 0; priority < todos.size(); priority++) cout << todos.getElementAt(priority) << endl;

##### Output

Sleep

Get groceries

Study for the CS exam

Complete these methods in PriorityList<Type> so it uses a vector to store the elements.

// Construct an empty PriorityList with capacity to store 20 elements

PriorityList();

// Return the number of elements currently in this PriorityList int size();

// Return true if size() == 0 or false if size() > 0 bool isEmpty();

// Insert the element at the given index. If the vector // is too small, resize it.

// precondition: index is on the range of 0 through size() void insertElementAt(int index, Type el);

// Return a reference to the element at the given index.

// precondition: index is on the range of 0 through size()-1

Type getElementAt(int index);

// Remove the element at the given index.

// precondition: index is on the range of 0 through size()-1 void removeElementAt(int index);

// Swap the element located at index with the element at index+1.

// Lower the priority of the element at index size()-1 has no effect.

// precondition: index is on the range of 0 through size() void lowerPriorityOf(int index);

// Swap the element located at index with the element at index-1.

// An attempt to raise the priority at index 0 has no effect.

// precondition: index is on the range of 0 through size() void raisePriorityOf(int index);

// Move the element at the given index to the end of this list.

// An attempt to move the last element to the last has no effect.

Programming Projects

// precondition: index is on the range of 0 through size()-1 void moveToLast(int index);

// Move the element at the given index to the front of this list.

// An attempt to move the top element to the top has no effect.

// precondition: index is on the range of 0 through size()-1 void moveToTop(int index);

To help you understand how these methods work, consider the program below which shows the changing list as each of the messages is sent to the list. *Recommended:* implement one method at a time, and write tests to ensure that it works.

#include <iostream>

#include <string> // Needed by Visual Studio

#include "PriorityList.h" using namespace std;

int main() {

PriorityList<string> list; list.insertElementAt(0, "a"); list.insertElementAt(1, "b"); list.insertElementAt(2, "c"); list.insertElementAt(3, "d");

for (int i = 0; i < list.size(); i++) // a b c d cout << list.getElementAt(i) << " "; cout << endl;

list.insertElementAt(1, "f");

for (int i = 0; i < list.size(); i++) // a f b c d cout << list.getElementAt(i) << " "; cout << endl;

list.removeElementAt(0);

for (int i = 0; i < list.size(); i++) // f b c d cout << list.getElementAt(i) << " "; cout << endl;

list.lowerPriorityOf(3); // no effect list.lowerPriorityOf(0); // move f right list.lowerPriorityOf(1); // move f right list.lowerPriorityOf(2); // move f right for (int i = 0; i < list.size(); i++) // b c d f cout << list.getElementAt(i) << " "; cout << endl;

list.raisePriorityOf(0); // no effect list.raisePriorityOf(2); // move d left list.raisePriorityOf(1); // move d left for (int i = 0; i < list.size(); i++) // d b c f cout << list.getElementAt(i) << " "; cout << endl;

list.moveToLast(list.size() - 1); // no effect

list.moveToLast(0); // move d from top priority to last priority

for (int i = 0; i < list.size(); i++) // b c f d cout << list.getElementAt(i) << " "; cout << endl;

list.moveToTop(0); // no effect

list.moveToTop(2); // move f to top priority again for (int i = 0; i < list.size(); i++) // f b c d cout << list.getElementAt(i) << " ";

return 0; }

## 11C PriorityList<Type> THROWS EXCEPTIONS

*Optional:* Change your code so it throws an exception when the index is out of range. To do this, fi rst add this #include to PriorityList<Type>:

#include <stdexcept>

Th en add an if statement to every method that takes index as a parameter. An exception will be thrown if the programmer supplies an incorrect index like -1 or an index > size(), which is a good thing:

// Insert the element at the given index.

// precondition: index is on the range of 0 through size() void insertElementAt(int index, Type element) { if (index < 0 || index > size()) { throw std::invalid\_argument(

"\ninsertElementAt: index must be 0..size()");

} // . . .

**C H A P T E R T W E L V E**

# Pointers and Memory Management

## SUMMING UP

Up until now, the memory needed to store all objects has been allocated behind the scenes. We access that memory with variables names or sending messages.

## COMING UP

Th is chapter introduces the notion of *indirection*. Indirection occurs when there is a substitute for something. Consider a library catalog card that holds the Dewey decimal number of a book. Th e card itself is not the book. Rather, the card is a reference to the book. Because the card names the location of the book, in a sense, the card contains an “address.” C++ has its own method for implementing indirection through *pointers*—variables that store addresses of, or pointers to, other variables. Th is chapter also introduces the primitive C array and memory management. After studying this chapter, you will be able to

* understand that pointer objects store addresses of other objects
* use primitive C++ arrays with no range checking
* use several methods for initializing pointers
* use the new and delete operators for memory management

### 12.1 MEMORY CONSIDERATIONS

Every object has a name, state, and set of available operations. Objects also have *scope*—where they are known—and *lifetime*—the length of time from when they are constructed to when they go out of existence. From initializations such as

int able = 123; int baker = 987;

most of these characteristics of the objects can be ascertained. However, the location of the object in memory—its address—is not so obvious. Until now, we have relied on the system to manage addresses. C++ allows programmers to manipulate those addresses directly.

Each object resides in a specifi c memory location, which is one or more bytes of computer memory. Each object is located by using the address of the fi rst byte in the object memory loca-

343

tion. For example, here is a machine-level view of values showing able stored at address 6300 and baker at address 6304. Th ese addresses are arbitrary and could be stored at any other address. Also with C++, ints are usually, but are not required to be, stored in four bytes of memory.

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | **Type** | **Name** | **State** |
| 6300 | int | able | 123 |
| 6304 | int | baker | 987 |

Th e object named able is shown to reside in the bytes 6300, 6301, 6302, and 6303. Th e address of able is the fi rst of those four bytes of memory, or 6300. Although we do not always need to know the exact addresses of objects, the concept of objects that store addresses eventually becomes important in the study of computing fundamentals with C++.

Th e memory allocated for many objects is determined at compile time. A char object might require one byte of memory, an int usually requires either two or four depending on the computer system, and a double object requires a specifi c and predictable (at least by machine) number of bytes. Th ese types are said to be *static* because the memory is allocated at compile time. Th e amount of memory allocated for a static variable is fi xed and will not change while the program is running.

Pointer objects allow programmers to write code that allows for runtime allocation of memory. Th e space is made available while the program is running. Th ese runtime-allocated objects are *dynamic* because they consume chunks of memory at runtime. Th e major benefi t is this: memory is allocated on an as-needed basis. Th e memory can also be deallocated, or returned, to the system when no longer needed so it can be used later.

Dynamic objects manage collections that may shrink and grow in size, where the size is limited only by available memory. Programmers can more eff ectively control computer resources. For example, string objects employ behind-the-scenes dynamic memory allocation that permits a runtime sizing of string objects. Th is implementation of the string class was chosen because there is no way to predict how many characters will be entered by the user at runtime:

string name; // Memory allocated during input cout << "Enter your name: "; cin >> name;

Th e string class also allows programmers to assign varying-length strings:

string a, b; // Appropriate memory is allocated on assignment

a = "The string a should have its own space"; // 38 chars b = "The string b should also"; // 24 chars

An alternative would be to allocate a vector of chars of arbitrary size for every string object during the call to the constructor. But what size should we use? We could pick a size large enough to accommodate most strings, but this would waste large amounts of memory. Imagine a vector of 1,000 strings in which each string is allocated 128 or 200 bytes of memory, even if the average length of the strings ends up to be only 9 characters. Without pointers, the programmer might be forced into this alternative of wasted computer memory. To understand memory management, pointers must be understood.

#### 12.1.1 POINTERS

*Pointers* store the addresses of other objects—they point *to* other objects. A pointer object is declared with an asterisk (\*) after the class name.

**General Form 12.1** *Declaring pointer variables*

*class-name*\* *identi er*;

Th e asterisk indicates that *identi er* can store the address of an object of type *class-name*. For example, in the declaration int\* intPtr;

the pointer object named intPtr may store the address of one int object. Th e object named intPtr does not represent an int—it represents the address of an int. A pointer object may have one of these states:

1. It may be undefi ned (garbage, as intPtr currently is).
2. It may contain the special pointer value nullptr, signifying the pointer points to nothing.
3. It may point to an instance of the class it was declared to point to.

Currently any attempt to use the undefi ned value of intPtr will result in undefi ned system behavior. One way to set the state of intPtr is to assign it the special pointer constant nullptr that means the pointer does not point to anything.

intPtr = nullptr; // intPtr points to nothing

Because pointer objects store addresses, their values become more meaningful when visibly written in a box with an arrow pointing to the object. So these statements

int anInt = 123; // Allocate memory for an int and initialize it int\* p; // Allocate memory to store the address of an int object

can be graphically represented as follows:

anInt p

|  |  |  |
| --- | --- | --- |
| 123 |  | ? |

Th e ? signifi es a pointer object that has not yet been assigned a value. Th e ? is a garbage value. To indicate the pointer is pointing to nothing, we use the C++ keyword nullptr:

p = nullptr;

When nullptr is assigned to the pointer p, the state of p could be pictured with a symbol such as the diagonal line show here:

p

Pointer objects may be assigned values through the *address of* operator &. Th e & operator returns the address of the object that follows it.

**General Form 12.2** *Obtaining the address of an object*

&*object-name*;

For example, the expression &anInt evaluates to the address of anInt. Th e following statement stores the address of anInt in the pointer object p (the expression &anInt is read as “address of anInt”):

p = &anInt; // &anInt returns memory location (address) of anInt

Th is assignment is best presented pictorially by moving the arrow from ? to the memory that holds the address of anInt:

p  123 anInt

|  |
| --- |
|  |

Th e arrow from p to anInt indicates that p is now pointing to the object anInt; however, the actual value stored in p is an address—the memory location of anInt.

Th e state of the object pointed to by a pointer object can be altered indirectly. For example, the state of anInt can be changed without even using the object name. Th is *indirect addressing* with the dereference, or indirection, operator \* allows the program to inspect or change the memory pointed to by the pointer object. Here is an example of how the memory for the anInt pointed to by p may be altered:

= 456; // Indirect addressing stores 456 in anInt

p

456

anInt

**General Form 12.3** *Indirect addressing*

\**pointer-object*;

Th e assignment to \*p does not change p. Instead, it changes the state of the object pointed to by p.

Note that the \* has two diff erent meanings for a pointer. First, when you declare a pointer, the asterisk means you are declaring a pointer, for example:

int\* pInteger; double\* pDouble;

Second, when you use the asterisk with an existing pointer, it means you are dereferencing the pointer:

\*pInteger = 456;

\*pDoube = 123.45;

In math, \* signifi es multiplication. As you can see, the asterisk is truly an overloaded operator. Pay close attention to how it is used in your code to determine the meaning of the asterisk. Th e asterisk that precedes a pointer object tells the pointer to go to the address that the pointer is storing and change the value at that address. So for example, if anInt were stored at address 6308, 6308 is stored in p:

p 6308 456 anInt \*p

To illustrate the diff erences between p, \*p, and &anInt, consider the indirect addressing method used in the following program that interchanges the value of two pointers. By the end of the program, the two pointer objects p1 and p2 point to each other’s original int object. Note that since the pointers are pointing to double values, the pointers must be declared as double pointers. Th is tells the compiler to go to the address stored in the double pointer and read enough bytes for a double object (usually 8 bytes).

// Interchange two pointer values. The pointers are switched // to point to the other's original int object.

#include <iostream> using namespace std;

int main() { double\* p1; double\* p2; double\* temp; double n1 = 99.9; double n2 = 88.8;

// Let p1 point to n1 and p2 point to n2 p1 = &n1; p2 = &n2;

cout << "\*p1 and \*p2 before switch" << endl;

// Get the integers indirectly with the \* operator

cout << (\*p1) << " " << (\*p2) << endl;

// Swap the pointers by letting p1 point to where p2 is pointing.

// Also let p2 point to where p1 is pointing.

temp = p1; p1 = p2; p2 = temp;

// Now the values of the pointers are switched to point to each

// other's int object. The ints themselves do not move.

cout << "\*p1 and \*p2 after switch" << endl; cout << (\*p1) << " " << (\*p2) << endl << endl;

cout << "Actual memory locations in hexadecimal:" << endl; cout << p1 << " " << p2 << endl;

return 0; }

##### Output

\*p1 and \*p2 before switch

99.9 88.8 \*p1 and \*p2 after switch

88.8 99.9

Actual memory locations in hexadecimal:

0x7fff5d00cbf0 0x7fff5d00cbf8

Th e values 99.9 and 88.8 were not moved in memory. Instead, the pointers to these double objects were interchanged. Th e following graphic representation traces this program execution. First, all fi ve objects are initialized as follows. *Note:* All boxes represent memory storing the state of an object.

n1n2

|  |  |  |
| --- | --- | --- |
| 99.9 |  | 88.8 |

p1p2temp

|  |
| --- |
| ? |

|  |  |  |
| --- | --- | --- |
| ? |  | ? |

Th e next two statements (p1 = &n1; and p2 = &n2;) store the addresses of the doubles in the pointers. Th e statement temp = p1; means that the pointer object temp is set to point to the same memory location as p1.

// Both temp and p1 are pointers to an integer

|  |
| --- |
| p1 = &n1; p2 = &n2; temp = p1; |

99.9

88.8

n2

n1

p1

p2

temp

Th e address (shown as arrows) of p1 was stored in temp. At this point the expression temp == p1 would be true. Th is change is indicated by the fact that arrows from both p1 and temp point to the same location—the object named n1.

Next, the assignment p1 = p2; causes p1 to point to the same place as p2. So p1 and p2 now store the same address. Th e two pointer values are equal.

99.9

88.8

n1

p1

n2

p2

temp

p1 = p2;

And fi nally, p2 = temp; causes p2 to point to the same double to which p1 was originally pointing.

p2 = temp

99.9

88.8

n2

n1

p2

p1

temp

Now that p2 points to n1 and p1 points to n2, cout << (\*p1) displays 88.8 rather than the original 99.9.

At fi rst, working with pointers is not easy. It takes a shift from understanding objects that store values to understanding objects that store addresses of other objects that store values. Algorithm design and debugging are diff erent when using pointers. One low-cost tool that helps during debugging is the use of arrows to represent pointer values. Algorithms are traced by moving the arrow rather than writing the address.

Also when writing debugging code, the value being pointed to is usually more telling than the address of where that object is located. So debug with \* as in cout << (\*aPointer); rather than cout << aPointer;. With this, you see the more useful values of the objects, not their addresses.

#### 12.1.2 POINTERS TO OBJECTS

Pointers to ints and doubles refer to one single value stored in those locations. Th ere are no associated member functions. Now consider what happens with a pointer to an object during a message. Because the dereference operator has a lower priority than a function call, this code will not work:

BankAccount anAcct("Functions > Dereference", 123.45);

BankAccount\* bp; bp = &anAcct; \*bp.deposit(123.45); // ERROR

One way to fi x this is to override the priority scheme by wrapping the pointer dereference in parentheses. Now \*bp returns the BankAccount object *before* the deposit function gets called:

(\*bp).deposit(123.45); // OKAY

Or you could use the C++ arrow operator -> as a shortcut to denote that the pointer is pointing to an instance of a true class:

bp->deposit(123.45); // SHORTCUT

Both techniques are used in the following program:

#include <iostream> using namespace std; #include "BankAccount.h"

int main() {

BankAccount anAcct("both (\*bp) and bp-> work ", 100.00);

BankAccount\* bp; bp = &anAcct;

// Wrap the dereference in parentheses because the dereference

// operator \* has lower precedence than function calls

(\*bp).deposit(123.45);

cout << (\*bp).getName() << (\*bp).getBalance() << endl;

// Use -> for pointers to objects other than int or double bp->withdraw(111.11);

cout << bp->getName() << bp->getBalance() << endl;

return 0; }

##### Output

both (\*bp) and bp-> work 223.45 both (\*bp) and bp-> work 112.34

#### SELF-CHECK

12-1 What do pointer objects store?

12-2 Use these statements to answer the questions below:

double\* doublePtr; double aDouble = 1.23; doublePtr = &aDouble;

1. What is the name of the pointer object?
2. What is the value of doublePtr?
3. What is the value of \*doublePtr?
4. Write code to *indirectly* change the value of aDouble from 1.23 to 2.23.
   1. What is the value of \*ptr after this code runs:

int anInt = 123; int\* ptr = &anInt; \*ptr += \*ptr;

* 1. What is the value of s3 after this code runs:

string s1 = string("one"); string\* p1 = &s1; string s3 = p1->c\_str(); s3 += p1->c\_str(); cout << s3;

* 1. Write an expression that evaluates to the sum of the two BankAccount balances indirectly.

BankAccount ba1("one", 100.00);

BankAccount ba2("two", 200.00);

BankAccount\* a = &ba1;

BankAccount\* b = &ba2;

* 1. Write the output generated by the following program:

#include <iostream> using namespace std; int main() { int\* p; int j = 12; p = &j;

cout << ((\*p) + (\*p)) << " " << ((\*p) \* (\*p)) << endl; return 0; }

* 1. Write statements that store the address of ch, a char object, into a char pointer object named charPtr.
  2. Write the minimum code that declares and initializes all the objects as shown in the diagram below:

n1n2n3 p1p2p3

12

34

56

* 1. Using the code from your answer to the previous question, write a statement that indirectly displays the sum of all the integers using \* (the dereference operator).
  2. Write the output generated by the following code:

int p = 111; int\* q = &p; p += 222;

cout << "p? " << p << endl; cout << "q? " << \*q << endl;

* 1. Write the output generated by the following code:

int n1 = 4; int n2 = 8; int\* ptr1; int\* ptr2; ptr1 = &n1; ptr2 = &n2;

cout << (\*ptr1) << " " << (\*ptr2) << endl; 12-12 Write the output generated by the following code:

double\* p = new double; double\* q = new double;

\*p = 1.23; \*q = 4.56; p = q;

cout << (\*p) << " " << (\*q);

12-13 In the preceding code of 12-12, is it possible to retrieve the value 1.23 by modifying the last line?

### 12.2 THE PRIMITIVE C ARRAY

Th e vector class is a relatively new addition to C++. In the past, the built-in, primitive C array was frequently used to store collections of objects. Because an array actually stores the address of the fi rst element, it is a useful example for illustrating pointer usage. Th e C array is used so frequently for implementing programs that you are likely to see it in existing code. But more importantly, the primitive C array illustrates the benefi ts of dynamic memory management. It

12.2: The Primitive C Array

provides a peek under the hood of vector::resize and string assignments. Both use pointers and dynamic allocation to better manage memory.

Th e primitive C array is a fi xed-size collection of elements that are of the same class. Arrays are homogeneous because they store collections of like objects. Th e objects in the collection may be one of the built-in classes char, int, long, or double. Th e objects may also be declared as a programmer-defi ned class such as BankAccount as long as the class has a default constructor. Here is the general form for declaring a primitive C array:

#### General Form 12.4 *Array declaration*

*type* *array-name*[*capacity*];

Th e *type* specifi es the type of objects stored under *array-name*. Th e *capacity* specifi es the maximum number of elements that can be stored under the array name. Th e capacity must be an integer constant (such as 100) or a named integer constant. An array cannot be sized or resized at runtime as a vector object can, at least not as a standard operation. Th e array shown next stores a maximum of one hundred:

double x[100];

Individual array elements are referenced through subscripts in the same manner seen with vector objects:

**General Form 12.5** *Referencing individual array elements*

*array-name*[*int-expression*];

Th e subscript range of a primitive array is the same as a vector going from 0 through capacity - 1.

#### 12.2.1 DIFFERENCES BETWEEN PRIMITIVE ARRAYS AND vectors

Th ere are many similarities between arrays and vectors—especially in the referencing of individual elements. In fact, the same vector-processing algorithms of chapter 10, “Vectors,” could also be applied to primitive C arrays. Th e most noticeable diff erence is that primitive C arrays cannot be made to automatically check for out-of-bounds subscripts. Th is is one of the drawbacks of the C array. It is safer to have the subscript range-checking feature available, especially when fi rst learning about arrays and vectors.

Some very strange errors occur when the code lets the computer “walk off ” the end of an array. Th e important state of other objects may be accidentally destroyed. With the subscript range checking of vector objects, the program can notify the programmer whenever there is an attempt to reference out-of-bounds memory. Th is can be a preferable situation.

Here are the diff erences between the vector class and primitive C arrays:

|  |  |  |
| --- | --- | --- |
| **Diff erence** | vector **Example** | **C Array Example** |
| vectors can initialize all vector elements at construction; arrays cannot. | vector <int> x(100, 0); // All elements are 0 | int x[100]; // Elements are garbage |
| vectors can be easily resized at runtime; arrays take a lot more work. | int n; cin >> n;  x.resize(n); | // See growing an array  // in a later chapter |
| vectors can be made to prevent out-of-range subscripts. | // You are told  // something is wrong cin >> x.at(100); | // Destroys other variables cin >> x[100]; |
| vectors require an #include; primitive, built-in arrays do not. | #include <vector> | // No #include required |

#### 12.2.2 THE ARRAY / POINTER CONNECTION

It turns out that all primitive C array variables actually store the pointer to—or the address of— the fi rst array element. Whenever the subscript operator is applied to an array object, an address is computed. For example, if x is an array of integers, each array element is four bytes long, and x has the value of address 6000, the following formula computes x[3] as 6000 + (3 \* 4).

##### Formula for Computing the Address of Individual Array Elements

*address of rst array element* +(*subscript* \* *size of one element*)

Th erefore, x[3] is stored at address 6012.

|  |  |  |
| --- | --- | --- |
| **Reference** | **Address** | **Value** |
| x[0] | 6000 | ? |
| x[1] | 6004 | ? |
| x[2] | 6008 | ? |
| x[3] | 6012 | ? |
| x[4] | 6016 | ? |

#### 12.2.3 PASSING PRIMITIVE ARRAY ARGUMENTS

When an array is passed to a function, the address of the fi rst array element gets sent. Arrays are automatically passed by reference. An array parameter is declared with the class and the parameter name followed by [ ]. Th is is illustrated in the next program where main() passes an array to the

12.2: The Primitive C Array

function init. Notice that when the function init alters the array parameter x, the associated array argument anArray is also altered. Both x and anArray have the fi rst three array elements assigned a value (90, 95, 99). Th is occurs even though & is not used for the parameter anArray. anArray is passed by value.

// Pass the address of the array to a function.

// The & is not required. An array stores an address.

#include <iostream> using namespace std;

void init(int x[], int & n) {

// x and n are reference parameters; however, x does not need & x[0] = 90; x[1] = 95; x[2] = 99; n = 3; }

int main() { int n = 5; int anArray[5];

init(anArray, n); // init will change x and anArray for (int index = 0; index < n; index++) { cout << anArray[index] << " ";

} cout << endl;

return 0; }

##### Output

90 95 99

Since the value stored in the array name is an address—a reference to the fi rst array element, passing the array name to a function actually passes an address. Th erefore, arrays are automatically passed by reference, which is the address of the fi rst array element. Subscripting parameter x results in the equivalent of subscripting argument anArray. Th e following fi gure depicts that anArray was passed by reference, even though & was not used:

void init(int x[], int & n) { x

|  |  |  |
| --- | --- | --- |
| 90 95 99 | ? | ? |

} // a change to x is a change to anArray in main

int main() { anArray

### 12.3 ALLOCATING MEMORY WITH new

Pointer objects are frequently assigned values through the new operator. When the new operator precedes a class name, the resulting expression allocates a contiguous block of memory large enough to store one instance of that class. Additionally, the same expression returns the address, or a pointer to, this memory.

**General Form 12.6** *Dynamic memory allocation (for one object only)*

new *class-name*;

Th e memory is allocated at runtime from the *free store*—a portion of computer memory reserved for this purpose (the free store is sometimes called the *heap*). For example, the following expression allocates enough memory to store one int value. Th e expression returns a pointer to that memory.

new int; // Allocate memory, return a pointer value (an address)

Instead of ignoring the returned pointer value (the address where an integer could be stored), such pointer expressions are usually combined with pointer objects in initializations.

int\* intPtr = new int; // Allocate memory, store address in intPtr Th e above is an abbreviated form of the following equivalent code:

int\* intPtr; intPtr = new int; // Allocate memory, store address in intPtr

Now we have a situation where intPtr holds the address of an int object—where an int could be stored. Th is is shown in the next fi gure where the undefi ned int value is signifi ed as ? and the pointer value is represented as an arrow indicating a value that points to that undefi ned int:

intPtr

|  |
| --- |
| ? |

Th is statement initializes that new allocation of memory:

\*intPtr = 123;

Th is resulting representation shows the state of the pointer and the int:

intPtr 123

Th e following program shows dynamic allocation of one int object:

// Illustrate one pointer object and one int object

#include <iostream>

12.3: Allocating Memory with new

using namespace std;

int main() {

// Declare an intPtr as a pointer to an int int\* intPtr;

// Allocate memory for an int and store address in intPtr intPtr = new int;

// Store 123 into memory referenced by intPtr

\*intPtr = 123;

cout << "\n The address stored in the pointer object: " << intPtr; cout << "\nThe value of the int pointed to by intPtr: " << \*intPtr;

return 0; }

#### Output (address shown in hexadecimal (a is 10, f is 15)

The address stored in the pointer object: 0x7fbd3bc04a20

The value of the int pointed to by intPtr: 123

Notice that the pointer object, with value 25,360 (0x7fbd3bc04a20 hexadecimal), is referenced as intPtr. Th e actual int with value 123 is dereferenced as \*intPtr.

#### 12.3.1 ALLOCATING MEMORY FOR ARRAYS AT RUNTIME

At times it is convenient to allocate arrays at runtime, when a maximum capacity is better known. Th e C++ new operator accomplishes this by allocating memory for many objects with [*capacity*], where *capacity* represents the number of objects to allocate.

**General Form 12.7** *Dynamic memory allocation (capacity objects)*

new *type*[*capacity*];

*Example:* allocate memory for 10 integers

new int[10]; // Allocate memory for 10 integers and return // a pointer to this newly allocated memory

Because new returns a pointer to the fi rst byte of the array, it can be used for pointer object initialization with this shortcut:

**General Form 12.8** *Initializing pointer objects*

*type*\* *identi er* = new *class-name*[*number of elements*];

*Example:*

int\* nums = new int[10];

Now the pointer object nums points to the fi rst of 4 \* 10 bytes of uninitialized (garbage) memory where each 4 bytes stores one integer:

nums

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ? | ? | ? | ? | ? | ? | ? | ? | ? | ? |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] |

One time you might fi nd this dynamic allocation of memory useful is when you have an array of objects set to an initial capacity, and then at runtime, you need to store more than the maximum capacity. Th is can be done with the following algorithm:

* Make a temporary array that is bigger than the instance variable.
* Copy the original contents (num[0] through nums[n - 1]) into this temporary array.
* Assign the reference to the temporary array to refer to the original array.

// This code dynamically (at runtime) "grows" an array

#include <iostream> using namespace std;

int main() { int n = 10;

int\* nums = new int[n]; // Some C++ compilers can not handle int[n] int anInt = 1;

// Initialize n array elements with a for loop for (int i = 0; i < n; i++) { nums[i] = anInt; anInt += 3; }

// Show the lled array for (int i = 0; i < n; i++) { cout << nums[i] << " "; }

// Need more room? Grow the array at runtime

int\* temp = new int[n+5]; // Some C++ compilers can not handle int[n+5]

// 2) copy the elements to the temporary array for (int i = 0; i < n; i++) { temp[i] = nums[i]; }

// Make the original array pointer refer to the "bigger" array nums = temp;

12.3: Allocating Memory with new

// Add 3 more elements to the bigger array nums[n++] = 997; nums[n++] = 998; nums[n++] = 999;

// Print the larger array with the added elements cout << endl << "Larger array" << endl; for (int i = 0; i < n; i++) { cout << nums[i] << " ";

} return 0; }

##### Output

1 4 7 10 13 16 19 22 25 28

Larger array

1 4 7 10 13 16 19 22 25 28 997 998 999

Here are the arrays through pictures—fi rst the original array fi lled to capacity:

nums

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 9 | 12 | 15 | 18 | 21 |
| [0] | [1] | [2] | [3] | [4] |

Th is is the new array with twice the capacity of the original:

temp

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | |

Finally, nums is made to point to the new array, the same one referenced by temp with this one assignment statement:

nums = temp;

After three integers are added to the larger-capacity array, we have this situation:

temp

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | ? | ? | ? | ? | ? | ? | ? | ? | ? | ? | | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | |

nums

#### SELF-CHECK

12-14 Write the output of the following code:

int\* x = new int[10]; x[0] = 4; x[1] = 8;

cout << x[0] + x[1] << endl;

12-15 Write one initialization using new to allocate an array that can store 1000 doubles.

12-16 Write the code that initializes all 1000 doubles of the previous question to -1.

12-17 Write the output generated by the following code:

const int MAX = 6; int\* x = new int[MAX];

for(int i = 0; i < MAX; i++) { x[i] = 2 \* i;

}

for(int i = 0; i < MAX; i++) { cout << x[i] << " "; }

12-18 Primitive arrays can be declared and initialized at the same time with array initializers such as the following:

int x[] = {3, -4, -3, 6, 1}; int n = 5;

Write the code that fi nds the range of the array elements in x. Range is defi ned as the largest minus the smallest integer. Your code must work for arrays that have array initializers with diff erent values and capacities.

12-19 Declare an array of strings with an array initializer that has the following strings in this order: "a", "b", "c", and "d".

### 12.4 THE delete OPERATOR

So far, the new operator examples allocated only small amounts of memory. However, consider what happens when dynamic data grows to a large size. Using new without returning memory to the free store results in a *memory leak*. Th is limits the amount of memory available to a program. At some point, a program no longer needs dynamically allocated memory. When this occurs, the unneeded memory should be allocated back to the free store. Th is makes it available for other

12.4: The delete Operator

objects that have yet to be dynamically allocated. Th is return of memory, or *deallocation*, is accomplished with the C++ built-in *delete* operator. Th e delete operator has two general forms:

**General Form 12.9** *Deallocating memory—a form of recycling*

delete *pointer-object*; delete[] *pointer-to-array*;

Th e fi rst form returns the memory allocated for one dynamic object back to the free store. Th e second form returns memory allocated for a group of objects with new and [ ]. In the following program, the delete operator allocates enough memory for one double pointed to by p, ten chars pointed to by charArray, and 100 integers pointed to by x.

// Allocate and deallocate memory at runtime

#include <iostream> using namespace std;

int main() { int\* p = new int; \*p = 123;

int\* x = new int[10000]; // claim 40,000 bytes from the free store x[0] = 76; x[1] = 89;

// ... x[9999] = \*p;

// When no longer needed, free the memory to avoid memory leaks delete p; delete[] x; // All the bytes of memory pointed to by p and x can be allocated later

return 0; }

After the two delete statements return the allocated memory back to the free store, the pointers should not be used. Using them at this point results in unpredictable behavior. Our programs with arrays should use delete to return the memory no longer needed. Th ere are no notifi cations or errors shown. Instead, we get memory leaks, which means memory no longer needed cannot be recycled later.

double\* temp = new double[n+5]; for (int i = 0; i < 10; i++) { temp[i] = nums[i]; } delete[] nums; // Avoid a memory leak by freeing up memory

**12.5 THE SINGLY LINKED STRUCTURE**

### WITH C structs

A singly linked data structure is an alternative way to store a collection of elements in a sequential fashion. Th ere is high probability that your standard C++ list class has been implemented using a linked structure with some of the concepts presented in this section.

Instead of having elements stored in contiguous memory, this singly linked structure will have a collection of linked node objects where each node stores an element and a link to the next node in the sequential collection. We also need a pointer to the fi rst node, which is named rst here.

To accomplish this, a pointer data member is added to a class or a struct. A struct is the same thing as a class except that by default, members of a class are private and members of a struct are public. However, if you use public and private explicitly, there is no diff erence other than the name. Th e struct is presented here for historical reasons and because structs typically have constructors and data members only. Because struct data members are public by default, adding public: is not necessary.

Casey

first

Kim

Chris

An example struct with two public data members and a start to LinkedList:

#ifndef LINKEDLIST\_H\_

#de ne LINKEDLIST\_H\_

/\*\*

* This le contains two types:

\*

* 1) struct node to hold an element and a link to another node
* 2) class LinkedList to hold an indexed sequential collection using
* the singly linked data structure

\*

* A LinkedList can only store string elements. Templates are not \* used here to allow focus on pointers and memory management.

\*/ struct node {

// Two public data members std::string data; node\* next;

// Two public constructors node() {

next = nullptr; }

node(std::string element) { data = element; next = nullptr; }

}; // class LinkedList will go here . . .

#endif /\* LINKEDLIST\_H\_ \*/

Th e following code constructs a new node object pointed to by rst and displays the value using the -> operator, which is necessary for dereferencing the public data members of node.

#include <iostream>

#include <string>

#include "LinkedList.h" using namespace std;

int main() {

// Let nodePointer reference a dynamically allocated node object node\* rst = new node("Kim"); // assert: nodePointer->next == nullptr

// Display the state of the public data member my\_data cout << " The value: " << rst->data << endl; cout << "#characters: " << rst->data.length() << endl; }

**Output:**

The value: Kim

#characters: 3

Here is a representation of what this looks like in memory:

Kim

first

A linked structure has the characteristic that one element can be referenced from another element. With a data member to store a pointer to another object of the same class, objects can be linked together in such a way that the pointer in the fi rst node object can be used to fi nd the second node. Th e following code constructs three node objects that are linked together. Notice that a reference to the data of the second node is made using the pointer p >next.

// Build the rst node node\* p = new node("One");

// Construct a second node pointed to by the rst node's next p->next = new node("Two");

// Build a third node pointed to by p->next->next p->next->next = new node("Three");

Here is a representation of what three linked nodes look like in memory:

Casey

Chris

p->next

p.next.next

p.next.next.next

p->data

p.next.data

p.next.next.data

Kim

p

Th ese three nodes can be traversed by allowing a pointer, named ptr here, to refer to all three nodes. It begins by having ptr point to the fi rst node. If ptr is not nullptr, the node’s data is displayed (inside the loop).

// Traverse the nodes until a next eld is nullptr node\* ptr = p; // Don't change p, which is a pointer to the rst node while(ptr != nullptr) { cout << ptr->data << endl; ptr = ptr->next; }

ptr is updated to point to the next node or it is set to nullptr at the end of each loop iteration with the statement ptr = ptr >next.

#### 12.5.1 A LIST CLASS USING THE SINGLY-LINKED DATA STRUCTURE

Th is section describes LinkedList member functions that use these node objects. Th e constructor establishes an empty list using a dummy header node. Th is makes the coding easier during add and remove.

class LinkedList {

private:

node\* header; node\* last; int n; public:

//--constructor

LinkedList() {

// Create a dummy header node to make things easier header = new node; // call node's default constructor last = rst; n = 0; }

Here is a representation of what an empty list with a dummy fi rst node looks like in memory:

header

last

##### 12.5.2 add(std::string)

Adding an element to a linked list has diff erent meanings for ordered and unordered lists. An ordered list stores objects in an ascending order based on the meaning of <. Th e linked list developed here is not ordered so the elements will not be in alphabetic order. Since this linked list here is unordered, all new elements can be added at the very end of the list. Th is is easy when a dummy headed node is employed to avoid the special case of adding to an empty list or removing an element. Elements are added by creating a new object pointed to by last->next. Th e member data last must then be updated to point to the last node. Th e current count must also be incremented.

void add(const std::string newElement) { // Allocate and initialize a new node last->next = new node(newElement);

// Update the last pointer last = last->next;

// Maintain current size n++; }

Th is one adds message results in the pictures of memory shown below:

LinkedList stringList; // n == 0 stringList.add("First"); // n == 1

First

header

last

stringList.add("Second"); // n == 2

stringList.add("Third"); // n == 3

Third

First

Second

header

last

##### 12.5.3 get(intindex)

Th e get operation uses a for loop to advance an external pointer ptr to the correct node. Notice that if index is 0, the for loop does not advance ptr, leaving it to point to the fi rst real node— the one pointed to by header->next, which would be the value " rst".

std::string get(int index) { node\* ptr = rst->next; for (int i = 0; i < index; i++) { ptr = ptr->next;

}

return ptr->data; }

##### 12.5.4 remove(string removalCandidate)

Th ese two possibilities must be considered when removing an element from a linked list:

1. the == operation does not match an element in the list
2. the == operation does match an element in the list

Th e search for a particular element in a linked list is similar to a sequential search through a vector or array. Th e diff erence is that now, instead of a subscript, a pointer will be used to access the data members.

Th e search for "Second" begins by pointing a variable named ptr to the fi rst element in the list. With the node header, we can peek one node ahead in the search while maintaining a pointer to the node that precedes the node to be removed.

Third

First

Second

header

ptr

last

ptr = header;

A sequential search continues until ptr->next->data equals the removalCandidate or there are no more elements to search. Since ptr->data == removalElement ("First" == "Second") is false, the loop advances ptr to the next node in the list.

Third

First

Second

header

last

ptr = ptr->next;

ptr

Now the node pointed to by ptr points to the node before the node to be removed: ptr->next->data == "Second" is true now. With the help of the dummy header node, this algorithm is able to peek at the data one node ahead. Th is comes in handy as we need to send a pointer around the node to be deleted with ptr->next = ptr->next->next.

First

Third

header

last

ptr->next = ptr->next->next;

ptr

Now the node pointed to ptr->next can safely be returned to the free store with delete.

bool remove(const std::string removalElement) {

// Create an external pointer to point to the node before the rst node node\* ptr = header;

// Search the remaining list elements until

// found or the end of the list is found

while (ptr->next != nullptr && ptr->next->data != removalElement) { ptr = ptr->next; }

// Don't delete a nonexistent node

if (ptr->next == nullptr) { // removalElement was not found return false;

} else {

// Check if the last node is being removed so last gets corrected if(ptr->next == last) { last = ptr;

}

// Send the link around the node to be removed ptr->next = ptr->next->next; if (ptr != header)

delete ptr->next; // Deallocate memory n--; // Maintain current size return true; // Report successful removal

}

}

If the last node is to be removed, last must be adjusted to the preceding node. If ptr points to the last node, the element was not found so remove returns false.

#### SELF-CHECK

12-20 Draw a picture of a linked list with two nodes before and after removing the fi rst node.

12-21 Draw a picture of a linked list with one node before and after removing the fi rst node.

12-22 What happens if removalCandidate is not found in the list?

12-23 True or False

Chapter Summary

1. Th e size of a dynamically linked list must be determined before the program begins to execute.
2. Elements in a linked list are referenced through subscripts.
3. Elements may be inserted into or deleted from a linked list at the beginning, end, or even the middle of a linked list.
4. When an element is to be inserted into or deleted from a linked list, the list should be checked to see if it is empty.
5. When an element has been removed from a dynamically linked list, the memory it used should be returned to the free store.

12-24 Write method bool removeLast() to remove the last element in a LinkedList. Return false if the list is empty. Th e program should generate the output shown in comments.

#include <iostream>

#include <string>

#include "LinkedList.h" using namespace std;

int main() {

LinkedList list;

cout << list.removeLast() << endl; // 0 list.add("A");

cout << list.removeLast() << endl; // 1 list.add("B"); list.add("C"); list.add("D");

cout << list.removeLast() << endl; // 1 list.add("E"); cout << list.get(0) << " "; cout << list.get(1) << " "; cout << list.get(2) << endl; // B C E cout << list.size() << endl; // 3 return 0; }

### CHAPTER SUMMARY

* Pointers store addresses of other objects. A pointer object points to some object. For example, ptr is a pointer and \*ptr is a reference to the double object x that starts as 99.9.

double \* ptr; double x = 99.9; ptr = &x;

\*ptr = 1.234;

In the last statement, the pointer changes the value stored in x to 1.234. Th e \* (asterisk) dereferences the pointer. Th is means the pointer goes to the address stored in the pointer, which in this case is the address of x and changes the value stored at that address. Th erefore, the pointer changes the value stored in x indirectly.

* Th e address of a variable is the fi rst address at the location where the state is stored. If it takes 4 bytes to store an int, the address of the int is the address of the fi rst byte of the int value, which is the address stored in the pointer. Th e pointer knows when to stop reading addresses because the pointer was declared as an int pointer. Th erefore, it reads 4 bytes (this may vary depending on your computer system).
* Th e address operator & gives us the address of a variable.
* Th e primitive C array—similar to the C++ vector class—is available on all compilers and will often be seen in existing C and C++ code.
* Th e new operator allocates memory from the free store. Th e delete operator deallocates memory. If more than one object is allocated as in char\* name = new char[10]; it must be deallocated with [ ] like this: delete [] name;
* You can allocate memory at runtime with new and return it to the free store with delete.

### PROGRAMMING TIPS

1. Draw linked structures when debugging programs with pointers. Th e value of a pointer object represents a location in the memory of the computer. Th ese values are diffi cult to use in a program trace. A diagram with arrows and boxes makes execution simulation and pointer debugging much clearer.
2. Pointers allow dynamic allocation of arrays. One problem when using arrays involves how big to make them at compile time. It may be big enough one time, but not another. Sometimes memory gets wasted when declared too big.
3. Use vectors instead of arrays. Th e standard vector type can be dynamically grown, or shrunk even, at runtime with resize messages. Let this well-tested class do the work for you.
4. Avoid memory leaks. Use the delete operator to return the memory back to the free store for single variables. Use the delete[] operator to free an array of values.

### EXERCISES

1. Write the values of the attributes supplied by this initialization:

double x = 987.65;

Exercises

|  |  |
| --- | --- |
| a. class | c. state |
| b. name | d. address |

1. Declare a pointer to an int and initialize the pointer somehow.
2. Use these statements to answer the questions that follow:

int\* intPtr; int anInt = 123; intPtr = &anInt;

* 1. What is the name of the pointer object?
  2. What is the value of \*intPtr?
  3. Without using anInt, write a statement that adds 100 to the memory storing 123.

1. Write the minimum declarations and statements that declare and initialize all the objects as they are shown in the diagram below.

a

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 7 | b | 11 | c | 8 |

p1 p2 p3

1. Using your code from the previous question, write the statements that will have a pointer object named largestPtr pointing to the largest integer no matter where it is stored among a, b, and c.
2. Using the declarations shown, which of the following are valid assignments that do not generate an error?

int j = 456; int\* p;

|  |  |
| --- | --- |
| a. p = j | g. p = &p |
| b. p = &j | h. p = 123 |
| c. p = 0 | i. \*p = "abc" |
| d. j = p | j. \*j = 123 |
| e. j = 123 | k. j = &p |
| f. \*p = j | l. \*p = \*p |

1. Write the output generated by this code:

int \* intPtr; int anInt = 987; intPtr = &anInt;

\*intPtr = \*intPtr + 111; cout << \*intPtr << " " << anInt;

1. Trace the following program segment by drawing pictures of the modifi ed objects:

n1 = 123; p1 = &n1; \*p1 = \*p1 + 111;

1. Trace the following program segment by drawing pictures of the modifi ed objects:

n2 = 999; p3 = &n2; p2 = p3;

1. Trace the following program segment by drawing pictures of the modifi ed objects:

int \* intPtr; intPtr = p3;

**PROGRAMMING PROJECTS**

## 12A ENHANCE LinkedList

Add two methods to the LinkedList class:

1. void toString() to return a string containing all elements with 10 elements separated by a new line “\n”.
2. void insertInOrder(std::string element) to insert string elements into the singly linked structure while maintaining alphabetical ordering.

## 12B CLASS LinkedStack WITH A SINGLY LINKED STRUCTURE

Implement class LinkedStack which allows elements to be added and removed in a last-in, fi rstout (LIFO) manner. Th is class must use a singly-linked structure to store the elements. Stacks have an operation called push to place elements at the “top” of the stack and another operation called pop to remove and return the element at the top of the stack. Th e only element on the stack that may be referenced is the one on the top. Th is means that if two elements are

Programming Projects

pushed onto the stack, the topmost element must be popped (removed) from the stack before the fi rst-pushed element can be referenced. Here is a stack program for storing strings:

#include <iostream>

#include <string> // Needed by Visual Studio

#include "LinkedStack.h" using namespace std;

int main() { LinkedStack stack; // stack of 20 strings

// Use intStack stack.push("a"); stack.push("b"); stack.push("c"); stack.push("d");

cout << "d? " << stack.peek() << endl; cout << "d? " << stack.pop() << endl; cout << "c? " << stack.peek() << endl;

cout << "isEmpty 0? " << stack.isEmpty() << endl; cout << "c b a? "; while(! stack.isEmpty()) { cout << stack.pop() << " ";

} cout << endl;

cout << "isEmpty 1? " << stack.isEmpty() << endl;

return 0; }

Using the class LinkedStack that you generate, this program must compile and generate the following output:

### Output

d? d d? d c? c isEmpty 0? 0 c b a? c b a isEmpty 1? 1

LinkedStack must use a singly linked structure. It is recommended you keep a pointer to the top of the stack and push new elements onto the front. If top == nullptr, you can have top refer to a new node with the element. When the stack is not empty, you can add to the fi rst so top references the most recently added. In this case, if you have pushed "First" then stack.push("Second") you could use code shown here in pictures of memory.

"First"

top

temp

"First"

"Second"

top

temp

### 12C LinkedPriorityList

Th is project asks you to implement a collection class LinkedPriorityList using a singly linked structure to store a sequence of string objects (no templates). Th is new type will store a collection of elements as a zero-based indexed list where the element at index 0 is considered to have higher priority than the element at index 1. Th e element at index size()-1 has the lowest priority. An instance of this collection class will be able to store just one type of element such as <string>.

#### Output

Sleep

Get groceries

Study for the CS exam

Complete these methods in LinkedPriorityList so it uses a singly linked structure to store elements. Don’t forget to add struct node in the same fi le as this class.

// Construct an empty LinkedPriorityList

LinkedPriorityList();

// Return the number of elements currently in this LinkedPriorityList int size();

// Return true if size() == 0 or false if size() > 0 bool isEmpty();

// Insert the element at the given index.

// precondition: index is on the range of 0 through size() void insertElementAt(int index, std::string el); // Return a reference to the element at the given index.

Programming Projects

// precondition: index is on the range of 0 through size()-1 std::string getElementAt(int index);

// Remove the element at the given index.

// precondition: index is on the range of 0 through size()-1 void removeElementAt(int index);

// Swap the element located at index with the element at index+1.

// Lower the priority of the element at index size()-1 has no effect.

// precondition: index is on the range of 0 through size() void lowerPriorityOf(int index);

// Swap the element located at index with the element at index-1.

// An attempt to raise the priority at index 0 has no effect.

// precondition: index is on the range of 0 through size() void raisePriorityOf(int index);

// Move the element at the given index to the end of this list.

// An attempt to move the last element to the last has no effect.

// precondition: index is on the range of 0 through size()-1 void moveToLast(int index);

// Move the element at the given index to the front of this list.

// An attempt to move the top element to the top has no effect.

// precondition: index is on the range of 0 through size()-1 void moveToTop(int index);

To help you understand how these methods work, consider the program below that shows the changing list as each of the messages is sent to list. Recommended: implement one member function at a time, and write tests to ensure that it works.

#include <iostream>

#include "LinkedPriorityList.h" using namespace std;

int main() {

LinkedPriorityList list; list.insertElementAt(0, "a"); list.insertElementAt(1, "b"); list.insertElementAt(2, "c"); list.insertElementAt(3, "d");

for (int i = 0; i < list.size(); i++) // a b c d cout << list.getElementAt(i) << " "; cout << endl;

list.insertElementAt(1, "f");

for (int i = 0; i < list.size(); i++) // a f b c d cout << list.getElementAt(i) << " "; cout << endl;

list.removeElementAt(0);

for (int i = 0; i < list.size(); i++) // f b c d cout << list.getElementAt(i) << " "; cout << endl;

list.lowerPriorityOf(3); // no effect list.lowerPriorityOf(0); // move f right list.lowerPriorityOf(1); // move f right list.lowerPriorityOf(2); // move f right for (int i = 0; i < list.size(); i++) // b c d f cout << list.getElementAt(i) << " "; cout << endl;

list.raisePriorityOf(0); // no effect list.raisePriorityOf(2); // move d left list.raisePriorityOf(1); // move d left for (int i = 0; i < list.size(); i++) // d b c f cout << list.getElementAt(i) << " "; cout << endl;

list.moveToLast(list.size() - 1); // no effect

list.moveToLast(0); // move d from top priority to last priority for (int i = 0; i < list.size(); i++) // b c f d cout << list.getElementAt(i) << " "; cout << endl;

list.moveToTop(0); // no effect

list.moveToTop(2); // move f to top priority again for (int i = 0; i < list.size(); i++) // f b c d cout << list.getElementAt(i) << " ";

return 0; }

### 12D LinkedPriorityList<Type> THROWS EXCEPTIONS

Change your code so it throws an exception when the index is out of range. To do this, fi rst add this #include to PriorityList<Type>:

#include <stdexcept>

Th en add an if statement to every method that takes index as a parameter. An exception will be thrown if the programmer supplies an incorrect index like -1 or an index > size(), which is a good thing:

// Insert the element at the given index.

// precondition: index is on the range of 0 through size() void insertElementAt(int index, Type element) { if (index < 0 || index > size()) { throw std::invalid\_argument(

"\ninsertElementAt: index must be 0..size()");

} // . . .

**C H A P T E R T H I R T E E N**

# Vector of Vectors (2D Arrays)

## SUMMING UP

You have now experienced the control structures necessary to implement any algorithm. You have also experienced ways to build programs with free functions and have written standalone classes that will help you understand the code in this chapter. Th e vector processing explained in the previous two chapters will also help.

## COMING UP

Th is chapter discusses a class that uses two subscripts to manage data logically stored in a tablelike format using rows and columns. Th is proves useful for storing and managing data in applications such as electronic spreadsheets, games, topographical maps, grade books, and many other data best viewed as collections of rows and columns. Th is chapter also reviews the C++ class construct, as the code examples use a class with the topic under study as a data member. After studying this chapter, you will be able to

* process data stored as a vector of vectors (rows and columns)
* use nested for loops

### 13.1 vectorofvectors

Data that conveniently presents itself in a tabular format is represented well with a vector of vector object.

**General Form 13.1** *Constructing a* vector *of* vector*s*

vector <vector<*type*> > *identi er*(*rows*, vector<*type*> (*cols*, *initialValueoptional*));

Th e following are example constructions:

vector <vector<double> > table(4, vector<double> (8, 0.0)); // 32 zeros vector <vector<string> > name(5, vector<string> (100, "TBA")); // 500 TBAs

377

Reference to individual elements in a vector of vectors requires two subscripts, one for the row and another for the column. Because of this, another name for this data structure is the *twodimensional (2D) array*.

**General Form 13.2** *Accessing individual elements*

*vectorName* [*row*][*column*]

Each subscript must be bracketed individually. It is the programmer’s responsibility to keep the subscripts in range. Th e fi rst subscript of a doubly subscripted object specifi es the row, and the second subscript specifi es the column.

*Nested looping* is commonly used to process the data of 2D arrays. Th is code begins with a vector of vectors that has 15 random garbage values. Th e nested loops then initialize all elements to the integers 1 . . . 15:

*Make sure you have one space here with older versions of C++.*

vector <vector<int> > nums(3, vector<int> (5));

int count = 1;

for(int row = 0; row < nums.size(); row++) {

// Initialize one row

for(int col = 0; col < nums[row].size(); col++) { nums[row][col] = count; count++; }

}

1

4

3

2

5

7

8

9

10

6

13

11

12

14

15

nums [0][2]

row 2

#### SELF-CHECK

13-1 Which type more appropriately manages lists of data—a vector or a vector of vectors?

13-2 Which type more appropriately manages data viewed in a row-by-column format—a vector or vector of vectors?

13-3 Construct a vector of vectors object named sales where 120 numbers can be stored in 10 rows.

13-4 Construct a vector of vectors object named sales2 where 120 numbers are stored in 10 columns.

### 13.2 class Matrix

In mathematics, a *matrix* (plural *matrices*) is a rectangular vector—of numbers, symbols, or expressions, arranged in rows and columns—that is treated in certain prescribed ways. One such way is to state the *order* of the matrix. For example, the order of this matrix is a 2 x 2 matrix because there are two rows and two columns:

[12 4 ]

–1 9

Th e individual items in a matrix are called its *elements* or *entries*.

Applications of matrices are found in most scientifi c fi elds including every branch of physics. Th ese include classical mechanics, optics, electromagnetics, quantum mechanics, quantum electrodynamics, and the study of physical phenomena such as the motion of rigid bodies. In computer graphics, matrices are used to project a three-dimensional image onto a two-dimensional screen. In probability theory and statistics, stochastic matrices are used to describe sets of probabilities; for instance, they are used within the PageRank algorithm that ranks the pages in a Google search.

Matrices can be modeled as a C++ class that maintains the number of rows, number of columns, and a vector of vectors to store the elements. Here is a header fi le containing the declaration of this particular Matrix type (diff erent designs are certainly possible):

// File name: Matrix.h

#ifndef MATRIX\_H\_

#de ne MATRIX\_H\_ #include <vector> class Matrix { private:

int rows, columns;

// Make sure there is a space between > and >

// || std::vector<std::vector<int> > table; public:

// Construct a new Matrix and read data from an input le Matrix(std::string leName);

// Construct a new Matrix given a vector of vectors

Matrix(const std::vector<std::vector<int> > & vecOfVecs);

// Return a string representation of this object.

std::string toString();

// Multiply each element by val void scalarMultiply(int val);

// Return the sum of this Matrix + other

Matrix add(Matrix other);

};

#endif // MATRIX\_H\_

In programs with little data required, interactive input suffi ces. Initialization of vector objects quite often involves large amounts of data. Th erefore, the data will come from an external fi le. Th is will also provide another example of fi le input. Th e fi rst line in the fi le of integers named matrix.data specifi es the number of rows and columns of the input fi le:

3 4

6 7 8 9

4 5 6 7

8 7 7 8

Each remaining line represents the quiz scores of one student. We’ll be using this intentionally small input fi le while showing examples of processing a vector of vectors in class Matrix. Th e ifstream object inFile will be associated with this external fi le in the constructor, and the number of rows and columns will be extracted from the fi rst line in the fi le (3 and 4) with this statement. Th en the vector gets resized to row (number of rows) and columns (number of columns). Th is is necessary because memory for the vector of vectors is being allocated at runtime, dynamically and in the way C++ has built these classes: with a Matrix precisely large enough to store three rows of data where each row has four integers and the vector of vectors gets initialized with the fi le data using a nested for loop (a loop inside another loop). Th ese steps are encapsulated in the Matrix constructor in the fi le Matrix.cpp:

/\*

\* Matrix.cpp

\*/

#include <string>

#include <fstream>

#include "Matrix.h" using namespace std;

// Constructs a new object and reads data from // the input le speci ed as the leName argument.

Matrix::Matrix(string leName) { rows = columns = 0; // Avoid a warning from one compiler // Make sure the le named lename is stored in the same directory ifstream inFile( leName); inFile >> rows >> columns;

// Resize the vector of vectors to any capacity at runtime (dynamically).

table.resize(rows, vector<int>(columns));

// Initialize the vector of vectors from le input for (int row = 0; row < rows; row++) { for (int col = 0; col < columns; col++) { inFile >> table[row][col];

}

}

}

As with vector objects, the antibugging technique of displaying all initialized elements of a Matrix can help prevent errors. Th is echo of the input data is accomplished again with the help of nested loops in Matrix::toString.

string Matrix::toString() { string result("");

// Concatenate all elements into one string for (int i = 0; i < rows; i++) { for (int j = 0; j < columns; j++) {

result = result + std::to\_string((int) table[i][j]) + " ";

}

result = result + "\n"; // new line"

}

return result; }

Th e following program initializes and displays the data stored in a Matrix after using fi le input to initialize the individual Matrix elements:

#include "Matrix.h" #include <iostream> using namespace std;

int main() {

Matrix m("matrix.data"); cout << m.toString();

return 0; }

#### Output

6 7 8 9

4 5 6 7

8 7 7 8

Th is Matrix object is now correctly initialized and stores 12 integers.

#### 13.2.1 SCALAR MULTIPLICATION

*Scalar multiplication* is the multiplication of a vector by a scalar where the product is a vector.

Th is operation is implemented below as a modifi er that changes the state of the matrix object:

void Matrix::scalarMultiply(int val) { for (int i = 0; i < rows; i++) { for (int j = 0; j < columns; j++) { table[i][j] \*= val;

}

}

}

Each element is multiplied by the argument, so this code produces the output shown:

m.scalarMultiply(3); cout << m.toString() << endl;

##### Output

18 21 24 27

12 15 18 21

24 21 21 24

#### 13.2.2 MATRIX ADDITION

*Matrix addition* occurs when the corresponding entries of two matrices are added together:

[ ] + [ ] = [ ]

12 4 7 –2 19 2

–1 9 5 –4 4 5

To allow code like this to return a new Matrix,

Matrix a("a.data");

Matrix b("b.data"); // Uses another input le to initialize Matrix c = a.add(b); we need a second constructor that can construct a Matrix object given a vector of vectors as the argument. Here is that second constructor that accepts a vector of vectors as a const reference parameter:

// Construct a new Matrix object given a vector of vectors

Matrix::Matrix(const std::vector<std::vector<int> > & vecOfVecs) { rows = vecOfVecs.size(); columns = vecOfVecs[0].size(); table = vecOfVecs; }

Th is will be called to construct a new Matrix object returned by add:

Matrix Matrix::add(Matrix other) {

vector<vector<int> > temp(rows, vector<int>(columns)); for (int i = 0; i < rows; i++) { for (int j = 0; j < columns; j++) {

temp[i][j] += table[i][j] + other.table[i][j];

}

}

Matrix result(temp); // Use the second constructor return result; }

Using input fi les representing the three matrices above, this code will generate the output shown:

cout << "Matrix a: " << endl << a.toString() << endl; cout << "Matrix b: " << endl << b.toString() << endl; cout << "Matrix c: " << endl << c.toString() << endl;

##### Output

Matrix a:

12 4

-1 9

Matrix b:

7 -2

5 -4

Matrix c:

19 2

4 5

#### SELF-CHECK

13-5 In row-by-row processing, which subscript increments more slowly—row or column?

13-6 In column-by-column processing, which subscript increments more slowly—row or column?

In the next problems, use this 2 × 2 Matrix:

12 4

-1 9

13-7 Complete method get as a member of Matrix to return the element at the given row and column. Matrix.get(1, 0) would return -1.

// Assume get is in class Matrix int Matrix::get(int row, int column) {

13-8 Complete method sum as a member of Matrix to return the sum of all elements. Th e sum of the 2 x 2 Matrix is 24.

// Assume sum is in class Matrix int Matrix::sum() {

### 13.3 PRIMITIVE 2D ARRAYS

Th e concepts of row-by-row and column-by-column processing also apply to primitive C arrays declared with two subscripts. A primitive C array declared with two subscripts use int expressions that specify the number of rows and columns. For example, x is declared here to store 10 rows and fi ve columns of data for a total of 50 numbers:

double x[10][5]; // Row subscripts 0...9, column subscripts 0...4

Th e other important diff erence is that the primitive C array has no range checking of the subscripts. Th e following table compares the Matrix class to the primitive C array declared with two subscripts:

|  |  |  |
| --- | --- | --- |
|  | **vector of vectors** | **Primitive C Array** |
| **General Form** | vector <vector<*type*> >  *identi er* (*rows*, vector<*type*>(*columns*)); | *type* *identi er*  [*rows*][*columns*]; |
| **Example** | vector <vector<int> > unitsSold(4, vector<int>(6)); | int unitsSold [4][6]; |
| **Range Check** | Yes | No |
| **Resizable** | Yes | No |
| **#include** | #include<vector> | Not needed |

Th e vector of vectors, unitsSold, manages four rows and six columns of integers (24 elements all together). Th e primitive C array of the same name (declared in the right column above) manages the same number of integers with the same subscript range. Diff erences include the fact that primitive arrays do not range check subscripts. Individual array elements are referenced in the same manner whether you are using a primitive C array or a vector of vectors. Subscripts always start at 0 in both data structures. Th is means that the following code may be used with either a vector of vectors or a primitive doubly subscripted C array:

int unitsSold[4][6]; // vector<vector<int> > unitsSold(4, vector<int>(6));

for (int r = 0; r < 4; r++) { for (int c = 0; c < 6; c++) unitsSold[r][c] = r + c; }

for (int r = 0; r < 4; r++) { for (int c = 0; c < 6; c++) { cout << unitsSold[r][c] << " ";

} cout << endl;

}

13.4: Arrays with More than Two Subscripts

#### Output with either a vector of vectors or primitive C array

1. 1 2 3 4 5
2. 2 3 4 5 6
3. 3 4 5 6 7
4. 4 5 6 7 8

#### SELF-CHECK

Use this declaration to answer the questions that follow:

int a[3][4];

13-9 What is the value of a[0][0]?

13-10 Does a get its subscripts checked?

13-11 How many int elements are properly managed by a?

13-12 What is the row (fi rst) subscript range for a?

13-13 What is the column (second) subscript range for a?

13-14 Write code to initialize all elements of a to 999.

13-15 Write code to display all elements in each row of a on separate lines with eight spaces for each element.

### 13.4 ARRAYS WITH MORE THAN TWO SUBSCRIPTS

Singly and doubly subscripted vectors occur more frequently than vectors with more than two subscripts. However, vectors with three or even more subscripts are sometimes useful. Triply subscripted arrays are possible because C++ does not limit the number of subscripts. For example, the declaration double q[3][11][6]

could represent the quiz grades for three courses, since 198 (3 × 11 × 6) grades can be stored under the same name (q). Th is triply subscripted object q[1][9][3]

is a reference to quiz index 3 of student index 9 in course index 1. In the following program, an array with three subscripts is initialized (with meaningless data). Th e fi rst subscript—representing a course—changes the most slowly. So the vector object q is initialized and then displayed in a course-by-course manner:

// Declare, initialize, and display a triply subscripted vector

// object. The primitive C subscripted object is used here, but we

// could also use a vector of Matrix objects to do the same thing.

#include <iostream> using namespace std;

int main() { const int courses = 3; const int students = 11; const int quizzes = 6; int q[courses][students][quizzes];

for (int c = 0; c < courses; c++) { for (int row = 0; row < students; row++) { for (int col = 0; col < quizzes; col++) {

// Give each quiz a value using a meaningless formula q[c][col][row] = (col + 1) \* (row + 2) + c + 25;

}

}

}

for (int course = 0; course < courses; course++) { cout << endl;

cout << "Course #" << course << endl; for (int row = 0; row < students; row++) { cout.width(3); cout << row << ": ";

for (int col = 0; col < quizzes; col++) { cout.width(4);

cout << q[course][col][row];

} cout << endl;

}

} return 0; }

#### Output with updated student line

Course #0

0: 27 33 41 49 57 65

1: 28 34 43 52 61 70

2: 29 35 45 55 65 75

3: 30 36 47 58 69 80

4: 31 37 49 61 73 85

5: 32 39 46 53 60 67

6: 33 41 49 57 65 73

7: 34 43 52 61 70 79

8: 35 45 55 65 75 85

9: 36 47 58 69 80 91

10: 37 49 61 73 85 97

Exercises

Course #1

0: 28 34 42 50 58 66

1: 29 35 44 53 62 71

2: 30 36 46 56 66 76

3: 31 37 48 59 70 81

4: 32 38 50 62 74 86

5: 33 40 47 54 61 68

6: 34 42 50 58 66 74

7: 35 44 53 62 71 80

8: 36 46 56 66 76 86

9: 37 48 59 70 81 92

10: 38 50 62 74 86 98

Course #2

0: 29 35 43 51 59 67

1: 30 36 45 54 63 72

2: 31 37 47 57 67 77

3: 32 38 49 60 71 82

4: 33 39 51 63 75 87

5: 34 41 48 55 62 69

6: 35 43 51 59 67 75

7: 36 45 54 63 72 81

8: 37 47 57 67 77 87

9: 38 49 60 71 82 93

10: 39 51 63 75 87 99

### CHAPTER SUMMARY

* A doubly subscripted vector of vectors and a primitive C++ 2D array both manage data that is logically organized in a tabular format—in rows and columns.
* Th e fi rst subscript specifi es the rows of data in a table; the second represents the columns.
* Th e elements stored in these data structures can be processed row by row or column by col-umn.
* Nested for loops are commonly used to process these data structures.
* Primitive 2D arrays do not check the subscript, which can lead to diffi cult errors. vector can check with at messages, as in nums.at(5).at(20); to reference the 21st element in the 6th row.

### EXERCISES

1. For each doubly subscripted object declaration below, determine:
   1. Th e total number of elements
   2. Th e value of all elements

vector<vector<string> > teacher(5, vector<string>(7, "to hire")); vector<vector<double> > quiz(10, vector<double>(32, 0.0)); vector<vector<int> > nums(10, vector<int>(10, -999)); double budget[6][100];

1. Detect the error(s) in the following attempts to declare a doubly subscripted vector:
   1. int x(5,6);
   2. double x[5,6];
   3. vector<vector<int> > x(5, 6);
2. Declare a doubly subscripted object identifi ed with three rows and four columns of fl oatingpoint numbers.
3. Write C++ code to accomplish the following tasks:
   1. Declare a doubly subscripted object called aTable that stores 10 rows and 14 columns of fl oating-point numbers.
   2. Set every element in aTable to 0.0.
   3. Write a for loop that sets all elements in row 4 to -1.0.
4. Show the output from the following program when the dialogue is:

|  |  |  |  |
| --- | --- | --- | --- |
| a. | # rows? ***2*** | d. | # rows? ***1*** |
|  | # cols? ***3*** |  | # cols? ***1*** |
| b. | # rows? ***3*** | e. | # rows? ***1*** |
|  | # cols? ***2*** |  | # cols? ***2*** |
| c. | # rows? ***4*** | f. | # rows? ***2*** |
|  | # cols? ***4*** |  | # cols? ***1*** |

#include <iostream>

#include <vector> using namespace std;

int main() { int maxRow, maxCol; cout << "# rows? "; cin >> maxRow; cout << "# cols? "; cin >> maxCol;

vector<vector<int> > aTable(maxRow, vector<int>(maxCol, -999));

// Initialize Matrix elements

Exercises

for (int row = 0; row < maxRow; row++) { for (int col = 0; col < maxCol; col++) { aTable[row][col] = row \* col;

}

}

// Display table elements for (int row = 0; row < maxRow; row++) { for (int col = 0; col < maxCol; col++) { cout.width(5); cout << aTable[row][col];

}

cout << endl;

} return 0; }

Use this class to answer questions 6 through 9:

class huh { public:

huh(int initLastRow, int initLastColumn); void add(int increment); void show() const; int rowSum(int currentRow) const; private:

int lastRow, lastCol;

std::vector <std:vector<int> > m; };

huh::huh(int initLastRow, int initLastColumn) { lastRow = initLastRow; lastCol = initLastColumn; // The vector of vectors must be initialized in the constructor.

// Use a resize message with two arguments to avoid a loop for each row.

m.resize(lastRow, vector<int>(lastCol));

for(int row = 0; row < lastRow; row++) { for(int col = 0; col < lastCol; col++) { // Give each item a meaningless formula m[row][col] = (row + 1) + (col + 1);

}

}

}

void huh::show() const { int row, col;

for(row = 0; row < lastRow; row++) { for(col = 0; col < lastCol; col++) { cout.width(4); cout << m[row][col];

}

cout << endl;

}

}

1. Write the output generated by the following program:

int main() { huh h(1, 2);

h.show(); return 0; }

1. Write the output generated by the following program:

int main() { huh h(3, 7);

h.show(); return 0; }

1. Complete the member function huh::rowSum that returns the sum of all the elements in a given row. Th e program below must generate the output of 22.

int main() { huh h(4, 4); cout << h.rowSum(2); return 0; }

1. Complete the member function huh::showDiagonal that prints all elements on the diagonal. Assume rows and columns are the same. Th e program to the left must generate the output on the right. *Hint:* You may use cout.width to get the required indentation.

#### Output

|  |  |
| --- | --- |
| int main() { huh h(4, 4);  h.showDiagonal(); return 0;  } | 2  4  6  8 |

10. To class Matrix, add member function transpose that changes the Matrix to its transpose. Th e transpose of a Matrix has the rows replace the columns and the columns replace the rows. You will need to declare a temporary vector of vectors.

Here is what the matrix should look like before:

1. 4
2. 5
3. 6

Here is what the matrix should look like after:

1 2 3

1. 5 6

### PROGRAMMING TIPS

1. When constructing vectors of vectors, be careful not to write >> in your constructions with some compilers.

vector<vector<int>> error(10, vector<int> (10, -1)); // Error: Need space between > and >

1. When using vectors of vectors, consider using the range-checking member function vector::at, especially when fi rst using two subscripts. Th e standard vector class does not automatically check the subscripts, but it can be made to do so with the vector::at member function.

vector<vector<int> > aTable(3, vector<int> (3, -1)); aTable.at(2).at(3) = 23; // Column 3 out of bounds aTable.at(3).at(2) = 32; // Row 3 out of bounds cout << aTable.at(0).at(0); // Output: -1

It is common to get a subscript variable that is out of bounds. Th e sooner you know about it, the better. With range checking on, you’ll know immediately.

1. Many of the programming tips for vectors with one subscript can be applied to doubly and triply subscripted objects:
   * Th e elements of any vector must be of the same class. For example, a Matrix cannot store both string and integer values.
   * Any object that uses a large amount of memory may be passed as a const reference parameter. As with singly subscripted vectors, memory is saved and only one value (the address of the Matrix) needs to be copied. However, when a Matrix is passed as a value parameter, every single element gets copied, making the program less effi cient. void function(const Matrix<double> & m) // Pass by const reference is more effi cient than

void function(Matrix<double> m)

* + Range checking should be employed while you are learning to manipulate doubly subscript-ed objects.

**PROGRAMMING PROJECTS**

## 13A MAGIC SQUARE

A magic square is an *n*-by-*n* vector of vectors where the integers 1 to *n*2 appear exactly once where *n* must be a positive integer like 1, 3, or 5. Additionally, the sum of the integers in every row, every column, and on both diagonals is the same. Implement class MagicSquare with two member functions: a constructor and display. Th e following code should generate the output shown:

|  |  |  |
| --- | --- | --- |
| MagicSquare magic(1); magic.display(); 1 by 1 magic square  1 | MagicSquare magic(3); magic.display(); 3 by 3 magic square  8 1 6   1. 5 7 2. 9 2 | MagicSquare magic(5); magic.display(); 5 by 5 magic square  17 24 1 8 15  23 5 7 14 16  4 6 13 20 22   1. 12 19 21 3 2. 18 25 2 9 |
|  | | |

You should be able to construct an *n*-by-*n* magic square for any odd value *n* from 1 to 15. When *j* is 1, place the value of *j* in the middle of the fi rst row. Th en, for a counter value ranging from 1 to *n*2, move up one row and to the right one column, and store the counter value—unless one of the following events occurs:

1. When the next row becomes 0, make the next row equal to *n* – 1.
2. When the next column becomes *n*, make the next column equal to 0.
3. If a position is already fi lled or the upper-right corner element has just obtained a value, place the next counter value in the position that is one row below the position where the last counter value has been placed.

You will need to resize the vector of vectors instance variable which can be done like this:

// An instance variable vector<vector<int> > magic;

// Resize the vector to be a size by size vector magic = vector<vector<int> >(size, vector<int>(size));

## 13B GAME OF LIFE

Th e Game of Life was invented by John Conway to simulate the birth and death of cells in a society. Th e following rules govern the birth and/or death of cells between two consecutive time periods. At time *T*:

* A cell is born if there was none at time *T* – 1 and exactly three of its neighbors were alive.
* An existing cell remains alive if at time *T* – 1 there were either two or three neighbors.
* A cell dies from isolation if at time *T* – 1 there were fewer than two neighbors.
* A cell dies from overcrowding if at time *T* – 1 there were more than three neighbors.

A neighborhood consists of the eight elements around any element (N represents one neighbor):

NNN

N N

NNN

Th e neighborhood can extend to the other side of the society. For example, a location in the fi rst row has a neighborhood that includes three locations in the last row. Th e following patterns would occur when T ranges from 1 to 5, with the initial society shown at T=1. O represents a live cell; a blank indicates that no cell exists at that particular location in the society.

T=0 T=1 T=2 T=3 T=4 ....... ....... ....... ....... .......

..O.O.. ..O.O.. ....... ....... .......

..OOO.. ..O.O.. ..O.O.. ...O... .......

....... ...O... ...O... ...O... .......

....... ....... ....... ....... .......

Other societies may stabilize like this:

T=0 T=1 T=2 T=3 T=4 ....... ....... ....... ....... .......

....... ...O... ....... .. O... .......

..OOO.. ...O... ..OOO.. ...O... ..OOO..

....... ...O... ....... ...O... .......

....... ....... ....... ....... .......

Use a test driver like this to see the fi rst fi ve versions of a society. Design your own fi le to be read by the GameOfLife constructor.

#include "GameOfLife.h" // For the GameOfLife class

int main() {

GameOfLife society("5by7");

for (int updates = 1; updates <= 5; updates++) { society.toString(); society.update();

} return 0; }

Implement the following member functions defi ned in this header fi le:

/\*

* File name: GameOfLife.h
* \* A model for John Conway's Game of Life to simulate the birth and death \* of cells. This is an example of cellular automata.

\*/

#ifndef GAMEOFLIFE\_H\_ #de ne GAMEOFLIFE\_H\_

#include <vector> #include <string> class GameOfLife {

private: std::vector<std::vector<bool> > theSociety; int nRows; int nCols;

public: /\*

* Write the constructor to initialize a vector of vectors so
* all elements are false. Also set nRows and nCols

\*

\*/

GameOfLife(int rows, int cols);

/\* \* Return the number of rows, which is indexed from 0..numberOfRows()-1.

\*/ int numberOfRows();

/\* \* The number of columns, which is indexed from 0..numberOfColumns()-1.

\*/

int numberOfColumns();

/\* \* Place a new cell in the society.

* Precondition: row and col are in range.
* \* row The row to grow the cell.
* col The column to grow the cell.

\*/

void growCellAt(int row, int col);

/\* \* Return true if there is a cell at the given row and column.

* Return false if there is none at the speci ed location.
* \* row The row to check.
* col The column to check.

\*/

bool cellAt(int row, int col);

/\*

* Return one big string of cells to represent the current state of the
* society of cells (see output below where '.' represents an empty space \* and 'O' is a live cell. There is no need to test toString. Simply use \* it to visually inspect. Here is one sample output from toString:

\*

* GameOfLife society(4, 14);
* society.growCellAt(1, 2);
* society.growCellAt(2, 3);
* society.growCellAt(3, 4);
* cout << society.toString();

\*

* Output \* ..............
* ..O...........
* ...O..........
* ....O.........
* \*/

std::string toString();

/\*

* Count the neighbors around the given location. Use wraparound. A cell
* in row 0 has neighbors in the last row if a cell is in the same column
* or the column to the left or right. In this example, cell 0,5 has two
* neighbors in the last row, cell 2,8 has four neighbors, cell 2,0 has \* four neighbors, cell 1,0 has three neighbors. Cell 3,8 has 3 neighbors.
* The potential location for a cell at 4,8 would have 3 neighbors.

\*

* .....O..O \* O........
* O.......O
* O.......O \* ....O.O..
* \* The return values should always be in the range of 0 through 8.

\*/

int neighborCount(int row, int col);

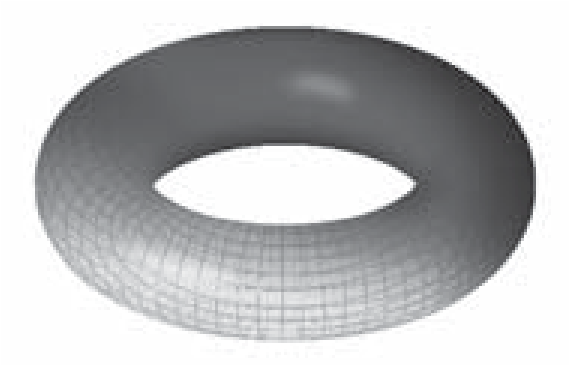
/\* \* Update the state to represent the next society.

* Typically, some cells will die off while others are born.

\*/ void update(); };

#endif /\* GAMEOFLIFE\_H\_ \*/

Th is GameOfLife has wraparound. It is recommended that you use nested for loops to visit all eight neighbors. Use an additional two int variables to be the actual row and column that are set, while checking to see if a loop index is negative or too large. Try to imagine the cells covering a torus:



Th e O below has eight neighbors labeled a through h. Wraparound is needed for neighbors labeled d through h. Th e labels are repeated to show where they need to be checked.

f g h

e d

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| O | a |  |  |  | e |
| c | b |  |  |  | d |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
| g | h |  |  |  |  |

*Another hint:* During update, initialize all elements of a temporary vector of vectors to false. Look at the vector instance variable. Only grow cells in the temporary vector. When done, assign this to the instance variable like this:

theSociety = temporary;

Here are two examples of updates in two diff erent cell societies. After 21 updates, the society on the right will return to its original form shifted fi ve spaces to the right. If you do 63 updates, you will see the form that begins on the right shifted to be on the left due to wraparound.

#include <iostream>#include <iostream> using namespace std;using namespace std; #include "GameOfLife.h"#include "GameOfLife.h"

int main() {int main() {

GameOfLife game(3, 8); GameOfLife society(5, 30);

game.growCellAt(1, 2); society.growCellAt(1, 6); game.growCellAt(1, 3); society.growCellAt(2, 7); game.growCellAt(1, 4); society.growCellAt(2, 8); cout << game.toString(); society.growCellAt(3, 7);

society.growCellAt(3, 6);

for (int t = 1; t <= 5; t++) {

game.update(); society.growCellAt(1, 16); cout << game.toString(); society.growCellAt(2, 17);

} society.growCellAt(2, 18); return 0; society.growCellAt(3, 17); } society.growCellAt(3, 16); *Output:*

|  |  |
| --- | --- |
| ........  ..OOO...  ........  ...O....  ...O....  ...O....  ..OOO...  ..OOO...  ..OOO...  .O...O..  .O...O..  .O...O..  OOO.OOO.  OOO.OOO.  OOO.OOO.  ........  ........  ........ | for (int t = 1; t <= 6; t++) { society.update();  cout << society.toString();  }  return 0; }  *Output:*  ..............................  .......O.........O............  ........O.........O...........  ......OOO.......OOO...........  ..............................  ..............................  ..............................  ......O.O.......O.O...........  .......OO........OO...........  .......O.........O............  ..............................  ..............................  ........O.........O...........  ......O.O.......O.O........... |

.......OO........OO...........

..............................

..............................

.......O.........O............

........OO........OO..........

.......OO........OO...........

..............................

..............................

........O.........O...........

.........O.........O..........

.......OOO.......OOO..........

........O.........O...........

..............................

..............................

.......O.O.......O.O..........

........OO........OO..........

*Note:* Th e sixth update shows wraparound.

© 2018 Franklin, Beedle & Associates All rights reserved

**A P P E N D I X**

# Answers to Self-Check Questions

## CHAPTER 1: PROBLEM SOLVING WITH C++

1-1 input: pounds or perhaps todaysConversionRate output: USDollars

1-2 cdCollection, selectedCD

1-3

|  |  |  |  |
| --- | --- | --- | --- |
| **Problem** | **Object Names** | **Input or Output** | **Sample Problem** |
| Compute the  future value of an investment | presentValue periods rate futureValue | Input  "  "  Output | 1000.00  360  0.0075  14730.58 |

1-4 Turn the oven off (you might have recognized some other activity has been omitted).

1-5 No (at least the author thinks it’s okay).

1-6 No (at least the author thinks it’s okay).

1-7 No. Th e courseGrade would be computed using undefi ned values for test1, test2, and nalExam.

1-8 No. Th e details of the process step are not present. Need a formula to compute a weighted average.

1-9 Th e program is wrong.

1-10 Th e prediction is wrong.

1-11 Th e program is wrong.

1-12 Numbers with a fractional part—fl oating-point numbers such as −1.2 or 1.023.

399

1-13 + - \* (also = output with cout<< and input with cin>>)

1-14 Integers—numbers without a decimal point. Th e actual range of integers is system dependent (unfortunately). Most C++ systems implement int to store integers in the range of –2,147,483,648 to 2,147,483,647.

1-15 + - \* (also = output with cout<< and input with cin>> ) 1-16 A collection of characters.

1-17 oat, double, int, bool, char, short, unsigned int, unsigned long

## CHAPTER 2: C++ FUNDAMENTALS

2-1 22 plus or minus two. Actually it is easy to miscount, so let the compiler worry about it.

2-2 a. VALID l. periods'.' not allowed

1. 1 can’t start identifi er m. double is a reserved word
2. VALID n. can’t start identifi ers with 5
3. # not allowed o. space not allowed
4. space not allowed p. VALID
5. # not allowed q. VALID
6. ! not allowed r. å) not allowed
7. VALID s. VALID (but weird)
8. ( ) are not allowed t. / not allowed
9. VALID (double is not) u. VALID
10. VALID
    1. + - other possible answers , : ; ! ( ) = { }
    2. << >> other possible answers != == <= >=
    3. cin and cout (also string vector width sqrt)
    4. thisIsOne and this Is\_YET\_Another\_1$
    5. a. string literals: "'" and "H"
11. integer constants: 234 and -123
12. fl oating-point constants: 1.0 and 1.0e+03
13. boolean literals: false true
14. char literals: '\n' 'h'
    1. a and d only
    2. double aNumber = -1.5; double anotherNumber = -1.5;
    3. string address;
    4. #include <iostream> using namespace std; int main() { cout << "Kim" << endl; cout << "Miller" << endl; return 0;

}

-or-

#include <iostream> int main() {

std::cout << "Kim" << std::endl; std::cout << "Miller" << std::endl; return 0;

}

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2-12 a. error, can’t assign int to boolean | | | | | d. error, can’t assign double to long | |
| b. 123 (truncation occurs) | | |  | | e. 'B' or 66 | |
| c. 123.0; |  |  | | f. error, ui is not a known symbol | | |
| 2-13 a. 10.5 |  |  | | d. -0.75 | | |
| b. 1.75 |  |  | | e. -0.5 | | |
| c. 3.5  2-14 97 % 25 % 10 / 5  22 % 10 / 5  2 / 5  0 |  |  | | f. 1.0 | | |
| 2-15 a. 0 |  |  | | d. 1 | | |
| b. 1 |  |  | | e. 0 | | |
| c. 0 |  |  | | f. 0 | | |
| 2-16 a. 0 |  |  | | d. 10 | | |
| b. 0.5555556 |  |  | | e. 12 | | |
| c. 0.5555556 |  |  | | f. 2 | | |
| 2-17 Dialogue #1: | Dialogue #2: | | | | | Dialogue #3: |
| a. **3.2** (16.0/5.0) | b. **1.9** (9.5/5.0) | | | | | c. **2.8** (14.0/5.0) |

* 1. Th at depends on the garbage value of x. Compilers usually warn about x being undefi ned.
  2. Th e predicted answer 25 does not match 0.04.
  3. change cin >> n to cin >> sum and cin >> sum to cin >> n.
  4. a. intent

1. compile time
2. compile time

## CHAPTER 3: USING FREE FUNCTIONS

3-1 pow(4.0, 3.0) is 4\*4\*4 or 64

3-2 pow(3.0, 4.0) is 3.0\*3.0\*3.0\*3.0 or 81

3-3 oor(1.6+0.5) is 2.0

3-4 ceil(1.6-0.5) is 2.0

3-5 1.0

3-6 4.0

3-7 Trace 9.99 rounded to 1 decimal place.

|  |  |  |
| --- | --- | --- |
|  | x | n |
| Input n | 9.99 | 1 |
| Let x become x \* 10n | 99.9 | 1 |
| Add 0.5 to x | 100.4 | " |
| Let x become oor(x) | 100.0 | " |
| Let x become x divided by 10n | 10.0 | " |

3-8 Th ree Sample Problems (other answers certainly possible)

|  |  |  |
| --- | --- | --- |
| x | n | changed x |
| 0.567 | 1 | 0.6 |
| 1234.56789 | 2 | 1234.57 |
| -1.5 | 1 | -1.0 |

3-9 3.2

3-10 x = x \* pow(10, n)

x = x - 0.5 // subtract 0.5 x = ceil(x) // take the ceiling of x x = x / pow(10,n) 3-11 a. 16.0 or 16 d. 1.0

1. 4.0 or 4 e. 23.4
2. -1.0 or -1 f. 16.0

3-12 a. valid d. incorrect type of argument b. wrong function name e. missing ( and )

c. too many arguments f. valid (the int is promoted to a double)

3-13 a. missing parameter type (need int, double, string, or, . . . )

1. missing two commas
2. no return type
3. okay only if myClass exists
4. extra , before )
5. attempt at parameter is a string literal

3-14 1. oor(1.9999)

1. oor(0.99999)
2. oor (-1.9)
3. oor(-1) (other answers possible)

3-15 1. 1.0 (or 1 will do)

1. 0.0
2. -2.0
3. -1.0
   1. "1st"
   2. 3.4
   3. a. double d. double b. pow e. double

c. 2 f. there is no third argument

3-19 pow(-81.0, 2) (other answers possible)

3-20 No, the preconditions are not met. Th e return value is undefi ned. Return could be NaN (not a number).

3-21 Yes, 100.0.

3-22 Yes, 32.0.

3-23 Yes, 2.0 (you might have needed a scientifi c calculator. x0.5 is the square root of x).

3-24 No, missing 2nd argument. Return value cannot be determined.

3-25 double remainder(double dividend, double divisor).

// pre: divisor is not zero

// post: return the oating point remainder of dividend/divisor

## CHAPTER 4: IMPLEMENTING FREE FUNCTIONS

4-1 a. -1.0 d. ERROR, one too many arguments b. 7.0 d. One two few arguments

c. 17.0 e. 66.28

4-2 0.375

4-3 No. Th e argument is supposed to be positive. Th e result depends on the system you are using. You could get Infi nity, NaN for not a number, or a square root of a negative number error.

4-4 a. Remove ; after ) d. return missing, you cannot assign a number to a function name.

1. j is unkown in f2 e. must return a number, not the double class name.
2. j is unkown in f3 f. must return an int, f6 tries to return a string instead.
   1. double times3(double x) { return 3 \* x;

}

* 1. cout f1, f2, and main a f1 only b f1 only d f2 only

cin f1, f2, and main f2 f2 and main

MAX f1, f2, and main main from nowhere inside this fi le

c f2 only e main only f1 f1, f2, and main

4-7 Parameters and variables declared within the function’s block.

4-8 Anywhere to the end of the fi le unless it is redeclared within a block, then the global is hidden from that particular function.

4-9 a. // arg1 5 arg2 5

b. // arg1 11 arg2 123

## CHAPTER 5: USING MEMBER FUNCTIONS

5-1 a. Missing second argument. b. Missing fi rst argument.

1. bankAccount is an undefi ned symbol. Change b to B.
2. Missing a numeric argument between ( and ).
3. Missing (, the argument, and ).
4. Wrong class of argument. Pass a number, not a string.
5. B1 is undefi ned.
6. Deposit is not a member of BankAccount. Change D to d.
7. Need an object and a dot before withdraw.
8. b4 is not a BankAccount object, it was never declared to be anything.
9. missing () after name.
10. name takes zero arguments, not one.
    1. Chris: 202.22

Kim: 545.55

* 1. 14

S k

7

18446744073709551615 or string::npos. Answer may vary on diff erent systems

Net

N Network o

5-4 a. UnSocial c. Socl

b. Societal d. NoTiaX

5-5 string aString = "abcd"; int midChar = aString.length() / 2; char mid = aString.at(midChar);

5-6 a. error, length is not a function d. 3

1. error , missing () e. y Str
2. error, length is unknown f. error

5-7 123456789012345

1 2.3 who

5-8 9.88

1

1.2

5-9 a. Enter an integer: ***123*** b. Enter an integer: ***XYZ***

Good? 1 Good? 0

5-10 a. istream c. string

1. Grid d. BankAccount
2. ostream e. istream
   1. . . . . . .

. . < .

. . . . .

. . . . .

. . . . . . . . . row: 1 row: 2

* 1. 1. Moving off the edge of the Grid (another answer is possible).

1. Moving through a block.
2. Attempting to pick up something that isn’t there.
   1. 1
   2. 35
   3. #include "Grid.h"

#include <iostream> using namespace std; int main() {

Grid g(5, 5, 2, 3, east); g.move(1);

g.face(north);

g.move(1);

g.face(west);

g.move(1);

g.face(south);

g.move();

g.display(); return 0;

}

5-16 Th ere is less code to write (another answer is possible).

Abstraction allows us to think of what the function does, not the details of the implementation.

Th e same code is often needed in more than one location. Writing that code as a function avoids duplicated code, which is a very bad thing.

5-17 Use your cell phone without worrying about how the network works.

Can do a lot without worrying about how to walk and breathe.

## CHAPTER 6: IMPLEMENTING MEMBER FUNCTIONS

6-1 LibraryBook

6-2 borrowBook returnBook

6-3 isAvailable getBorrower getBookInfo

6-4 author title borrower available

6-5 string

6-6 bool

6-7 LibraryBook aBook("Computing Fundamentals with Java, "River Tanner");

6-8 abook.borrowBook("Madison");

6-9 abook.getBorrower();

6-10 Add the class name and :: before the function name (after the return type). Also match the rest of the function heading in the class defi nition. Th e parameter names may diff er.

6-11 Yes

6-12 No

6-13 'Tinker Tailor Soldier Spy' by John le Carre

CAN BORROW

1

Charlie Archer

0

1

CAN BORROW

6-14 Th e function is not supposed to change the state. Th is is enforced when passing the argument by const reference: const&.

6-15 Th e constructor(s).

6-16 Allow access to the state of any object so humans or other objects can either inspect or use that state. An accessor may return a data member, or some sort of processing may occur to return some information about the state of the object.

6-17 Modify the state of the object. At least one data member gets changed for each modifying message—otherwise it is an accessor.

6-18 Allow programmers to initialize objects with either the default state or their own initial values.

6-19 To store the state of any object. Each instance of the class has its own copy of the data members.

6-20 Lines 2 and 3 are attempts to send a modifying message to a const object (the parameter b).

## CHAPTER 7: SELECTION

7-1 a. true e. true

1. false f. true
2. false h. true (165 is non-zero)
3. true g. false (= is assignment, not equality,j = 0 is evaluates to false)

7-2 a. addRecord e. dubious

1. deleteRecord f. g: 45
2. None option is lower case at cutoff
3. dubious g: 70 failing you get one

g: 1

* 1. Tune-up due in 0 miles
  2. a. 38.0 c. 43.0

b. 40.0 d. 45.25

7-5 a. true c. x is low after if...else d. neg

b. zero or pos

7-6 if (option == 1) cout << "My name" << endl;

else

cout << "My school" << endl;

7-7 a. true e. true

1. false f. false
2. true g. true
3. false h. true
   1. (score >= 1) && (score <= 10)
   2. (test > 100) || (score < 0)
   3. President's list (always true because = was used instead of ==).
   4. Row Column Output

|  |  |  |
| --- | --- | --- |
| 3 4 |  | not |
| 4 3 |  | not |
| 2 2 |  | not |
| 0 2 |  | On edge |
| 2 0 |  | On edge |
| 7-12 a. true |  | c. false |
| b. false |  | d. false |

* 1. All four evaluate. Th e fourth expression (¦¦ g.column()==g.nColumns()-1) had to be evaluated because the fi rst three are false (as is the fourth).
  2. Th e last three couts were not evaluated. Th is is weird code meant to vividly demonstrate short circuit Boolean Evaluation.

1. okay c. okay
2. failed d. failed
   1. 70
   2. Th is unfortunate student gets a D instead of the deserved C.
   3. I wouldn’t be happy, and I doubt you would either.
   4. -40: extremely frigid 20: warm -1: below freezing

42: toast 15: freezing to mild 31: very hot

* 1. 20 through 29 inclusive.
  2. 0 through 19 inclusive. 7-21 int main() {

assert("extremely frigid" == weather(-41)); assert("extremely frigid" == weather(-40)); assert("below freezing" == weather(-39)); assert("below freezing" == weather(-1)); assert("freezing to mild" == weather(0)); assert("freezing to mild" == weather(19)); assert("warm" == weather(20)); assert("warm" == weather(29)); assert("very hot" == weather(30)); assert("very hot" == weather(39)); assert("toast" == weather(40)); assert("toast" == weather(41)); return 0;

}

* 1. AAA
  2. BBB
  3. Invalid
  4. Invalid 7-26 switch(choice) { case 1: cout << "Favorite music is Jazz" << endl; break; case 2: cout << "Favorite food is Tacos" << endl; break; case 3: cout << "Favorite teacher is you" << endl; break; default;

cout << "Error" << endl;

}

## CHAPTER 8: REPETITION

8-1 No, the init-statement happens fi rst (and only once).

8-2 No, you can use increments of any amount, including negative increments (or decrements). 8-3 No, consider the example for (int i = 1; i < n; i++) when n==0.

8-4 Consider if the update step does not increment j, or j is decremented as much as it is incremented inside the loop: for (j = 1; j < n; j){ } *or* for (j = 1; j < n; j++){j--;} 8-5 a. 1 2 3 4 d. 0 1 2 3 4

1. 1 2 3 4 5 e. 5 4 3 2 1
2. -3 -1 1 3 f. before after
   1. for (int i = 1; i <= 100; i++) { cout << i << endl;

}

* 1. for (int i = 10; i >= 1; i--) { cout << i << " ";

}

* 1. An attempt is made to block an intersection at a non-existent row; the program terminates.
  2. It’s no big deal. Th e right corners would be blocked twice.
  3. Th e function would alter a copy of the Grid, not the Grid in main. Th e border would be set locally in setBorder, but it would not modify the arguments aGrid or anotherGrid in main.
  4. Range = 3
  5. Range = 29 (this is correct).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| highest | -2147483648 | -5 | 8 | 22 | 22 | 22 |
| lowest | 2147483647 | -5 | -5 | -5 | -7 | -7 |

* 1. Range = 4 (this is correct).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| highest | -2147483648 | 5 | 5 | 5 | 5 | 5 |
| lowest | 2147483647 | 2147483647 | 4 | 3 | 2 | 1 |

* 1. Range = Range: -2147483642 (this is obviously incorrect)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| highest | -2147483648 | 1 | 2 | 3 |
| lowest | 2147483647 | 2147483647 | 2147483647 | 2147483647 |

* 1. b. When the input is entered in ascending order.
  2. Get rid of the else.
  3. Th e client code will never know in advance how many moves must be made. It cannot be determined in advance
  4. a. 56.33333 the fi rst test (70.0) is destroyed before its added to the accumulator. Additionally, the sentinel-1 is incorrectly added to the accumulator.

b. 80.0

8-19 a. Observe the location of the second cin>>testScore. Redo till you arrive at the preceding answers of 56.3333 for a. and 80.0 for b.

8-20 b. Th e input statement comes immediately before it is compared to –1.

8-21 zero

8-22 Input another cin >> testscore at the bottom of the loop.

8-23 A tricky question: remove the ; after ). Th is loop does nothing infi nitely because ; represents the null statement. It is legal code, but it was not likely intended.

|  |  |
| --- | --- |
| 8-24 a. 1 2 3 | b. 2 4 6 8 10 |
| 8-25 a. unknown | d. forever |
| b. forever | e. 5 |
| c. 0 | f. forever notice the ; after count >= 0); |

8-26 int sum = 0; int x = 0;

while ( (cin >> x) && (x != 999) ) { sum += x;

}

|  |  |
| --- | --- |
| 8-27 a. 1 | b. -1 |
| 2 | -0.5 |
| 3 | 0 |
|  | 0.5 |
|  | 1 |

8-28 int x;

do {

cout << "Enter a number in the range of 1 through 10: "; cin >> x;

} while ( x < 1 || x > 10);

8-29 do {

cout << "Enter A)dd W)ithdraw Q)uit: ";

cin >> option; option = toupper(option);

} while ( option != 'A' && option != 'W' && option != 'Q');

8-30 a. determinate for loop

1. determinate for loop
2. indeterminate while loop
3. indeterminate do-while loop

8-31 a. value == -1

b. while (value != -1)

8-32 For each loop, which objects are not initialized but should be? a. count and n

b. n and inc

## CHAPTER 9: FILE STREAMS

9-1 // le name: THISPROG.CPP

#include <fstream> // for class ifstream

#include <iostream> // for cout #include <string>

using namespace std;

int main() { string aString; ifstream inFile("THISPROG.CPP"); for(int j = 1; j <= 4; j++) { inFile >> aString;

cout << aString << " ";

}

cout << endl; return 0;

}

9-2 Can't average 0 numbers.

9-3 a. Failed to nd the le numbers.dat b. iteration # 1: 0.001

End of le reached. 1 numbers found. c. End of le reached. 0 numbers found.

9-4 a. 6 c. 6

b. 15 d. 1 the period '.' sets inFile to a bad state and the loop terminates.

9-5 Th e loop would terminate since there would be no last name for Kline. Th e status would be Kline, the last name Sue. Th e employee Kline would never be constructed.

9-6 Th e loop would terminate when S was encountered for exempts. Th e employee Kline would never be constructed.

9-7 a. 1313 Mocking Bird Lane

b. 1214 West Walnut Tree Drive

## CHAPTER 10: VECTORS

10-1 100

10-2 0

10-3 99

10-4 0

10-5 x[0] = 78; 10-6 int n = 100; for(int j = 0; j < n; j++) { x[j] = n-j;

}

10-7 for(j = 0; j < n; j++) { cout << x[j] << endl;

}

10-8 It depends. Th e computer may “crash.” You may destroy the state of another object. Or, with subscript range checking, you may get a runtime error before the program terminates.

10-9 vector::resize and vector::capacity

10-10 0 1 2 3 4

0 1 2 3 4 0 0 0 0 0

10-11 -1

10-12 1

10-13 5

10-14 4

10-15 n

10-16 0 (because of short circuit Boolean evaluation).

10-17 irst econ hir ourt

10-18 account[12] = BankAccount("A12thCustomer", 1212.12); account[13] = BankAccount("Cust13", 1313.13);

10-19 Th e 21st account on the 21st line of the fi le would not become part of the account database. Th e vector size would not be big enough and the loop would terminate because numberOfAccounts < account.capacity() would be false.

10-20 #include <iostream>

#include <fstream> using namespace std; int main() {

vector <int> vectorOfInts(1000);

// File name will do if it is in the working directory

ifstream inFile("int.dat");

int n = 0; int el;

while( (inFile >> el) && (n < vectorOfInts.capacity()) ) {

vectorOfInts[n] = el; n++;

}

return 0;

}

10-21 n

10-22 cout << "Number of meaningful ints in vectorOfInts is " << n << endl;

cout << "Here they are" << endl; for(int j = 0; j < n; j++) { cout << j << ". " << vectorOfInts[j] << endl;

}

10-23 Grid and vector objects are much bigger than int and double. Th at is, it takes more memory to store a grid than an int (appoximately 800 bytes versus 4 bytes). A vector of 1,000 doubles is 1,000 times larger than one double.

10-24 a. 100,000 **×** 57 or 5.7 million bytes.

1. 4
2. 4
   1. ascending
   2. Th e fi rst element in the vector is swapped with itself. Th at means three extra assignments, but it is not worth worrying about this special case.
   3. double largest = x[0]; for(int j = 1; j < n; j++) { if( x[j] > largest ) largest = x[j]; }
   4. Th e vector is sorted and the binary search knows whether it is either ascending or descending order.
   5. 1: 1024 2: 512 3: 256 4: 128 5: 64 6: 32 7: 16 8: 8 9: 4 10: 2 11:

Th erefore, the largest number of comparisons is 11.

* 1. When rst exceeds last, that is the beginning and end of the vector no longer make any sense. For example when rst == 1028 and last == 1026.
  2. Swap the location of the two statements.

last = mid - 1; and

rst = mid + 1; or change the expression if (searchString < str[mid]

to

if (str[mid] < searchString) but NOT both changes.

## CHAPTER 11: GENERIC COLLECTIONS

11-1 Any number can be used as long as the computer has more memory to grow the vector.

11-2 cout << intSet.size() << endl;

11-3 intSet.insert(89);

11-4 intSet.remove(89);

11-5 three

11-6 a. 5

1. 40
2. 0

11-7 #include <iostream> using namespace std;

#include "Set.h" // For a generic Set class

#include "BankAccount.h"

int main() {

Set<BankAccount> set; // Store a set of 4 BankAccounts set.insert(BankAccount("Chris", 300.00)); set.insert(BankAccount("Devon", 100.00)); set.insert(BankAccount("Kim", 444.44)); set.insert(BankAccount("Dakota", 99.99));

double largest = 0.0;

set. rst(); // Initialize an iteration over all elements

while (set.hasMore()) { double currentBalance = set.current().getBalance(); if (currentBalance > largest) largest = currentBalance;

set.next();

}

cout << "Max balance is " << largest << endl; return 0;

}

## CHAPTER 12: POINTERS AND MEMORY MANAGEMENT

12-1 Th e addresses of other objects

12-2 a. doublePtr

1. Can’t know. One run with cout << & doublePtr; was 0x7fff5b44cc78.
2. 1.23
3. \*doublePtr += 1.0;
   1. 246
   2. 1
   3. a->getBalance() + b->getBalance();

-or-

(\*a).getBalance() + (\*b).getBalance();

* 1. 24 144
  2. char ch = 'C'; char\* charPtr = &ch; 12-8 int n1 = 12; int n2 = 34; int n3 = 56; int \*p1 = &n1; int \*p2 = &n2; int \*p3 = &n3;
  3. cout << \*p1 + \*p2 + \*p3 <<endl
  4. p? 333

q? 333

* 1. 4 8
  2. 4.56 4.56
  3. Th ere is no way to get the value memory that was found at \*p.
  4. 12
  5. int array[1000];
  6. for(int i = 0; i < 1000; i++) { array[i] = -1;

}

* 1. 0 2 4 6 8 10
  2. **int** min = x[0];

**int** max = x[0]; **for** (int i = 1; i < n; i++) { **if** (x[i] > max) max = x[i]; **if** (x[i] < min) min = x[i];

}

* 1. string strs[] = {"one", "two", "three", "four"}; 12-20 Before:

First

Second

header

last

After:

First

header

last

12-21

Before

:

First

header

last

After:

header

last

* 1. Remove returns false. No changes are made to the state of the list.
  2. a. False, it can grow at runtime.

1. False, not in this list anyway. Use get(int). Th e [] could be overridden.
2. True
3. False, this is not necessary with the unused header node.
4. True, to avoid a memory leak. In small program it doesn’t matter, but in large pro-grams a lot of time is spent chasing down and removing memory leaks unfortunately.

12-24 bool removeLast() {

if (n == 0) return false;

// Get ptr to point to the last node

node\* ptr = header; while (ptr->next != last) {

ptr = ptr->next;

}

// Adjust last to the node before it, clean up memory, decrease size

last = ptr; delete ptr->next;

n--;

return true;

}

## CHAPTER 13: VECTOR OF VECTORS

13-1 vector

13-2 matrix (or a vector of vectors)

13-3 matrix<double> sales(10, 12);

13-4 matrix<double> sales2(12, 10);

13-5 row

©

13-6 column

13-7 int Matrix::get(int row, int column) { return table[row][column];

}

13-8 int Matrix::sum() { int result = 0; for (int i = 0; i < rows; i++) { for (int j = 0; j < columns; j++) {

result += table[i][j];

}

}

return result;

}

13-9 Undefi ned, could be anything (just had a[0][0] return 1550093504).

13-10 No, this is a primitive array.

13-11 12

13-12 0 through 2

13-13 0 through 3

13-14 for (int row = 0; row < 3; row++) { for(int col = 0; col < 4; col++) {

a[row][col] = 999;

}

}

13-15 for (int row = 0; row < 3; row++) { for (int col = 0; col < 4; col++) {

cout.width(8);

cout << a[row][col];

}

cout << endl;

}

-- ++ += -= incrementing operators, 226

! || && Boolean operators, 188

{ } block, 185

* / % + - arithmetic operators, 31
* with pointers, 346

\n \" \' \t escape sequences, 204

& address operator, 346

& reference parameter, 92

#include directive, 119 % remainder operator, 32

> < <= >= == != operators, 178

**A**

abs function, 67 abstraction, 128 access mode, 156 accessing methods 112, 124, 154 accessor method, 160 address of operator &, 346 addressing, indirect, 346 argument, 65, 326 algorithm, 6 algorithmic patterns, 2 input process output, 5 prompt then input, 35 guarded action, 176 alternative action, 181 multiple selection, 195 determinate loop, 223

### I N D E X

indeterminate loop, 236 iterator, 333

analysis, 1 analysis design implementation, 83, 232

course grade computation, 3–9 round to n decimals, 58–60 distance between two points, 83–86 range of temperature readings, 230–235

argument / parameter associations, 99 arithmetic expressions, 30 array, primitive C, 352 arguments, 354 two-dimensional, 384 diff erences from vector, 353

assert function, 200 assignment statement 27 assignment compatibility, 27 at (string method), 113 at (vector method), 284 average, weighted, 15, 52

**B**

bank teller objects, 106

BankAccount class, 107, 143 binary operator, 31 binary search, 303 block, 77

with if and if…else, 184

boundary testing, 200

421

Boolean, 22, 69, 176, 186 with if else, 184 functions, 193 short circuit evaluation, 191

branch coverage testing, 199 bug, 13

**C**

calendar project 6 functions, 214 capacity (vector method), 285 cctype, 68 ceil function, 56 char, 204 functions, 68, 70

cin, 28 as loop test, 240 class construct, 141

BankAccount, 112 defi nition, 141, 143 diagram, 109

Grid, 121 ifstream, 263 iostream, 18 List<Type>, 364 Matrix, 379 ofstream, 274 string, 112 Set<Type>, 330 vector<Type>, 279 why use classes?, 128

cmath functions, 55 cohesion, 159 collections, 325 comments, 22 compile time error, 38 compiler, 38, 157 const, 34, 160 const&, 92, 161 const& with vectors, 296 const reference parameter, 93, 94 constructor, 146, 151 constant objects, 34

**D**

data member, 107, 143, 146–149 default constructors, 151 delete operator, 360 design, 2 determinate loop, 223, 228, 282 debugging, 22, 235, 236, 254 distance formula, 83 division, integer (5/2 is 2), 50 do while, 244 dynamic memory allocation, 356

**E**

Einstein's number project, 53

Elevator class project, 261

Employee class project, 171 federal income tax, 218

encapsulation, 129 end of fi le, 267 erase (string method), 114 errors, 38

compile time, 39 intent errors, 44 link time errors, 43 run time errors, 43 with functions, 98 with messages, 110 escape sequence, 204 evaluations, short circuit, 191 exception, 342, 376 executable program, 43 expression, arithmetic, 30

**F** fabs function, 56 false, 28, 69, 177 fi le streams, 263 end of fi le 267 fi le input, 291

nd (string method), 113 fl oating-point, 23 oor function, 56 fl owcharts of

if statement, 177 if . . . else statement, 182 for loop, 225 while loop, 237

for loops 225

free functions implementing, 77 using, 55

fstream, 263 functions

constructor function, 146 free functions, 55 function block, 77 function heading, 62, 119 member functions, 148 overloading functions, 153 why use functions?, 128

**G**

g++, 157 GameOfLife project, 392 garbage, 42, 248 generic collections, 325

getline, 29, 272

global identifi er, 89 good (ostream method), 117

Grid class 121 member functions move, turnLeft, 124 guarded action pattern, 176

Index

**H**

header .h fi le, 141 headings, class member, 118

**I**

identifi er, 20 local, 86 global, 86 scope of, 86

if statement, 186 if . . . else statement, 181 ifstream, 292 implementation, 5, 8, 156

#include directive, 119 increment operators ++ -- +=, 226 indeterminate loop, 236, 266, 269 indirect addressing, 346 indirection, 343 infi nite loop, 243

INT\_MIN, 233 INT\_MAX, 233 input with cin, 28 input process output (IPO) pattern, 5, 58 input/output, 1 insert (string method), 114 integer, 23

constants, 23 maximum / minimum, 233

mixed with fl oats, 34 quotient / remainder (99 % 2 is 1), 32, 50

integer constants, 23 intent error, 44, 235 interface, 156 isalpha, 70 isblank, 70 isdigit, 70 islower, 68 isupper, 68 iterator algorithmic pattern, 333

423

**K**

keyword, 21

**L**

leap year, 214 length (string method), 112 link time error, 42 linked structure, singly, 362

LinkedPriorityList class, 374 LinkedStack project, 372 linker, 38, 157

List class with add, get, remove, 364

literals, 22 local identifi er, 86 logical expression, 176 loop

design, 247 infi nite, 243 project (6 functions), 257 testing, 247

**M**

main function, 18 maintenance, 13 MasterMind project, 259

Matrix class, 379 addition, 382 scalar multiplication, 381

max, 67 member functions, 118, 148

implementation, 146

memory, 11 memory leak, 360

messages, 109, 289

()always needed, 127

BankAccount messages, 110 istream, 116 Grid messages, 122

ostream, 116 string, 108

methods, 108

accessor methods, 155 accessors are const, 160 BankAccount methods, 107 istream methods, 116 modifying methods, 114, 154 ostream methods, 116 string methods, 108

min function, 67 minimum coins, 54 modifying methods, 114, 124, 154 multiple selection pattern, 195 naming conventions, 155 new, 356

with arrays, 357

**O**

objects, 11

BankAccount, 107 const, 34 cout, 11 diagram, 109 istream, 264 modeling the real world, 105 ofstream, 274 string, 112

Object-Oriented Design Guidelines,

158, 159, 161

operators arithmetic, 30 boolean, 188 delete, 360 new, 356 operator precedence (table), 189 relational, 178 out of range subscripts, 284 output, 1

with cout, 26

overloading functions, 153

**P**

parameters, 65, 94

const reference, 93 value, 92 reference, 90

pass by value, reference, const&, 94

PiggyBank project, 170 pointer, 345

to objects, 350

population prediction project, 101 postcondition, 61 pow function, 55 precedence, 189

arithmetic operators, 31

precision (ostream method), 117 precondition, 61, 127

PriorityList project, 339 private, 155 problem solving, 1 program, 8, 18 prompt then input, 35 public, 155

**Q**

quadratic formula project, 102 quotient remainder %, 32

**R**

range of a projectile, 74 reference parameter, 92, 94 relational operators, 178 repetition, 221

why?, 222 replace (string method), 114

Index

resize (vector modifi er), 285 return statement, 78 ignored 194 multiple, 198

rounding, 74 runtime error, 38, 43

**S**

scope, 86

within classes, 147 within functions, 89

search

binary, 303 sequential, 288

selection, 175

project with six selection functions, 212

sequential search, 288

Set<Type> class, 330 short circuit Boolean evaluation, 191 singly-linked structure, 362 sorting, 297 sqrt function, 55 stack project, 338 state of objects, 11, 12, 154 state object pattern, 154 statements

assignment, 27 declaration, 25 do while, 244 for loop, 225 if, 176 if . . . else, 181 initialization, 25 return, 78 switch, 203 while loop, 237

static, 344 std, 19

425

streams, fi le, 263 string, 113

<< + >> [], 115 methods, 113 operators, 114 project with 10 string functions, 127

struct, 362 structured programming, 129 Student class project, 216 subscripts, 283 substring (string method), 113

swap

functions, 90 vector elements, 301 with pointers, 349

switch statement, 203

**T**

templates, 327 testing, 10, 80, 234 boundary, 200 multiple selection, 199 test driver, 80, 84, 100, 101, 137, 165,

169, 170, 173, 178, 200, 212, 214–216, 257–258, 324

using the assert function, 200

time travel project, 75 tokens, 20 true, 28, 69, 117, 118, 177 torus, 395

type

arguments, 326 compound, 12 promotion, 34

**V**

value parameter, 92, 94 vectors, 279 accessing methods

arguments, 294 at, 285 capacity, 285 of vectors, 377 processing, 283 project with 8 vector functions, 320 resize, 285 sorting, 297 vector / array, diff erences, 353

void functions, 90

**W**

warnings, 38, 41 weighted average, 15, 52 while statement, 237 wholesale cost, 15 width (ostream method), 116

**Z**

zero (false), 177

© 2018 Franklin, Beedle & Associates All rights reserved

© 2018 Franklin, Beedle & Associates All rights reserved

© 2018 Franklin, Beedle & Associates All rights reserved

© 2018 Franklin, Beedle & Associates All rights reserved