Student MANUAL

##### COM 203

Intermediate Game Programming C++



****



# TABLE OF CONTENTS

USING THIS INSTRUCTOR GUIDE

Purpose of this Instructor Guide 3

Structure of the Guide 3

COURSE OUTLINE 4

Description of the Course 4

Learning Outcomes 4

Learning Objectives 4

Information Resources for this Course 5

Textbook: 5

Required Web Site Readings 5

Course Materials 7

Lesson Topics for this Course 5

Grading Policies and Procedures 8

Requirements 10

Coleman College Policy on Academic Honesty 11

Course Assignments

Summary of Assignments

Lesson 5:

Assignment 1: Tic-Tac-Toe

Lesson 8:

Assignment 2: Maze

Lesson 14:

Assignment 3: Black Jack

Lesson 16:

Assignment 4:Hangman

Lesson 19 :

Assignment 5: Pong

ATTACHMENT 1:

Lesson 1: 13

Lesson 2: 19

Lesson 3: 20

Lesson 4: 25

Lesson 5: 31

Lesson 6: 37

Lesson 7: 42

Lesson 8: 48

Lesson 9: 52

Lesson 10: 60

Lesson 11: 66

Lesson 12: 73

Lesson 13: 78

Lesson 14: 81

Lesson 15: 85

Lesson 16: 93

Lesson 17: 99

CHARTS

Chart 1: Summary of Lessons/Topics, Reading Assignments, and Class Projects/Assignments

Chart 2. Distribution of Points for Class Assignments

Chart 3. Coleman College Grade Assignment Policy

Chart 4. Summary of Course Assignments

# **USING THIS INSTRUCTOR GUIDE**

# **COM 203 Intermediate Game Programming C++**

## Purpose of this Instructor Guide

The primary purpose of this Guide is to facilitate consistency in the substance and quality of offerings of COM 203 by various instructors who will teach this course as part of the Coleman College curriculum. The Guide is *not* intended as a strict prescription for exactly what must be taught, or how the content is to be taught. Rather, the Guide is intended to ensure that students taking this course at different times or in different places have roughly equivalent experiences.

## Structure of the Guide

The Course Outline in this Instructor Guide contains the following sections:

* A brief description of the course, reflecting the course description in the College catalog.
* Learning Outcomes expected to result from participation in the course. These Learning Outcomes define the “big picture” intentions of the course in terms of student learning.
* Learning Objectives, specific statements of the details of what is to be taught and learned in the course, and under what conditions students should be able to demonstrate what they have learned.
* Instructional resources, what reading students are expected to complete in the course, including any required textbook, Websites, or other instructor-provided learning materials.
* A summary of topics to be addressed in the course, organized by Lessons.
* Grading policies and procedures to be used in this course.
* Course assignments: In this section, detailed information about each suggested assignment is provided. These are suggested assignments; individual instructors may choose to utilize different assignments.
* Attachment 1 contains a compendium of resources for instructors to use in planning and implementing their teaching of this course. Attachment 1 is organized by Lessons, which often (but not always) correspond with weeks in the academic calendar. Each Lesson in Attachment 1 is further divided into two sections:
  + The first section offers a suggested structure for the Lesson, describing a sequence of activities the Instructor may wish to implement in teaching the lesson.
  + The second section, Lesson Notes, contains content the Instructor may wish to consult for his or her own learning, or may be used as notes for lecture and discussion sessions. This section also contains an inventory of Websites the Instructor may wish to consult, or may wish to assign or suggest students look at. The Websites listed in this section are reproduced in the Student Manual.
* Attachment 2 contains materials the Instructor may wish to use to implement or support the instructional activities suggested in Attachment 1. Materials include problem scenarios, checklists, survey instruments, small group activity materials, etc. Again, these are meant as suggestions, not prescriptions.

Suggested allocation of class time across Lessons

The content of this course has been grouped into 8 Lessons. As the course takes place over a period of 10 weeks, the Instructor has discretion over which lessons may take longer than one week. Following is a suggestion for time allocation to the various lessons:

Lesson 1 1 Class Lesson 10 2 Classes

Lesson 2 1 Class Lesson 11 1 Class

Lesson 3 1 Class Lesson 12 1 Class

Lesson 4 1 Class Lesson 13 1 Class

Lesson 5 1 Class Lesson 14 2 Classes

Lesson 6 1 Class Lesson 15 3 Classes

Lesson 7 1 Class

Lesson 8 2 Classes

Lesson 9 1 Class

# COURSE OUTLINE

## Description of the Course

This course enables the student to write well structured C++ programs. The students are introduced to the Object Oriented Programming (OOP) concepts such as abstraction, inheritance, encapsulation and polymorphism along with basic C++ concepts. The students are also introduced to OGRE library which will enable them to understand the basic concepts of graphics and game programming.

## Learning Outcomes

***Upon successful completion of the course, students will be able to:***

* Design and implement structured, modular object orientated programs in C/C++.
* Understand Object Technology and the UML Concepts
* Use existing C++ code and libraries in projects and produce libraries and share code for use in other projects.
* Design, implement and test computer programs and games using Visual Studio and Ogre Engine

## Learning Objectives

To achieve the learning outcomes specified for this course, students will upon successful completion of the course:

* Create a program that uses pointers.
* Create a program that uses files.
* Create a program that uses the data structures.
* Create a program that uses operator overloading.
* Create a game program that uses OGRE Engine.

## Information Resources for this Course

### 🕮 Textbook: Deitel & Deitel. 2009 C++ How to Program – Seventh Edition, Publisher: Prentice Hall. ISBN-13: 978-0136117261

## Lesson Topics for this Course

COM 203 has been organized into [15] lessons or topical areas, as follows:

* Lesson 1: Introduction to C++ and OOAD
* Lesson 2: Control Structures
* Lesson 3: Control Structures Part 2
* Lesson 4: Functions
* Lesson 5: Arrays and Vectors
* Lesson 6: Pointers
* Lesson 7: Classes: A Deeper Look Part 1
* Lesson 8: Classes: A Deeper Look Part 2
* Lesson 9: Operator Overloading
* Lesson 10: Object-Oriented Programming: Inheritance
* Lesson 11: Object-Oriented Programming: Polymorphism
* Lesson 12: Exception Handling
* Lesson 13: File Processing
* Lesson 14: Data Structures
* Lesson 15: Ogre Library

## Topics To Be Addressed In This Course

Chart 1 contains a list of the lessons and topics to be addressed in the course, the readings you are expected to complete for each topic, and the assignments associated with each lesson or topic.

Chart 1. Summary of Lessons/Topics, Reading Assignments, and Class Projects/Assignments

| ***CLASS*** | ***Topic*** | ***Reading*** | ***Project***  ***Assigned*** | ***Project***  ***Due*** |
| --- | --- | --- | --- | --- |
| 1 | Lesson 1: Introduction to C++ and OOAD | Chapter 2 and Chapter 3 |  |  |
| 2 | Lesson 2: Control Structures | Chapter 4 |  |  |
| 3 | Lesson 3: Control Structures Part 2 | Chapter 5 |  |  |
| 4 | Lesson 4: Functions | Chapter 6 |  |  |
| 5 | Lesson 5: Arrays and Vectors | Chapter 7 | Tic- Tac -Toe |  |
| 6 | Lesson 6: Pointers | Chapter 8 |  |  |
| 7 | Lesson 7: Classes: A Deeper Look Part 1 | Chapter 9 |  |  |
| 8 | Lesson 8: Classes: A Deeper Look Part 2 | Chapter 10 | Maze |  |
| 9 | Lesson 9: Operator Overloading |  |  |  |
| 10 | Midterm | Chapter 11 |  |  |
| 11 | Lesson 10: Object-Oriented Programming: Inheritance | Chapter 12 |  |  |
| 12 | continued |  |  |  |
| 13 | Lesson 11: Object-Oriented Programming: Polymorphism | Chapter 13 | BlackJack |  |
| 14 | Lesson 12: Exception Handling | Chapter 16 |  |  |
| 15 | Lesson 13: File Processing | Chapter 17 | Hangman |  |
| 16 | Lesson 14: Data Structures | Chapter 20 |  |  |
| 17 | continued |  |  |  |
| 18 | Lesson 15: OGRE | Chapter 27 | Pong |  |
| 19 | Continued |  |  |  |
| 20 | Final |  |  |  |

## Grading Policies and Procedures

Students in this course should be graded following Coleman College assessment practices and policies. A point system is used in the College to indicate student performance on various required activities or projects. For this course, it is recommended that points be distributed as follows:

Chart 2. Distribution of points for class assignments

|  |  |  |  |
| --- | --- | --- | --- |
| Lesson | Assignment | Possible  Points | Percent of  Grade |
| 5 | 1 | 10 | 10% |
| 8 | 2 | 10 | 10% |
| 11 | 3 | 10 | 10% |
|  | Midterm | 20 | 20% |
| 13 | 4 | 15 | 15% |
|  | Final | 20 | 20% |
| 15 | Final Project | 15 | 15% |
| **TOTAL** |  | **100** | **100%** |

The Coleman College guidelines for the assignment of grades to total points earned is as follows:

Chart 3. Coleman College Grade Assignment Policy

|  |  |  |
| --- | --- | --- |
| **Percent** | **Letter Grade** | **Grade Points** |
| 94-100 | A | 4.0 |
| 90-93 | A- | 3.7 |
| 87-89 | B+ | 3.3 |
| 84-86 | B | 3.0 |
| 80-83 | B- | 2.7 |
| 77-79 | C+ | 2.3 |
| 74-76 | C | 2.0 |
| 70-73 | C- | 1.7 |
| 0-69 | Fail | 0 |
|  | INC | 0 |
|  | W | 0 |
|  | WP | 0 |
|  | WF | 0 |
| 74 or above | CR | 0 |
| 73 or below | NC | 0 |
| 70 or above | PASS | 0 |

## Requirements

**Assignments:** All assignments (including projects, lab work, quizzes and exams) must be completed as scheduled. The following will apply to late assignments:

1st day late = 20% off point value

2nd day late = 60% off point value

3rd day late = No points given

If an assignment equals less than 5 points, no points will be given for late work. If there are extenuating circumstances, the student must submit a written explanation to the department Senior Instructor. Upon evaluation, points will be given according to the Senior Instructor’s discretion.

**Attendance:** Classes begin and end as indicated in the published schedule. It is required that students be present at the beginning of each class session and stay until class is dismissed, including lab periods. Excessive tardiness, leaving early and/or absences (from either lecture or lab sessions) are causes for dismissal from the College.

**Conduct:** Students are expected to conduct themselves in a professional manner while on campus. Rules of conduct are outlined in the College Catalog and students are required to adhere to such policies.

**Coleman College Policy on Academic Honesty**

***Revised 04/05***

**Coleman College insists upon academic honesty.** All work submitted by students for a grade must be ***independent, original work.***

**Academic dishonesty is cause for dismissal from Coleman College.**

Presenting another person's coursework, ideas, methods, or test answers with the intention that they are taken as one's own is theft of a special kind. It defrauds the originator of the work, the institution, its graduates, its current students and its future students.

**Students are required to do their own work.** ***The student has full responsibility for the authenticity of all academic work and examinations submitted***

**Academic dishonesty includes both plagiarism and other offenses**

***Plagiarism is the presentation of someone else's ideas as your own.*** Coleman College course facilitators are encouraged to check for plagiarism.

The following are examples of ***other types of academic dishonesty:***

* Allowing someone else to do your work on a program or graded assignment.
* Copying from or using someone else's program, documentation, or graded assignment.
* Being in possession of another student's program code listing, graded assignment, or disk (that contains program code listings or graded assignments).
* Sharing your password with another student or allowing another student to log on to your network account with your sign on, or granting another student unauthorized access to your network directory.
* Unauthorized access of someone else's network directory or network account.
* Copying another student's exam questions for a graded test.
* Sharing test questions with another student for a graded test.

**The following are Coleman College policies on academic dishonesty**

When academic dishonesty is found, the student(s) involved may be asked to explain the situation. The facilitator will evaluate the situation and notify the student and Coleman College by email and indicate to the student, by email, that Coleman College has been notified of the offense. (This may be accomplished in the same email.)

Coleman College will then check the student's record to see if there have been previous offenses.

If there are no previous offenses, Coleman College will record the offense in the student's record and notify the facilitator who will record a zero or reduced grade for that assignment and warn the student that further offenses will result in more serious penalties, including dismissal from the course with a failing grade and suspension from school.

If there is one previous offense, Coleman College will record the offense in the student's record and notify the student that he or she has been placed on academic probation. The student may also be dismissed from class with a failing grade or receive a lesser penalty, such as a zero for the assignment.

If there are two or more previous offenses, the Dean of Academics will notify the student that he or she is suspended from the College.

Regardless of the existence of previous offenses, if any offense is particularly objectionable, Coleman College or Dean of Academics may impose more serious penalties, including dismissal from the course with a failing grade and/or suspension from school.

**Take responsibility**

***When an incident of plagiarism or other academic dishonesty occurs, both the student who gave the information and the student who received the information may be subject to penalties.*** Do not allow yourself to be put in this position. When you need assistance, contact your instructor.

***If you have any questions about the policy on academic dishonesty, contact your instructor or Coleman College.***

Course Assignments

This section of the Instructor Guide offers suggestions for assignments to be given to students in COM 203. Each assignment pertains to a specific lesson topic, and contributes to student pursuit of the Learning Objectives of the course. The material contained herein is reproduced in the Student Manual.

#### Summary of Course Assignments

Chart 4. Summary of Course Assignments, COM 203

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Lesson | Assignment | Task | Possible  Points | Week  Assigned | Week  Due |
| 5 | 1 | Game Tic-Tac-Toe | 10 | 1 | 3 |
| 8 | 2 | Game Maze | 10 | 3 | 4 |
| 11 | 3 | Game BlackJack | 10 | 4 | 7 |
| 13 | 4 | Game Hangman | 15 | 7 | 10 |
| 15 | 5 | Pong | 15 | 9 |  |
| **TOTAL** | **100** |  |  |  |  |

###### Lesson 5: Tic- Tac- Toe

Assignment 1: Create a text based game Tic-Tac-Toe using pointers There are no graphics involved in this project. The program should accept the location on the text-based board and automatically generate the move by the computer and print the result.

###### Lesson 8: Maze

**Assignment 3: Create a text based game Maze.**

###### Lesson 11: Blackjack

**Assignment4: Create a text based game Black Jack.**

###### Lesson 13: Hangman

Assignment 2: Create a text based game Hangman. This program should use a file that stores words and accept input from the user.

###### Lesson 15: Pong

**Assignment4: Create a Pong game using the OGRE Engine.**

# Attachment 1 **Suggested Lesson Structure and Notes**

**Lesson 1: Introduction to C++ and OOAD**

**C++ Standard Library**

C++ programs consist of pieces called classes and functions. You can program each piece that you may need to form a C++ program. However, most C++ programmers take advantage of the rich collections of existing classes and functions in the **C++ Standard Library**. Thus, there are really two parts to learning the C++ "world." The first is learning the C++ language itself; the second is learning how to use the classes and functions in the C++ Standard Library. Throughout the book, we discuss many of these classes and functions.

The standard class libraries generally are provided by compiler vendors. Many special-purpose class libraries are supplied by independent software vendors.

*Software Engineering Observation 1:*

Use a "building-block" approach to create programs. Avoid reinventing the wheel. Use existing pieces wherever possible. Called **software reuse**, this practice is central to object-oriented programming.

*Software Engineering Observation 2:*

When programming in C++, you typically will use the following building blocks: Classes and functions from the C++ Standard Library, classes and functions you and your colleagues create and classes and functions from various popular third-party libraries.

**Software Engineering Case Study: Introduction to Object Technology and the UML Basic Object Technology Concepts**

Humans think in terms of objects. Objects can be seen everywhere you look in the real world; people, animals, plants, cars, planes, buildings, computers and so on.

We sometimes divide objects into two categories: animate and inanimate. Animate objects are "alive" in some sense they move around and do things. Inanimate objects, on the other hand, do not move on their own. Objects of both types, however, have some things in common. They all have **attributes** (e.g., size, shape, color and weight), and they all exhibit **behaviors** (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We will study the kinds of attributes and behaviors that software objects have.

Humans learn about existing objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees.

**Object-oriented design (OOD)** models software in terms similar to those that people use to describe real-world objects. It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics cars, trucks, little red wagons and roller skates have much in common. OOD takes advantage of **inheritance** relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class "convertible" certainly has the characteristics of the more general class "automobile," but more specifically, the roof goes up and down.

Object-oriented design provides a natural and intuitive way to view the software design process namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.

OOD **encapsulates** (i.e., wraps) attributes and **operations** (behaviors) into objects an object's attributes and operations are intimately tied together. Objects have the property of **information hiding**. This means that objects may know how to communicate with one another across well-defined **interfaces**, but normally they are not allowed to know how other objects are implemented, implementation details are hidden within the objects themselves. We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally as long as we know how to use the accelerator pedal, the brake pedal, the steering wheel and so on. Information hiding, as we will see, is crucial to good software engineering.

Languages like C++ are object oriented. Programming in such a language is called **object-oriented programming (OOP)**, and it allows computer programmers to implement an object-oriented design as a working software system. Languages like C, on the other hand, are **procedural**, so programming tends to be action oriented. In C, the unit of programming is the **function**. In C++, the unit of programming is the **class** from which objects are eventually **instantiated** (an OOP term for "created"). C++ classes contain functions that implement operations and data that implements attributes.

C programmers concentrate on writing functions. Programmers group actions that perform some common task into functions, and group functions to form programs. Data is certainly important in C, but the view is that data exists primarily in support of the actions that functions perform. The verbs in a system specification help the C programmer determine the set of functions that will work together to implement the system.

#### Classes, Data Members and Member Functions

C++ programmers concentrate on creating their own **user-defined types** called classes. Each class contains data as well as the set of functions that manipulate that data and provide services to clients (i.e., other classes or functions that use the class). The data components of a class are called **data members**. For example, a bank account class might include an account number and a balance. The function components of a class are called **member functions** (typically called **methods** in other object-oriented programming languages such as Java). For example, a bank account class might include member functions to make a deposit (increasing the balance), make a withdrawal (decreasing the balance) and inquire what the current balance is. The programmer uses built-in types (and other user-defined types) as the "building blocks" for constructing new user-defined types (classes). The nouns in a system specification help the C++ programmer determine the set of classes from which objects are created that work together to implement the system.

Classes are to objects as blueprints are to houses a class is a "plan" for building an object of the class. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class. You cannot cook meals in the kitchen of a blueprint; you can cook meals in the kitchen of a house. You cannot sleep in the bedroom of a blueprint; you can sleep in the bedroom of a house.

Classes can have relationships with other classes. For example, in an object-oriented design of a bank, the "bank teller" class needs to relate to other classes, such as the "customer" class, the "cash drawer" class, the "safe" class, and so on. These relationships are called **associations**.

Packaging software as classes makes it possible for future software systems to reuse the classes. Groups of related classes are often packaged as reusable components. Just as realtors often say that the three most important factors affecting the price of real estate are "location, location and location," people in the software development community often say that the three most important factors affecting the future of software development are "reuse, reuse and reuse."

#### Introduction to Object-Oriented Analysis and Design (OOAD)

Soon you will be writing programs in C++. How will you create the code for your programs? Perhaps, like many beginning programmers, you will simply turn on your computer and start typing. This approach may work for small programs, but what if you were asked to create a multiplayer game with all the bells and whistles? Or what if you were asked to work on a team of 1,000 game developers building the next generation strategy game? For projects so large and complex, you could not simply sit down and start writing programs.

To create the best solutions, you should follow a detailed process for **analyzing** your project's requirements (i.e., determining what the system is supposed to do) and developing a **design** that satisfies them (i.e., deciding how the system should do it). Ideally, you would go through this process and carefully review the design (or have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it is called **object-oriented analysis and design (OOAD)**. Experienced programmers know that analysis and design can save many hours by helping avoid an ill-planned system development approach that has to be abandoned partway through its implementation, possibly wasting considerable time, money and effort.

OOAD is the generic term for the process of analyzing a problem and developing an approach for solving it. Small problems like the ones discussed in these first few chapters do not require an exhaustive OOAD process. It may be sufficient, before we begin writing C++ code, to write **pseudocode** an informal text-based means of expressing program logic. It is not actually a programming language, but we can use it as a kind of outline to guide us as we write our code.

As problems and the groups of people solving them increase in size, the methods of OOAD quickly become more appropriate than pseudocode. Ideally, a group should agree on a strictly defined process for solving its problem and a uniform way of communicating the results of that process to one another. Although many different OOAD processes exist, a single graphical language for communicating the results of any OOAD process has come into wide use. This language, known as the Unified Modeling Language (UML), was developed in the mid-1990s under the initial direction of three software methodologists: Grady Booch, James Rumbaugh and Ivar Jacobson.

#### History of the UML

In the 1980s, increasing numbers of organizations began using OOP to build their applications, and a need developed for a standard OOAD process. Many methodologists including Booch, Rumbaugh and Jacobson individually produced and promoted separate processes to satisfy this need. Each process had its own notation, or "language" (in the form of graphical diagrams), to convey the results of analysis and design.

By the early 1990s, different organizations, and even divisions within the same organization, were using their own unique processes and notations. At the same time, these organizations also wanted to use software tools that would support their particular processes. Software vendors found it difficult to provide tools for so many processes. Clearly, a standard notation and standard processes were needed.

In 1994, James Rumbaugh joined Grady Booch at Rational Software Corporation (now a division of IBM), and the two began working to unify their popular processes. They soon were joined by Ivar Jacobson. In 1996, the group released early versions of the UML to the software engineering community and requested feedback. Around the same time, an organization known as the **Object Management Group™ (OMG™)** invited submissions for a common modeling language. The OMG ([www.omg.org](http://www.omg.org)) is a nonprofit organization that promotes the standardization of object-oriented technologies by issuing guidelines and specifications, such as the UML. Several corporations among them HP, IBM, Microsoft, Oracle and Rational Software had already recognized the need for a common modeling language. In response to the OMG's request for proposals, these companies formed UML Partners the consortium that developed the UML version 1.1 and submitted it to the OMG. The OMG accepted the proposal and, in 1997, assumed responsibility for the continuing maintenance and revision of the UML. In March 2003, the OMG released UML version 1.5. The UML version 2which had been adopted and was in the process of being finalized at the time of this publication marks the first major revision since the 1997 version 1.1 standard. Many books, modeling tools and industry experts are already using the UML version 2, so we present UML version 2 terminology and notation throughout this course.

#### What Is the UML?

The Unified Modeling Language is now the most widely used graphical representation scheme for modeling object-oriented systems. It has indeed unified the various popular notational schemes. Those who design systems use the language (in the form of diagrams) to model their systems, as we do throughout this book.

An attractive feature of the UML is its flexibility. The UML is **extensible** (i.e., capable of being enhanced with new features) and is independent of any particular OOAD process. UML modelers are free to use various processes in designing systems, but all developers can now express their designs with one standard set of graphical notations.

The UML is a complex, feature-rich graphical language. Learning this will help you to create efficient software designs for your Capstone project.

Exercise:

1. The \_\_\_\_\_\_\_\_\_\_\_ in a system specification help the C++ programmer determine the set of classes to implement the system.
2. The \_\_\_\_\_\_\_ or \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_ is now the most widely used graphical representation scheme for modeling object-oriented systems.
3. Pseudocode is \_\_\_\_\_\_\_\_.
4. another term for OOAD
5. a programming language used to display UML diagrams
6. an informal means of expressing program logic
7. a graphical representation scheme for modeling object-oriented systems
8. The UML is used primarily to \_\_\_\_\_\_\_\_.
9. test object-oriented systems
10. design object-oriented systems
11. implement object-oriented systems
12. Both a and b

**First Program in C++: Printing a Line of Text**

C++ programs consist of pieces called classes and functions. You can program each piece that you may need to form a C++ program.

*Refer to Pages 40 – 43 in the textbook for the program*

NOTE: Indent the entire body of each function one level within the braces that delimit the body of the function. This makes a program's functional structure stand out and helps make the program easier to read.

Our next program uses the input stream object **std::cin** and the **stream extraction operator**, >>, to obtain two integers typed by a user at the keyboard, computes the sum of these values and outputs the result using std::cout.

*Refer to Pages 45 – 49 in the textbook*

Class Assignment 1:

Write a program to accept 2 numbers from the user. Perform addition, subtraction, multiplication and division of the the numbers and print the result on the screen. Use appropriate names for all the variables, provide necessary comments.

Exercises:

Fill in the blanks in each of the following.

1. Every C++ program begins execution at the function \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.
2. The \_\_\_\_\_\_\_\_\_ begins the body of every function and the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ends the body of every function.
3. Every C++ statement ends with a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.
4. The escape sequence \n represents the \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_character, which causes the cursor to position to the beginning of the next line on the screen.
5. The \_\_\_\_\_\_\_\_\_ statement is used to make decisions.

State whether each of the following is true or false. If false, explain why. Assume the statement using std::cout; is used.

1. Comments cause the computer to print the text after the // on the screen when the program is executed.
2. The escape sequence \n, when output with cout and the stream insertion operator, causes the cursor to position to the beginning of the next line on the screen.
3. All variables must be declared before they are used.
4. All variables must be given a type when they are declared.
5. C++ considers the variables number and NuMbEr to be identical.
6. Declarations can appear almost anywhere in the body of a C++ function.
7. The modulus operator (%) can be used only with integer operands.
8. The arithmetic operators \*, /, %, + and all have the same level of precedence.
9. A C++ program that prints three lines of output must contain three statements using cout and the stream insertion operator.

Write a single C++ statement to accomplish each of the following (assume that using declarations have not been used):

1. Declare the variables c, thisIsAVariable, q76354 and number to be of type int.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Prompt the user to enter an integer. End your prompting message with a colon (:) followed by a space and leave the cursor positioned after the space.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Read an integer from the user at the keyboard and store the value entered in integer variable age.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. If the variable number is not equal to 7, print "The variable number is not equal to 7".

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Print the message "This is a C++ program" on one line.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Print the message "This is a C++ program" on two lines. End the first line with C++.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Print the message "This is a C++ program" with each word on a separate line.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Print the message "This is a C++ program" with each word separated from the next by a tab.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Write a statement (or comment) to accomplish each of the following (assume that using declarations have been used):

1. State that a program calculates the product of three integers.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Declare the variables x, y, z and result to be of type int (in separate statements).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Prompt the user to enter three integers.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Read three integers from the keyboard and store them in the variables x, y and z.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Compute the product of the three integers contained in variables x, y and z, and assign the result to the variable result.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Print "The product is " followed by the value of the variable result.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Return a value from main indicating that the program terminated successfully.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Using the statements you wrote above, write a complete program that calculates and displays the product of three integers. Add comments to the code where appropriate. [Note: You will need to write the necessary using declarations.]

Identify and correct the errors in each of the following statements (assume that the statement using std::cout; is used):

1. if ( c < 7 );

cout << "c is less than 7\n";

1. if ( c => 7 )

cout << "c is equal to or greater than 7\n";

**Introduction to Classes and Objects**

Modern OO languages provide the programmer with three capabilities that improve and simplify the design of programs: encapsulation, inheritance, and polymorphism (or generic functionality).

Related topics involve *objects, classes,* and *data abstraction.*

***Abstraction***in the object oriented world generalizes the characteristics and behaviors of an item. Example: Automobile.

***Encapsulation*** conceals the functional details of an object from other objects, i.e., the object operates as a black box. Example: The functionality of brake is hidden.

***Inheritance*** is a way to form new classes using classes that have already been defined. Inheritance is employed to help reuse of existing code with little or no modification.Example: Animal, Tiger.

***Polymorphism*** refers to the ability of executing different operations in response to the same message.Example: Movement in various types of animals such as Frog, Bird, Snake, Fish etc.

**Classes and Objects**

Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal. What must happen before you can do this? Well, before you can drive a car, someone has to design it and build it. A car typically begins as engineering drawings, similar to the blueprints used to design a house. These drawings include the design for an accelerator pedal that the driver will use to make the car go faster. In a sense, the pedal "hides" the complex mechanisms that actually make the car go faster, just as the brake pedal "hides" the mechanisms that slow the car, the steering wheel "hides" the mechanisms that turn the car and so on. This enables people with little or no knowledge of how cars are engineered to drive a car easily, simply by using the accelerator pedal, the brake pedal, the steering wheel, the transmission shifting mechanism and other such simple and user-friendly "interfaces" to the car's complex internal mechanisms.

Unfortunately, you cannot drive the engineering drawings of a car before you can drive a car, it must be built from the engineering drawings that describe it. A completed car will have an actual accelerator pedal to make the car go faster. But even that's not enough the car will not accelerate on its own, so the driver must press the accelerator pedal to tell the car to go faster.

Performing a task in a program requires a function (such as main). The function describes the mechanisms that actually perform its tasks. The function hides from its user the complex tasks that it performs, just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster. In C++, we begin by creating a program unit called a ***class*** to house a function, just as a car's engineering drawings house the design of an accelerator pedal. A function belonging to a class is called a member function. In a class, you provide one or more member functions that are designed to perform the class's tasks.

Just as you cannot drive an engineering drawing of a car, you cannot "drive" a class. Just as someone has to build a car from its engineering drawings before you can actually drive the car, you must create an object of a class before you can get a program to perform the tasks the class describes. That is one reason C++ is known as an object-oriented programming language. Note also that just as many cars can be built from the same engineering drawing, many objects can be built from the same class.

Thus classes enable data abstraction and data encapsulation.

When you drive a car, pressing its gas pedal sends a message to the car to perform a task that is, make the car go faster. Similarly, you send **messages** to an object each message is known as a **member-function call** and tells a member function of the object to perform its task. This is often called requesting a service from an object.

In addition to the capabilities a car provides, it also has many **attributes**, such as its color, the number of doors, the amount of gas in its tank, its current speed and its total miles driven (i.e., its odometer reading). Like the car's capabilities, these attributes are represented as part of a car's design in its engineering diagrams. As you drive a car, these attributes are always associated with the car. ***Every car maintains its own attributes***. For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars. Similarly, an object has attributes that are carried with the object as it is used in a program. These attributes are specified as part of the object's class.

A class is created using the keyword **class**.

A class can contain private as well as public members. By default, all items defined in a class are **private**. When the members are declared **private**, only the members of the class can access them but not any other part of your program.

This is one way encapsulation is achieved—you can tightly control access to certain items of data by keeping them private. You can also define **private** functions, which can be called only by other members of the class.

To make parts of a class public (i.e., accessible to other parts of your program), you must declare them after the **public** keyword.

All variables or functions defined after the **public** specifier are accessible by all other functions in your program.

Typically, your program will access the private members of a class through its public functions.

General form of a class definition:

**class class\_name**

**{**

**private: (optional)**

**private data and functions**

**public:**

**public data and functions**

**}object\_list;**

Note : You cannot initialize the variables inside the class declaration. Variables can be initialized using constructors.

**Defining a Class with a Member Function**

*Refer to Pages 71 – 73 in the textbook for the program*

**Class GradeBook**

The GradeBook class definition contains a member function called displayMessage that displays a message on the screen . Recall that a class is like a blueprint so we need to make an object of class GradeBook and call its displayMessage member function execute and display the welcome message.

The **class definition** begins with the keyword class followed by the class name GradeBook. By convention, the name of a user-defined class begins with a capital letter, and for readability, each subsequent word in the class name begins with a capital letter. This capitalization style is often referred to as **camel case**, because the pattern of uppercase and lowercase letters resembles the silhouette of a camel.

Every class's **body** is enclosed in a pair of left and right braces ({ and }). The class definition terminates with a semicolon.

Recall that the function main is always called automatically when you execute a program. Most functions do not get called automatically. As you will soon see, you must call member function displayMessage explicitly to tell it to perform its task.

The keyword **public** is called an **access specifier**. The member function appears after access specifier public: to indicate that the function is "available to the public" that is, it can be called by other functions in the program and by member functions of other classes. Access specifiers are always followed by a colon (:).

Each function in a program performs a task and may return a value when it completes its task for example, a function might perform a calculation, then return the result of that calculation. When you define a function, you must specify a **return** type to indicate the type of the value returned by the function when it completes its task. The keyword **void** to the left of the function name displayMessage is the function's return type. Return type void indicates that displayMessage will perform a task but will not return (i.e., give back) any data to its **calling function** (in this example, main, as we'll see in a moment) when it completes its task.

The name of the member function, displayMessage, follows the return type. By convention, function names begin with a lowercase first letter and all subsequent words in the name begin with a capital letter. The parentheses after the member function name indicate that this is a function. An empty set of parentheses indicates that this member function does not require additional data to perform its task.

The body of a function contains statements that perform the function's task. In this case, member function displayMessage contains one statement (line 15) that displays the message "Welcome to the Grade Book!” After this statement executes, the function has completed its task.

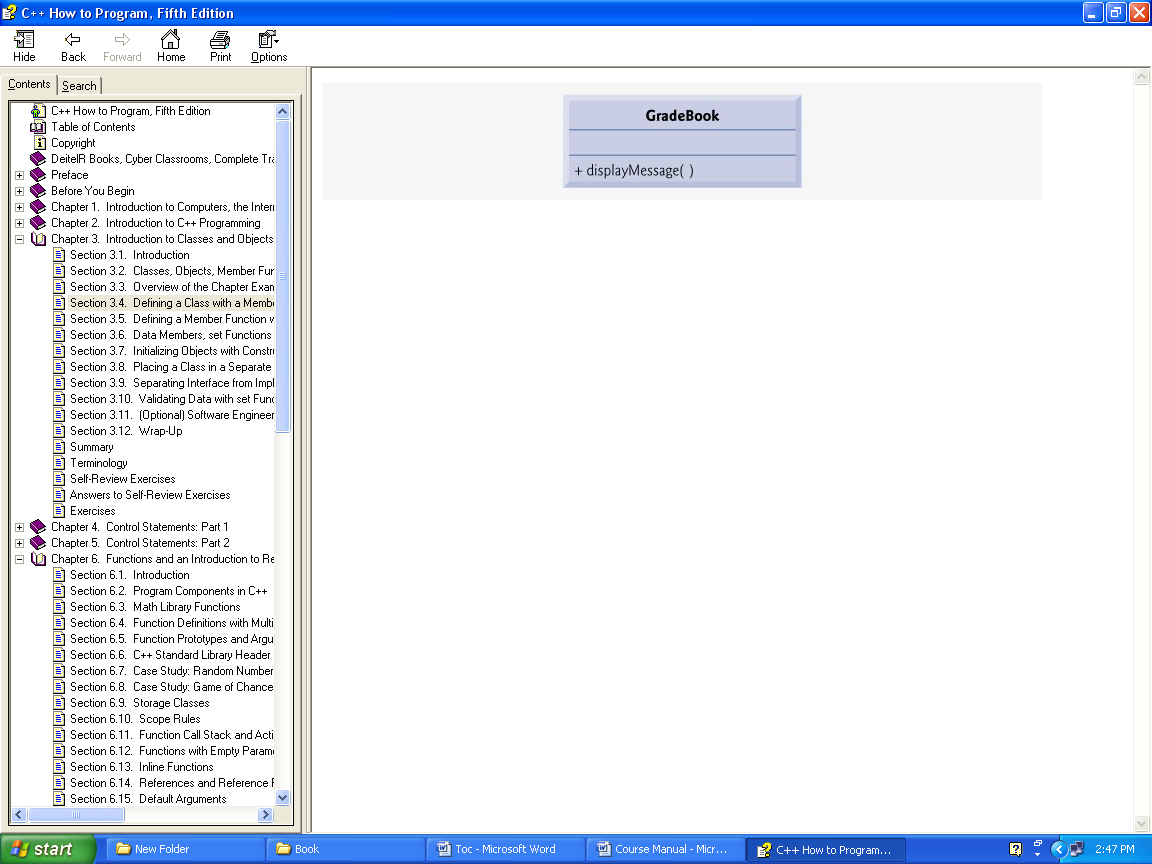
#### UML Class Diagram for Class GradeBook

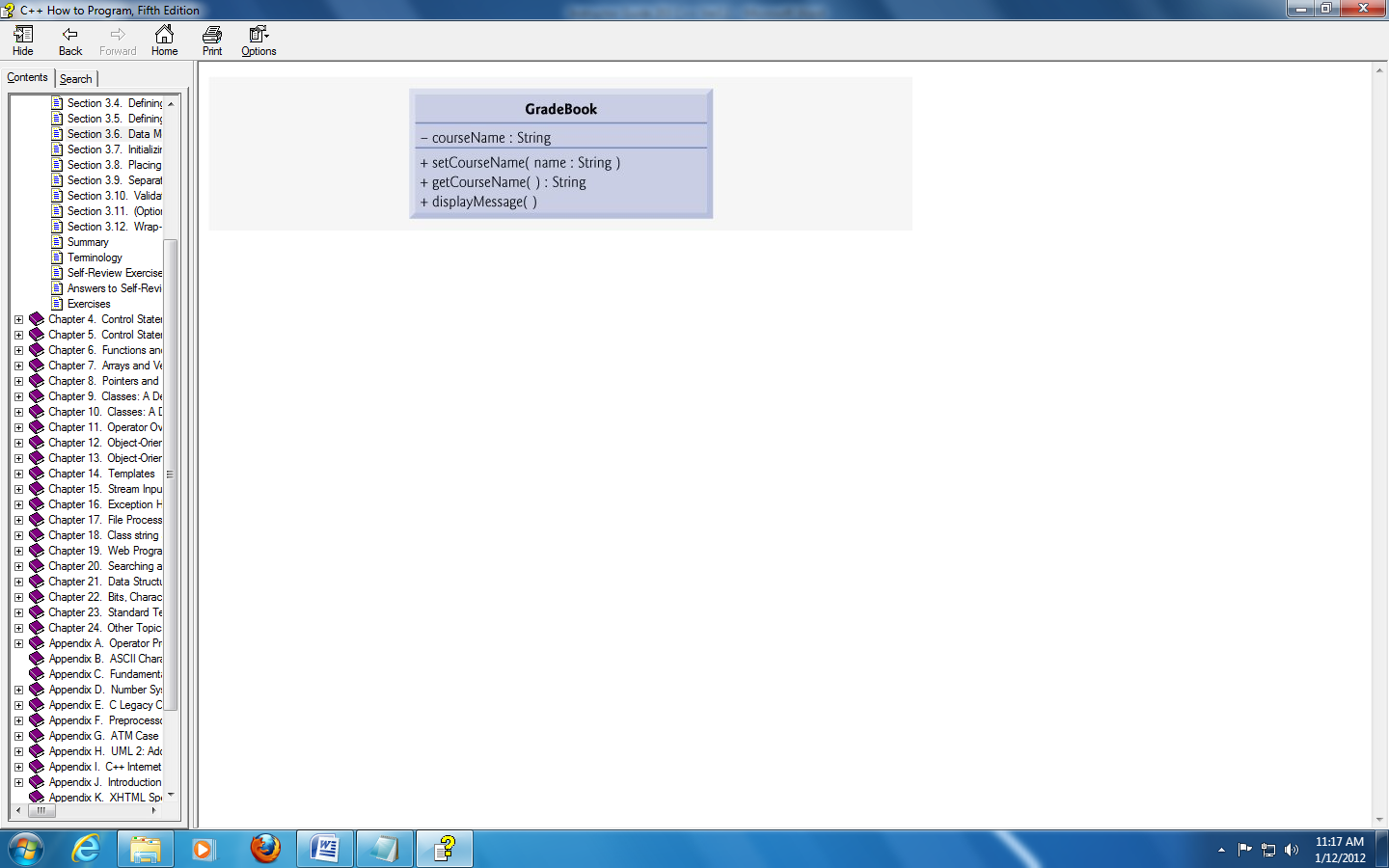
The UML is a graphical language used by programmers to represent their object-oriented systems in a standardized manner. In the UML, each class is modeled in a class diagram as a rectangle with three compartments. The following figure presents a **UML class diagram** for class GradeBook. The top compartment contains the name of the class, centered horizontally and in boldface type. The middle compartment contains the class's attributes, which correspond to data members in C++. The middle compartment is empty, because the version of class GradeBook does not have any attributes. The bottom compartment contains the class's operations, which correspond to member functions in C++. The UML models operations by listing the operation name followed by a set of parentheses. The class GradeBook has only one member function, displayMessage, so the bottom compartment lists one operation with this name. Member function displayMessage does not require additional information to perform its tasks, so the parentheses following displayMessage in the class diagram are empty, just as they are in the member function's header. The plus sign (+) in front of the operation name indicates that displayMessage is a public operation in the UML (i.e., a public member function in C++).

The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type. Data member are private in C++, so the class diagram lists a minus sign (-) in front of the corresponding attribute's name. The minus sign in the UML is equivalent to the private access specifier in C++. The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. Member function getCourseName of class GradeBook has a string return type in C++, so the class diagram shows a String return type in the UML. Note that operations setCourseName and displayMessage do not return values (i.e., they return void), so the UML class diagram does not specify a return type after the parentheses of these operations. The UML does not use void as C++ does when a function does not return a value.

We frequently use UML class diagrams to summarize class attributes and operations.

**UML Class Diagram**





**Member function:**

A member function definition need not be placed inside the class definition. If you want to put it outside the class definition, you need to put the prototype for the function inside the class.

To implement a function that is a member of a class, you must tell the compiler to which class the function belongs by qualifying the function name with the class name.

The**::** is called the scope resolution operator.

**Defining a Member Function with a Parameter**

*Refer to Pages 75 – 78 in the textbook for the program*

In our car analogy, we mentioned that pressing a car's gas pedal sends a message to the car to perform a task make the car go faster. But how fast should the car accelerate? As you know, the farther down you press the pedal, the faster the car accelerates. So the message to the car includes both the task to perform and additional information that helps the car perform the task. This additional information is known as a **parameter** the value of the parameter helps the car determine how fast to accelerate. Similarly, a member function can require one or more parameters that represent additional data it needs to perform its task. A function call supplies values called **arguments** for each of the function's parameters.

Variables declared in a function definition's body are known as **local variables** and can be used only from the line of their declaration in the function to the immediately following closing right brace (}) of the function definition. A local variable must be declared before it can be used in a function. A local variable cannot be accessed outside the function in which it is declared. When a function terminates, the values of its local variables are lost.

A class normally consists of one or more member functions that manipulate the attributes that belong to a particular object of the class. Attributes are represented as variables in a class definition. Such variables are called **data members** and are declared inside a class definition but outside the bodies of the class's member-function definitions. Each object of a class maintains its own copy of its attributes in memory. The example in this section demonstrates a GradeBook class that contains a courseName data member to represent a particular GradeBook object's course name.

**GradeBook Class with a Data Member, a set Function and a get Function**

*Refer to Pages 78 – 79 in the textbook for the program*

**Access Specifiers public and private**

Most data member declarations appear after the access-specifier label **private:** . Like public, keyword private is an access specifier. Variables or functions declared after access specifier private (and before the next access specifier) are accessible only to member functions of the class for which they are declared. Thus, data member courseName can be used only in member functions setCourseName, getCourseName and displayMessage of (every object of) class GradeBook. Data member courseName, because it is private, cannot be accessed by functions outside the class (such as main) or by member functions of other classes in the program. Attempting to access data member courseName in one of these program locations with an expression such as myGradeBook.courseName would result in a compilation error containing a message similar to

cannot access private member declared in class 'GradeBook'

The default access for class members is private so all members after the class header and before the first access specifier are private. The access specifiers public and private may be repeated, but this is unnecessary and can be confusing.

Declaring data members with access specifier private is known as **data hiding**. When a program creates (instantiates) an object of class GradeBook, data member courseName is encapsulated (hidden) in the object and can be accessed only by member functions of the object's class. In class GradeBook, member functions setCourseName and getCourseName manipulate the data member courseName directly (and displayMessage could do so if necessary).

#### Member Functions setCourseName and getCourseName

Member function setCourseName does not return any data when it completes its task, so its return type is void. The member function receives one parametername which represents the course name that will be passed to it as an argument. Suppose, for instance, that a university can print student transcripts containing course names of only 25 characters or fewer. In this case, we might want class GradeBook to ensure that its data member courseName never contains more than 25 characters.

Member function getCourseName returns a particular GradeBook object's courseName. The member function has an empty parameter list, so it does not require additional data to perform its task. The function specifies that it returns a string. When a function that specifies a return type other than void is called and completes its task, the function returns a result to its calling function. For example, when you go to an automated teller machine (ATM) and request your account balance, you expect the ATM to give you back a value that represents your balance. Similarly, when a statement calls member function getCourseName on a GradeBook object, the statement expects to receive the GradeBook's course name (in this case, a string, as specified by the function's return type).

#### Software Engineering with Set and Get Functions

A class's private data members can be manipulated only by member functions of that class. So a client of an object that is, any class or function that calls the object's member functions from outside the object calls the class's public member functions to request the class's services for particular objects of the class. This is why the statements in function main call member functions setCourseName, getCourseName and displayMessage on a GradeBook object. Classes often provide public member functions to allow clients of the class to set (i.e., assign values to) or get (i.e., obtain the values of) private data members. The names of these member functions need not begin with set or get, but this naming convention is common. In this example, the member function that sets the courseName data member is called setCourseName, and the member function that gets the value of the courseName data member is called getCourseName. Note that set functions are also sometimes called **mutators** (because they mutate, or change, values), and get functions are also sometimes called **accessors** (because they access values).

The set and get functions of a class also should be used by other member functions within the class to manipulate the class's private data, although these member functions can access the private data directly.

**Member Function displayMessage**

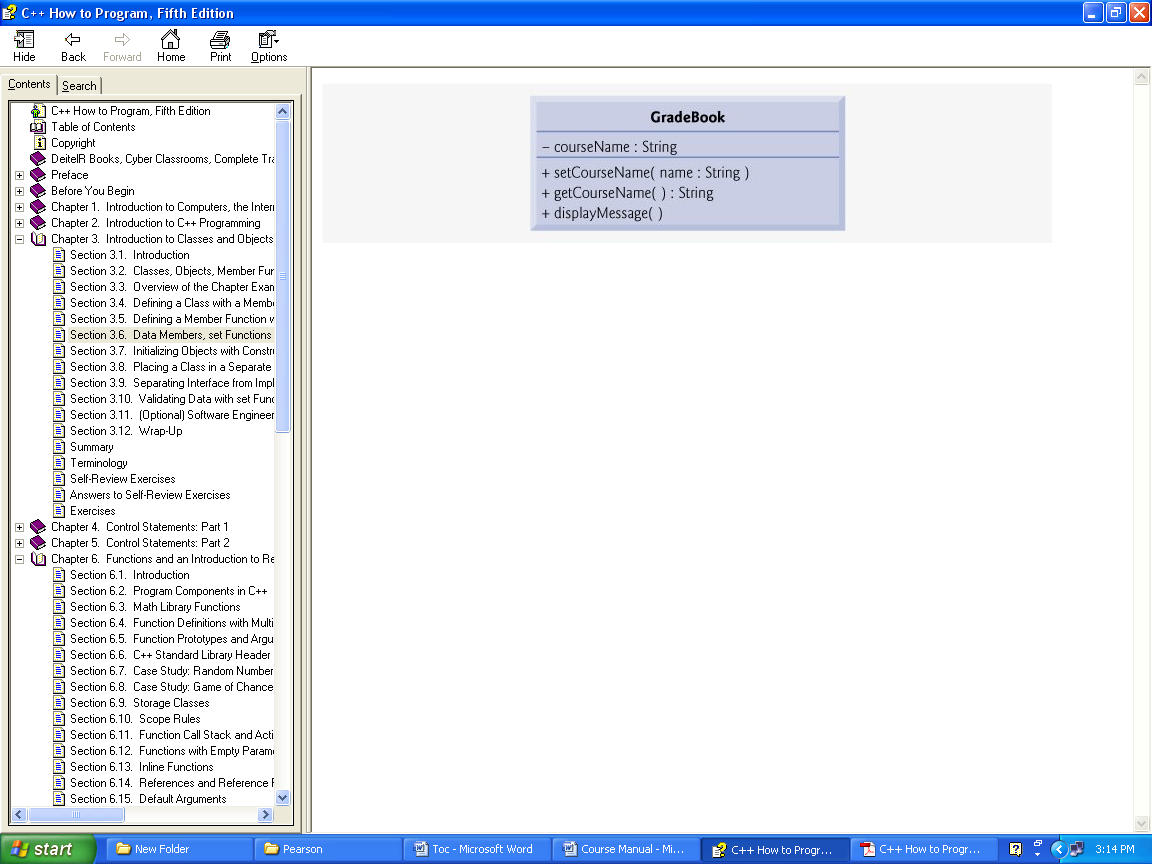
Member function displayMessage does not return any data when it completes its task, so its return type is void. The function does not receive parameters, so its parameter list is empty.

**Testing Class GradeBook**

The main function creates one object named myGradeBook of class GradeBook and uses each of its member functions. Main also displays the initial course name by calling the object's getCourseName member function. Note that the first line of the output does not show a course name, because the object's courseName data member (i.e., a string) is initially empty by default, the initial value of a string is the so-called **empty string**, i.e., a string that does not contain any characters. Nothing appears on the screen when an empty string is displayed.

The main then prompts the user to enter a course name. Local string variable nameOfCourse is set to the course name entered by the user, which is obtained by the call to the getline function. When the function is called, the argument's value is copied to parameter name of member function setCourseName. Then the parameter's value is assigned to data member courseName. The function call for the object using myGradeBook's displayMessage member function to display the welcome message containing the course name.

**UML Class Diagram**

****

**Initializing Objects with Constructors**

*Refer to Pages 85 – 86 in the textbook for the program*

**Constructors**:

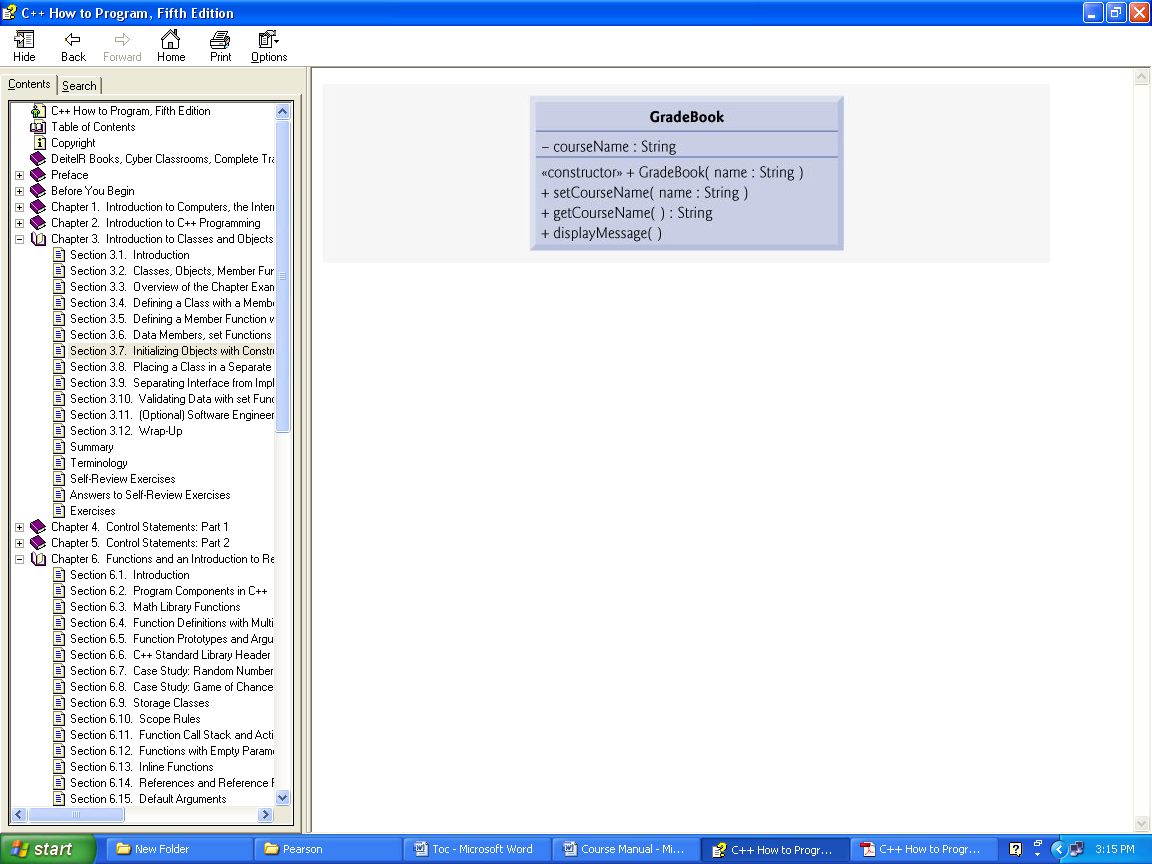
A constructor is a special function that is a member of a class and that has the same name as the class and it is called when an object is created. Notice that the constructor has no return type. In C++, constructors do not return values and, therefore, have no return type. (Not even **void** may be specified.) An object’s constructor is called when the object is created. This means that it is called when the object’s declaration is executed.

For global objects, the constructor is called when the program begins execution, prior to the call to **main( )**. For local objects, the constructor is called each time the object declaration is encountered. A constructor can have parameters. This allows you to give member variables program defined initial values when an object is created. You do this by passing arguments to an object’s constructor.

**Destructors**:

The complement of the constructor is the *destructor*. In many circumstances, an object will need to perform some action or series of actions when it is destroyed. Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. There are many reasons why a destructor may be needed. For example, an object may need to deallocate memory that it had previously allocated. In C++, it is the destructor that handles deactivation. The destructor has the same name as the constructor, but the destructor’s name is preceded by ~. Like constructors, destructors do not have return types.

**UML Class Diagram with Constructor**

****

**Class Assignment 2 (a) – Game Class:**

* 1. Create a class named Game using the following UML class diagram. Ensure that the program executes with no errors. 

The objective of the above program is to help the student write a class on their own with one or more attributes. You are free to modify the above version of the program or use a different program to achieve the objective.

Class in a Separate File for Reusability



To the Instructor: Explain how to Place a Class in a Separate File for Reusability

If client code does know how a class is implemented, the client code programmer might write client code based on the class's implementation details. Ideally, if implementation changes then the class's client should not have to change. Hiding the class's implementation details makes it easier to change the class's implementation while minimizing, and hopefully eliminating, changes to client code.

One of the benefits of creating class definitions is that, when packaged properly, our classes can be reused by programmers potentially worldwide.

#### Header Files (.h)

When building an object-oriented C++ program, it is customary to define reusable source code (such as a class) in a file that by convention has a .h filename extension known as a **header file**. Programs use #include preprocessor directives to include header files and take advantage of reusable software components, such as type string provided in the C++ Standard Library and user-defined types like class Game. This file should contain the class definition.

**Separating Interface from Implementation - Source Files (.cpp)**

By convention, member-function definitions are placed in a source-code file of the same base name (e.g., Game) as the class's header file but with a .cpp filename extension. Since the interface of the class is part of the class definition in the Game.h header file, the source files must have access to this file and #include in it.

**Client File**

The client code needs to know only Game's interface to use the class and must be able to link its object code. Since the interface of the class is part of the class definition in the Game.h header file, the client-code programmer must have access to this file and #include it in the client's source-code file. When the client code is compiled, the compiler uses the class definition in Game.h to ensure that the main function creates and manipulates objects of class Game correctly.

**Class Assignment 2(b)- Player Class:**

* 1. Using the Game program as a reference

1. Draw a UML class diagram for a class called Player.
   1. This class has 2 member variables namely playerName and playerScore. (Use appropriate data types to declare the member variables).
   2. The class contains following member functions: Constructor, set functions, get functions, display function.
2. Now using Visual Studio write code:
   1. To initialize the member variables use rand() to generate a random score for the player.
   2. Write appropriate code for the other member functions.
   3. Create two objects of the Player class and execute the program.

**The output should be as follows:**

**Player Mark has scored 45 points**

Things to note about Classes:

* The class declaration must end with a semi-colon.
* The public and private members are grouped. It is generally considered a good idea to put the public members first, but that is not a C++ requirement.
* The class's constructor function must have the same name as the class, and no return type (not even *void*).
* In general, member functions that do not change any of the data members of the class should be declared *const*
* A class can contain **static** data members. Every instance of the class will include its own copy of each of the non-static data but there will be only one copy of each static data member for the whole class.
* Only static data members can be initialized as part of the class declaration. Other data members are initialized by the class's constructor function(s).

Once an object of a class is created (instantiated) you can access the public members of that class using a dot(.) operator.

Exercises:

Fill in the blanks in each of the following:

1. A house is to a blueprint as a(n) \_\_\_\_\_\_\_\_\_ is to a class.
2. Every class definition contains keyword \_\_\_\_\_\_\_\_\_ followed immediately by the class's name.
3. A class definition is typically stored in a file with the \_\_\_\_\_\_\_\_\_ filename extension.
4. Each parameter in a function header should specify both a(n) \_\_\_\_\_\_\_\_\_ and a(n) \_\_\_\_\_\_\_\_\_.
5. When each object of a class maintains its own copy of an attribute, the variable that represents the attribute is also known as a(n) \_\_\_\_\_\_\_\_\_.
6. Keyword public is a(n) \_\_\_\_\_\_\_\_\_.
7. Return type \_\_\_\_\_\_\_\_\_ indicates that a function will perform a task but will not return any information when it completes its task.
8. When a member function is defined outside the class definition, the function header must include the class name and the \_\_\_\_\_\_\_\_\_, followed by the function name to "tie" the member function to the class definition.
9. The source-code file and any other files that use a class can include the class's header file via an \_\_\_\_\_\_\_\_\_ preprocessor directive.
10. ***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***in the object oriented world generalizes the characteristics and behaviors of an item.
11. *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_* conceals the functional details of an object from other objects, i.e., the object operates as a black box.
12. ***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***is a way to form new classes using classes that have already been defined.
13. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ refers to the ability of executing different operations in response to the same message.

Multiple Choice:

* + - 1. C++ functions other than main are executed:

1. Before main executes.
2. After main completes execution.
3. When they are explicitly called by another function.
4. Never.
   * + 1. Calling a member function of an object requires which item?
5. The dot operator.
6. Open and close braces.
7. The class name.
8. None of the above.
   * + 1. In the UML, the top compartment of the rectangle modeling a class contains:
9. The class’s name.
10. The class’s attributes.
11. The class’s behaviors.
12. All of the above.
    * + 1. Attributes of a class are also known as:
13. Constructors.
14. Local variables.
15. Data members.
16. Classes.
    * + 1. What type of member functions allow a client of a class to assign values to private data members?
17. *Client* member functions.
18. *Get* member functions.
19. *Set* member functions.
20. None of the above.

6. A default constructor has how many parameters?

1. 0.
2. 1.
3. 2.
4. Variable number.
   1. A constructor can specify the return type:
5. int.
6. string.
7. void.
8. A constructor cannot specify a return type.
   1. The compiler will implicitly create a default constructor if:
9. The class does not contain any data members.
10. The programmer specifically requests that the compiler do so.
11. The class does not define any constructors.
12. The class already defines a default constructor.
    1. A header file is typically given the filename extension:
13. .h
14. .hdr
15. .header
16. .cpp
    1. In the source-code file containing a class’s member function definitions, each member function definition must be tied to the class definition by preceding the member function name with the class name and ::, which is known as the:
17. Member definition linker.
18. Class implementation connector.
19. Source code resolver.
20. Binary scope resolution operator.

State whether each of the following is true or false. If false, explain why.

1. By convention, function names begin with a capital letter and all subsequent words in the name begin with a capital letter.
2. Empty parentheses following a function name in a function prototype indicate that the function does not require any parameters to perform its task.
3. Data members or member functions declared with access specifier private are accessible to member functions of the class in which they are declared.
4. Variables declared in the body of a particular member function are known as data members and can be used in all member functions of the class.
5. Every function's body is delimited by left and right braces ({ and }).
6. Any source-code file that contains int main() can be used to execute a program.
7. The types of arguments in a function call must match the types of the corresponding parameters in the function prototype's parameter list.

What is the difference between a local variable and a data member?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Lesson 2: Control Structures**

**Algorithms**

Any solvable computing problem can be solved by the execution of a series of actions in a specific order. A **procedure** for solving a problem in terms of

1. the **actions** to execute and
2. the **order** in which these actions execute

is called an **algorithm**. The following example demonstrates that correctly specifying the order in which the actions execute is important.

Consider the "rise-and-shine algorithm" followed by one junior executive for getting out of bed and going to work: (1) Get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work. In this case, our junior executive shows up for work soaking wet. Specifying the order in which statements (actions) execute in a computer program is called **program control**.

**Project 1 (a)**  2 POINTS

Write an algorithm for the game TicTacToe that describes ***in detail*** the steps as to how the game is played (follow the instruction provided by your instructor).

**Pseudocode**

**Pseudocode** (or "fake" code) is an artificial and informal language that helps a programmer to develop algorithms without having to worry about the strict details of C++ language syntax. The pseudocode we present here is particularly useful for developing algorithms that will be converted to structured portions of C++ programs. Pseudocode is similar to everyday English; it is convenient and user friendly, although it is not an actual computer programming language.

Pseudocode does not execute on computers. Rather, it helps the programmer "think out" a program before attempting to write it in a programming language, such as C++.

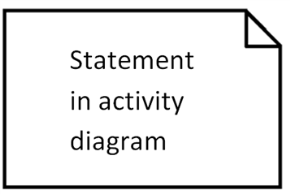
The style of pseudocode we present consists purely of characters, so programmers can type pseudocode conveniently, using any editor program. The computer can produce a freshly printed copy of a pseudocode program on demand. A carefully prepared pseudocode program can easily be converted to a corresponding C++ program. In many cases, this simply requires replacing pseudocode statements with C++ equivalents.

Pseudocode normally describes only **executable statements**, which cause specific actions to occur after a programmer converts a program from pseudocode to C++ and the program is run on a computer. Declarations (that do not have initializers or do not involve constructor calls) are not executable statements. We typically do not include variable declarations in our pseudocode. However, some programmers choose to list variables and mention their purposes at the beginning of pseudocode programs.

We now look at an example of pseudocode that may be written to help a programmer create the addition program.

1. Prompt the user to enter the first integer
2. Input the first integer
3. Prompt the user to enter the second integer
4. Input the second integer
5. Add first integer and second integer, store result
6. Display result

Activity Diagram for the above pseudocode:



The sequence structure is built into C++. Unless directed otherwise, the computer executes C++ statements one after the other in the order in which they are written that is, in sequence.

The Unified Modeling Language (UML) **activity diagram** above illustrates a typical sequence structure in which calculations are performed in order. C++ allows us to have as many actions as we want in a sequence structure.

Activity diagrams are part of the UML. An activity diagram models the **workflow** (also called the activity) of a portion of a software system. Such workflows may include a portion of an algorithm, such as the sequence structure. Activity diagrams are composed of special-purpose symbols, such as **action state symbols** (an elongated circle), **diamonds** (used usually for decisions/condition) and **small circles** (known as start and end states); these symbols are connected by **transition arrows**, which represent the flow of the activity.

Like pseudocode, activity diagrams help programmers develop and represent algorithms, although many programmers prefer pseudocode. Activity diagrams clearly show how control structures operate.

Consider the above activity diagram. It contains several **action states** that represent actions to perform. Each action state contains an **action expression** e.g., "prompt for an integer" or "display result" that specifies a particular action to perform. Other actions might include calculations or input/output operations. The arrows in the activity diagram are called transition arrows. These arrows represent **transitions**, which indicate the order in which the actions represented by the action states occur the program that implements the activities.

The **solid circle** located at the top of the activity diagram represents the activity's **initial state** (or the start state)the beginning of the workflow before the program performs the modeled activities. The solid circle surrounded by a hollow circle that appears at the bottom of the activity diagram represents the **final state** (or the end state)the end of the workflow after the program performs its activities.

The diagram also includes rectangles with the upper-right corners folded over. These are called **notes** in the UML. Notes are explanatory remarks that describe the purpose of symbols in the diagram. Notes can be used in any UML diagram not just activity diagrams. A **dotted line** connects each note with the element that the note describes. Activity diagrams normally do not show the C++ code that implements the activity. We use notes for this purpose here to illustrate how the diagram relates to C++ code. For more information on the UML visit [www.uml.org](http://www.uml.org).

if ( grade >= 60 )

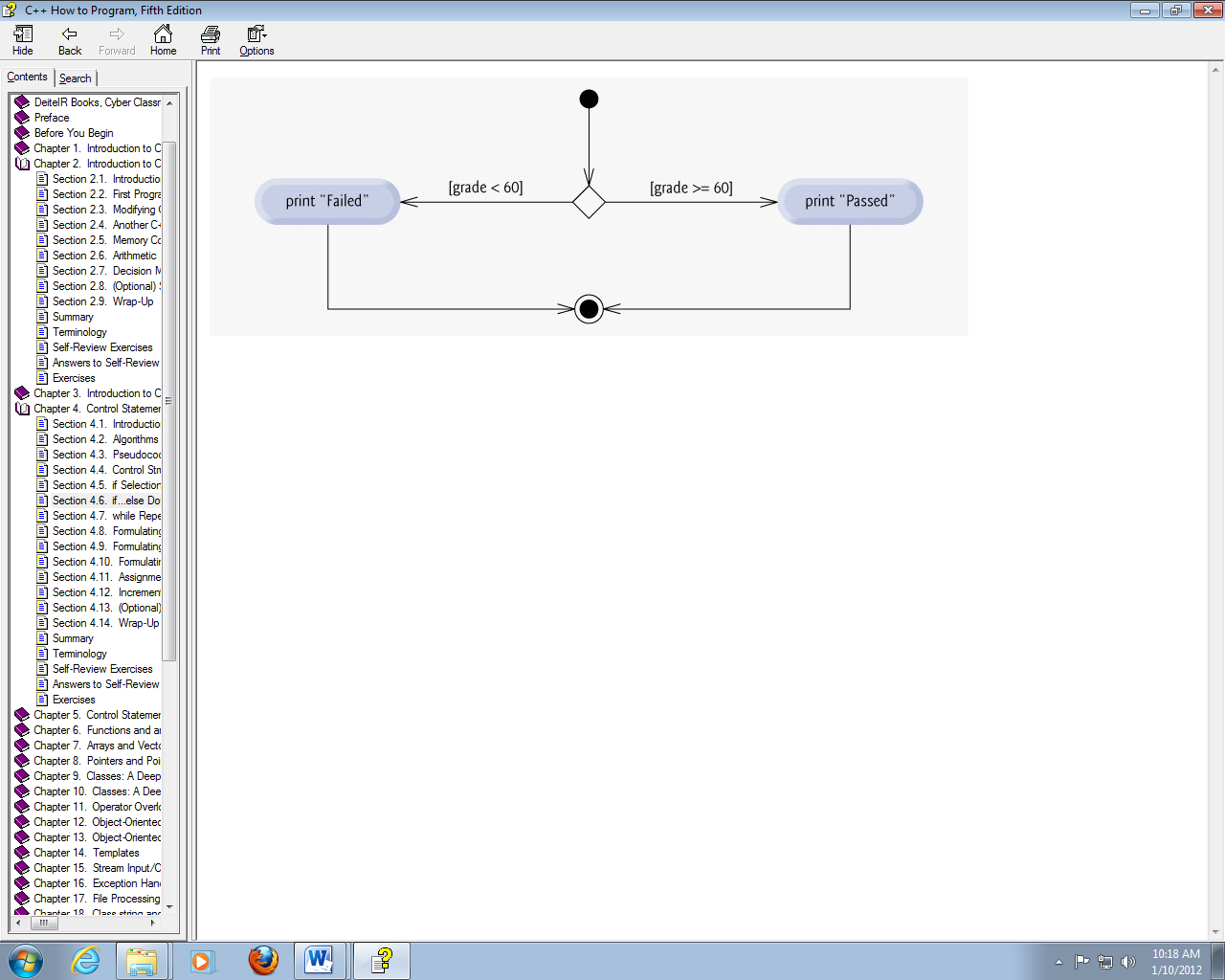
Statement in activity diagram

Statement in activity diagram

cout << "Passed";

else

cout << "Failed";



The following diagram illustrates the single-selection if statement. It contains what is perhaps the most important symbol in an activity diagram the diamond or **decision symbol**, which indicates that a decision is to be made. A decision symbol indicates that the workflow will continue along a path determined by the symbol's associated **guard conditions**, which can be true or false. Each transition arrow emerging from a decision symbol has a guard condition (specified in square brackets above or next to the transition arrow). If a particular guard condition is true, the workflow enters the action state to which that transition arrow points. In the above diagram, if the grade is greater than or equal to 60, the program prints "Passed" to the screen, then transitions to the final state of this activity. If the grade is less than 60, the program immediately transitions to the final state without displaying a message

### Exercises:

* + - 1. Specifying the order in which statements are to be executed in a computer program is called:
         1. An algorithm.
         2. Transfer of control.
         3. Program control.
         4. Pseudocode.

2. Which of the following is *true* of pseudocode?

1. It is executed by the computer.
2. It helps the programmer “think out” a program.
3. It includes declarations and all types of statements.
4. All of the above are false.

3. Pseudocode does *not* include:

1. Declarations.
2. Input/output.
3. Algorithms.
4. Control structures.

**Control Structures**

Normally, statements in a program execute one after the other in the order in which they are written. This is called **sequential execution**. Various C++ statements enable the programmer to specify that the next statement to execute may be other than the next one in sequence. This is called **transfer of control**.

All programs could be written in terms of only three control structures, namely, the sequence structure, the selection structure and the repetition structure. The term "control structures" comes from the field of computer science.

**Sequence Structure in C++**

The sequence structure is built into C++. Unless directed otherwise, the computer executes C++ statements one after the other in the order in which they are written that is, in sequence.

In the above pseudocode we execute with statement 1 which is prompting the user to enter a integer, and then we execute the statement 2 which is get the input from the user and so on in an sequential order until we reach the end of the statements.

C++ allows us to have as many actions as we want in a sequence structure.

**Selection Statements in C++**

C++ provides three types of selection statements.

The **if** selection statement either performs (selects) an action if a condition (predicate) is true or skips the action if the condition is false. The **if...else** selection statement performs an action if a condition is true or performs a different action if the condition is false. The **switch** selection statement performs one of many different actions, depending on the value of an integer expression.

The if selection statement is a **single-selection statement** because it selects or ignores a single action (or, as we will soon see, a single group of actions). The if...else statement is called a **double-selection statement** because it selects between two different actions (or groups of actions). The switch selection statement is called a **multiple-selection statement** because it selects among many different actions (or groups of actions).

**Repetition Statements in C++**

C++ provides three types of repetition statements (also called **looping statements** or loops) that enable programs to perform statements repeatedly as long as a condition (called the **loop-continuation condition**) remains true. The repetition statements are the **while**, do...while and for statements.

The while and for statements perform the action (or group of actions) in their bodies zero or more times if the loop-continuation condition is initially false, the action (or group of actions) will not execute. The do...while statement performs the action (or group of actions) in its body at least once.

Each of the words if, else, switch, while, do and for is a C++ keyword. These words are reserved by the C++ programming language to implement various features, such as C++'s control statements. Keywords must not be used as identifiers, such as variable names.

NOTE: Using a keyword as an identifier is a syntax error.

NOTE: Spelling a keyword with any uppercase letters is a syntax error. All of C++'s keywords contain only lowercase letters.

### if Selection Statement

Programs use selection statements to choose among alternative courses of action. For example, suppose the passing grade on an exam is 60. The pseudocode statement

If student's grade is greater than or equal to 60

Print "Passed"

determines whether the condition "student's grade is greater than or equal to 60" is true or false. If the condition is true, then "Passed" is printed and the next pseudocode statement in order is "performed" (remember that pseudocode is not a real programming language). If the condition is false, the print statement is ignored and the next pseudocode statement in order is performed. Note that the second line of this selection statement is indented. Such indentation is optional, but it is recommended because it emphasizes the inherent structure of structured programs. When you convert your pseudocode into C++ code, the C++ compiler ignores white-space characters (like blanks, tabs and newlines) used for indentation and vertical spacing.

Consistently applying reasonable indentation conventions throughout your programs greatly improves program readability.

The preceding pseudocode If statement can be written in C++ as

if ( grade >= 60 )

cout << "Passed";

Notice that the C++ code corresponds closely to the pseudocode. This is one of the properties of pseudocode that makes it such a useful program development tool.

In C++, a decision can be based on any expression if the expression evaluates to zero, it is treated as false; if the expression evaluates to nonzero, it is treated as true. C++ provides the data type **bool** for variables that can hold only the values true and false each of these is a C++ keyword.

### if...else Double-Selection Statement

The if single-selection statement performs an indicated action only when the condition is True; otherwise the action is skipped. The if...else double-selection statement allows the programmer to specify an action to perform when the condition is true and a different action to perform when the condition is false. For example, the pseudocode statement

If student's grade is greater than or equal to 60

Print "Passed"

Else

Print "Failed"

prints "Passed" if the student's grade is greater than or equal to 60, but prints "Failed" if the student's grade is less than 60. In either case, after printing occurs, the next pseudocode statement in sequence is "performed."

The preceding pseudocode If...Else statement can be written in C++ as

if ( grade >= 60 )

cout << "Passed";

else

cout << "Failed";

Note that the body of the else is also indented. Whatever indentation convention you choose should be applied consistently throughout your programs. It is difficult to read programs that do not obey uniform spacing conventions.

#### Conditional Operator (?:)

C++ provides the **conditional operator (?:)**, which is closely related to the if...else statement. The conditional operator is C++'s only **ternary operator** it takes three operands. The operands, together with the conditional operator, form a **conditional expression**. The first operand is a condition, the second operand is the value for the entire conditional expression if the condition is True and the third operand is the value for the entire conditional expression if the condition is false. For example, the output statement

cout << ( grade >= 60 ? "Passed" : "Failed" );

contains a conditional expression, grade >= 60 ? "Passed" : "Failed", that evaluates to the string "Passed" if the condition grade >= 60 is True, but evaluates to the string "Failed" if the condition is false. Thus, the statement with the conditional operator performs essentially the same as the preceding if...else statement. As we will see, the precedence of the conditional operator is low, so the parentheses in the preceding expression are required.

The preceding conditional expression is read, "If grade is greater than or equal to 60, then cout << "Passed"; otherwise, cout << "Failed"." This, too, is comparable to the preceding if...else statement. Conditional expressions can appear in some program locations where if...else statements cannot.

#### Nested if...else Statements

Nested **if...else** statements test for multiple cases by placing if...else selection statements inside other if...else selection statements. For example, the following pseudocode if...else statement prints A for exam grades greater than or equal to 90, B for grades in the range 80 to 89, C for grades in the range 70 to 79, D for grades in the range 60 to 69 and F for all other grades:

If student's grade is greater than or equal to 90  
     Print "A"  
Else  
     If student's grade is greater than or equal to 80  
         Print "B"  
     Else  
        If student's grade is greater than or equal to 70  
            Print "C"  
        Else  
            If student's grade is greater than or equal to 60  
                   Print "D"  
            Else  
                   Print "F"

This pseudocode can be written in C++ as

if ( studentGrade >= 90 ) // 90 and above gets "A"

cout << "A";

else

if ( studentGrade >= 80 ) // 80-89 gets "B"

cout << "B";

else

if ( studentGrade >= 70 ) // 70-79 gets "C"

cout << "C";

else

if ( studentGrade >= 60 ) // 60-69 gets "D"

cout << "D";

else // less than 60 gets "F"

cout << "F";

If studentGrade is greater than or equal to 90, the first four conditions will be true, but only the cout statement after the first test will execute. After that cout executes, the program skips the else-part of the "outermost" if...else statement. Most C++ programmers prefer to write the preceding if...else statement as

if ( studentGrade >= 90 ) // 90 and above gets "A"

cout << "A";

else if ( studentGrade >= 80 ) // 80-89 gets "B"

cout << "B";

else if ( studentGrade >= 70 ) // 70-79 gets "C"

cout << "C";

else if ( studentGrade >= 60 ) // 60-69 gets "D"

cout << "D";

else // less than 60 gets "F"

cout << "F";

The two forms are identical except for the spacing and indentation, which the compiler ignores. The latter form is popular because it avoids deep indentation of the code to the right. Such indentation often leaves little room on a line, forcing lines to be split and decreasing program readability.

NOTE: nested if...else statement can perform much faster than a series of single-selection if statements because of the possibility of early exit after one of the conditions is satisfied. In a nested if...else statement, test the conditions that are more likely to be true at the beginning of the nested if...else statement. This will enable the nested if...else statement to run faster and exit earlier than testing infrequently occurring cases first.

#### Dangling-else Problem

The C++ compiler always associates an else with the immediately preceding if unless told to do otherwise by the placement of braces ({ and }). This behavior can lead to what is referred to as the **dangling-else problem**. For example,

if ( x > 5 )

if ( y > 5 )

cout << "x and y are > 5";

else

cout << "x is <= 5";

appears to indicate that if x is greater than 5, the nested if statement determines whether y is also greater than 5. If so, "x and y are > 5" is output. Otherwise, it appears that if x is not greater than 5, the else part of the if...else outputs "x is <= 5".

Beware! This nested if...else statement does not execute as it appears. The compiler actually interprets the statement as

if ( x > 5 )

if ( y > 5 )

cout << "x and y are > 5";

else

cout << "x is <= 5";

in which the body of the first if is a nested if...else. The outer if statement tests whether x is greater than 5. If so, execution continues by testing whether y is also greater than 5. If the second condition is true, the proper string"x and y are > 5"is displayed. However, if the second condition is false, the string "x is <= 5" is displayed, even though we know that x is greater than 5.

To force the nested if...else statement to execute as it was originally intended, we must write it as follows:

if ( x > 5 )

{

if ( y > 5 )

cout << "x and y are > 5";

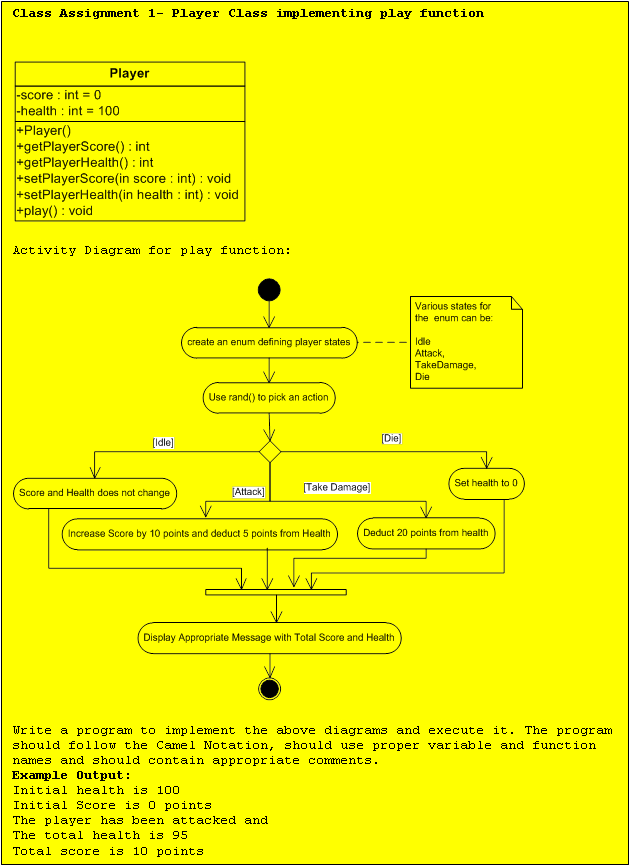
}

else

cout << "x is <= 5";

The braces ({}) indicate to the compiler that the second if statement is in the body of the first if and that the else is associated with the first if.

NOTE: Placing a semicolon after the condition in an if statement leads to a logic error in single-selection if statements and a syntax error in double-selection if...else statements (when the if part contains an actual body statement).



### while Repetition Statement

A **repetition statement** (also called a looping statement or a **loop**) allows the programmer to specify that a program should repeat an action while some condition remains true. The pseudocode statement

**While there are more items on my shopping list**

**Purchase next item and cross it off my list**

describes the repetition that occurs during a shopping trip. The condition, "there are more items on my shopping list" is either true or false. If it is true, then the action, "Purchase next item and cross it off my list" is performed. This action will be performed repeatedly while the condition remains true. The statement contained in the While repetition statement constitutes the body of the While, which can be a single statement or a block. Eventually, the condition will become false (when the last item on the shopping list has been purchased and crossed off the list). At this point, the repetition terminates, and the first pseudocode statement after the repetition statement executes.

As an example of C++'s while repetition statement, consider a program segment designed to find the first power of 3 larger than 100. Suppose the integer variable product has been initialized to 3. When the following while repetition statement finishes executing, product contains the result:

int product = 3;

while ( product <= 100)

product = 3 \* product;

When the while statement begins execution, the value of product is 3. Each repetition of the while statement multiplies product by 3, so product takes on the values 9, 27, 81 and 243 successively. When product becomes 243, the while statement condition product <= 100 becomes false. This terminates the repetition, so the final value of product is 243. At this point, program execution continues with the next statement after the while statement.

NOTE: Not providing, in the body of a while statement, an action that eventually causes the condition in the while to become false normally results in a logic error called an infinite loop, in which the repetition statement never terminates. This can make a program appear to "hang" or "freeze" if the loop body does not contain statements that interact with the user.

**Formulating Algorithms: Counter-Controlled Repetition**

*Refer to Pages 123 – 129 in the textbook for the program*

**Pseudocode Algorithm with Counter-Controlled Repetition**

Let's use pseudocode to list the actions to execute and specify the order in which these actions should occur. We use **counter-controlled repetition** to input the grades one at a time. This technique uses a variable called a **counter** to control the number of times a group of statements will execute (also known as the number of **iterations** of the loop).

Counter-controlled repetition is often called **definite repetition** because the number of repetitions is known before the loop begins executing. In this example, repetition terminates when the counter exceeds 10. This section presents a fully developed pseudocode algorithm that implements the algorithm in a C++ member function.

*Refer to Pages 124 for the pseudocode and to Pages 125 – 128 in the textbook for the program*

Class Assignment 2 - Player Class Bonus Problem:

Suppose the player has collected 10 bonus objects and each bonus object increases the player score. The teacher will provide with points 10 bonus objects.

* + 1. Write Pseudocode algorithm that uses counter-controlled repetition to solve the bonus calculation problem.
    2. Use the previous program and add a new function called CalculateBonusAndAverage(), calculate and display the collective bonus points using the Counter-Controlled Repetition pseudocode that you have written, display the total score after adding the bonus points and calculate the average bonus points and display the results.



Total Bonus collected is 59 points

Total score is 69 points

The average Bonus is 5.9 points

**Sentinel-Controlled Repetition**

*Refer to Pages 129 –136 in the textbook for pseudocode and implementation of Sentinel-Controlled Repetition*

Class Assignment 3 – Player Class Sentinel Controlled Repetition

Modify class assignment 3 to implement the sentinel controlled repetition.

**Formulating Algorithms: Nested Control Statements**

For the next example, we once again formulate an algorithm by using pseudocode and topdown, stepwise refinement, and write a corresponding C++ program. We have seen that control statements can be stacked on top of one another (in sequence) just as a child stacks building blocks. In this case study, we examine the only other structured way control statements can be connected, namely, by nesting one control statement within another.

*Refer to Pages 140 –143 in the textbook for pseudocode and implementation of Nested-Control Statements*

Class Assignment 4 – Player Class Nested Control Statements

Modify class assignment 2 to implement the nested control statements. The program should keep picking random enum and calculate the player score until the health of the player becomes 0 or the player dies.

Your program should analyze the results of the player class as follows:

1. Count the number of test results of each type.
2. Display a summary of the test results indicating the number of times the the player was attacked, idle or took damage.
3. If the players attack rate was high then print the message "Well Played." Etc.

**Assignment Operators**

C++ provides several assignment operators for abbreviating assignment expressions. For example, the statement

c = c + 3;

can be abbreviated with the **addition assignment operator +=** as

c += 3;

The += operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator. Any statement of the form

variable = variable operator expression;

in which the same variable appears on both sides of the assignment operator and operator is one of the binary operators +, -, \*, /, or % (or others we'll discuss later in the text), can be written in the form

variable operator= expression;

Thus the assignment c += 3 adds 3 to c.

| **Assignment operator** | **Sample expression** | **Explanation** | **Assigns** |
| --- | --- | --- | --- |
| Assume: int c = 3, d = 5, e = 4, f = 6, g = 12; | | | |
| += | c += 7 | c = c + 7 | 10 to c |
| -= | d -= 4 | d = d - 4 | 1 to d |
| \*= | e \*= 5 | e = e \* 5 | 20 to e |
| /= | f /= 3 | f = f / 3 | 2 to f |
| %= | g %= 9 | g = g % 9 | 3 to g |

**Increment and Decrement Operators**

In addition to the arithmetic assignment operators, C++ also provides two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable. These are the unary **increment operator, ++**, and the unary **decrement operator, --**. A program can increment by 1 the value of a variable called c using the increment operator, ++, rather than the expression c = c + 1 or c += 1. An increment or decrement operator that is prefixed to (placed before) a variable is referred to as the **prefix increment** or **prefix decrement operator**, respectively. An increment or decrement operator that is postfixed to (placed after) a variable is referred to as the **postfix increment** or **postfix decrement operator**, respectively.

| **Operator** | **Called** | **Sample expression** | **Explanation** |
| --- | --- | --- | --- |
| ++ | preincrement | ++a | Increment a by 1, then use the new value of a in the expression in which a resides. |
| ++ | postincrement | a++ | Use the current value of a in the expression in which a resides, then increment a by 1. |
| -- | predecrement | --b | Decrement b by 1, then use the new value of b in the expression in which b resides. |
| -- | postdecrement | b-- | Use the current value of b in the expression in which b resides, then decrement b by 1. |

Using the prefix increment (or decrement) operator to add (or subtract) 1 from a variable is known as **preincrementing** (or **predecrementing**) the variable. Preincrementing (or predecrementing) causes the variable to be incremented (decremented) by 1, and then the new value of the variable is used in the expression in which it appears. Using the postfix increment (or decrement) operator to add (or subtract) 1 from a variable is known as **postincrementing** (or **postdecrementing**) the variable. Postincrementing (or postdecrementing) causes the current value of the variable to be used in the expression in which it appears, and then the variable's value is incremented (decremented) by 1.

**Operator precedence**

| **Operators** | | | | | | **Associativity** | **Type** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| () |  |  |  |  |  | left to right | parentheses |
| ++ | -- | static\_cast< type >() | | | | left to right | unary (postfix) |
| ++ | -- | + | - |  |  | right to left | unary (prefix) |
| \* | / | % |  |  |  | left to right | multiplicative |
| + | - |  |  |  |  | left to right | additive |
| << | >> |  |  |  |  | left to right | insertion/extraction |
| < | <= | > | >= |  |  | left to right | relational |
| == | != |  |  |  |  | left to right | equality |
| ?: |  |  |  |  |  | right to left | conditional |
| = | += | -= | \*= | /= | %= | right to left | assignment |

### Exercises

Answer each of the following questions.

1. All programs can be written in terms of three types of control structures:\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_and\_\_\_\_\_\_\_\_\_.
2. The\_\_\_\_\_\_\_\_\_selection statement is used to execute one action when a condition is True or a different action when that condition is false.
3. Repeating a set of instructions a specific number of times is called\_\_\_\_\_\_\_\_\_repetition.
4. When it is not known in advance how many times a set of statements will be repeated, a(n)\_\_\_\_\_\_\_\_\_value can be used to terminate the repetition.

Write C++ statements to accomplish each of the following:

1. In one statement, assign the sum of the current value of x and y to z and postincrement the value of x.
2. Determine whether the value of the variable count is greater than 10. If it is, print "Count is greater than 10."
3. Predecrement the variable x by 1, then subtract it from the variable total.
4. Calculate the remainder after q is divided by divisor and assign the result to q. Write this statement two different ways.

Write C++ statements to accomplish each of the following tasks.

1. Declare variables sum and x to be of type int.
2. Set variable x to 1.
3. Set variable sum to 0.
4. Add variable x to variable sum and assign the result to variable sum.
5. Print "The sum is: " followed by the value of variable sum.
6. Combine the above statements into a program that calculates and prints the sum of the integers from 1 to 10. Use the while statement to loop through the calculation and increment statements. The loop should terminate when the value of x becomes 11.

Write single C++ statements that do the following:

1. Input integer variable x with cin and >>.
2. Input integer variable y with cin and >>.
3. Set integer variable i to 1.
4. Set integer variable power to 1.
5. Multiply variable power by x and assign the result to power.
6. Postincrement variable i by 1.
7. Determine whether i is less than or equal to y.
8. Output integer variable power with cout and <<.
9. Write a C++ program that uses the statements in Exercise 4.7 to calculate x raised to the y power. The program should have a while repetition statement.

Identify and correct the errors in each of the following:

1. while ( c <= 5 )

{

product \*= c;

c++;

1. cin << value;
2. if ( gender == 1 )

cout << "Woman" << endl;

else;

cout << "Man" << endl;

1. while ( z >= 0 )

sum += z;

**Lesson 3: Control Structures Part 2**

**Essentials of Counter-Controlled Repetition**

Counter-controlled repetition requires

1. the **name** of a control variable (or loop counter)
2. the **initial value** of the control variable
3. the **loop-continuation condition** that tests for the **final value** of the control variable (i.e., whether looping should continue)
4. the **increment** (or **decrement**) by which the control variable is modified each time through the loop.

*Refer to Pages 165 in the textbook for the program*

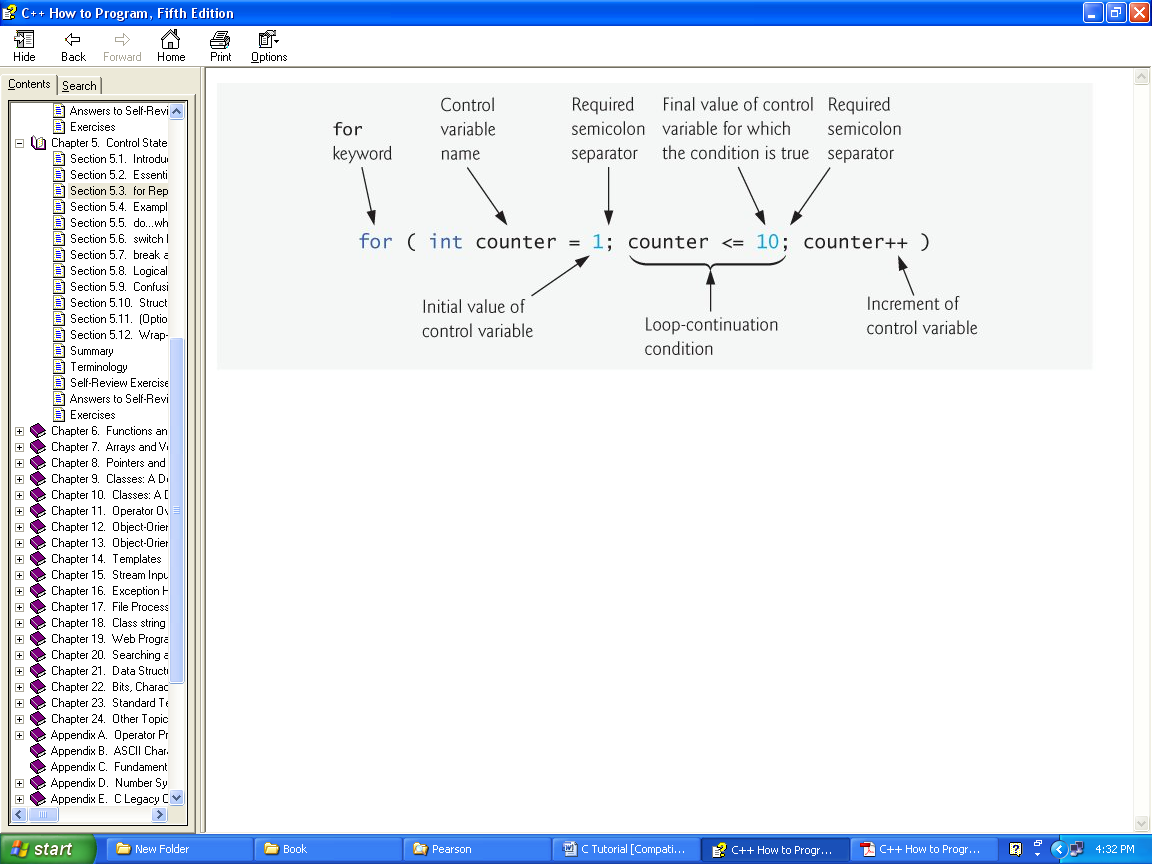
for **Repetition Statement**

The while statement can be used to implement any counter-controlled loop. C++ also provides the **for repetition statement**, which specifies the counter-controlled repetition details in a single line of code.

*Refer to Pages 166 in the textbook for the program*

When the for statement begins executing, the control variable counter is declared and initialized to 1. Then, the loop-continuation condition counter <= 10 is checked. The initial value of counter is 1, so the condition is satisfied and the body statement (line 12) prints the value of counter, namely 1. Then, the expression counter++ increments control variable counter and the loop begins again with the loop-continuation test. The control variable is now equal to 2, so the final value is not exceeded and the program performs the body statement again. This process continues until the loop body has executed 10 times and the control variable counter is incremented to 11this causes the loop-continuation test to fail and repetition to terminate. The program continues by performing the first statement after the for statement.

**for statement header components**



The general form of the for statement is

for ( initialization; loopContinuationCondition; increment )

statement

where the initialization expression initializes the loop's control variable, loopContinuationCondition determines whether the loop should continue executing (this condition typically contains the final value of the control variable for which the condition is true) and increment increments the control variable. The initialization and increment expressions can be comma-separated lists of expressions. The commas, as used in these expressions, are **comma operators**, which guarantee that lists of expressions evaluate from left to right. The comma operator has the lowest precedence of all C++ operators. The value and type of a comma-separated list of expressions is the value and type of the rightmost expression in the list. The comma operator most often is used in for statements. Its primary application is to enable the programmer to use multiple initialization expressions and/or multiple increment expressions. For example, there may be several control variables in a single for statement that must be initialized and incremented.

The three expressions in the for statement header are optional (but the two semicolon separators are required). If the loopContinuationCondition is omitted, C++ assumes that the condition is true, thus creating an infinite loop. One might omit the initialization expression if the control variable is initialized earlier in the program. One might omit the increment expression if the increment is calculated by statements in the body of the for or if no increment is needed.

In most cases, the for statement can be represented by an equivalent while statement, as follows:

initialization;

while ( loopContinuationCondition )

{

statement

increment;

}

If the initialization expression in the for statement header declares the control variable (i.e., the control variable's type is specified before the variable name), the control variable can be used only in the body of the for statement the control variable will be unknown outside the for statement. This restricted use of the control variable name is known as the variable's **scope.** The scope of a variable specifies where it can be used in a program.

*Refer to Pages 170 in the textbook for (Fig 5.4) UML Activity diagrams for the for loop*

Class Assignment 1 – Sum of Even numbers in a range:

* 1. Type and execute the code (Fig 5.5) on page 171. Observe what happened.
  2. Modify the program so that the program sums the even numbers from 2 to 100

**do...while Repetition Statement**

The do...while repetition statement is similar to the while statement. In the while statement, the loop-continuation condition test occurs at the beginning of the loop before the body of the loop executes. The do...while statement tests the loop-continuation condition after the loop body executes; therefore, the loop body always executes at least once. When a do...while terminates, execution continues with the statement after the while clause. Note that it is not necessary to use braces in the do...while statement if there is only one statement in the body; however, most programmers include the braces to avoid confusion between the while and do...while statements. For example,

while ( condition )

normally is regarded as the header of a while statement. A do...while with no braces around the single statement body appears as

do

statement

while ( condition );

which can be confusing. The last linewhile( condition );might be misinterpreted by the reader as a while statement containing as its body an empty statement. Thus, the do...while with one statement is often written as follows to avoid confusion:

do

{

statement

} while ( condition );

*Refer to Page 176 in the textbook for (Fig 5.8) UML Activity diagrams for the do..while loop*

*Refer to Pages 175 in the textbook implementation of do..while*

### switch Multiple-Selection Statement

We discussed the if single-selection statement and the if...else double-selection statement previously. C++ provides the switch multiple-selection statement to perform many different actions based on the possible values of a variable or expression. Each action is associated with the value of a **constant integral expression** (i.e., any combination of character constants and integer constants that evaluates to a constant integer value) that the variable or expression on which the switch is based may assume.

*Refer to Page 184 in the textbook for (Fig 5.12) UML Activity diagrams for* the switch statement

**GradeBook Class with switch Statement to Count A, B, C, D and F Grades**

*Refer to Pages 177- 183 in the textbook implementation of switch*

Class Assignment 2 – Player Class implementing Switch Statement

Modify class assignment 4 from previous lesson to implement the switch multiple control statements. The program should keep picking random enum and calculate the player score until the health of the player becomes 0 or the player dies.

Your program should analyze the results of the player class as follows:

* + - 1. Count the number of test results of each type.
      2. Display a summary of the test results indicating the number of times the the player was attacked, idle or took damage.
      3. If the players attack rate was high then print the message "Well Played." Etc.

**break and continue Statements**

In addition to the selection and repetition statements, C++ provides statements break and continue to alter the flow of control. The preceding section showed how break can be used to terminate a switch statement's execution. This section discusses how to use break in a repetition statement.

**break Statement**

The break statement, when executed in a while, for, do...while or switch statement, causes immediate exit from that statement. Program execution continues with the next statement. Common uses of the break statement are to escape early from a loop or to skip the remainder of a switch statement

**continue Statement**

The continue statement, when executed in a while, for or do...while statement, skips the remaining statements in the body of that statement and proceeds with the next iteration of the loop. In while and do...while statements, the loop-continuation test evaluates immediately after the continue statement executes. In the for statement, the increment expression executes, then the loop-continuation test evaluates.

*Refer to Pages 185- 186 in the textbook implementation of break and continue*

Class Assignment 3 - Sum of even numbers divisible by 5

Modify the class assignment 1 so that the program adds only the numbers that are divisible by 5. Use break and continue statements if needed.

**Logical Operators**

So far we have studied only **simple conditions**, such as counter <= 10, total > 1000 and number != sentinelValue. We expressed these conditions in terms of the relational operators >, <, >= and <=, and the equality operators == and !=. Each decision tested precisely one condition. To test multiple conditions while making a decision, we performed these tests in separate statements or in nested if or if...else statements.

C++ provides **logical operators** that are used to form more complex conditions by combining simple conditions. The logical operators are && (logical AND), || (logical OR) and ! (logical NOT, also called logical negation).

**Logical AND (&&) Operator**

Suppose that we wish to ensure that two conditions are both True before we choose a certain path of execution. In this case, we can use the **&& (logical AND)** operator, as follows:

if ( gender == 1 && age >= 65 )

seniorFemales++;

This if statement contains two simple conditions. The condition gender == 1 is used here to determine whether a person is a female. The condition age >= 65 determines whether a person is a senior citizen. The simple condition to the left of the && operator evaluates first, because the precedence of == is higher than the precedence of &&. If necessary, the simple condition to the right of the && operator evaluates next, because the precedence of >= is higher than the precedence of &&. As we will discuss shortly, the right side of a logical AND expression is evaluated only if the left side is true. The if statement then considers the combined condition

gender == 1 && age >= 65

This condition is true if and only if both of the simple conditions are true. Finally, if this combined condition is indeed true, the statement in the if statement's body increments the count of seniorFemales. If either of the simple conditions is false (or both are), then the program skips the incrementing and proceeds to the statement following the if. The preceding combined condition can be made more readable by adding redundant parentheses:

( gender == 1 ) && ( age >= 65 )

**Logical OR (||) Operator**

Now let us consider the || (**logical OR**) operator. Suppose we wish to ensure at some point in a program that either or both of two conditions are True before we choose a certain path of execution. In this case, we use the || operator, as in the following program segment:

if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )

cout << "Student grade is A" << endl;

This preceding condition also contains two simple conditions. The simple condition semesterAverage >= 90 evaluates to determine whether the student deserves an "A" in the course because of a solid performance throughout the semester. The simple condition finalExam >= 90 evaluates to determine whether the student deserves an "A" in the course because of an outstanding performance on the final exam. The if statement then considers the combined condition

( semesterAverage >= 90) || ( finalExam >= 90 )

and awards the student an "A" if either or both of the simple conditions are True. Note that the message "Student grade is A" prints unless both of the simple conditions are false.

**Logical Negation (!) Operator**

C++ provides the **!** (**logical NOT**, also called **logical negation**) operator to enable a programmer to "reverse" the meaning of a condition. Unlike the && and || binary operators, which combine two conditions, the unary logical negation operator has only a single condition as an operand. The unary logical negation operator is placed before a condition when we are interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

if ( !( grade == sentinelValue ) )

cout << "The next grade is " << grade << endl;

The parentheses around the condition grade == sentinelValue are needed because the logical negation operator has a higher precedence than the equality operator.

In most cases, the programmer can avoid using logical negation by expressing the condition with an appropriate relational or equality operator. For example, the preceding if statement also can be written as follows:

if ( grade != sentinelValue )

cout << "The next grade is " << grade << endl;

This flexibility often can help a programmer express a condition in a more "natural" or convenient manner.

**Confusing Equality (==) and Assignment (=) Operators**

There is one type of error that C++ programmers, no matter how experienced, tend to make so frequently that we feel it requires a separate section. That error is accidentally swapping the operators == (equality) and = (assignment). What makes these swaps so damaging is the fact that they ordinarily do not cause syntax errors. Rather, statements with these errors tend to compile correctly and the programs run to completion, often generating incorrect results through runtime logic errors. [Note: Some compilers issue a warning when = is used in a context where == normally is expected.]

There are two aspects of C++ that contribute to these problems. One is that any expression that produces a value can be used in the decision portion of any control statement. If the value of the expression is zero, it is treated as false, and if the value is nonzero, it is treated as True. The second is that assignments produce a value namely, the value assigned to the variable on the left side of the assignment operator. For example, suppose we intend to write

if ( payCode == 4 )

cout << "You get a bonus!" << endl;

but we accidentally write

if ( payCode = 4 )

cout << "You get a bonus!" << endl;

The first if statement properly awards a bonus to the person whose payCode is equal to 4. The second if statementthe one with the errorevaluates the assignment expression in the if condition to the constant 4. Any nonzero value is interpreted as True, so the condition in this if statement is always true and the person always receives a bonus regardless of what the actual paycode is! Even worse, the paycode has been modified when it was only supposed to be examined!

Variable names are said to be **lvalues** (for "left values") because they can be used on the left side of an assignment operator. Constants are said to be **rvalues** (for "right values") because they can be used on only the right side of an assignment operator. Note that lvalues can also be used as rvalues, but not vice versa.

Exercises

Write a C++ statement or a set of C++ statements to accomplish each of the following:

1. Sum the odd integers between 1 and 99 using a for statement. Assume the integer variables sum and count have been declared.
2. Print the integers from 1 to 20 using a while loop and the counter variable x. Assume that the variable x has been declared, but not initialized. Print only 5 integers per line. [Hint: Use the calculation x % 5. When the value of this is 0, print a newline character; otherwise, print a tab character.]

Find the error(s) in each of the following code segments and explain how to correct it (them).

1. x = 1;

while ( x <= 10 );

x++;

}

1. for ( y = .1; y != 1.0; y += .1 )

cout << y << endl;

1. switch ( n )

{

case 1:

cout << "The number is 1" << endl;

case 2:

cout << "The number is 2" << endl;

break;

default:

cout << "The number is not 1 or 2" << endl;

break;

}

1. The following code should print the values 1 to 10.

n = 1;

while ( n < 10 )

cout << n++ << endl;

1. In an activity diagram for an algorithm, what does a solid circle surrounded by a hollow circle represent?

1. Initial state.
2. Final state.
3. Action state.
4. Transition.

2.Which of the following is a double-selection statement?

1. if.
2. if…else.
3. do…while.
4. switch.

3.Which of the following is a repetition structure?

1. if.
2. if…else.
3. do…while.
4. switch.

4. If grade has the value of 60, what will the following code display?

if ( grade >= 60 )

cout << "Passed";

1. nothing.
2. 60
3. Passed
4. cout << "Passed";

5. The data type bool:

1. Can take on values true and false.
2. Can take on any expression as a value.
3. Can take on values -1, 0 or 1.
4. Can only be used in a selection statement.

6. The conditional operator (?:):

1. Is the only ternary operator in C++.
2. Is a unary operator.
3. Associates from left to right.
4. Accepts two operands.

7. Which of the following does *not* perform the following task: display correct if answer is equal to 7 and incorrect if answer is not equal to 7?

1. if ( answer == 7 )  
    cout << "correct";  
   else  
    cout << "incorrect";
2. cout << answer == 7 ? "correct" : "incorrect";
3. cout << ( answer == 7 ? "correct" : "incorrect" );
4. answer == 7 ? cout << "correct" : cout << "incorrect";

8. A block:

1. Must contain exactly three statements.
2. Cannot contain declarations.
3. Is a compound statement.
4. Is represented by placing a semicolon (;) where a statement would normally be.

9. What is wrong with the following while loop?

while ( sum <= 1000 )

sum = sum – 30;

1. The parentheses should be braces.
2. Braces are required around sum = sum – 30;.
3. There should be a semicolon after while ( sum <= 1000 ).
4. sum = sum – 30 should be sum = sum + 30 or else the loop may never end.

10. How many times will the following loop print hello?

i = 1;

while ( i <= 10 )

cout << "hello";

1. 0.
2. 9.
3. 10.
4. An infinite number of times.

11. Indefinite repetition is controlled by a:

1. Counter.
2. Sentinel value.
3. Absence of a condition.
4. Non-constant condition.

12. Having a loop within a loop is known as:

1. Recursion.
2. Doubling up.
3. Nesting.
4. Stacking.

13. If x initially contains the value 3, which of the following sets x to 7?

1. x ++ 4;
2. x += 4;
3. x =+ 4;
4. x + 4 = x;

14. Assuming that x and y are equal to 3 and 2, respectively, after the statement x -= y executes, the values of x and y will be:

1. x: 5; y: 3
2. x: 3; y: -1
3. x: 3; y: 5
4. x: 1; y: 2

15. Which of the following will not increment c by 1?

1. c + 1;
2. c++;
3. ++c;
4. c += 1;

16. Assuming that x is equal to 4, which of the following statements will *not* result in y containing the value 5 after execution?

1. y = 5;
2. y = x++;
3. y = ++x;
4. y = x + 1

17. Which of the following operations has the highest precedence?

1. Postincrement.
2. Multiplication.
3. Addition.
4. Assignment.

18.Which of the following for headers is *not* valid?

1. for ( int i = 0; i < 10; i++ )
2. int i = 0;  
   for ( ; i < 10; i++ )
3. for ( int i = 0; int j = 5; ; i++ )
4. All of the above.

19. If a do…while structure is used:

1. An infinite loop cannot take place.
2. Counter-controlled repetition is not possible.
3. The body of the loop will execute at least once.
4. An off-by-one error cannot occur.

20. What will the following program segment do?

int counter = 1;  
do   
{

cout << counter << " ";

} while ( ++counter <= 10 );

1. Print the numbers 1 through 11.
2. Print the numbers 1 through 10.
3. Print the numbers 1 through 9.
4. Cause a syntax error.

21. A switch statement should be used:

1. As a single-selection structure.
2. As a double-selection structure.
3. As a multiple-selection structure.
4. To replace all if…else statements.

22. In a switch structure:

1. A break is required after each case.
2. Multiple actions in a case do not need to be enclosed in braces.
3. A default case is required.
4. A break is required after the default case.

23. Which of the following is *correct* when labeling cases in a switch structure?

1. case1
2. Case1
3. case 1
4. Case 1

24. switch *can* be used to test:

1. int constants.
2. float constants.
3. string constants.
4. all types of constants.

25. The \_\_\_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_\_\_ are the only three forms of control necessary.

1. switch, if, else.
2. sequence, selection, repetition.
3. break, continue, if…else.
4. for, while, do…while.

26. Which of the following is *not* one of the C++ control structures?

1. if
2. switch
3. break
4. do…while

27. Which of the following data types *can* be used to represent integers?

1. char
2. long
3. short
4. All of the above.

28.Which of the following is *false*?

1. break and continue statements alter the flow of control.
2. continue statements skip the remaining statements in current iteration of the body of the loop in which they are embedded.
3. break statements exit from the loop in which they are embedded.
4. continue and break statements may be embedded only within repetition statements.

29. The OR (||) operator:

1. Has higher precedence than the AND (&&) operator.
2. Stops evaluation upon finding one condition to be true.
3. Associates from right to left.
4. Is a ternary operator.

30. An example of a unary operator is:

1. The < relational operator.
2. The = assignment operator.
3. The % arithmetic operator.
4. The ! logical operator.

**Lesson 4: Functions**

Functions (called **methods** or **procedures** in other programming languages) allow the programmer to modularize a program by separating its tasks into self-contained units. You have used functions in every program you have written. These functions are sometimes referred to as **user-defined functions** or **programmer-defined functions**. The statements in the function bodies are written only once, are reused from perhaps several locations in a program and are hidden from other functions.

There are several motivations for modularizing a program with functions. One is the divide-and-conquer approach, which makes program development more manageable by constructing programs from small, simple pieces. Another is software reusability using existing functions as building blocks to create new programs. For example, in earlier programs, we did not have to define how to read a line of text from the keyboard C++ provides this capability via the getline function of the <string> header file. A third motivation is to avoid repeating code. Also, dividing a program into meaningful functions makes the program easier to debug and maintain.

**NOTE:** To promote software reusability, every function should be limited to performing a single, well-defined task, and the name of the function should express that task effectively. Such functions make programs easier to write, test, debug and maintain.

**NOTE:** A small function that performs one task is easier to test and debug than a larger function that performs many tasks.

A function is invoked by a function call, and when the called function completes its task, it either returns a result or simply returns control to the caller. An analogy to this program structure is the hierarchical form of management. A boss (similar to the calling function) asks a worker (similar to the called function) to perform a task and report back (i.e., return) the results after completing the task. The boss function does not know how the worker function performs its designated tasks. The worker may also call other worker functions, unbeknownst to the boss. This hiding of implementation details promotes good software engineering.

We have written classes containing simple functions that had at most one parameter. Functions often require more than one piece of information to perform their tasks. We now consider functions with multiple parameters.

There must be one argument in the function call for each parameter (also called a **formal parameter**) in the function definition.

*Refer to Pages 212-214 in the textbook for the program*

Class Assignment 1 – Maximum Bonus Problem:

Modify the Player Class Bonus Problem to accept 3 bonus points from the user and display the max bonus the player had acquired.

**Return control to the caller**

There are three ways to return control to the point at which a function was invoked. If the function does not return a result (i.e., the function has a void return type), control returns when the program reaches the function-ending right brace, or by execution of the statement

return;

If the function does return a result, the statement

return expression;

evaluates expression and returns the value of expression to the caller.

**Function Prototypes and Argument Coercion**

A function prototype (also called a **function declaration**) tells the compiler the name of a function, the type of data returned by the function, the number of parameters the function expects to receive, the types of those parameters and the order in which the parameters of those types are expected.

**Function Signatures**

The portion of a function prototype that includes the name of the function and the types of its arguments is called the **function signature** or simply the **signature**. The function signature does not specify the function's return type. Function in the same scope must have unique signatures. The scope of a function is the region of a program in which the function is known and accessible.

**Argument Coercion**

An important feature of function prototypes is **argument coercion** i.e., forcing arguments to the appropriate types specified by the parameter declarations. For example, a program can call a function with an integer argument, even though the function prototype specifies a double argument the function will still work correctly.

**Argument Promotion Rules**

Sometimes, argument values that do not correspond precisely to the parameter types in the function prototype can be converted by the compiler to the proper type before the function is called. These conversions occur as specified by C++'s **promotion rules**. The promotion rules indicate how to convert between types without losing data. An int can be converted to a double without changing its value. However, a double converted to an int truncates the fractional part of the double value. Keep in mind that double variables can hold numbers of much greater magnitude than int variables, so the loss of data may be considerable. Values may also be modified when converting large integer types to small integer types (e.g., long to short), signed to unsigned or unsigned to signed.

The promotion rules apply to expressions containing values of two or more data types; such expressions are also referred to as **mixed-type expressions**. The type of each value in a mixed-type expression is promoted to the "highest" type in the expression (actually a temporary version of each value is created and used for the expressionthe original values remain unchanged). Promotion also occurs when the type of a function argument does not match the parameter type specified in the function definition or prototype.

Converting values to lower fundamental types can result in incorrect values. Therefore, a value can be converted to a lower fundamental type only by explicitly assigning the value to a variable of lower type (some compilers will issue a warning in this case) or by using a cast operator

*Refer to Pages 220-228 in the textbook for the case studies 6.7 and 6.8*

### Scope Rules

The portion of the program where an identifier can be used is known as its scope. For example, when we declare a local variable in a block, it can be referenced only in that block and in blocks nested within that block. This section discusses four scopes for an identifier **function scope**, **file scope**, **block scope** and **function-prototype scope**.

An identifier declared outside any function or class has file scope. Such an identifier is "known" in all functions from the point at which it is declared until the end of the file. Global variables, function definitions and function prototypes placed outside a function all have file scope.

Identifiers declared inside a block have block scope. Block scope begins at the identifier's declaration and ends at the terminating right brace (}) of the block in which the identifier is declared. Local variables have block scope, as do function parameters, which are also local variables of the function. Any block can contain variable declarations. When blocks are nested and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is "hidden" until the inner block terminates. While executing in the inner block, the inner block sees the value of its own local identifier and not the value of the identically named identifier in the enclosing block. Local variables declared static still have block scope, even though they exist from the time the program begins execution. Storage duration does not affect the scope of an identifier.

**Labels** (identifiers followed by a colon such as start:) are the only identifiers with function scope. Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body. Labels are used in goto statements

The only identifiers with function prototype scope are those used in the parameter list of a function prototype. As mentioned previously, function prototypes do not require names in the parameter list only types are required. Names appearing in the parameter list of a function prototype are ignored by the compiler. Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity. In a single prototype, a particular identifier can be used only once.

### References and Reference Parameters

Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**. When an argument is passed by value, a copy of the argument's value is made and passed (on the function call stack) to the called function. Changes to the copy do not affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Each argument that has been passed in the programs in this chapter so far has been passed by value.

### Reference Parameters

This section introduces **reference parameters** the first of two means C++ provides for performing pass-by-reference. With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data if the called function chooses to do so.

A reference parameter is an alias for its corresponding argument in a function call. To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand (&); use the same convention when listing the parameter's type in the function header. For example, the following declaration in a function header

int &count

when read from right to left is pronounced "count is a reference to an int." In the function call, simply mention the variable by name to pass it by reference. Then, mentioning the variable by its parameter name in the body of the called function actually refers to the original variable in the calling function, and the original variable can be modified directly by the called function. As always, the function prototype and header must agree.

*Refer to Pages 242-243 in the textbook for the program*

### Default Arguments

It is not uncommon for a program to invoke a function repeatedly with the same argument value for a particular parameter. In such cases, the programmer can specify that such a parameter has a **default argument**, i.e., a default value to be passed to that parameter. When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument to be passed as an argument to the function call.

Default arguments must be the rightmost (trailing) arguments in a function's parameter list. When calling a function with two or more default arguments, if an omitted argument is not the rightmost argument in the argument list, then all arguments to the right of that argument also must be omitted. Default arguments should be specified with the first occurrence of the function name typically, in the function prototype. If the function prototype is omitted because the function definition also serves as the prototype, then the default arguments should be specified in the function header. Default values can be any expression, including constants, global variables or function calls. Default arguments also can be used with inline functions.

*Refer to Pages 245-246 in the textbook for the program*

### Function Overloading

C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as the parameter types or the number of parameters or the order of the parameter types are concerned). This capability is called **function overloading**. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call. Function overloading is commonly used to create several functions of the same name that perform similar tasks, but on different data types. For example, many functions in the math library are overloaded for different numeric data types

*Refer to Pages 248-249 in the textbook for the program*

### Recursion

A **recursive function** is a function that calls itself, either directly, or indirectly

Recursive problem-solving approaches have a number of elements in common. A recursive function is called to solve a problem. The function actually knows how to solve only the simplest case(s), or so-called **base case(s)**. If the function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, it typically divides the problem into two conceptual pieces a piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version. This new problem looks like the original problem, so the function launches (calls) a fresh copy of itself to work on the smaller problem this is referred to as a **recursive call** and is also called the **recursion step**. The recursion step often includes the keyword return, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller, possibly main.

The recursion step executes while the original call to the function is still open, i.e., it has not yet finished executing. The recursion step can result in many more such recursive calls, as the function keeps dividing each new sub-problem with which the function is called into two conceptual pieces. In order for the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually converge on the base case. At that point, the function recognizes the base case and returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original function call eventually returns the final result to main.

*Refer to Pages 255-256 in the textbook for the program*

### Recursion vs. Iteration

Both iteration and recursion are based on a control statement: Iteration uses a repetition structure; recursion uses a selection structure. Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls. Iteration and recursion both involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized. Iteration with counter-controlled repetition and recursion both gradually approach termination: Iteration modifies a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion produces simpler versions of the original problem until the base case is reached. Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base case.

Recursion has many negatives. It repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space. Each recursive call causes another copy of the function (actually only the function's variables) to be created; this can consume considerable memory. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.

### So why choose recursion?

Any problem that can be solved recursively can also be solved iteratively (non-recursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.

**Storage Classes**

The programs you have seen so far use identifiers for variable names. The attributes of variables include name, type, size and value. Actually, each identifier in a program has other attributes, including **storage class**, scope and **linkage**.

C++ provides five **storage-class specifiers**: **auto**, **register**, **extern**, **mutable** and **static**. This section discusses storage-class specifiers auto, register, extern and static.

**Storage Class, Scope and Linkage**

An identifier's storage class determines the period during which that identifier exists in memory. Some identifiers exist briefly, some are repeatedly created and destroyed and others exist for the entire execution of a program. This section discusses two storage classes: static and automatic.

An identifier's scope is where the identifier can be referenced in a program. Some identifiers can be referenced throughout a program; others can be referenced from only limited portions of a program.

An identifier's linkage determines whether an identifier is known only in the source file where it is declared or across multiple files that are compiled, then linked together. An identifier's storage-class specifier helps determine its storage class and linkage.

**Storage Class Categories**

The storage-class specifiers can be split into two storage classes: automatic storage class and static storage class. Keywords auto and register are used to declare variables of the automatic storage class. Such variables are created when program execution enters the block in which they are defined, they exist while the block is active and they are destroyed when the program exits the block.

**Local Variables**

Only local variables of a function can be of automatic storage class. A function's local variables and parameters normally are of automatic storage class. The storage class specifier auto explicitly declares variables of automatic storage class. For example, the following declaration indicates that double variables x and y are local variables of automatic storage class they exist only in the nearest enclosing pair of curly braces within the body of the function in which the definition appears:

auto double x, y;

Local variables are of automatic storage class by default, so keyword auto rarely is used. For the remainder of the text, we refer to variables of automatic storage class simply as automatic variables.

**Register Variables**

Data in the machine-language version of a program is normally loaded into registers for calculations and other processing.

The compiler might ignore register declarations. For example, there might not be a sufficient number of registers available for the compiler to use. The following definition suggests that the integer variable counter be placed in one of the computer's registers; regardless of whether the compiler does this, counter is initialized to 1:

register int counter = 1;

The register keyword can be used only with local variables and function parameters.

**Static Storage Class**

Keywords extern and static declare identifiers for variables of the static storage class and for functions. Static-storage-class variables exist from the point at which the program begins execution and last for the duration of the program. A static-storage-class variable's storage is allocated when the program begins execution. Such a variable is initialized once when its declaration is encountered. For functions, the name of the function exists when the program begins execution, just as for all other functions. However, even though the variables and the function names exist from the start of program execution, this does not mean that these identifiers can be used throughout the program.

**Identifiers with Static Storage Class**

There are two types of identifiers with static storage classexternal identifiers (such as **global variables** and global function names) and local variables declared with the storage class specifier static. Global variables are created by placing variable declarations outside any class or function definition. Global variables retain their values throughout the execution of the program. Global variables and global functions can be referenced by any function that follows their declarations or definitions in the source file.

Local variables declared with the keyword static are still known only in the function in which they are declared, but, unlike automatic variables, static local variables retain their values when the function returns to its caller. The next time the function is called, the static local variables contain the values they had when the function last completed execution. The following statement declares local variable count to be static and to be initialized to 1:

static int count = 1;

All numeric variables of the static storage class are initialized to zero if they are not explicitly initialized by the programmer, but it is nevertheless a good practice to explicitly initialize all variables.

Class Assignment 2 - Craps:

Modify the craps program of Fig. 6.11 to allow wagering. Package as a function the portion of the program that runs one game of craps. Initialize variable bankBalance to 1000 dollars. Prompt the player to enter a wager. Use a while loop to check that wager is less than or equal to bankBalance and, if not, prompt the user to reenter wager until a valid wager is entered. After a correct wager is entered, run one game of craps. If the player wins, increase bankBalance by wager and print the new bankBalance. If the player loses, decrease bankBalance by wager, print the new bankBalance, check on whether bankBalance has become zero and, if so, print the message "Sorry. You busted!" As the game progresses, print various messages to create some "chatter" such as "Oh, you're going for broke, huh?", "Aw cmon, take a chance!" or "You're up big. Now's the time to cash in your chips!".

Class Assignment 3 – Prime Numbers

Refer to page 275 in the text book and follow instructions to complete exercise 6.29

Class Assignment 4 – Towers of Hanoi

Refer to page 276 in the text book and follow instructions to complete exercise 6.38

**Exercises**

Answer each of the following:

1. Program components in C++ are called \_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_.
2. A function is invoked with a(n) \_\_\_\_\_\_\_\_.
3. A variable that is known only within the function in which it is defined is called a(n) \_\_\_\_\_\_\_\_.
4. The \_\_\_\_\_\_\_\_ statement in a called function passes the value of an expression back to the calling function.
5. The keyword \_\_\_\_\_\_\_\_ is used in a function header to indicate that a function does not return a value or to indicate that a function contains no parameters.
6. The \_\_\_\_\_\_\_\_ of an identifier is the portion of the program in which the identifier can be used.
7. The three ways to return control from a called function to a caller are \_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_.
8. A(n)\_\_\_\_\_\_\_\_ allows the compiler to check the number, types and order of the arguments passed to a function.
9. Function \_\_\_\_\_\_\_\_ is used to produce random numbers.
10. Function \_\_\_\_\_\_\_\_ is used to set the random number seed to randomize a program.
11. The storage-class specifiers are mutable, \_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_.
12. Variables declared in a block or in the parameter list of a function are assumed to be of storage class \_\_\_\_\_\_\_\_ unless specified otherwise.
13. Storage-class specifier \_\_\_\_\_\_\_\_ is a recommendation to the compiler to store a variable in one of the computer's registers.
14. A variable declared outside any block or function is a(n) \_\_\_\_\_\_\_\_ variable.
15. For a local variable in a function to retain its value between calls to the function, it must be declared with the \_\_\_\_\_\_\_\_ storage-class specifier.
16. The six possible scopes of an identifier are \_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_.
17. A function that calls itself either directly or indirectly (i.e., through another function) is a(n) \_\_\_\_\_\_\_\_ function.
18. A recursive function typically has two components: One that provides a means for the recursion to terminate by testing for a(n) \_\_\_\_\_\_\_\_ case and one that expresses the problem as a recursive call for a slightly simpler problem than the original call.
19. In C++, it is possible to have various functions with the same name that operate on different types or numbers of arguments. This is called function \_\_\_\_\_\_\_\_.

Give the function header for each of the following functions:

1. Function hypotenuse that takes two double-precision, floating-point arguments, side1 and side2, and returns a double-precision, floating-point result.
2. Function smallest that takes three integers, x, y and z, and returns an integer.
3. Function instructions that does not receive any arguments and does not return a value. [Note: Such functions are commonly used to display instructions to a user.]
4. Function intToDouble that takes an integer argument, number, and returns a double-precision, floating-point result.

Find the error in each of the following program segments, and explain how the error can be corrected

1. int g( void )

{

cout << "Inside function g" << endl;

int h( void )

{

cout << "Inside function h" << endl;

}

}

1. int sum( int x, int y )

{

int result;

result = x + y;

}

1. int sum( int n )

{

if ( n == 0 )

return 0;

else

n + sum( n - 1 );

}

1. void f ( double a);

{

float a;

cout << a << endl;

}

1. void product( void )

{

int a;

int b;

int c;

int result;

cout << "Enter three integers: ";

cin >> a >> b >> c;

result = a \* b \* c;

cout << "Result is " << result;

return result;

}

**Project 1 (b) 3 POINTS**

Using the algorithm given above write a fully developed pseudo code for the **run** function for the TicTacToe game based on the description given above. Check if the description is sufficient. If not, what else needs to be done?

**Lesson 5: Arrays and Vectors**

Arrays are data structures consisting of related data items of the same type. Elements in an array are stored in a consecutive group of memory locations and all have the same type. To refer to a particular location or element in the array, we specify the name of the array and the **position number** of the particular element in the array.

Figure 7.1 shows an integer array called c. This array contains 12 elements. A program refers to any one of these elements by giving the name of the array followed by the position number of the particular element in square brackets ([]). The position number is more formally called a **subscript** or **index** (this number specifies the number of elements from the beginning of the array). The first element in every array has subscript 0 (**zero**) and is sometimes called the **zeroth element**. Thus, the elements of array c are c[ 0 ] (pronounced "c sub zero"), c[ 1 ], c[ 2 ] and so on. The highest subscript in array c is 11, which is 1 less than 12the number of elements in the array. Array names follow the same conventions as other variable names, i.e., they must be identifiers.

A subscript must be an integer or integer expression (using any integral type). If a program uses an expression as a subscript, then the program evaluates the expression to determine the subscript. For example, if we assume that variable a is equal to 5 and that variable b is equal to 6, then the statement

c[ a + b ] += 2;

adds 2 to array element c[ 11 ]. Note that a subscripted array name is an lvalueit can be used on the left side of an assignment, just as non-array variable names can.

### Declaring Arrays

Arrays occupy space in memory. The programmer specifies the type of the elements and the number of elements required by an array as follows:

type arrayName [ arraySize ];

and the compiler reserves the appropriate amount of memory. The arraySize must be an integer constant greater than zero. For example, to tell the compiler to reserve 12 elements for integer array c, use the declaration

int c[ 12 ]; // c is an array of 12 integers

Memory can be reserved for several arrays with a single declaration. The following declaration reserves 100 elements for the integer array b and 27 elements for the integer array x.

int b[ 100 ], // b is an array of 100 integers

x[ 27 ]; // x is an array of 27 integers

Arrays can be declared to contain values of any non-reference data type. For example, an array of type char can be used to store a character string. Until now, we have used string objects to store character strings.

*Refer to Pages 286-299 in the textbook for the program*

**Passing Arrays to Functions**

To pass an array argument to a function, specify the name of the array without any brackets. For example, if array hourlyTemperatures has been declared as

int hourlyTemperatures[ 24 ];

the function call

modifyArray( hourlyTemperatures, 24 );

passes array hourlyTemperatures and its size to function modifyArray. When passing an array to a function, the array size is normally passed as well, so the function can process the specific number of elements in the array.

C++ passes arrays to functions by referencethe called functions can modify the element values in the callers' original arrays. The value of the name of the array is the address in the computer's memory of the first element of the array. Because the starting address of the array is passed, the called function knows precisely where the array is stored in memory. Therefore, when the called function modifies array elements in its function body, it is modifying the actual elements of the array in their original memory locations.

Although entire arrays are passed by reference, individual array elements are passed by value exactly as simple variables are. Such simple single pieces of data are called **scalars** or **scalar quantities**. To pass an element of an array to a function, use the subscripted name of the array element as an argument in the function call.

For a function to receive an array through a function call, the function's parameter list must specify that the function expects to receive an array. For example, the function header for function modifyArray might be written as

void modifyArray( int b[], int arraySize )

indicating that modifyArray expects to receive the address of an array of integers in parameter b and the number of array elements in parameter arraySize. The size of the array is not required between the array brackets. If it is included, the compiler ignores it. Because C++ passes arrays to functions by reference, when the called function uses the array name b, it will in fact be referring to the actual array in the caller (i.e., array hourlyTemperatures discussed at the beginning of this section).

Note the strange appearance of the function prototype for modifyArray

void modifyArray( int [], int );

This prototype could have been written

void modifyArray( int anyArrayName[], int anyVariableName );

Class Assignment 1:

Modify the Bonus Problem by declaring an integer array named bonus containing 10 bonus elements and pass the array to the player class to calculate the total bonus the player has received. Display the collective bonus points and display the total score after adding the bonus points. Also calculate the average bonus points and display the results. Use appropriate function and variables when necessary.



Total Bonus collected is 59 points

Total score is 69 points

The average Bonus is 5.9 points

**Constant Keyword**

There may be situations in your programs in which a function should not be allowed to modify array elements. C++ provides the type qualifier const that can be used to prevent modification of array values in the caller by code in a called function. When a function specifies an array parameter that is preceded by the const qualifier, the elements of the array become constant in the function body, and any attempt to modify an element of the array in the function body results in a compilation error. This enables the programmer to prevent accidental modification of array elements in the function's body.

*Figure 7.14* demonstrates the const qualifier. Function tryToModifyArray is defined with parameter const int b[], which specifies that array b is constant and cannot be modified. Each of the three attempts by the function to modify array b's elements results in a compilation error. The Microsoft Visual C++.NET compiler, for example, produces the error "l-value specifies const object." [Note: The C++ standard defines an "object" as any "region of storage," thus including variables or array elements of fundamental data types as well as instances of classes (what we've been calling objects).] This message indicates that using a const object (e.g., b[ 0 ]) as an lvalue is an error you cannot assign a new value to a const object by placing it on the left of an assignment operator. Note that compiler error messages vary between compilers.

### 7.6. Case Study: Class GradeBook Using an Array to Store Grades

### Searching Arrays with Linear Search

*Refer to Pages 304-309 in the textbook for the program*

### Searching Arrays with Linear Search

Often a programmer will be working with large amounts of data stored in arrays. It may be necessary to determine whether an array contains a value that matches a certain **key value**. The process of finding a particular element of an array is called **searching**. In this section we discuss the simple linear search. Exercise 7.33 at the end of this chapter asks you to implement a recursive version of the linear search.

### Linear Search

The **linear search** (Fig. 7.19) compares each element of an array with a **search key**. Because the array is not in any particular order, it is just as likely that the value will be found in the first element as the last. On average, therefore, the program must compare the search key with half the elements of the array. To determine that a value is not in the array, the program must compare the search key to every element in the array.

The linear searching method works well for small arrays or for unsorted arrays (i.e., arrays whose elements are in no particular order). However, for large arrays, linear searching is inefficient. If the array is sorted (e.g., its elements are in ascending order), you can use the high-speed binary search technique that you will learn about in Searching and Sorting.

*Refer to Pages 310-311 in the textbook for the program*

Class Assignment 2

Refer to page 343 in the text book and follow instructions to complete exercise 7.33

### Sorting Arrays with Insertion Sort

**Sorting** data (i.e., placing the data into some particular order such as ascending or descending) is one of the most important computing applications. A bank sorts all checks by account number so that it can prepare individual bank statements at the end of each month. Telephone companies sort their phone directories by last name and, within that, by first name to make it easy to find phone numbers. Virtually every organization must sort some data and, in many cases, massive amounts of data. Sorting data is an intriguing problem that has attracted some of the most intense research efforts in the field of computer science. In this chapter, we discuss a simple sorting scheme.

### Insertion Sort

The program in Fig. 7.20 sorts the values of the 10-element array data into ascending order. The technique we use is called **insertion sort** a simple, but inefficient, sorting algorithm. The first iteration of this algorithm takes the second element and, if it is less than the first element, swaps it with the first element (i.e., the program inserts the second element in front of the first element). The second iteration looks at the third element and inserts it into the correct position with respect to the first two elements, so all three elements are in order. At the ith iteration of this algorithm, the first i elements in the original array will be sorted.

*Refer to Pages 312-313 in the textbook for the program*

The chief virtue of the insertion sort is that it is easy to program; however, it runs slowly. This becomes apparent when sorting large arrays.

### Multidimensional Arrays

**Multidimensional arrays** with two dimensions are often used to represent **tables of values** consisting of information arranged in rows and columns. To identify a particular table element, we must specify two subscripts. By convention, the first identifies the element's row and the second identifies the element's column. Arrays that require two subscripts to identify a particular element are called **two-dimensional arrays** or **2-D arrays**. Note that multidimensional arrays can have more than two dimensions (i.e., subscripts). Figure 7.21 illustrates a two-dimensional array, a. The array contains three rows and four columns, so it is said to be a 3-by-4 array. In general, an array with m rows and n columns is called an m**-by-**n **array**.

Every element in array a is identified in Fig. 7.21 by an element name of the form a[ i ][ j ], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a. Notice that the names of the elements in row 0 all have a first subscript of 0; the names of the elements in column 3 all have a second subscript of 3.

A multidimensional array can be initialized in its declaration much like a one-dimensional array. For example, a two-dimensional array b with values 1 and 2 in its row 0 elements and values 3 and 4 in its row 1 elements could be declared and initialized with

int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };

The values are grouped by row in braces. So, 1 and 2 initialize b[ 0 ][ 0 ] and b[ 0 ][ 1 ], respectfully, and 3 and 4 initialize b[ 1 ][ 0 ] and b[ 1 ][ 1 ], respectfully. If there are not enough initializers for a given row, the remaining elements of that row are initialized to 0. Thus, the declaration

int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };

initializes b[ 0 ][ 0 ] to 1, b[ 0 ][ 1 ] to 0, b[ 1 ][ 0 ] to 3 and b[ 1 ][ 1 ] to 4.

*Refer to Pages 314 - 315 in the textbook for the program*

Class Assignment 3

Type the program listed on Pages 314 - 315 in the textbook and execute it. What do you observe?

**Case Study: Class GradeBook Using a Two-Dimensional Array**

*Refer to Pages 212-214 in the textbook for the program*

### Introduction to C++ Standard Library Class Template vector

We now introduce C++ Standard Library class template vector, which represents a more robust type of array featuring many additional capabilities. C-style pointer-based arrays (i.e., the type of arrays presented thus far) have great potential for errors. For example, a program can easily "walk off" either end of an array, because C++ does not check whether subscripts fall outside the range of an array. Two arrays cannot be meaningfully compared with equality operators or relational operators.

Pointer variables (known more commonly as pointers) contain memory addresses as their values. Array names are simply pointers to where the arrays begin in memory, and, of course, two arrays will always be at different memory locations. When an array is passed to a general-purpose function designed to handle arrays of any size, the size of the array must be passed as an additional argument. Furthermore, one array cannot be assigned to another with the assignment operator(s) array names are const pointers, and, a constant pointer cannot be used on the left side of an assignment operator. These and other capabilities certainly seem like "naturals" for dealing with arrays, but C++ does not provide such capabilities. However, the C++ Standard Library provides class template vector to allow programmers to create a more powerful and less error-prone alternative to arrays.

The vector class template is available to anyone building applications with C++. The notations that the vector example uses might be unfamiliar to you, because vectors use template notation. For now, you should feel comfortable using class template vector by mimicking the syntax in the example we show in this section. You will deepen your understanding as we study class templates.

Vector containers are implemented as dynamic arrays; just as regular arrays, vector containers have their elements stored in contiguous storage locations, which means that their elements can be accessed not only using iterators but also using offsets on regular pointers to elements.  
  
But unlike regular arrays, storage in vectors is handled automatically, allowing it to be expanded and contracted as needed.  
  
Vectors are good at:

1. Accessing individual elements by their position index (constant time).
2. Iterating over the elements in any order (linear time).
3. Add and remove elements from its end (constant amortized time).

Compared to arrays, they provide almost the same performance for these tasks, plus they have the ability to be easily resized. Although, they usually consume more memory than arrays when their capacity is handled automatically (this is in order to accommodate extra storage space for future growth).

Internally, vectors -like all containers- have a size, which represents the amount of elements contained in the vector. But vectors also have a capacity, which determines the amount of storage space they have allocated, and which can be either equal or greater than the actual [size](http://www.cplusplus.com/vector::size). The extra amount of storage allocated is not used, but is reserved for the vector to be used in the case it grows. This way, the vector does not have to reallocate storage on each occasion it grows, but only when this extra space is exhausted and a new element is inserted (which should only happen in logarithmic frequence in relation with its size).

Declaring and initializing vectors

// creates an empty vector of type int

vector<int> first;

// creates four int elements, each with value 100

vector<int> second (4,100);

// creates an vector named third by iterating through vector second and copying all the elements from it.

vector<int> third (second.begin(),second.end());

// creates an vector named fourth that is a copy of vector named thirdvector<int> fourth (third);

**Non-Modifying Operations:**

|  |  |
| --- | --- |
| Operation | Effect |
| v1.size() | Returns the actual number of elements |
| v1.empty() | Returns whether the container is empty (equivalent to size() == 0, but might be faster) |
| v1.max\_size() | Returns the maximum number of elements possible |
| v1.capacity() | Returns the maximum possible number of elements without reallocation |
| v1.reserve() | Enlarges capacity, if not enough yet |
| v1 == v2 | Returns whether v1 is equal to v2 |
| v1 != v2 | Returns whether v1 is not equal to v2 |

**Assignments:**

|  |  |
| --- | --- |
| Operation | Effect |
| v1 = v2 | Assigns all elements of v2 to v1 |
| v1.assign(n elem) | Assigns n copies of element elem |
| v1.assign(beg, end) | Assigns the elements of the range [beg, end] |
| v1.swap(v2) | Swaps the data of v1 and v2 |
| swap(v1, v2) | global function ( same as v1,swap(v2)) |

**Element Access:**

|  |  |
| --- | --- |
| Operation | Effect |
| v.at( idx ) | Returns the element with index idx (throws range error exception if idx is out of range) |
| v[idx] | Returns the element with index idx (no range checking) |
| v.front() | Returns the first element ( no check whether a first element exists) |
| v.back() | Returns the last element( no check whether a last element exits) |

**Inserting and removing elements:**

|  |  |
| --- | --- |
| Operation | Effect |
| v.insert(pos, elem) | Inserts at iterator position “pos” a copy of “elem” and returns the position of the new element |
| v.insert(pos, n, elem) | Inserts at iterator position “pos” “n” copies of “elem” (returns nothing) |
| v.insert(pos,beg,end) | Inserts at iterator position “pos” a copy of all elements of the range [beg, end] (returns nothing) |
| v.push\_back( elem ) | Appends a copy of element at the end |
| v.pop\_back() | Removes the last element ( does not return it) |
| v.erase( pos) | Removes the element at iterator position “pos” and returns the position of the next element |
| v.erase(beg, end) | Removes all elements of the of the range [beg,end] and returns the position of the next element |
| v.resize(num) | Changes the number of elements to num( if size() grows, new elements are created by their default constructor) |
| v.resize(num, elem) | Changes the number of elements to num( if size() grows, new elements are copies of elem) |
| v.clear() | Removes all elements (makes the container empty) |

**Iterator functions:**

|  |  |
| --- | --- |
| Operation | Effect |
| v.begin() | Returns a random access iterator for the first element |
| v.end() | Returns a random access iterator for the position after the last element |
| v.rbegin() | Returns a reverse iterator for the first element of a reverse iteration |
| v.rend() | Returns a reverse iterator for the position after the last element of a reverse iteration |

*Refer to Pages 324 - 327 in the textbook for the program*

//====================================================================

// Example Program

// Traversing through the vector elements using an vector iterators

//====================================================================

#include <iostream>

#include <vector>

using namespace std;

int main ()

{

vector<int> myVector; //Creating an empty integer vector named myVector

//Using a for loop to insert 5 elements in the vector. The elements 0 through 4 will be inserted in the vector.

for (int i = 0; i < 5; i++)

{

myVector.push\_back(i); //this statement inserts the value I in the //vector

}

vector<int>::iterator iter; //declaring a vector iterator of the type int

//The following statement prints the string "myVector contains:";

cout << "myVector contains:";

//The for loop is used to traverse through the vector starting from the first //element to the last. myVector.begin() points to the first element of the //vector. myVector.end() points to the position after the last element.

for ( iter = myVector.begin() ; iter < myVector.end(); iter ++ )

{

cout << " " << \*iter;

}

cout << endl;

return 0;

}

Class Assignment 4:

Create a string vector named wordList. Create a string variable named userInput. Using a while loop keep prompting the user to enter a word until the user enters ctrl+z and store the input in the variable named userInput. After accepting the input from the user, insert the word in the vector named wordlist. Once all the words have been entered, display all the words the user entered by displaying the contents of the vector wordList.

Exercises

1. An array is *not*:
2. A consecutive group of memory locations.
3. Subscripted by integers.
4. Declared using braces, [].
5. Made up of different data types.
   * + 1. Which of the following is *false*?
6. The last element of an array has position number one less than the array size.
7. The position number contained within square brackets is called a subscript.
8. A subscript cannot be an expression.
9. All of the above.
   * + 1. Which statement would be used to declare a 10-element integer array c?
10. array c = int[ 10 ];
11. c = int[ 10 ];
12. int array c[ 10 ];
13. int c[ 10 ];

4. Which of the following is *not* a correct way to initialize an array?

1. int n[ 5 ] = { 0, 7, 0, 3, 8, 2 };
2. int n[] = { 0, 7, 0, 3, 8, 2 };
3. int n[ 5 ] = { 7 };
4. int n[ 5 ] = { 9, 1, 9 };

5. Constant variables:

1. Can be assigned values in executable statements.
2. Do not have to be initialized when they are declared.
3. Can be used to specify array sizes, thereby making programs more scalable.
4. Can be used to specify array sizes, but this makes programs harder to understand.

6. Unless otherwise specified, entire arrays are passed \_\_\_\_\_\_\_\_\_\_ and individual array elements are passed \_\_\_\_\_\_\_\_\_\_.

1. By value, by reference.
2. By reference, by value.
3. By value, by value.
4. By reference, by reference.

7. To *prevent* modification of array values passed to a function:

1. The array must be declared static in the function.
2. The array parameter can be preceded by the const qualifier.
3. A copy of the array must be made inside the function.
4. The array must be passed by reference.

8. Linear search can be used on:

1. Unsorted arrays.
2. Sorted arrays.
3. Integer arrays.
4. Any of the above.

9. Linear search is highly inefficient compared to binary search when dealing with:

1. Small, unsorted arrays.
2. Small, sorted arrays.
3. Large, unsorted arrays.
4. Large, sorted arrays.

10. Which statement about insertion sort is *true*?

1. A maximum of *n* comparisons are needed to sort the array, where *n* is the number of elements.
2. The algorithm is simple compared to other sorting procedures.
3. No temporary variables are needed.
4. Performance is maximized.

11. A double subscripted array declared as int a[ 3 ][ 5 ]; has how many elements?

1. 15
2. 13
3. 10
4. 8

12. Given the following declaration, what is the value of b[ 1 ][ 0 ]?

int b[ 2 ][ 2 ] = { { 1 }, { 3 , 4 } };

1. 0
2. 1
3. 3
4. This is not a valid declaration.

13. Which of the following does *not* declare a 2-by-2 array and set all four of its elements to 0?

1. int b [ 2 ][ 2 ];  
   b[ 0 ][ 0 ] = b[ 0 ][ 1 ] = b[ 1 ][ 0] = b[ 1 ][ 1 ] = 0;
2. int b[ 2 ][ 2 ] = { 0 };
3. int b[ 2 ][ 2 ];  
   for ( int i = 0; i < 2; i++ )  
    for ( int j = 0; j < 2; j++ )  
    b[ i ][ j ] = 0;
4. All of the above initialize all four of the array elements to 0.

14. In a typical *nested* for-loop used to process a two-dimensional array, following the end of the each execution of the inner for loop:

1. The outer for loop initializes its counter variable.
2. The outer for loop increments its counter variable.
3. The inner for loop initializes its counter variable.
4. The inner for loop increments its counter variable.

15. Which of the following is *not* true of class template vector?

1. The size of a vector can be changed after it is declared.
2. A vector can be assigned to another vector by using the assignment operator.
3. A vector object can be initialized with a copy of another vector by invoking the copy constructor.
4. A vector can store only data of type int.

16. Using square brackets ([]) to retrieve vector elements \_\_\_\_\_\_\_\_\_\_ perform bounds checking; using member function at to retrieve vector elements \_\_\_\_\_\_\_\_\_\_ perform bounds checking.

1. Does not, does not.
2. Does not, does.
3. Does, does not.
4. Does, does.

Answer each of the following:

1. Lists and tables of values can be stored in \_\_\_\_\_\_\_\_\_\_ or \_\_\_\_\_\_\_\_\_\_.
2. The elements of an array are related by the fact that they have the same \_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_\_.
3. The number used to refer to a particular element of an array is called its \_\_\_\_\_\_\_\_.
4. A(n) \_\_\_\_\_\_\_\_\_\_ should be used to declare the size of an array, because it makes the program more scalable.
5. The process of placing the elements of an array in order is called \_\_\_\_\_\_\_\_ the array.
6. The process of determining if an array contains a particular key value is called \_\_\_\_\_\_\_\_\_ the array.
7. An array that uses two subscripts is referred to as a(n) \_\_\_\_\_\_\_\_\_ array.

State whether the following are true or false. If the answer is false, explain why.

1. An array can store many different types of values.
2. An array subscript should normally be of data type float.
3. If there are fewer initializers in an initializer list than the number of elements in the array, the remaining elements are initialized to the last value in the initializer list.
4. It is an error if an initializer list contains more initializers than there are elements in the array.
5. An individual array element that is passed to a function and modified in that function will contain the modified value when the called function completes execution.

Write one or more statements that perform the following tasks for and array called fractions:

1. Define a constant variable arraySize initialized to 10.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Declare an array with arraySize elements of type double, and initialize the elements to 0.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Name the fourth element of the array.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Refer to array element 4.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Assign the value 1.667 to array element 9.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Assign the value 3.333 to the seventh element of the array.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Print array elements 6 and 9.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Print all the array elements using a for statement. Define the integer variable i as a control variable for the loop. Show the output.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Answer the following questions regarding an array called table:

1. Declare the array to be an integer array and to have 3 rows and 3 columns. Assume that the constant variable arraySize has been defined to be 3.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. How many elements does the array contain?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Use a for repetition statement to initialize each element of the array to the sum of its subscripts. Assume that the integer variables i and j are declared as control variables.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Write a program segment to print the values of each element of array table in tabular format with 3 rows and 3 columns. Assume that the array was initialized with the declaration:

int table[ arraySize ][ arraySize ] = { { 1, 8 }, { 2, 4, 6 }, { 5 } };

and the integer variables i and j are declared as control variables. Show the output.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Find the error in each of the following program segments and correct the error:

1. #include <iostream>;
2. arraySize = 10; // arraySize was declared const
3. Assume that int b[ 10 ] = { 0 };

for ( int i = 0; <= 10; i++ )

b[ i ] = 1;

1. Assume that int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };

a[ 1, 1 ] = 5;

**Project 1 (C) 5 POINTS**

**Using the specifications, algorithm and the pseudo code, write the code and execute the TicTacToe game.**

**Lesson 6: Pointers**

Pointers are the most powerful features of C++. **Pointers are variables that contain memory addresses as their values.**

int y = 5; //declares an ordinary variable called y

int \*yPtr; //declares a pointer variable called yPtr

int \*yPtr = &y; //creates a pointer yPtr that points to the variable y.

y -- Returns the value of the variable

yPtr -- Returns the address of the variable it points to.

\*yPtr -- Returns the value of the variable it points to.

The **\* operator**, is commonly referred to as the **indirection operator** or **dereferencing operator**

The **& operator**, is commonly referred to as **address of operator** or **referencing operator**

The & (ampersand) read as "the address of", when associated with a variable, returns the memory address of the variable.

**A pointer may be initialized to 0, NULL or an address.** A pointer with the value 0 or NULL points to nothing and is known as a **null pointer**. Symbolic constant NULL is defined in header file <iostream> (and in several other standard library header files) to represent the value 0. Initializing a pointer to NULL is equivalent to initializing a pointer to 0. The value 0 is the only integer value that can be assigned directly to a pointer variable without casting the integer to a pointer type first.

*Refer to Pages 349 - 350 in the textbook for the program*

Class Assignment 1

1. Declare a double variable called weight and initialize it to 128.4.
2. Create a pointer named weightPtr and point it to the variable weight.
3. Display the value directly using the variable itself.
4. Display the address directly using the address of operator.
5. Using the pointer display the value of the variable.
6. Display the address of the variable using the pointer.

Test the above code in Visual Studio and correct compilation errors (if any) and execute the code.

Class Assignment 2

Modify assignment 1 by adding the following lines of code:

1. Set the value of the variable weight to 149.3
2. Repeat steps 3 through 6.
3. Now set \*weightPointer to 173.8
4. Repeat steps 3 through 6.

**Advantages of using a pointer:**

Accessing variables directly by pointers rather than through their memory location results in increased efficiency and flexibility of written code.

**Using const with Pointers**

Recall that the const qualifier enables the programmer to inform the compiler that the value of a particular variable should not be modified.

NOTE: If a value must not change in the body of a function to which it is passed, the parameter should be declared const to ensure that it is not accidentally modified

If an attempt is made to modify a const value, a warning or an error is issued, depending on the particular compiler.

**Nonconstant Pointer to Nonconstant Data**

The highest access is granted by a **nonconstant pointer to nonconstant data**the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data. The declaration for a nonconstant pointer to nonconstant data does not include const. Such a pointer can be used to receive a null-terminated string in a function that changes the pointer value to process (and possibly modify) each character in the string.

**Nonconstant Pointer to Constant Data**

A **nonconstant pointer to constant data** is a pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer. Such a pointer might be used to receive an array argument to a function that will process each element of the array, but should not be allowed to modify the data.

*Refer to Page 356 in the textbook for the program*

**Constant Pointer to Nonconstant Data**

A **constant pointer to nonconstant data** is a pointer that always points to the same memory location; the data at that location can be modified through the pointer. This is the default for an array name. An array name is a constant pointer to the beginning of the array. All data in the array can be accessed and changed by using the array name and array subscripting. A constant pointer to nonconstant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation. Pointers that are declared const must be initialized when they are declared. (If the pointer is a function parameter, it is initialized with a pointer that is passed to the function.)

*Refer to Page 357 in the textbook for the program*

**Constant Pointer to Constant Data**

The least amount of access privilege is granted by a **constant pointer to constant data**. Such a pointer always points to the same memory location, and the data at that memory location cannot be modified using the pointer. This is how an array should be passed to a function that only reads the array, using array subscript notation, and does not modify the array.

*Refer to Page 358 in the textbook for the program*

**Selection Sort Using Pass-by-Reference**

We use the **selection sort** algorithm, which is an easy-to-program, but unfortunately inefficient, sorting algorithm. The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element. The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element. The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index. After the ith iteration, the smallest i items of the array will be sorted into increasing order in the first i elements of the array.

*Refer to Pages 359 - 361 in the textbook for the program*

**Pointer Arithmetic**

Several arithmetic operations may be performed on pointers. A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=) or one pointer may be subtracted from another.

Assume that array int v[ 5 ] has been declared and that its first element is at memory location 3000. Assume that pointer vPtr has been initialized to point to v[ 0 ] (i.e., the value of vPtr is 3000).

Note that vPtr can be initialized to point to array v with either of the following statements (because the name of an array is equivalent to the address of its first element):

int \*vPtr = v;

int \*vPtr = &v[ 0 ];

In conventional arithmetic, the addition 3000 + 2 yields the value 3002. This is normally not the case with pointer arithmetic. When an integer is added to, or subtracted from, a pointer, the pointer is not simply incremented or decremented by that integer, but by that integer times the size of the object to which the pointer refers. The number of bytes depends on the object's data type. For example, the statement

vPtr += 2;

would produce 3008 (3000 + 2 \* 4), assuming that an int is stored in four bytes of memory. In the array v, vPtr would now point to v[ 2 ]

If vPtr had been incremented to 3016, which points to v[4], the statement

vPtr -= 4;

would set vPtr back to 3000the beginning of the array. If a pointer is being incremented or decremented by one, the increment (++) and decrement (--) operators can be used. Each of the statements increments the pointer to point to the next element of the array.

++vPtr;

vPtr++;

Each of the statements decrements the pointer to point to the previous element of the array.

--vPtr;

vPtr--;

Pointer variables pointing to the same array may be subtracted from one another. For example, if vPtr contains the location 3000 and v2Ptr contains the address 3008, the statement

x = v2Ptr - vPtr;

would assign to x the number of array elements from vPtr to v2Ptrin this case, Pointer arithmetic is meaningless unless performed on a pointer that points to an array. We cannot assume that two variables of the same type are stored contiguously in memory unless they are adjacent elements of an array.

NOTE: Subtracting or comparing two pointers that do not refer to elements of the same array is a logic error.

A pointer can be assigned to another pointer if both pointers are of the same type. Otherwise, a cast operator must be used to convert the value of the pointer on the right of the assignment to the pointer type on the left of the assignment. The exception to this rule is the pointer to void (i.e., void \*), which is a generic pointer capable of representing any pointer type. All pointer types can be assigned to a pointer of type void \* without casting. However, a pointer of type void \* cannot be assigned directly to a pointer of another type the pointer of type void \* must first be cast to the proper pointer type.

A void \* pointer cannot be dereferenced. For example, the compiler "knows" that a pointer to int refers to four bytes of memory on a machine with four-byte integers, but a pointer to void simply contains a memory address for an unknown data type the precise number of bytes to which the pointer refers and the type of the data are not known by the compiler. The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer for a pointer to void, this number of bytes cannot be determined from the type.

Pointers can be compared using equality and relational operators. Comparisons using relational operators are meaningless unless the pointers point to members of the same array. Pointer comparisons compare the addresses stored in the pointers. A comparison of two pointers pointing to the same array could show, for example, that one pointer points to a higher numbered element of the array than the other pointer does. A common use of pointer comparison is determining whether a pointer is 0 (i.e., the pointer is a null pointer; it does not point to anything).

**Pointers and Arrays**

An array name is a constant pointer to the beginning of the array. The relationship between C++ pointers and arrays are closely related and its use is almost interchangeable.

**Note:** Although array names are pointers to the beginning of the array and pointers can be modified in arithmetic expressions, array names cannot be modified in arithmetic expressions, because array names are constant pointers.

int arr[5] = {1,2,3,4,5};

arr++; //will throw an error: C2105: '++' needs l-value

*Refer to Page 369 in the textbook for the program*

Class Assignment 3

Create an array named myArray of size 10 and using an initializer list set it with random values. Create a pointer to the array named myArrayPtr. Display the elements of the array using the pointer. (Hint use a for loop and pointer arithmetic)

**Array of Pointers**

It is possible to create an array of pointers in c++. The most common example is an array of strings.

char \*suit[4] = {“Hearts”, “Diamonds”, “Clubs”, “Spades”};

**Review:**

Q1: Pointers are variables that contain *\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Q2: Pointers may be assigned which of the following values?

1. Any integer values.
2. An address.
3. NULL.
4. Both (b) and (c).

Q3: What does the following statement declare?

int \*countPtr, count;

1. Two int variables.
2. One pointer to an int and one int variable.
3. Two pointers to ints.
4. The declaration is invalid.

Q4: Three of the following expressions have the same value. Which of the following expressions has a value *different* from the others’?

1. \*&ptr
2. &\*ptr
3. \*ptr
4. ptr

Q5: The following is the principle of least privilege:

1. A nonconstant pointer to nonconstant data.
2. A nonconstant pointer to constant data.
3. A constant pointer to nonconstant data.
4. A constant pointer to constant data.

Q6: Which of the following *can* have a pointer as an operand?

1. ++
2. \*=
3. %
4. /

Q7: Given that k is an integer array starting at location 2000, kPtr is a pointer to k and each integer is stored in 4 bytes of memory, what location does kPtr + 3 point to?

1. 2003
2. 2006
3. 2012
4. 2024

Q8: Assuming that t is an array and tPtr is a pointer to that array, which expression refers to the address of element 3 of the array?

1. \*( tPtr + 3 )
2. tPtr[ 3 ]
3. &t[ 3 ]
4. \*( t + 3 )

### Arrays of Pointers

Arrays may contain pointers. A common use of such a data structure is to form an array of pointer-based strings, referred to simply as a **string array**. Each entry in the array is a string, but in C++ a string is essentially a pointer to its first character, so each entry in an array of strings is simply a pointer to the first character of a string. Consider the declaration of string array suit that might be useful in representing a deck of cards:

const char \*suit[ 4 ] =

{ "Hearts", "Diamonds", "Clubs", "Spades" };

The suit[4] portion of the declaration indicates an array of four elements. The const char \* portion of the declaration indicates that each element of array suit is of type "pointer to char constant data." The four values to be placed in the array are "Hearts", "Diamonds", "Clubs" and "Spades". Each is stored in memory as a null-terminated character string that is one character longer than the number of characters between quotes. The four strings are seven, nine, six and seven characters long (including their terminating null characters), respectively. Although it appears as though these strings are being placed in the suit array, only pointers are actually stored in the array, as shown in Fig. 8.19. Each pointer points to the first character of its corresponding string. Thus, even though the suit array is fixed in size, it provides access to character strings of any length. This flexibility is one example of C++'s powerful data-structuring capabilities.

##### Refer to Pages 373 Fig 8.19 for Graphical representation of the suit array.

The suit strings could be placed into a two-dimensional array, in which each row represents one suit and each column represents one of the letters of a suit name. Such a data structure must have a fixed number of columns per row, and that number must be as large as the largest string. Therefore, considerable memory is wasted when we store a large number of strings, of which most are shorter than the longest string. We use arrays of strings to help represent a deck of cards in the next section.

### Function Pointers

A pointer to a function contains the address of the function in memory. The name of a function is actually the starting address in memory of the code that performs the function's task. Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.

#### Multipurpose Selection Sort Using Function Pointers

To illustrate the use of pointers to functions, Fig. 8.28 modifies the selection sort program of Fig. 8.15. Figure 8.28 consists of main and the functions selectionSort, swap, ascending and descending. Function selectionSort receives a pointer to a function either function ascending or function descending as an argument in addition to the integer array to sort and the size of the array. Functions ascending and descending determine the sorting order. The program prompts the user to choose whether the array should be sorted in ascending order or in descending order. If the user enters 1, a pointer to function ascending is passed to function selectionSort, causing the array to be sorted into increasing order. If the user enters 2, a pointer to function descending is passed to function selectionSort, causing the array to be sorted into decreasing order.

*Refer to Pages 374 - 376 in the textbook for the program.*

**Fundamentals of Characters and Pointer-Based Strings**

Characters are the fundamental building blocks of C++ source programs. Every program is composed of a sequence of characters that when grouped together meaningfully is interpreted by the compiler as a series of instructions used to accomplish a task. A program may contain **character constants**. A character constant is an integer value represented as a character in single quotes. The value of a character constant is the integer value of the character in the machine's character set. For example, 'z' represents the integer value of z (122 in the ASCII character set; see Appendix B), and '\n' represents the integer value of newline (10 in the ASCII character set).

A string is a series of characters treated as a single unit. A string may include letters, digits and various **special characters** such as +, -, \*, / and $. String literals, or **string constants**, in C++ are written in double quotation marks as follows:

| "John Q. Doe" | (a name) |
| --- | --- |
| "9999 Main Street" | (a street address) |
| "Maynard, Massachusetts" | (a city and state) |
| "(201) 555-1212" | (a telephone number) |

A pointer-based string in C++ is an array of characters ending in the null character ('\0'), which marks where the string terminates in memory. A string is accessed via a pointer to its first character. The value of a string is the address of its first character. Thus, in C++, it is appropriate to say that a string is a constant pointer in fact, a pointer to the string's first character. In this sense, strings are like arrays, because an array name is also a pointer to its first element.

A string literal may be used as an initializer in the declaration of either a character array or a variable of type char \*. The declarations

char color[] = "blue";

const char \*colorPtr = "blue";

each initialize a variable to the string "blue". The first declaration creates a five-element array color containing the characters 'b', 'l', 'u', 'e' and '\0'. The second declaration creates pointer variable colorPtr that points to the letter b in the string "blue" (which ends in '\0') somewhere in memory. String literals have static storage class (they exist for the duration of the program) and may or may not be shared if the same string literal is referenced from multiple locations in a program. Also, string literals in C++ are constant their characters cannot be modified.

The declaration char color[] = "blue"; could also be written

char color[] = { 'b', 'l', 'u', 'e', '\0' };

When declaring a character array to contain a string, the array must be large enough to store the string and its terminating null character. The preceding declaration determines the size of the array, based on the number of initializers provided in the initializer list.

A string can be read into a character array using stream extraction with cin. For example, the following statement can be used to read a string into character array word[ 20 ]:

cin >> word;

The string entered by the user is stored in word. The preceding statement reads characters until a white-space character or end-of-file indicator is encountered. Note that the string should be no longer than 19 characters to leave room for the terminating null character. The setw stream manipulator can be used to ensure that the string read into word does not exceed the size of the array. For example, the statement

cin >> setw( 20 ) >> word;

specifies that cin should read a maximum of 19 characters into array word and save the 20th location in the array to store the terminating null character for the string. The setw stream manipulator applies only to the next value being input. If more than 19 characters are entered, the remaining characters are not saved in word, but will be read in and can be stored in another variable.

In some cases, it is desirable to input an entire line of text into an array. For this purpose, C++ provides the function **cin.getline** in header file <iostream>. The cin.getline function takes three argumentsa character array in which the line of text will be stored, a length and a delimiter character. For example, the program segment

char sentence[ 80 ];

cin.getline( sentence, 80, '\n' );

declares array sentence of 80 characters and reads a line of text from the keyboard into the array. The function stops reading characters when the delimiter character '\n' is encountered, when the end-of-file indicator is entered or when the number of characters read so far is one less than the length specified in the second argument. (The last character in the array is reserved for the terminating null character.) If the delimiter character is encountered, it is read and discarded. The third argument to cin.getline has '\n' as a default value, so the preceding function call could have been written as follows:

cin.getline( sentence, 80 );

Class Assignment 4

Refer to page 384 in the text book and follow instructions to complete exercise 8.11

Class Assignment 5

Refer to page 384 in the text book and follow instructions to complete exercise 8.12

Class Assignment 6

Refer to pages 385 and 286 in the text book and follow instructions to complete exercises 8.13 and 8.14

Class Assignment 7

Refer to page 388 in the text book and follow instructions to complete exercise 8.17

**Lesson 7: Classes: A Deeper Look Part 1**

**Preprocessor Wrapper**

A Preprocessor Wrapper is an important C++ software engineering concept which is used in header files to prevent the code in the header from being included into the same source code file more than once. Since a class can be defined only once, using such preprocessor directives prevents multiple-definition errors.

In Fig. 9.1, note that the class definition is enclosed in the following **preprocessor wrapper**:

// prevent multiple inclusions of header file

#ifndef TIME\_H

#define TIME\_H

...

#endif

When we build larger programs, other definitions and declarations will also be placed in header files. The preceding preprocessor wrapper prevents the code between #ifndef (which means "if not defined") and #endif from being included if the name TIME\_H has been defined. If the header has not been included previously in a file, the name TIME\_H is defined by the #define directive and the header file statements are included. If the header has been included previously, TIME\_H is defined already and the header file is not included again. Attempts to include a header file multiple times (inadvertently) typically occur in large programs with many header files that may themselves include other header files. [Note: The commonly used convention for the symbolic constant name in the preprocessor directives is simply the header file name in upper case with the underscore character replacing the period.]

*Refer to Pages 397 - 402 in the textbook for the program*

**Defining Member Functions Outside the Class Definition; Class Scope**

Even though a member function declared in a class definition may be defined outside that class definition (and "tied" to the class via the binary scope resolution operator), that member function is still within that class's scope; i.e., its name is known only to other members of the class unless referred to via an object of the class, a reference to an object of the class, a pointer to an object of the class or the binary scope resolution operator.

If a member function is defined in the body of a class definition, the C++ compiler attempts to inline calls to the member function. Member functions defined outside a class definition can be inlined by explicitly using keyword inline. Remember that the compiler reserves the right not to inline any function.

[NOTE: Only the simplest and most stable member functions (i.e., whose implementations are unlikely to change) should be defined in the class header.]

**Member Functions vs. Global Functions**

It is interesting to note that the member functions take no arguments. This is because these member functions implicitly know that they are to print the data members of the particular Time object for which they are invoked. This can make member function calls more concise than conventional function calls in procedural programming.

#### Object Size

People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions. Logically, this is true the programmer may think of objects as containing data and functions (and our discussion has certainly encouraged this view); physically, however, this is not true.

**Objects contain only data, so objects are much smaller than if they also contained member functions. Applying operator sizeof to a class name or to an object of that class will report only the size of the class's data members. The compiler creates one copy (only) of the member functions separate from all objects of the class. All objects of the class share this one copy. Each object, of course, needs its own copy of the class's data, because the data can vary among the objects. The function code is non-modifiable (also called** **reentrant code or** **pure procedure) and, hence, can be shared among all objects of one class.**

### Class Scope and Accessing Class Members

A class's data members (variables declared in the class definition) and member functions (functions declared in the class definition) belong to that class's scope. Nonmember functions are defined at **file scope**.

Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name. Outside a class's scope, public class members are referenced through one of the **handles** on an object an object name, a reference to an object or a pointer to an object. The type of the object, reference or pointer specifies the interface (i.e., the member functions) accessible to the client.

Member functions of a class can be overloaded, but only by other member functions of that class. To overload a member function, simply provide in the class definition a prototype for each version of the overloaded function, and provide a separate function definition for each version of the function.

Variables declared in a member function have block scope and are known only to that function. If a member function defines a variable with the same name as a variable with class scope, the class-scope variable is hidden by the block-scope variable in the block scope. Such a hidden variable can be accessed by preceding the variable name with the class name followed by the scope resolution operator (::). Hidden global variables can be accessed with the unary scope resolution operator.

The dot member selection operator (.) is preceded by an object's name or with a reference to an object to access the object's members. The arrow member selection operator (->) is preceded by a pointer to an object to access the object's members.

*Refer to Pages 404 - 405 in the textbook for the program*

**Access Functions and Utility Functions**

**Access functions** can read or display data. Another common use for access functions is to test the truth or falsity of conditions such functions are often called **predicate functions**. An example of a predicate function would be an isEmpty function for any container class a class capable of holding many objects such as a linked list, a stack or a queue. A program might test isEmpty before attempting to read another item from the container object. An isFull predicate function might test a container-class object to determine whether it has no additional room. Useful predicate functions for our Time class might be isAM and isPM.

*Refer to Pages 406 - 409 in the textbook for the demonstration of a utility function (also called a* ***helper function****).*

A utility function is not part of a class's public interface; rather, it is a private member function that supports the operation of the class's public member functions. Utility functions are not intended to be used by clients of a class.

### Time Class Case Study: Constructors with Default Arguments

*Refer to Pages 409 - 413 in the textbook for the demonstration of how arguments are implicitly passed to a constructor.*

[NOTE: If a member function of a class already provides all or part of the functionality required by a constructor (or other member function) of the class, call that member function from the constructor (or other member function). This simplifies the maintenance of the code and reduces the likelihood of an error if the implementation of the code is modified. As a general rule: Avoid repeating code.]

A constructor can call other member functions of the class, such as set or get functions, but because the constructor is initializing the object, the data members may not yet be in a consistent state. Using data members before they have been properly initialized can cause logic errors.

Class Assignment 1

Modify the Player class and add a new constructor that takes default arguments for health and score.

### Destructors

A **destructor** is another type of special member function. The name of the destructor for a class is the **tilde character (~)** followed by the class name. This naming convention has intuitive appeal, because as we will see in a later chapter, the tilde operator is the bitwise complement operator, and, in a sense, the destructor is the complement of the constructor.

A class's destructor is called implicitly when an object is destroyed. This occurs, for example, as an automatic object is destroyed when program execution leaves the scope in which that object was instantiated. The destructor itself does not actually release the object's memory it performs **termination housekeeping** before the system reclaims the object's memory, so the memory may be reused to hold new objects.

A destructor receives no parameters and returns no value. A destructor may not specify a return type, not even void. A class may have only one destructor, destructor overloading is not allowed.

Even though destructors have not been provided for the classes presented so far, every class has a destructor. If the programmer does not explicitly provide a destructor, the compiler creates an "empty" destructor.

### When Constructors and Destructors Are Called

Constructors and destructors are called implicitly by the compiler. The order in which these function calls occur depends on the order in which execution enters and leaves the scopes where the objects are instantiated. Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but as we will see in Figs. 9.11- 9.13, the storage classes of objects can alter the order in which destructors are called.

Constructors are called for objects defined in global scope before any other function (including main) in that file begins execution (although the order of execution of global object constructors between files is not guaranteed). The corresponding destructors are called when main terminates. Function exit forces a program to terminate immediately and does not execute the destructors of automatic objects. The function often is used to terminate a program when an error is detected in the input or if a file to be processed by the program cannot be opened. Function **abort** performs similarly to function exit but forces the program to terminate immediately, without allowing the destructors of any objects to be called. Function abort is usually used to indicate an abnormal termination of the program. The constructor for an automatic local object is called when execution reaches the point where that object is defined the corresponding destructor is called when execution leaves the object's scope (i.e., the block in which that object is defined has finished executing). Constructors and destructors for automatic objects are called each time execution enters and leaves the scope of the object. Destructors are not called for automatic objects if the program terminates with a call to function exit or function abort.

The constructor for a static local object is called only once, when execution first reaches the point where the object is defined the corresponding destructor is called when main terminates or the program calls function exit. Global and static objects are destroyed in the reverse order of their creation. Destructors are not called for static objects if the program terminates with a call to function abort.

*Refer to Pages 416 - 418 in the textbook for the demonstration of the order in which constructors and destructors are called for objects of class CreateAndDestroy (Fig. 9.11 and Fig. 9.12) of various storage classes in several scopes.*

Each object of class CreateAndDestroy contains an integer (objectID) and a string (message) that are used in the program's output to identify the object.

### Default Memberwise Assignment

The assignment operator (=) can be used to assign an object to another object of the same type. By default, such assignment is performed by **memberwise assignment** each data member of the object on the right of the assignment operator is assigned individually to the same data member in the object on the left of the assignment operator. [Caution: Member-wise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory]

Objects may be passed as function arguments and may be returned from functions. Such passing and returning is performed using pass-by-value by default a copy of the object is passed or returned. In such cases, C++ creates a new object and uses a **copy constructor** to copy the original object's values into the new object. For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object. Like member-wise assignment, copy constructors can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory.

*Refer to Pages 421 - 422 in the textbook for the program*

Class Assignment 2

Draw a class diagrams for Weapon class, Armor Class and Enemy Class. Ensure that each of these classes contain atleast:

1. Default Constructor
2. Overloaded Constructor with Default Arguments
3. Destructor
4. Appropriate Set and Get functions
5. Display Function

Ensure to use proper variable names and function names. The classes should contain appropriate information for the classes. Example: Armor class needs a variable that will enable us to determine the defense/attack strengths.

The Destructors just prints appropriate messages enabling us to understand that the object has ceased to exist.

Class Assignment 3

Implement the above class diagrams into code

Exercises

Fill in the blanks in each of the following:

1. Class members are accessed via the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ operator in conjunction with the name of an object (or reference to an object) of the class or via the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ operator in conjunction with a pointer to an object of the class.
2. Class members specified as \_\_\_\_\_\_\_\_\_\_\_\_\_ are accessible only to member functions of the class and friends of the class.
3. Class members specified as \_\_\_\_\_\_\_\_\_\_\_\_\_ are accessible anywhere an object of the class is in scope.
4. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ can be used to assign an object of a class to another object of the same class.

Find the error(s) in each of the following and explain how to correct it (them):

1. Assume the following prototype is declared in class Time:

void ~Time( int );

1. The following is a partial definition of class Time:

class Time

{

public:

// function prototypes

private:

int hour = 0;

int minute = 0;

int second = 0;

}; // end class Time

1. Assume the following prototype is declared in class Employee:

int Employee( const char \*, const char \* );

Multiple Choice:

* + - 1. Member access specifiers (public and private) can appear:

1. In any order and multiple times.
2. In any order (public first or private first) but not multiple times.
3. In any order and multiple times, if they have brackets separating each type.
4. Outside a class definition.

2. Which of the following preprocessor directives does not constitute part of the preprocessor wrapper?

1. #define
2. #endif
3. #ifndef
4. #include

3. Member function definitions:

1. Always require the binary scope operator (::).
2. Require the binary scope operator only when being defined outside of the definition of their class.
3. Can use the binary scope operator anywhere, but become public functions.
4. Must use the binary scope operator in their function prototype.

4. Every object of the same class:

1. Gets a copy of every member function and member variable.
2. Gets a copy of every member variable.
3. Gets a copy of every member function.
4. Shares pointers to all member variables and member functions.

5. Classes *cannot*:

1. Be derived from other classes.
2. Initialize data members in the class definition.
3. Be used to model attributes and behaviors of objects.
4. Include objects from other classes as members.

6. Variables defined inside a member function of a class have:

1. File scope.
2. Class scope.
3. Block scope.
4. Class or block scope, depending on whether the binary scope resolution operator (::) is used.

7. A class-scope variable hidden by a block-scope variable can be accessed by preceding the variable name with the class name followed by:

1. ::
2. :
3. .
4. ->

8. When independent software vendors provide class libraries to clients, they typically give the \_\_\_\_\_\_\_\_\_\_ for the class’s interface and the \_\_\_\_\_\_\_\_\_\_ for the class’s implementation.

1. Source code file, source code file.
2. Source code file, object file.
3. Object file, source code file.
4. Object file, object file.

9. The type of function a client would use to check the balance of a bank account would be:

1. A utility function.
2. A predicate function.
3. An access function.
4. A constructor.

10. Utility functions:

1. Are private member functions that support operations of the class’s other member functions.
2. Are part of a class’s interface.
3. Are intended to be used by clients of a class.
4. Are a type of constructor.

11. A default constructor:

1. Is a constructor that must receive no arguments.
2. Is the constructor generated by the compiler when no constructor is provided by the programmer.
3. Does not perform any initialization.
4. Both (a) and (b).

12. If a member function of a class already provides all or part of the functionality required by a constructor or another member function then:

1. Copy and paste that member function’s code into this constructor or member function.
2. Call that member function from this constructor or member function.
3. That member function is unnecessary.
4. This constructor or member function is unnecessary.

13. Which of the following is *not* true of a destructor?

1. It performs termination housekeeping.
2. It is called before the system reclaims the object’s memory.
3. If the programmer does not explicitly provide a destructor, the compiler creates an “empty” destructor.
4. It releases the object’s memory.

14. Given the class definition:

class CreateDestroy

{

public:

CreateDestroy() { cout << "constructor called, "; }

~CreateDestroy() { cout << "destructor called, "; }

};

What will the following program output?

int main()

{

CreateDestroy c1;

CreateDestroy c2;

return 0;

}

1. constructor called, destructor called, constructor called, destructor called,
2. constructor called, destructor called,
3. constructor called, constructor called,
4. constructor called, constructor called, destructor called, destructor called,

15. Given the class definition:

class CreateDestroy

{

public:

CreateDestroy() { cout << "constructor called, "; }

~CreateDestroy() { cout << "destructor called, "; }

};

What will the following program output?

int main()

{

for ( int i = 1; i <= 2; i++ )

CreateDestroy cd;

return 0;

}

1. constructor called, destructor called, constructor called, destructor called,
2. constructor called, constructor called,
3. constructor called, constructor called, destructor called, destructor called,
4. Nothing.

16. The assignment operator (=) *can* be used to:

1. Test for equality.
2. Copy data from one object to another.
3. Compare two objects.
4. Copy a class.

**Project 2 Maze POINTS 10**

**Complete the project as per the instructions provided by your instructor.**

**Lesson 8: Classes: A Deeper Look Part 2**

**const (Constant) Objects and const Member Functions**

Some objects need to be modifiable and some do not. The programmer may use keyword const to specify that an object is not modifiable and that any attempt to modify the object should result in a compilation error. The statement

const Time noon( 12, 0, 0 );

declares a const object noon of class Time and initializes it to 12 noon.

Declaring an object as const helps enforce the principle of least privilege. Attempts to modify the object are caught at compile time rather than causing execution-time errors. Using const properly is crucial to proper class design, program design and coding

Declaring variables and objects const can improve performance today's sophisticated optimizing compilers can perform certain optimizations on constants that cannot be performed on variables.

C++ compilers disallow member function calls for const objects unless the member functions themselves are also declared const. This is true even for get member functions that do not modify the object. In addition, the compiler does not allow member functions declared const to modify the object.

A function is specified as const both in its prototype (Fig. 10.1) and in its definition (Fig. 10.2 by inserting the keyword const after the function's parameter list and, in the case of the function definition, before the left brace that begins the function body.

An interesting problem arises for constructors and destructors, each of which typically modifies objects. The const declaration is not allowed for constructors and destructors. A constructor must be allowed to modify an object so that the object can be initialized properly. A destructor must be able to perform its termination housekeeping chores before an object's memory is reclaimed by the system.

#### Defining and Using const Member Functions

*Refer to Pages 431 - 434 in the textbook for the program*

#### Initializing a const Data Member with a Member Initializer

All data members can be initialized using member initializer syntax, but const data members and data members that are references must be initialized using member initializers.

In this example, count is initialized with the value of constructor parameter c and increment is initialized with the value of constructor parameter i. Note that multiple member initializers are separated by commas. Also, note that the member initializer list executes before the body of the constructor executes.

*Refer to Pages 435 - 436 in the textbook for the program*

[NOTE: A const object cannot be modified by assignment, so it must be initialized. When a data member of a class is declared const, a member initializer must be used to provide the constructor with the initial value of the data member for an object of the class. The same is true for references.]

### Composition: Objects as Members of Classes

An AlarmClock object needs to know when it is supposed to sound its alarm, so why not include a Time object as a member of the AlarmClock class? Such a capability is called **composition** and is sometimes referred to as a has**-a relationship**. A class can have objects of other classes as members.

When an object is created, its constructor is called automatically. Previously, we saw how to pass arguments to the constructor of an object we created in main. This section shows how an object's constructor can pass arguments to member-object constructors, which is accomplished via member initializers. Member objects are constructed in the order in which they are declared in the class definition (not in the order they are listed in the constructor's member initializer list) and before their enclosing class objects (sometimes called **host objects**) are constructed.

The program of Figs. 10.10-10.14 uses class Date (Figs. 10.10 and 10.11) and class Employee (Figs. 10.12and 10.13) to demonstrate objects as members of other objects. The definition of class Employee (Fig. 10.12) contains private data members firstName, lastName, birthDate and hireDate. Members birthDate and hireDate are const objects of class Date, which contains private data members month, day and year. The Employee constructor's header (Fig. 10.13) specifies that the constructor receives four parameters (first, last, dateOfBirth and dateOfHire). The first two parameters are used in the constructor's body to initialize the character arrays firstName and lastName. The last two parameters are passed via member initializers to the constructor for class Date. The colon (:) in the header separates the member initializers from the parameter list. The member initializers specify the Employee constructor parameters being passed to the constructors of the member Date objects. Parameter dateOfBirth is passed to object birthDate's constructor (Fig. 10.13), and parameter dateOfHire is passed to object hireDate's constructor (Fig. 10.13). Again, member initializers are separated by commas. As you study class Date (Fig. 10.10), notice that the class does not provide a constructor that receives a parameter of type Date.

**friend Functions and friend Classes**

A **friend function** of a class is defined outside that class's scope, yet has the right to access the non-public (and public) members of the class. Standalone functions or entire classes may be declared to be friends of another class. Using friend functions can enhance performance. This section presents a mechanical example of how a friend function works.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend. To declare all member functions of class ClassTwo as friends of class ClassOne, place a declaration of the form

friend class ClassTwo;

in the definition of class ClassOne.

[NOTE: Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

Member access notions of private, protected and public are not relevant to friend declarations, so friend declarations can be placed anywhere in a class definition.]

**Friendship is granted, not taken i.e., for class B to be a friend of class A, class A must explicitly declare that class B is its friend. Also, the friendship relation is neither symmetric nor transitive; i.e., if class A is a friend of class B, and class B is a friend of class C, you cannot infer that class B is a friend of class A (again, friendship is not symmetric), that class C is a friend of class B (also because friendship is not symmetric), or that class A is a friend of class C (friendship is not transitive).**

**Modifying a Class's private Data With a Friend Function**

Figure 10.15 is a mechanical example in which we define friend function setX to set the private data member x of class Count. Note that the friend declaration appears first (by convention) in the class definition, even before public member functions are declared. Again, this friend declaration can appear anywhere in the class.

Function setX is a C-style, stand-alone function it is not a member function of class Count. For this reason, when setX is invoked for object counter, and passes counter as an argument to setX rather than using a handle (such as the name of the object) to call the function, as in

counter.setX( 8 );

As we mentioned, Fig. 10.15 is a mechanical example of using the friend construct. It would normally be appropriate to define function setX as a member function of class Count. It would also normally be appropriate to separate the program of Fig. 10.15 into three files:

1. A header file (e.g., Count.h) containing the Count class definition, which in turn contains the prototype of friend function setX
2. An implementation file (e.g., Count.cpp) containing the definitions of class Count's member functions and the definition of friend function setX
3. A test program (e.g., fig10\_15.cpp) with main

#### Erroneously Attempting to Modify a private Member with a Non-friend Function

The program of Fig. 10.16 demonstrates the error messages produced by the compiler when non-friend function cannotSetX is called to modify private data member x.

It is possible to specify overloaded functions as friends of a class. Each overloaded function intended to be a friend must be explicitly declared in the class definition as a friend of the class.

#### Using the this Pointer

We have seen that an object's member functions can manipulate the object's data. How do member functions know which object's data members to manipulate? Every object has access to its own address through a pointer called **this** (a C++ keyword). An object's this pointer is not part of the object itself i.e., the size of the memory occupied by the this pointer is not reflected in the result of a sizeof operation on the object. Rather, the this pointer is passed (by the compiler) as an implicit argument to each of the object's non-static member functions. Section 10.7 introduces static class members and explains why the this pointer is not implicitly passed to static member functions.

Objects use the this pointer implicitly (as we have done to this point) or explicitly to reference their data members and member functions. The type of the this pointer depends on the type of the object and whether the member function in which this is used is declared const. For example, in a nonconstant member function of class Employee, the this pointer has type Employee \* const (a constant pointer to a nonconstant Employee object). In a constant member function of the class Employee, the this pointer has the data type const Employee \* const (a constant pointer to a constant Employee object).

Our first example in this section shows implicit and explicit use of the this pointer; later in this chapter, we show some substantial and subtle examples of using this.

#### Implicitly and Explicitly Using the this Pointer to Access an Object's Data Members

Figure 10.17 demonstrates the implicit and explicit use of the this pointer to enable a member function of class Test to print the private data x of a Test object.

For illustration purposes, member function print first prints x by using the this pointer implicitly only the name of the data member is specified. Then print uses two different notations to access x through the this pointer the arrow operator (->) off the this pointer and the dot operator (.) off the dereferenced this pointer.

Note the parentheses around \*this when used with the dot member selection operator (.). The parentheses are required because the dot operator has higher precedence than the \* operator. Without the parentheses, the expression \*this.x would be evaluated as if it were parenthesized as \*( this.x), which is a compilation error, because the dot operator cannot be used with a pointer. One interesting use of the this pointer is to prevent an object from being assigned to itself.

### Dynamic Memory Management with Operators new and delete

C++ enables programmers to control the allocation and deallocation of memory in a program for any built-in or user-defined type. This is known as **dynamic memory management** and is performed with operators **new** and **delete**. Recall that class Employee (Figs. 10.12 and 10.13) uses two 25-character arrays to represent the first and last name of an Employee. The Employee class definition (Fig. 10.12) must specify the number of elements in each of these arrays when it declares them as data members, because the size of the data members dictates the amount of memory required to store an Employee object. As we discussed earlier, these arrays may waste space for names shorter than 24 characters. Also, names longer than 24 characters must be truncated to fit in these fixed-size arrays.

Wouldn't it be nice if we could use arrays containing exactly the number of elements needed to store an Employee's first and last name? Dynamic memory management allows us to do exactly that. As you will see in the example of Section 10.7, if we replace array data members firstName and lastName with pointers to char, we can use the new operator to dynamically **allocate** (i.e., reserve) the exact amount of memory required to hold each name at execution time. Dynamically allocating memory in this fashion causes an array (or any other built-in or user-defined type) to be created in the **free store** (sometimes called the **heap**)a region of memory assigned to each program for storing objects created at execution time. Once the memory for an array is allocated in the free store, we can gain access to it by aiming a pointer at the first element of the array. When we no longer need the array, we can return the memory to the free store by using the delete operator to **deallocate** (i.e., release) the memory, which can then be reused by future new operations.

Again, we present the modified Employee class as described here in the example of Section 10.7. First, we present the details of using the new and delete operators to dynamically allocate memory to store objects, fundamental types and arrays.

Consider the following declaration and statement:

Time \*timePtr;

timePtr = new Time;

The new operator allocates storage of the proper size for an object of type Time, calls the default constructor to initialize the object and returns a pointer of the type specified to the right of the new operator (i.e., a Time \*). Note that new can be used to dynamically allocate any fundamental type (such as int or double) or class type. If new is unable to find sufficient space in memory for the object, it indicates that an error occurred by "throwing an exception."

To destroy a dynamically allocated object and free the space for the object, use the delete operator as follows:

delete timePtr;

This statement first calls the destructor for the object to which timePtr points, then deallocates the memory associated with the object. After the preceding statement, the memory can be reused by the system to allocate other objects.

[NOTE: Not releasing dynamically allocated memory when it is no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "**memory leak**."]

C++ allows you to provide an initializer for a newly created fundamental-type variable, as in

double \*ptr = new double( 3.14159 );

which initializes a newly created double to 3.14159 and assigns the resulting pointer to ptr. The same syntax can be used to specify a comma-separated list of arguments to the constructor of an object. For example,

Time \*timePtr = new Time( 12, 45, 0 );

initializes a newly created Time object to 12:45 PM and assigns the resulting pointer to timePtr.

As discussed earlier, the new operator can be used to allocate arrays dynamically. For example, a 10-element integer array can be allocated and assigned to gradesArray as follows:

int \*gradesArray = new int[ 10 ];

which declares pointer gradesArray and assigns it a pointer to the first element of a dynamically allocated 10-element array of integers. Recall that the size of an array created at compile time must be specified using a constant integral expression. However, the size of a dynamically allocated array can be specified using any integral expression that can be evaluated at execution time. Also note that, when allocating an array of objects dynamically, the programmer cannot pass arguments to each object's constructor. Instead, each object in the array is initialized by its default constructor. To delete the dynamically allocated array to which gradesArray points, use the statement

delete [] gradesArray;

The preceding statement deallocates the array to which gradesArray points. If the pointer in the preceding statement points to an array of objects, the statement first calls the destructor for every object in the array, then deallocates the memory. If the preceding statement did not include the square brackets ([]) and gradesArray pointed to an array of objects, only the first object in the array would receive a destructor call.

[NOTE: Using delete instead of delete [] for arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with *operator delete []*. Similarly, always delete memory allocated as an individual element with operator delete.]

### static Class Members

There is an important exception to the rule that each object of a class has its own copy of all the data members of the class. In certain cases, only one copy of a variable should be shared by all objects of a class. A **static data member** is used for these and other reasons. Such a variable represents "class-wide" information (i.e., a property of the class shared by all instances, not a property of a specific object of the class). The declaration of a static member begins with keyword static. Recall that the versions of class GradeBook in Chapter 7 use static data members to store constants representing the number of grades that all GradeBook objects can hold.

Let us further motivate the need for static class-wide data with an example. Suppose that we have a video game with Martians and other space creatures. Each Martian tends to be brave and willing to attack other space creatures when the Martian is aware that there are at least five Martians present. If fewer than five are present, each Martian becomes cowardly. So each Martian needs to know the martianCount. We could endow each instance of class Martian with martianCount as a data member. If we do, every Martian will have a separate copy of the data member. Every time we create a new Martian, we will have to update the data member martianCount in all Martian objects. Doing this would require every Martian object to have, or have access to, handles to all other Martian objects in memory. This wastes space with the redundant copies and wastes time in updating the separate copies. Instead, we declare martianCount to be static. This makes martianCount class-wide data. Every Martian can access martianCount as if it were a data member of the Martian, but only one copy of the static variable martianCount is maintained by C++. This saves space. We save time by having the Martian constructor increment static variable martianCount and having the Martian destructor decrement martianCount. Because there is only one copy, we do not have to increment or decrement separate copies of martianCount for each Martian object.

Although they may seem like global variables, a class's static data members have class scope. Also, static members can be declared public, private or protected. A fundamental-type static data member is initialized by default to 0. If you want a different initial value, a static data member can be initialized once (and only once). A const static data member of int or enum type can be initialized in its declaration in the class definition. However, all other static data members must be defined at file scope (i.e., outside the body of the class definition) and can be initialized only in those definitions. Note that static data members of class types (i.e., static member objects) that have default constructors need not be initialized because their default constructors will be called.

A class's private and protected static members are normally accessed through public member functions of the class or through friends of the class. A class's static members exist even when no objects of that class exist. To access a public static class member when no objects of the class exist, simply prefix the class name and the binary scope resolution operator (::) to the name of the data member. For example, if our preceding variable martianCount is public, it can be accessed with the expression Martian::martianCount when there are no Martian objects. (Of course, using public data is discouraged.)

A class's public static class members can also be accessed through any object of that class using the object's name, the dot operator and the name of the member (e.g., myMartian.martianCount). To access a private or protected static class member when no objects of the class exist, provide a public static member function and call the function by prefixing its name with the class name and binary scope resolution operator.

[NOTE: A class's static data members and static member functions exist and can be used even if no objects of that class have been instantiated.]

The program of Figs. 10.21 - 10.23 demonstrates a private static data member called count (Fig. 10.21) and a public static member function called getCount (Fig. 10.21). In Fig. 10.22, line 14 defines and initializes the data member count to zero at file scope and lines 1821 define static member function getCount. When static is applied to an item at file scope, that item becomes known only in that file. The static members of the class need to be available from any client code that accesses the file, so we cannot declare them static in the .cpp file we declare them static only in the .h file. Data member count maintains a count of the number of objects of class Employee that have been instantiated. When objects of class Employee exist, member count can be referenced through any member function of an Employee objecti n Fig. 10.22, count is referenced by both the constructor and the destructor. Also, note that since count is an int, it could have been initialized in the header file at line 21 of Fig. 10.21.

A member function should be declared static if it does not access non-static data members or non-static member functions of the class. Unlike non-static member functions, a static member function does not have a this pointer, because static data members and static member functions exist independently of any objects of a class. The this pointer must refer to a specific object of the class, and when a static member function is called, there might not be any objects of its class in memory.

Exercises

Fill in the blanks in each of the following:

1. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ must be used to initialize constant members of a class.
2. A nonmember function must be declared as a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ of a class to have access to that class's private data members.
3. The \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ operator dynamically allocates memory for an object of a specified type and returns a \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ to that type.
4. A constant object must be \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_; it cannot be modified after it is created.
5. A(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ data member represents class-wide information.
6. An object's non-static member functions have access to a "self pointer" to the object called the \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ pointer.
7. The keyword \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ specifies that an object or variable is not modifiable after it is initialized.
8. If a member initializer is not provided for a member object of a class, the object's \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ is called.
9. A member function should be declared static if it does not access \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ class members.
10. The \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ operator reclaims memory previously allocated by new.

Find the errors in the following class and explain how to correct them:

class Example

{

public:

Example( int y = 10 )

: data( y )

{

// empty body

} // end Example constructor

int getIncrementedData() const

{

return data++;

} // end function getIncrementedData

static int getCount()

{

cout << "Data is " << data << endl;

return count;

} // end function getCount

private:

int data;

static int count;

}; // end class Example

Multiple Choice

1.Which of the following statements will not produce a syntax error?

1. Defining a const member function that modifies a data member of the object.
2. Invoking a non-const member function on a const object.
3. Declaring an object to be const.
4. Declaring a constructor to be const.

2. The code fragment:

Increment::Increment( int c, int i ) : increment ( i )

{

count = c;

}

does *not* cause any compilation errors. This tells you that:

1. count must be a non-const variable.
2. count must be a const variable.
3. increment must be a non-const variable.
4. increment must be a const variable.

3. When composition (one object having another object as a member) is used:

1. The host object is constructed first and then the member objects are placed into it.
2. Member objects are constructed first, in the order they appear in the host constructor’s initializer list.
3. Member objects are constructed first, in the order they are declared in the host’s class.
4. Member objects are destructed last, in the order they are declared in the host’s class.

4. An error occurs if:

1. A non-reference, non-const, primitive data member is initialized in the member initialization list.
2. An object data member is not initialized in the member initialization list.
3. An object data member does not have a default constructor.
4. An object data member is not initialized in the member initialization list and does not have a default constructor.

5. If the line:

friend class A;

appears in class B, and the line:

friend class B;

appears in class C, then:

1. Class A is a friend of class C.
2. Class A can access private variables of class B.
3. Class C can call class A’s private member functions.
4. Class B can access class A’s private variables.

6. Which of the following is not true about friend functions and friend classes?

1. A class can either grant friendship to or take friendship from another class using the friend keyword.
2. A friend declaration can appear anywhere in a class definition.
3. A friend of a class can access all of its private data member and member functions.
4. The friendship relationship is neither symmetric nor transitive.

7. Inside a function definition for a member function of an object with data element x, which of the following is *not* equivalent to this->x:

1. \*this.x
2. (\*this).x
3. x
4. (\* (& (\*this) ) ).x

8. If Americans are objects of the same class, which of the following attributes would most likely be represented by a staticvariable of that class?

1. Age.
2. The President.
3. Place of birth.
4. Favorite food.

9. static data members of a certain class:

1. Can be accessed only if an object of that class exists.
2. Cannot be changed, even by objects of the same *that* class.
3. Have class scope.
4. Can only be changed by static member functions.

10. static member functions:

1. Can use the this pointer.
2. Can access only other static member functions and static data members.
3. Cannot be called until an object of their class is instantiated.
4. Can be declared const as well.

11. Which of the following is *not* an abstract data type?

1. An int.
2. A user-defined class.
3. A for loop.
4. None of the above are abstract data types.

12. Which of the following are *true* about an abstract data type?

I. Captures a data representation.

II. Defines the operations that are allowed on its data.

III. Replaces structured programming.

1. I, II and III.
2. I and II.
3. I and III.
4. II and III

**Lesson 9: Operator Overloading**

**Operator Overloading**

C++ enables the programmer to overload most operators to be sensitive to the context in which they are used. The compiler generates the appropriate code based on the context (in particular, the types of the operands). Although operator overloading sounds like an exotic capability, most programmers implicitly use overloaded operators regularly. For example, the C++ language itself overloads the addition operator (+) and the subtraction operator (-). These operators perform differently, depending on their context in integer arithmetic, floating-point arithmetic and pointer arithmetic.

**Fundamentals of Operator Overloading**

In C++ programmers can use fundamental data types and can define new data types. The fundamental types can be used with C++'s rich collection of operators. Operators provide programmers with a concise notation for expressing manipulations of objects of fundamental types.

Programmers can use operators with user-defined types as well. Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when these operators are used with objects, the operators have meaning appropriate to those objects. This is a powerful capability.

An operator is overloaded by writing a non-static member function definition or global function definition as you normally would, except that the function name now becomes the keyword operator followed by the symbol for the operator being overloaded. For example, the function name operator+ would be used to overload the addition operator (+). When operators are overloaded as member functions, they must be non-static, because they must be called on an object of the class and operate on that object.

To use an operator on class objects, that operator must be overloaded with three exceptions.

* Assignment operator (=)
  + May be used with every class to perform memberwise assignment between objects
* Address operator (&)
  + Returns address of object
* Comma operator (,)
  + Evaluates expression to its left then the expression to its right
  + Returns the value of the expression to its right

Overloading provides concise notation

object2 = object1.add( object2 );

vs.

object2 = object2 + object1;

Operator overloading is not automatic, you must write operator-overloading functions to perform the desired operations. Sometimes these functions are best made member functions; sometimes they are best as friend functions; occasionally they can be made global, non-friend functions.

The jobs performed by overloaded operators can also be performed by function calls, but operator notation is often clearer and more familiar to programmers. An operator is overloaded by writing a non-static member-function definition or global function definition in which the function name is the keyword operator followed by the symbol for the operator being overloaded. When operators are overloaded as member functions, they must be non-static, because they must be called on an object of the class and operate on that object.

**Restrictions on Operator Overloading**

##### See Figure 11.1. Operators that can be overloaded and Figure 11.2. Operators that cannot be overloaded on page 469 in the textbook

We cannot change:

* Precedence of operator (order of evaluation)
  + Use parentheses to force order of operators
* Associativity (left-to-right or right-to-left)
* "arity" - Number of operands
  + e.g., & is unary, can only act on one operand
* How operators act on built-in data types (i.e., cannot change integer addition)
* Cannot create new operators; only existing operators can be overloaded
* Operators must be overloaded explicitly
* Overloading + and = does not overload +=
* Operator ?: cannot be overloaded

Overloading an assignment operator and an addition operator to allow statements like

object2 = object2 + object1;

does not imply that the += operator is also overloaded to allow statements such as

object2 += object1;

Such behavior can be achieved only by explicitly overloading operator += for that class.

**Operator Functions as Class Members vs. Global Functions**

Operator functions can be member functions or global functions; global functions are often made friends for performance reasons. Member functions use the this pointer implicitly to obtain one of their class object arguments (the left operand for binary operators). Arguments for both operands of a binary operator must be explicitly listed in a global function call.

When an operator function is implemented as a member function:

* Leftmost object must be of same class as operator function
* Use this keyword to implicitly get left operand argument
* Operators (), [], -> or any assignment operator must be overloaded as a class member function
* Operator member functions of a specific class are called when
  + Left operand of binary operator is of this class
  + Single operand of unary operator is of this class
* As global functions
  + Need parameters for both operands
  + Can have object of different class than operator
  + Can be a friend to access private or protected data

When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.

If the left operand must be an object of a different class or a fundamental type, this operator function must be implemented as a global function.

A global operator function can be made a friend of a class if that function must access private or protected members of that class directly.

**Overloaded Stream Insertion and Stream Extraction Operators Are Overloaded as Global Functions**

Another example of an overloaded operator built into C++ is operator <<, which is used both as the stream insertion operator and as the bitwise left-shift operator. Similarly, >> is also overloaded; it is used both as the stream extraction operator and as the bitwise right-shift operator.

The overloaded stream insertion operator (<<) is used in an expression in which the left operand has type ostream &, as in cout << classObject. To use the operator in this manner where the right operand is an object of a user-defined class, it must be overloaded as a global function. To be a member function, operator << would have to be a member of the ostream class. This is not possible for user-defined classes, since we are not allowed to modify C++ Standard Library classes.

Similarly, the overloaded stream extraction operator (>>) is used in an expression in which the left operand has type istream &, as in cin >> classObject, and the right operand is an object of a user-defined class, so it, too, must be a global function. Also, each of these overloaded operator functions may require access to the private data members of the class object being output or input, so these overloaded operator functions can be made friend functions of the class for performance reasons.

It is possible to overload an operator as a global, non-friend function, but such a function requiring access to a class's private or protected data would need to use set or get functions provided in that class's public interface. The overhead of calling these functions could cause poor performance, so these functions can be inlined to improve performance.

The overloaded stream insertion operator (<<) is used in an expression in which the left operand has type ostream &.

The function operator<< accepts 2 arguments; a reference to ostream and the appropriate data type.

=====================================================================  
Function prototype to be included in the class definition is:  
=====================================================================

friend ostream& operator<<(ostream&, const ClassName&);

friend istream& operator>>(istream&, ClassName&);

//----------------------------------------------------------------

// Player.h This contains the class definition.

//----------------------------------------------------------------

#include <iostream>

#include <string>

#include <ctime>

using namespace std;

class Player

{

// Function prototypesOverloaded insertion and

// extration operators. Note that the left

// parameter is a reference to object of the output/input

// streams and the right parameter is a reference

// to an object of a user defined datatype

friend ostream &operator<<(ostream&, const Player&);

friend istream &operator>>(istream& , Player&);

private:

string playerName; //stores the name of the player

int playerScore; //stores the player's score

};

//----------------------------------------------------------------

// Player.cpp

//----------------------------------------------------------------

#include "Player.h"

ostream& operator<<(ostream& output, const Player& player)

{

output<<"Name is: "<<player.playerName<<"Score is: "

<<player.playerScore<<endl;

return output;

}

istream& operator>>(istream& input, Player& player)

{

input >> player.playerName;

input >> player.playerScore;

return input;

}

//----------------------------------------------------------------

// Main.cpp

//----------------------------------------------------------------

#include "Player.h"

int main()

{

Player player;

cout<<"Please enter your name and score separated by spaces: ";

//Notice that we are able to use the name of the object

//directly without using any member functions.

cin>>player;

cout<<"You entered: "<<endl;

//Notice that we are able to use the name of the object

//directly without using any member functions.

cout<<player;

}

**Overloading Binary Operators**

A binary operator can be overloaded as a non-static member function with one argument or as a global function with two arguments (one of those arguments must be either a class object or a reference to a class object).

==============================================================  
Function prototype to be included in the class definition for overloading the + and = = operator as a *member function* is:  
==============================================================  
ClassName operator+(const ClassName& right) const;

bool operator==(const ClassName& right) const;

========================================================================  
Function prototype to be included in the class definition for overloading the + and = = operator as a *global function* is:  
========================================================================

friend ClassName operator+(const ClassName& left , const ClassName& right);

friend bool operator==(const ClassName& left, const ClassName& right);

//----------------------------------------------------------------

// BinaryOperator.h

// In this program the binary operators + and == are overloaded

// as member functions

//----------------------------------------------------------------

#include <iostream>

#include <string>

using namespace std;

class BinaryOperator

{

//Overloaded insertion and extraction operators

friend ostream& operator<<(ostream&, const BinaryOperator&);

friend istream& operator>>(istream&, BinaryOperator&);

public:

BinaryOperator(string ="", int =0); //constructor

//Overloaded + operator

BinaryOperator operator+(const BinaryOperator&) const;

//Overloaded == operator

bool operator==(const BinaryOperator&) const;

//Get Function

string getName();

private:

//Member variables

int playerScore;

string playerName;

};

//----------------------------------------------------------------

// BianryOperator.cpp

//----------------------------------------------------------------

#include "BinaryOperator.h"

BinaryOperator::BinaryOperator(string name , int score )

{

playerName = name;

playerScore = score;

}

string BinaryOperator::getName()

{

//returns the contents of the member variable playerName

return playerName;

}

BinaryOperator BinaryOperator::operator+(const BinaryOperator &right) const

{

BinaryOperator temp; //create a temp object to store the result

//adding the score of the current object and the score of the

// object right and storing in temp object

temp.playerScore = playerScore + right.playerScore;

temp.playerName = ""; //setting the name of the temp obj to null

return temp; //return temp object

}

bool BinaryOperator::operator ==(const BinaryOperator & right) const

{

//tests if the player score of the object rigth is same as

// score of the current object

if(right.playerScore == playerScore)

{

return true; // returns true if scores are same

}

else

{

return false; // returns false if scores are not same

}

}

//----------------------------------------------------------------

// Main.cpp

//----------------------------------------------------------------

#include "BinaryOperator.h"

int main()

{

// create an object called alien and assign name and score

BinaryOperator alien("Zork", 250);

// create an object called cowBoy and assign name and score

BinaryOperator cowBoy("Buffalo Bill", 540);

// Notice that we are using the + operator directly to add the

// contents of 2 objects and display them. This is not

// possible if we did not overload the + operator and the

// << operator

cout<<"Total score of alien and cowboy is: "<< alien + cowBoy <<endl;

// Notice that we are using the == operator directly to

// check the contents of 2 objects. This is not possible if

// we did not overload the == operator

if(alien == cowBoy)

{

cout<<alien.getName() << " And " << cowBoy.getName() <<" have same scores"<<endl;

}

else

{

cout<<alien.getName() << " And " << cowBoy.getName() <<" do not have same scores"<<endl;

}

cout<<"\nEnter new score for cowBoy: ";

// Notice that we are using the >> operator directly to get the

// score for cowBoy object. This is not possible if we did not

// overload the >> operator

cin>>cowBoy;

cout<<"\nYou have entered: "<<cowBoy<<endl;

if(alien == cowBoy)

{

cout<<alien.getName() << " And " << cowBoy.getName() <<" have same scores"<<endl;

}

else

{

cout<<alien.getName() << " And " << cowBoy.getName() <<" do not have same scores"<<endl;

}

return 0;

}

ostream& operator<<(ostream& output, const BinaryOperator& object)

{

output << object.playerScore; // storing the score in the output

return output; //returning the output object

}

istream& operator>>(istream& input, BinaryOperator& object)

{

input >> object.playerScore; //gets the score stored

return input;

}

**Overloading Unary operator**

The process of overloading unary operators is similar to that of binary operators. In case of unary the operand is only one. Therefore to overload the unary class:  
 1. If the operator function is a member of class, it has no parameters  
 2. If the operator is a non- member (friend), it has one parameter.

**Overloading ++ and --**

The prefix and postfix versions of the increment and decrement operators can all be overloaded. We will see how the compiler distinguishes between the prefix version and the postfix version of an increment or decrement operator. To overload the increment operator to allow both prefix and postfix increment usage, each overloaded operator function must have a distinct signature, so that the compiler will be able to determine which version of ++ is intended. The prefix versions are overloaded exactly as any other prefix unary operator would be.

==============================================================  
Function prototype to be included in the class definition for overloading the ++ prefix and postfix operators as member functions ==============================================================  
ClassName& operator++(); // prefix ++ operator

ClassName operator++ (int) //postfix ++ operator

==============================================================  
Function prototype to be included in the class definition for overloading the ++ prefix and postfix operators as global functions ==============================================================  
ClassName& operator++ (ClassName&); // prefix ++ operator

ClassName operator++ (ClassName&, int) //postfix ++ operator

The integer argument usually passed as 0 is strictly a "dummy value" that enables the compiler to distinguish between the prefix and postfix increment operator functions.

Exercises

Fill in the blanks in each of the following:

1. Suppose a and b are integer variables and we form the sum a + b. Now suppose c and d are floating-point variables and we form the sum c + d. The two + operators here are clearly being used for different purposes. This is an example of \_\_\_\_\_\_\_\_\_\_.
2. Keyword \_\_\_\_\_\_\_\_\_\_ introduces an overloaded-operator function definition.
3. To use operators on class objects, they must be overloaded, with the exception of operators \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_.
4. The \_\_\_\_\_\_\_\_\_\_, \_\_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_ of an operator cannot be changed by overloading the operator.

Multiple Choice

* + - 1. The correct function name for overloading the addition (+) operator is:

1. operator+
2. operator(+)
3. operator:+
4. operator\_+

2. Which of the following operators cannot be overloaded?

1. The . operator.
2. The -> operator.
3. The & operator.
4. The [ ] operator.

3. Which statement about operator overloading is false?

1. New operators can never be created.
2. Certain overloaded operators can change the number of arguments they take.
3. The precedence of an operator cannot be changed by overloading.
4. Overloading cannot change how an operator works on built-in types.

4. To implicitly overload the += operator:

1. Only the + operator needs to be overloaded.
2. Only the = operator needs to be overloaded.
3. Both the + and = operators need to be overloaded.
4. The += operator cannot be overloaded implicitly.

5. Which situation would require the operator to be overloaded as a global function?

1. The overloaded operator is =.
2. The left most operand must be a class object (or a reference to a class object).
3. The left operand is an int.
4. The operator returns a reference.

6. Suppose you have a programmer-defined data type Data and want to overload the << operator to output your data type to the screen in the form cout << dataToPrint; and allow cascaded function calls. The first line of the function definition would be:

1. ostream &operator<<( ostream &output, const Data &dataToPrint )
2. ostream operator<<( ostream &output, const Data &dataToPrint )
3. ostream &operator<<( const Data &dataToPrint, ostream &output )
4. ostream operator<<( const Data &dataToPrint, ostream &output )

7. For operators overloaded as non-static member functions:

1. Binary operators can have two arguments and unary operators can have one.
2. Both binary and unary operators take one argument.
3. Binary operators can have one argument, and unary operators cannot have any.
4. Neither binary nor unary operators can have arguments.

8. The conventional way to distinguish between the overloaded preincrement and postincrement operators (++) is:

1. To assign a dummy value to preincrement.
2. To make the argument list of postincrement include an int.
3. To have the postincrement operator call the preincrement operator.
4. Implicitly done by the compiler.

9. There exists a data type Date with member function Increment that increments the current Date object by one. The ++ operator is being overloaded to postincrement an object of type Date. Select the correct implementation:

1. Date Date::operator++( int )  
   {  
    Date temp = \*this;  
    Increment();  
    return \*temp;  
   }
2. Date Date::operator++( int )  
   {  
    Increment();  
    Date temp = \*this;  
    return temp;  
   }
3. Date Date::operator++( int )  
   {  
    Date temp = \*this;  
    return this;  
    temp.Increment();  
   }
4. Date Date::operator++( int )  
   {  
    Date temp = \*this;  
    Increment();  
    return temp;  
   }

**Lesson 10: Object-Oriented Programming: Inheritance**

Inheritance is a form of software reuse in which the programmer creates a class that absorbs an existing class's data and behaviors and enhances them with new capabilities. An is-a relationship represents inheritance. In an is-a relationship, an object of a derived class also can be treated as an object of its base class. For example, a car is a vehicle, so any property and behavior of a vehicle are also a property of a car.

**Advantages:**

Software reusability saves time during program development. It also encourages the reuse of proven, debugged, high-quality software, which increases the likelihood that a system will be implemented effectively.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should **inherit** the members of an existing class. This existing class is called the **base class (also called super class or parent class)**, and the new class is referred to as the **derived class (also called sub class or child class).**

A derived class represents a more specialized group of objects. Typically, a derived class contains behaviors inherited from its base class plus additional behaviors.

**Class Hierarchy**

A **direct base class** is the base class from which a derived class explicitly inherits.

An **indirect base class** is inherited from two or more levels up in the **class hierarchy**.

**Single inheritance**, a class inherits from one base class.

**Multiple inheritance**, a derived class inherits from multiple (possibly unrelated) base classes. Multiple inheritance can be complex and error prone.

With inheritance, the common data members and member functions of all the classes in the hierarchy are declared in a base class. When changes are required for these common features, software developers need to make the changes only in the base class. The derived classes then inherit the changes. Without inheritance, changes would need to be made to all the source code files that contain a copy of the code in question.

**Abstraction**

Focuses on commonalities among objects in system

“**is-a” vs. “has-a”**

The **is-a** relationship represents inheritance. In an is-a relationship, an object of a derived class also can be treated as an object of its base class. Example: Car *is a* vehicle, so any properties and behaviors of a vehicle are also properties of a car.

The **has-a** relationship represents composition. In a has-a relationship, an object contains one or more objects of other classes as members. Example: Car *has a* steering wheel. A car includes many components; it has a brake pedal, has a transmission and has many other components.

**Base Classes and Derived Classes**

Often, an object of one class is an object of another class, as well. For example, in geometry, a rectangle is a quadrilateral (as are squares, parallelograms and trapezoids). Thus, in C++, class Rectangle can be said to inherit from class Quadrilateral. In this context, class Quadrilateral is a base class, and class Rectangle is a derived class. A rectangle is a specific type of quadrilateral, but it is incorrect to claim that a quadrilateral is a rectangle. The quadrilateral could be a parallelogram or some other shape.

Because every derived-class object is an object of its base class, and one base class can have many derived classes, the set of objects represented by a base class typically is larger than the set of objects represented by any of its derived classes. For example, the base class Vehicle represents all vehicles, including cars, trucks, boats, airplanes, bicycles and so on. By contrast, derived class Car represents a smaller, more specific subset of all vehicles.

Inheritance relationships form treelike hierarchical structures. A base class exists in a hierarchical relationship with its derived classes. Although classes can exist independently, once they are employed in inheritance relationships, they become affiliated with other classes. A class becomes either a base class supplying members to other classes, a derived class inheriting its members from other classes, or both.

**Deriving from Base Class**

A derived class automatically contains all the members of the base class with some restrictions. The only members of a base class that are not inherited by a derived class are:

1. Constructors
2. Copy Constructors
3. Destructors
4. Overloaded Assignment Operators

This is because the derived class requires having its own copies of the above.

**Access Control Specifiers**

**Private**

This is the default access mode for C++ classes. Only members of that class can access the members declared as private. Usually data members of a class should be kept private. This enables data hiding.

**Public**

Outside programs / functions /classes can access the publicly declared members. Usually the functions are declared as public. Public functions can be used to set the value of private data members.

**Protected**

These members are not accessible outside the class. However, these members are accessible by its derived class.

**Controlling Access under Inheritance**

A derived class's member functions cannot access its base class's private data. To enable a derived class to be able to access the members of a base class, we can declare those as protected in the base class.

**Advantages and Disadvantages of protected**

Inheriting protected data members slightly increases performance, because we can directly access the members without incurring the overhead of calls to set or get member functions. In most cases, however, it is better to use private data members to encourage proper software engineering, and leave code optimization issues to the compiler. This means that the code will be easier to maintain, modify and debug.

Using protected data members creates two major problems. First, the derived-class object does not have to use a member function to set the value of the base-class's protected data member. Therefore, a derived-class object easily can assign an invalid value to the protected data member, thus leaving the object in an inconsistent state.

The second problem with using protected data members is that derived-class member functions are more likely to be written so that they depend on the base-class implementation. In practice, derived classes should depend only on the base-class services (i.e., non private member functions) and not on the base-class implementation. With protected data members in the base class, if the base-class implementation changes, we may need to modify all derived classes of that base class. In such a case, the software is said to be **fragile** or **brittle**, because a small change in the base class can "break" derived-class implementation.

**Relationship between Base Classes and Derived Classes**

*Refer to Pages 528 - 531 in the textbook for the program*

**Including the Base Class Header File in the Derived Class Header File with #include**

We #include the base class's header file in the derived class's header file. This is necessary for three reasons. First, we must tell the compiler that the base class exists.

The second reason is that the compiler uses a class definition to determine the size of an object of that class. A client program that creates an object of a class must #include the class definition to enable the compiler to reserve the proper amount of memory for the object. When using inheritance, a derived-class object's size depends on the data members declared explicitly in its class definition and the data members inherited from its direct and indirect base classes.

The last reason is to allow the compiler to determine whether the derived class uses the base class's inherited members properly.

//----------------------------------------------------------

// Filename: Boss.h

// Base class definition

//----------------------------------------------------------

#ifndef BOSS\_H

#define BOSS\_H

#include <iostream>

using namespace std;

class Boss

{

public:

Boss();

~Boss();

void setDamage(int att);

int getDamage();

void displayAttack();

private:

int damage;

};

#endif

//----------------------------------------------------------

// Filename: BigBoss.h

// Derived class definition

//----------------------------------------------------------

#ifndef BIGBOSS\_H

#define BIGBOSS\_H

#include "Boss.h"

//public inheritance of Boss class - BigBoss class automatically inherits

// all the public and protected members of the Boss class

class BigBoss : public Boss

{

public:

BigBoss();

~BigBoss();

void setDamageMultiplier(int dm);

int getDamageMultiplier();

void specialAttack();

private:

int damageMultiplier;

};

#endif

//----------------------------------------------------------

// Filename: Boss.cpp

// Base class implementation

//----------------------------------------------------------

#include "Boss.h"

Boss::Boss()

{

cout<<"Boss Constructor Called !"<<endl;

setDamage(10);

}

Boss::~Boss()

{

cout<<"Boss Destructor Called !"<<endl;

}

void Boss::setDamage(int att)

{

damage = att;

}

int Boss::getDamage()

{

return damage;

}

void Boss::displayAttack()

{

cout<<"Attack causes "<<getDamage()<<" damage"<<endl;

}

//----------------------------------------------------------

// Filename: BigBoss.cpp

// Derived class implementation

//----------------------------------------------------------

#include "BigBoss.h"

BigBoss::BigBoss()

{

cout<<"Big Boss Constructor Called !"<<endl;

setDamageMultiplier(3);

}

BigBoss::~BigBoss()

{

cout<<"Big Boss Destructor Called !"<<endl;

}

void BigBoss::setDamageMultiplier(int dm)

{

damageMultiplier = dm;

}

int BigBoss::getDamageMultiplier()

{

return damageMultiplier;

}

void BigBoss::specialAttack()

{

cout<<"Special Attack causes "<<getDamageMultiplier() \* getDamage()<<" damage"<<endl;

}

//----------------------------------------------------------

// Filename: Main.cpp

// Main Function

//----------------------------------------------------------

#include "Boss.h"

#include "BigBoss.h"

int main()

{

cout<<"Boss: "<<endl;

Boss boss; //creating an object of the base class Boss

boss.displayAttack();

cout<<endl<<endl;

cout<<"Big Boss"<<endl;

BigBoss bb; //creating an object of the derived class BigBoss

bb.displayAttack(); //boss

bb.specialAttack(); //bigboss

cout<<endl<<endl;

return 0;

}

**Initializing data members**

C++ requires a derived-class constructor to call its base-class constructor to initialize the base-class data members that are inherited into the derived class. A compilation error occurs if a derived-class constructor calls one of its base-class constructors with arguments that are inconsistent with the number and types of parameters specified in one of the base-class constructor definitions. In a derived-class constructor, initializing member objects and invoking base-class constructors explicitly in the member initializer list prevents duplicate initialization in which a default constructor is called, then data members are modified again in the derived-class constructor's body.

//----------------------------------------------------------

// Filename: Boss.h

// Base class definition

//----------------------------------------------------------

#ifndef BOSS\_H

#define BOSS\_H

#include <iostream>

using namespace std;

class Boss

{

public:

Boss(int damageValue);

~Boss();

void setDamage(int att);

int getDamage();

void displayAttack();

private:

int damage;

};

//----------------------------------------------------------

// Derived class definition

//public inheritance of Boss class - BigBoss class automatically inherits

// all the public and protected members of the Boss class

//----------------------------------------------------------

class BigBoss : public Boss

{

public:

BigBoss(int damage, int damageMultiplierValue);

~BigBoss();

void setDamageMultiplier(int dm);

int getDamageMultiplier();

void specialAttack();

private:

int damageMultiplier;

};

#endif

//----------------------------------------------------------

// Filename: Boss.cpp

// Base class implementation

//----------------------------------------------------------

#include "Boss.h"

Boss::Boss(int damageValue)

{

cout<<"Boss Constructor Called !"<<endl;

setDamage(damageValue);

}

Boss::~Boss()

{

cout<<"Boss Destructor Called !"<<endl;

}

void Boss::setDamage(int damageValue)

{

damage = damageValue;

}

int Boss::getDamage()

{

return damage;

}

void Boss::displayAttack()

{

cout<<"Attack causes "<<getDamage()<<" damage"<<endl;

}

//----------------------------------------------------------

// Derived class implementation

//----------------------------------------------------------

BigBoss::BigBoss(int damage, int damageMultiplierValue): Boss(damage)

{

cout<<"Big Boss Constructor Called !"<<endl;

setDamageMultiplier(damageMultiplierValue);

}

BigBoss::~BigBoss()

{

cout<<"Big Boss Destructor Called !"<<endl;

}

void BigBoss::setDamageMultiplier(int dm)

{

damageMultiplier = dm;

}

int BigBoss::getDamageMultiplier()

{

return damageMultiplier;

}

void BigBoss::specialAttack()

{

cout<<"Special Attack causes "<<getDamageMultiplier() \* getDamage()<<" damage"<<endl;

}

//----------------------------------------------------------

// Filename: Main.cpp

// Main Function

//----------------------------------------------------------

#include "Boss.h"

int main()

{

cout<<"Boss: "<<endl;

Boss boss(25); //creating an object of the base class Boss

boss.displayAttack();

cout<<endl<<endl;

cout<<"Big Boss"<<endl;

BigBoss bb(50, 2); //creating an object of the derived class BigBoss

bb.displayAttack(); //boss

bb.specialAttack(); //bigboss

cout<<endl<<endl;

return 0;

}

**Constructor And Destructors in Derived Classes**

Instantiating a derived-class object begins a chain of constructor calls in which the derived-class constructor, before performing its own tasks, invokes its direct base class's constructor either explicitly (via a base-class member initializer) or implicitly (calling the base class's default constructor). Similarly, if the base class is derived from another class, the base-class constructor is required to invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in this chain is the constructor of the class at the base of the hierarchy, whose body actually finishes executing first. The original derived-class constructor's body finishes executing last. Each base-class constructor initializes the base-class data members that the derived-class object inherits.

When a derived-class object is destroyed, the program calls that object's destructor. This begins a chain (or cascade) of destructor calls in which the derived-class destructor and the destructors of the direct and indirect base classes and the classes' members execute in reverse of the order in which the constructors executed. When a derived-class object's destructor is called, the destructor performs its task, then invokes the destructor of the next base class up the hierarchy. This process repeats until the destructor of the final base class at the top of the hierarchy is called. Then the object is removed from memory.

Base-class constructors, destructors and overloaded assignment operators are not inherited by derived classes. Derived-class constructors, destructors and overloaded assignment operators, however, can call base-class constructors, destructors and overloaded assignment operators.

*Refer to Pages 557 - 564 in the textbook for the program*

**public, protected and private Inheritance**

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. Use of protected and private inheritance is rare, and each should be used only with great care; we normally use public inheritance in this book.

If you omit the access specification for a base class, the compiler assumes that it’s private.

When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

// Inherit from Base publicly

class Pub: public Base

{

};

When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

// Inherit from Base protectedly

class Pro: protected Base

{

};

When deriving from a private base class, public and protected members of the base class become private members (e.g., the functions become utility functions) of the derived class.

// Inherit from Base privately

class Pri: private Base

{

};

class Def: Base // Defaults to private inheritance

{

};

Private and protected inheritance are not is-a relationships.



**Class Assignment:**

Base Class:

class Creature

{

public:

Creature(string newName);

void setName(string name);

string getName();

void setHealth(short health);

short getHealth();

void setDamage(short damage);

short getDamage();

void attack();

protected:

string creatureName;

short creatureHealth;

short creatureDamage;

};

Derived Class:

class Monster

{

public:

Monster(string name);

private:

static const short claws = 2;

};

Creature Class Instructions

Constructor:

1. In the body of the constructor, create a short variable named damage and store a random number between 1 and 5 (inclusive).
2. Initialize the name of the creature by calling the setHealth function.
3. Initialize the health of the creature to 100 by calling the setHealth function.
4. Initialize the damage inflicted by the creature by calling the setDamage function and passing the random damage generated as its argument.

Setter And Getter Functions

1. Use appropriate code for these functions
2. Ensure that the functions have valid return types, class names and parameters

Attack Function

Print the name,health and the total damage of the object using appropriate messages.

Monster Class Instructions

Constructor:

1. Call the base class constructor to initialize the member variable for the Monster class
2. In the body of the constructor, create 2 short variables named damage and totalDamage.
3. Store the actual damage the Monster can inflict in the variable damage.
4. The variable totalDamage will store the actual damage plus the damage inflicted by the claws.
5. Call the setDamage function and pass the variable totalDamage as its argument.
6. Print the actual damage and the total damage using appropriate messages

Main

1. Create an object of the type Creature class.
2. Call the attack function using Creature object.
3. Create an object of the type Monster class.
4. Call the attack function using Monster object.

Exercises:

Fill in the blanks in each of the following statements:

1. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ is a form of software reuse in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.
2. A base class's \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ members can be accessed only in the base-class definition or in derived-class definitions.
3. In a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ relationship, an object of a derived class also can be treated as an object of its base class.
4. In a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_relationship, a class object has one or more objects of other classes as members.
5. A base class's \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ members are accessible within that base class and anywhere that the program has a handle to an object of that base class or to an object of one of its derived classes.
6. A base class's protected access members have a level of protection between those of public and \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ access.
7. C++ provides for \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, which allows a derived class to inherit from many base classes, even if these base classes are unrelated.
8. When an object of a derived class is instantiated, the base class's \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.
9. When deriving a class from a base class with public inheritance, public members of the base class become \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ members of the derived class, and protected members of the base class become \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ members of the derived class.
10. When deriving a class from a base class with protected inheritance, public members of the base class become \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ members of the derived class, and protected members of the base class become \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ members of the derived class.

State whether each of the following is true or false. If false, explain why.

1. Base-class constructors are not inherited by derived classes.
2. A has-a relationship is implemented via inheritance.
3. A Car class has an is-a relationship with the SteeringWheel and Brakes classes.
4. Inheritance encourages the reuse of proven high-quality software.
5. When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors.

Multiple Choice:

1. Select the false statement regarding inheritance.

1. A derived class can contain more attributes and behaviors than its base class.
2. A derived class can be the base class for other derived classes.
3. Some derived classes can have multiple base classes.
4. Base classes are usually more specific than derived classes.

2. Which of the following is not a kind of inheritance in C++?

1. public.
2. private.
3. static.
4. protected.

3. The is-a relationship represents.

1. Composition.
2. Inheritance.
3. Information Hiding.
4. A friend.

4. Which of the following is most likely a base class of the other three?

1. automobile.
2. convertible.
3. miniVan.
4. sedan.

5. Which of the following is not a good example of a hierarchy likely to be modeled by inheritance?

1. Airplanes.
2. Geometric shapes.
3. Animals.
4. Prime numbers.

6. To declare class subClass a privately derived class of superClass one would write:

1. class subclass : private superClass
2. class subclass :: private superClass
3. class subclass < private superClass >
4. class subclass inherits private superClass

7. From most restrictive to least restrictive, the access modifiers are:

1. protected, private, public
2. private, protected, public
3. private, public, protected
4. protected, public, private

8. Assuming the following is the beginning of the constructor definition for class BasePlus-CommissionEmployee which inherits from class Point,

BasePlusCommissionEmployee::BasePlusCommissionEmployee( string first,   
 string last, string ssn, double sales, double rate, double salary )  
 : CommissionEmployee( first, last, ssn, sales, rate )

The second line:

1. Invokes the CommissionEmployee constructor with arguments.
2. Causes a compiler error.
3. Is unnecessary because the CommissionEmployee constructor is called automatically.
4. Indicates inheritance.

9. Which of the following is not one of the disadvantages of using the “copy-and-paste” approach to duplicating code from one class into another class?

1. Errors are prone to be spread around.
2. It is time consuming.
3. It forces the system to store many physical copies of the code, creating a code-maintenance nightmare.
4. All of the above are disadvantages of the “copy-and-paste” approach.

10. When should base class members be declared protected?

1. When all clients should be able to access these members.
2. When these members are used only by member functions of this base class.
3. When these members should be available only to derived classes (and friends), but not to other clients.
4. The protected access specified should never be used.

11. When an object of a derived class is instantiated, the \_\_\_\_\_\_\_\_\_\_ constructor initializes the \_\_\_\_\_\_\_\_\_ members.

1. Base class, base class.
2. Derived class, base class.
3. Base class, derived class.
4. Derived class, public.

12. Suppose class A inherits from base class B. What is the order in which their constructors and destructors will be called when an object of class A is instantiated and then destroyed?

1. B constructor, A constructor, A destructor, B destructor.
2. B constructor, A constructor, B destructor, A destructor.
3. A constructor, B constructor, A destructor, B destructor.
4. A constructor, B constructor, B destructor, A destructor.

13. Which forms of inheritance are is-a relationships?

1. All forms of inheritance are is-a relationships.
2. Only public and private.
3. Only public and protected.
4. Only public.

14. When deriving a class from a protected base class, the public members of the base class become \_\_\_\_\_\_\_\_\_ and the protected members of the base class become \_\_\_\_\_\_\_\_\_\_?

1. protected, private
2. public, private
3. protected, protected
4. public, protected

**Lesson 11: Object-Oriented Programming: Polymorphism**

**Polymorphism**

Polymorphism enables us to "program in the general" rather than "program in the specific." In particular, polymorphism enables us to write programs that process objects of classes that are part of the same class hierarchy as if they are all objects of the hierarchy's base class.

Consider the following example of polymorphism:

Example: Animal hierarchy

* Base class: Animal
  + every derived class has function move
* Different animal objects maintained as a vector of Animal pointers
* Program issues same message (move) to each animal generically
* Proper function gets called
  + A Fish will move by swimming
  + A Frog will move by jumping
  + A Bird will move by flying

With polymorphism, one function can cause different actions to occur, depending on the type of the object on which the function is invoked. This gives the programmer tremendous expressive capability.

The polymorphism occurs when a program invokes a virtual function through a base-class (i.e., Animal) pointer or reference C++ dynamically (i.e., at execution time) chooses the correct function for the class from which the object was instantiated.

An object of a derived class can be treated as an object of its base class. This enables various interesting manipulations. Despite the fact that the derived-class objects are of different types, the compiler allows this because each derived-class object is an object of its base class. However, we cannot treat a base-class object as an object of any of its derived classes. Treating a base-class object as a(n) derived-class object can cause errors.

Whenever you create a base class and a derived class with the same function name and compile it, the call to the base class function is set once and for all by the compiler as the version defined in the base class. The compiler will have no knowledge of any other derived class function with the same name exists. This is called **static resolution** of the function call since the function call is fixed before the program is executed. This is also sometimes called **early binding** because the particular base class function chosen is bound to the call during the compilation of the program.

The **dynamic linkage** or **late binding** resolves this issue and ensures that the correct function is called. Late binding refers to events that must occur at run time. A late bound function call is one in which the address of the function to be called is unknown until the program runs. A virtual function is a late bound object.

C++ supports polymorphism using virtual functions. A virtual function can be called just like any other member function. However, what makes a virtual function interesting is the capability of supporting run-time polymorphism. It signals to the compiler that you don’t want static linkage for this function. What you *do* want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called.

Virtual functions are declared by preceding the function’s prototype with the keyword virtual in base class. They enable derived classes override function as appropriate. Once declared virtual, a function remains virtual all the way down the hierarchy.

**Static binding**

When calling a virtual function using specific object with dot operator, function invocation resolved at compile time

**Dynamic binding**

Dynamic binding occurs only off pointer and reference handles

//----------------------------------------------------------------

// Animal Class. This is the Base class.

//----------------------------------------------------------------

class Animal

{

public:

Animal(const string& type);

virtual string move(); // Notice keyword virtual

string getAnimalType();

void setAnimalType(string type);

private:

string animalType;

};

//----------------------------------------------------------------

// Fish Class. This is the Derived class. This inherits from the

// base class Animal.

//----------------------------------------------------------------

class Fish : public Animal

{

public:

Fish(const string& type);

string move();//keyword virtual can also be used here but is optional, but should be used if its derived classes needs to override this funtion

};

//----------------------------------------------------------------

// Frog Class. This is the Derived class. This inherits from the

// base class Animal.

//----------------------------------------------------------------

class Frog : public Animal

{

public:

Frog(const string& type);

string move();

};

//----------------------------------------------------------------

// Bird Class. This is the Derived class. This inherits from the

// base class Animal.

//----------------------------------------------------------------

class Bird : public Animal

{

public:

Bird(const string& type);

string move();

};

//---------------------------------------------------------------------

// Implementation File

//---------------------------------------------------------------------

Animal::Animal(const string &type)

{

// Indirect initialization using the set function

setAnimalType(type);

}

void Animal::setAnimalType(const std::string type)

{

animalType = type; // sets the member variable with new value

}

string Animal::getAnimalType()

{

return animalType; // returns the value of the member variable

}

string Animal::move()

{

// returns the animal type along with a string

return getAnimalType() + " Walks";

}

//invokes the base class constructor Animal

Fish::Fish(const std::string &type): Animal(type)

{ }

string Fish::move()

{

// returns the animal type along with a string

return getAnimalType() + " Swims";

}

//invokes the base class constructor Animal

Frog::Frog(const string &type): Animal(type)

{}

string Frog::move()

{

// returns the animal type along with a string

return getAnimalType() + " Leaps";

}

//invokes the base class constructor Animal

Bird::Bird(const std::string &type) : Animal(type)

{}

string Bird::move()

{

// returns the animal type along with a string

return getAnimalType() + " Flies";

}

//---------------------------------------------------------------------

// Client File

//---------------------------------------------------------------------

int main()

{

// Instantiation of the Class object, Here we are creating 3 new // Animal objects dynamically of the type Fish, Bird, Frog. This // is possible because these are a type of Animal

Animal\* animals[] =

{

//dynamically allocates memory to 3 new objects

new Fish("Fish"),

new Bird("Bird"),

new Frog("Frog")

};

for(int i = 0; i < 3; i++)

{

cout << animals[i]->move() << endl;

delete animals[i];

}

return 0;

}

**Allowed Assignments between Base-Class and Derived-Class Objects and Pointers**

Although a derived-class object also is a base-class object, the two objects are nevertheless different. Although derived-class objects can be treated as if they are base-class objects. This is a logical relationship, because the derived class contains all the members of the base class. However, base-class objects cannot be treated as if they are derived-class objects, the derived class can have additional derived-class-only members. For this reason, aiming a derived-class pointer at a base-class object is not allowed without an explicit cast and such an assignment would leave the derived-class-only members undefined on the base-class object.

// Creating Base class object

Animal animalObj;

// Creating Base class pointer. Assigning the value

// 0 or NULL ensures that the pointer does not point

// to anything (garbage value)

Animal \*animalPtr = 0;

// Creating Derived class object

Bird birdObj;

//Creating Derived class pointer. Assigning the value

// 0 or NULL ensures that the pointer does not point

// to anything (garbage value)

Bird \*birdPtr = 0;

There are four ways to aim base-class pointers and derived-class pointers at base-class objects and derived-class objects:

1. Aiming a base-class pointer at a base-class object is straight forward. Calls made off the base-class pointer simply invoke base-class functionality.

// Pointing the base class pointer to base class object

animalPtr = &animalObj;

1. Aiming a derived-class pointer at a derived-class object is straight forward. Calls made off the derived-class pointer simply invoke derived-class functionality.

// Pointing the derived class pointer to derived class

// object

birdPtr = &birdObj;

1. Aiming a base-class pointer at a derived-class object is safe, because the derived-class object is an object of its base class. However, this pointer can be used to invoke only base-class member functions. If the programmer attempts to refer to a derived-class-only member through the base-class pointer, the compiler reports an error. To avoid this error, the programmer must cast the base-class pointer to a derived-class pointer. The derived-class pointer can then be used to invoke the derived-class object's complete functionality. However, this technique is called downcasting and is a potentially dangerous operation.

// Downcasting:

// Base class pointer aimed at derived class object

// Can invoke only base class functions

animalPtr = &birdObj;

1. Aiming a derived-class pointer at a base-class object generates a compilation error. The is-a relationship applies only from a derived class to its direct and indirect base classes, and not vice versa. A base-class object does not contain the derived-class-only members that can be invoked off a derived-class pointer.

// Derived class pointer aimed at base class object

birdPtr = &animalObj; // WILL THROW ERROR !!!

**Differences between overloading and virtual functions**

The redefinition of a virtual function inside a derived class might seem somewhat similar to function overloading. However, the two processes are distinctly different:

* An overloaded function must differ in type and/or number of parameters, while a redefined virtual function must have precisely the same type and number of parameters and the same return type. (In fact, if you change either the number or type of parameters when redefining a virtual function, it will simply become an overloaded function and its virtual nature is lost.)
* Virtual functions must be class members.
* While destructor functions can be virtual, constructors cannot.

Because of the differences between overloaded functions and redefined virtual functions, the term overriding is used to describe virtual function redefinition.

Virtual functions are hierarchical in order of inheritance.

When a derived class does not override a virtual function, the function defined within its base class is used.

**Abstract Classes and Pure virtual Functions**

Sometimes when a virtual function is declared in the base class there is no meaningful operation for it to perform. This situation is common because often a base class does not define a complete class by itself. It simply supplies a core set of member functions and variables to which the derived class supplies the remainder.

When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class must override this function. To ensure that this will occur, C++ supports pure virtual functions:

**virtual type func-name (parameter-list) = 0;**

The key part of this declaration is the setting of the function equal to 0.

When a virtual function is made pure, it forces any derived class to override it. Thus, making a virtual function pure is a way to guarantee that a derived class will provide its own redefinition.

A *pure virtual function* has a prototype, but no definition. Used when a default implementation does not make sense.

The "=0" is known as a pure specifier. Pure virtual functions do not provide implementations. Every concrete derived class must override all base-class pure virtual functions with concrete implementations of those functions. The difference between a virtual function and a pure virtual function is that a virtual function has an implementation and gives the derived class the option of overriding the function; by contrast, a pure virtual function does not provide an implementation and requires the derived class to override the function (for that derived class to be concrete; otherwise the derived class remains abstract). A class with a pure virtual function is called a pure virtual class and cannot be instantiated. (However, its subclasses can).

**Abstract Classes**

When a class contains ***at least one pure virtual function***, it is referred to as an abstract class. Since an abstract class contains at least one function for which no body exists, it is an incomplete type, and no object of that class can be created. Thus, abstract classes exist only to be inherited.

In many situations it is useful to define abstract classes for which the programmer never intends to create objects. Because these are used only as base classes, we refer to them as abstract base classes. No objects of an abstract class may be instantiated. Classes from which objects can be instantiated are called concrete classes.

//====================================================================

// Creature.h

// Abstract Class -- Cannot instantiate an object of this class

//====================================================================

#include <iostream>

using namespace std;

class Creature

{

public:

Creature(int health = 100); // Default constructor

void setHealth(int health);

int getHealth();

virtual void displayHealth();

//abstract function -- does not contain any code

// Notice that it is assigned a value of 0

virtual void greet() = 0;

private:

int health;

};

//====================================================================

// Griffin.h

// Concrete class -- instantiation of objects is possible, i.e,

// An object of this class can be created

//====================================================================

class Griffin : public Creature

{

public:

Griffin (int health = 200);

//concrete function -- contains code

virtual void greet();

};

//====================================================================

// Creature.cpp

//====================================================================

#include "Creature.h"

Creature::Creature(int health)

{

setHealth(health);

}

void Creature::setHealth(int health)

{

this->health = health;

}

int Creature::getHealth()

{

return health;

}

void Creature::displayHealth()

{

cout<<"Health : "<<getHealth()<<endl;

}

//====================================================================

// Griffin.cpp

//====================================================================

#include "Griffin.h"

//calling the base class constructor

Griffin ::Griffin (int health):Creature(health)

{}

void Griffin ::greet()

{

cout<<"Hello, I am the Griffin!!!"<<endl;

}

//====================================================================

// Main.cpp

//====================================================================

#include "Creature.h"

#include "Griffin.h"

int main()

{

Griffin \*opinicus = new Griffin ();

opinicus->greet();

opinicus->displayHealth();

cout<<endl;

Creature \*pCreature = new Griffin();

cout<<"Displaying the greeting of the creature: "<<endl;

pCreature->greet();

cout<<endl;

pCreature->displayHealth();

return 0;

}

**Virtual Destructors**

A problem can occur when using polymorphism to process dynamically allocated objects of a class hierarchy. So far you have seen nonvirtual destructors destructors that are not declared with keyword virtual. If a derived-class object with a nonvirtual destructor is destroyed explicitly by applying the delete operator to a base-class pointer to the object, the C++ standard specifies that the behavior is undefined.

The simple solution to this problem is to create a virtual destructor (i.e., a destructor that is declared with keyword virtual) in the base class. This makes all derived-class destructors virtual even though they do not have the same name as the base-class destructor. Now, if an object in the hierarchy is destroyed explicitly by applying the delete operator to a base-class pointer, the destructor for the appropriate class is called based on the object to which the base-class pointer points. Remember, when a derived-class object is destroyed, the base-class part of the derived-class object is also destroyed, so it is important for the destructors of both the derived class and base class to execute. The base-class destructor automatically executes after the derived-class destructor.

//====================================================================

// This program shows the working of the virtual destructor

//====================================================================

class Boss

{

public:

Boss(int damage = 10);

// constructors cannot be virtual only destructors can be

// virtual used deallocate any memory reserved when calling

// calling the functions from the base class

virtual ~Boss();

void setDamage(int att);

int getDamage();

virtual void displayAttack();

private:

int \*damagePtr;

};

class BigBoss: public Boss

{

public:

BigBoss(int multiplier = 3);

virtual ~BigBoss();

void setDamageMultiplier(int dm);

int getDamageMultiplier();

virtual void displayAttack();

private:

int \*damageMultiplierPtr;

};

//====================================================================

// Implementation

//====================================================================

#include "Boss.h"

Boss::Boss(int damage)

{

cout<<"Boss Constructor called"<<endl;

setDamage(damage);

}

Boss::~Boss()

{

cout<<"Boss Destructor: Deleting the damagePtr"<<endl;

delete damagePtr;

damagePtr = NULL;

}

void Boss::setDamage(int att)

{

damagePtr = new int(att);

}

int Boss::getDamage()

{

return \*damagePtr;

}

void Boss::displayAttack()

{

cout<<"Attack causes "<<getDamage()

<<" damage points"<<endl<<endl;

}

BigBoss::BigBoss(int multiplier)

{

cout<<"Big Boss Constructor called"<<endl<<endl;

setDamageMultiplier(multiplier);

}

BigBoss::~BigBoss()

{

cout << "BigBoss Destructor: Deleting the damageMultiplierPtr"

<< endl;

delete damageMultiplierPtr;

damageMultiplierPtr = NULL;

}

void BigBoss::setDamageMultiplier(int dm)

{

damageMultiplierPtr = new int(dm);

}

int BigBoss::getDamageMultiplier()

{

return \*damageMultiplierPtr;

}

void BigBoss::displayAttack()

{

cout << "Special Attack causes "

<< getDamageMultiplier() \* getDamage()

<<" damage points"<<endl<<endl;

}

//====================================================================

// Client

//====================================================================

int main()

{

// Dynamically creates a pointer

Boss \*pBadGuy = new BigBoss();

pBadGuy->displayAttack();

// Deleting the pointer

delete pBadGuy;

pBadGuy = NULL;

return 0;

}

Exercises:

Fill in the blanks in each of the following statements:

1. Treating a base-class object as a(n) \_\_\_\_\_\_\_\_ can cause errors.
2. If a class contains at least one pure virtual function, it is a(n) \_\_\_\_\_\_\_\_\_\_ class.
3. Classes from which objects can be instantiated are called \_\_\_\_\_\_\_\_ classes.
4. \_\_\_\_\_\_\_\_\_ involves using a base-class pointer or reference to invoke virtual functions on base-class and derived-class objects.
5. Overridable functions are declared using keyword \_\_\_\_\_\_\_\_\_ .
6. Casting a base-class pointer to a derived-class pointer is called \_\_\_\_\_\_\_\_\_ .

State whether each of the following is true or false. If false, explain why.

1. All virtual functions in an abstract base class must be declared as pure virtual functions.
2. Referring to a derived-class object with a base-class handle is dangerous.
3. A class is made abstract by declaring that class virtual.
4. If a base class declares a pure virtual function, a derived class must implement that function to become a concrete class.

Multiple Choices:

1. Polymorphism is implemented via:

1. Member functions.
2. virtual functions and dynamic binding.
3. inline functions.
4. Non-virtual functions.

2. Which of the following would not be a member function that derived classes Fish, Frog and Bird should inherit from base class Animal and then provide their own definitions for, so that the function call can be performed polymorphically?

1. eat
2. sleep
3. move
4. flapWings

3. Employee is a base class and HourlyWorker is a derived class, with a redefined non-virtual print function. Given the following statements, will the output of the two print function calls be identical?

HourlyWorker h;

Employee \*ePtr = &h;

ePtr->print();

ePtr->Employee::print();

1. Yes.
2. Yes, if print is a static function.
3. No.
4. It would depend on the implementation of the print function.

4. Which of the following assignments would be a compilation error?

1. Assigning the address of a base-class object to a base-class pointer.
2. Assigning the address of a base-class object to a derived-class pointer.
3. Assigning the address of a derived-class object to a base-class pointer.
4. Assigning the address of a derived-class object to a derived-class pointer.

5.If objects of all the classes derived from the same base class all need to draw themselves, the draw() function would most likely be declared:

1. private
2. virtual
3. protected
4. friend

6. virtual functions must:

1. Be overridden in every derived class.
2. Be declared virtualin every derived class.
3. Be declared virtual in the base class.
4. Have the same implementation in every derived class.

7. Which of the following statements about virtual functions is false?

1. They allow the program to select the correct implementation at execution time.
2. They can use either static or dynamic binding, depending on the handles on which the functions are called.
3. They do not remain virtual down the inheritance hierarchy.
4. They can be called using the dot operator.

8. The line:

virtual double earnings() const = 0;  
appears in a class definition. You cannot deduce that:

1. All classes that directly inherit from this class will override this method.
2. This class is an abstract class.
3. Any concrete class derived from this class will have an earnings function.
4. This class will probably be used as a base class for other classes.

9. Abstract classes:

1. Contain at most one pure virtual function.
2. Can have objects instantiated from them if the proper permissions are set.
3. Cannot have abstract derived classes.
4. Are defined, but the programmer never intends to instantiate any objects from them.

10. The main difference between a pure virtual function and a virtual function is:

1. The return type.
2. The member access specifier.
3. That a pure virtual function cannot have an implementation.
4. The location in the class.

11. Which of the following is not allowed?

1. Objects of abstract classes.
2. Multiple pure virtual functions in a single abstract class.
3. References to abstract classes.
4. Arrays of pointers to abstract classes.

12. What mistake prevents the following class declaration from functioning properly as an abstract class?

class Shape

{

public:

virtual double print() const;

double area() const { return base \* height; }

private:

double base;

double height;

};

1. There are no pure virtual functions.
2. There is a non-virtual function.
3. private variables are being accessed by a public function.
4. Nothing, it functions fine as an abstract class.

13. virtual destructors must be used when:

1. The constructor in the base class is virtual.
2. delete is used on a base-class pointer to a derived-class object.
3. delete is used on a derived-class object.
4. A constructor in either the base class or derived class is virtual.

**Project 3 Blackjack POINTS 10**

**Complete the project as per the instructions provided by your instructor.**

**Lesson 12: Exception Handling**

**Exception Handling**

An **exception** is an indication of a problem that occurs during a program's execution. In many cases, handling an exception allows a program to continue executing as if no problem had been encountered. A more severe problem could prevent a program from continuing normal execution, instead of requiring the program to notify the user of the problem before terminating in a controlled manner. The features presented in this chapter enable programmers to write **robust** and **fault-tolerant programs** that are able to deal with problems that may arise and continue executing or terminate gracefully.

If the potential problems occur infrequently, intermixing program logic and error-handling logic can degrade a program's performance, because the program must (potentially frequently) perform tests to determine whether the task executed correctly and the next task can be performed.

**Handling an Attempt to Divide by Zero**

*Refer to Page 685 for the pseudocode and Pages 686 – 688 in the textbook for the program*

**Try- Catch Blocks**

**try blocks:**

Keyword try followed by braces ({}). Try blocks should enclose the

1. Statements that might cause exceptions and
2. Statements that should be skipped in case of an exception

**catch handlers:**

The catch handlers should immediately follow a try block. There can be one or more catch handlers for each try block. The catch block requires an exception parameter which should be enclosed in parentheses. This parameter represents the type of exception to process. If an exception parameter includes an optional parameter name, the catch handler can use that parameter name to interact with a caught exception object in the body of the catch handler, which is delimited by braces ({ and }). A catch handler typically reports the error to the user, logs it to a file, terminates the program gracefully or tries an alternate strategy to accomplish the failed task.

**Note:** It is a syntax error to place code between a try block and its corresponding catch handlers. Each catch handler can have only a single parameter—specifying a comma-separated list of exception parameters is a syntax error.

If an exception occurs as the result of a statement in a try block, the try block terminates immediately. Next, the program searches for the first catch handler that can process the type of exception that occurred. The program locates the matching catch by comparing the thrown exception's type to each catch's exception-parameter type until the program finds a match. A match occurs if the types are identical or if the thrown exception's type is a derived class of the exception-parameter type. When a match occurs, the code contained within the matching catch handler executes. When a catch handler finishes processing by reaching its closing right brace (}), the exception is considered handled and the local variables defined within the catch handler (including the catch parameter) go out of scope. Program control does not return to the point at which the exception occurred (known as the **throw point**), because the try block has expired. Rather, control resumes with the first statement after the last catch handler following the try block. This is known as the **termination model of exception handling**.

If the try block completes its execution successfully (i.e., no exceptions occur in the try block), then the program ignores the catch handlers and program control continues with the first statement after the last catch following that try block. If no exceptions occur in a try block, the program ignores the catch handler(s) for that block.

If an exception that occurs in a try block has no matching catch handler, or if an exception occurs in a statement that is not in a try block, the function that contains the statement terminates immediately, and the program attempts to locate an enclosing try block in the calling function. This process is called **stack unwinding.**

**Keyword throw**

Use keyword **throw** followed by an operand to represent the type of exception being thrown. The operand of a throw can be of any type. If the operand is an object, it is called an **exception object.** However, a throw operand also can assume other values, such as the value of an expression (e.g., throw x > 5) or the value of an int (e.g., throw 5). The throw operand initializes the exception parameter in the matching catch handler, if one is found.

**Note**: Use caution when throwing the result of a conditional expression (?:), because promotion rules could cause the value to be of a type different from the one expected. For example, when throwing an int or a double from the same conditional expression, the conditional expression converts the int to a double. However, the catch handler always catches the result as a double, rather than catching the result as a double when a double is thrown, and catching the result as an int when an int is thrown.

Catching an exception object by reference eliminates the overhead of copying the object that represents the thrown exception. Associating each type of runtime error with an appropriately named exception object improves program clarity

**When to Use Exception Handling**

Exception handling is designed to process **synchronous errors**, which occur when a statement executes. Common examples of these errors are out-of-range array subscripts, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid function parameters and unsuccessful memory allocation (due to lack of memory). Exception handling is not designed to process errors associated with **asynchronous** events (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program's flow of control.

Incorporate your exception-handling strategy into your system from the design process's inception. Including effective exception handling after a system has been implemented can be difficult.

The exception-handling mechanism also is useful for processing problems that occur when a program interacts with software elements, such as member functions, constructors, destructors and classes. Rather than handling problems internally, such software elements often use exceptions to notify programs when problems occur. This enables programmers to implement customized error handling for each application. Functions with common error conditions should return 0 or NULL (or other appropriate values) rather than throw exceptions. A program calling such a function can check the return value to determine success or failure of the function call.

**Rethrowing an Exception**

It is possible that an exception handler, upon receiving an exception, might decide either that it cannot process that exception or that it can process the exception only partially. In such cases, the exception handler can defer the exception handling (or perhaps a portion of it) to another exception handler. In either case, the handler achieves this by **rethrowing the exception** via the statement

throw;

Regardless of whether a handler can process (even partially) an exception, the handler can rethrow the exception for further processing outside the handler. The next enclosing try block detects the rethrown exception, which a catch handler listed after that enclosing try block attempts to handle. Executing an empty throw statement that is situated outside a catch handler causes a call to function **terminate**, which abandons exception processing and terminates the program immediately.

//=================================================================

// Main.cpp

// Check if the user has chosen a valid option. Code executes

// without errors if valid option is chosen. Otherwise we rethrow

// an exception from the global function and the main handles the

// exception instead the global function where an exception occurs

//=================================================================

#include <iostream>

#include <iomanip>

#include <string>

#include <exception>

using namespace std;

void displayMenu()

{

int userChoice;

string gameArray[5] = {" ","Playing Tic Tac Toe", "Playing Hangman", "Blank Screen", "Exiting"};

cout<<endl<<endl;

cout<<setw(48)<<"========================================"<<endl;

cout<<setw(48)<<"| MENU |"<<endl;

cout<<setw(48)<<"========================================"<<endl;

cout<<setw(48)<<"| |"<<endl;

cout<<setw(48)<<"| 1. TicTacToe Game |"<<endl;

cout<<setw(48)<<"| |"<<endl;

cout<<setw(48)<<"| 2. Hangman Game |"<<endl;

cout<<setw(48)<<"| |"<<endl;

cout<<setw(48)<<"| 3. Do Nothing (stare at a blank screen) |" << endl;

cout<<setw(48)<<"| |"<<endl;

cout<<setw(48)<<"| 4. Exit |"<<endl;

cout<<setw(48)<<"| |"<<endl;

cout<<setw(48)<<"========================================"<<endl;

cout<<endl;

cout<<"Please choose an option: ";

cin >> userChoice;

try

{

if(userChoice<=0 || userChoice>=5)

{

throw exception(); //throw an exception

}

//If an exception occurs the following statement

// will not be executed

cout<<gameArray[userChoice]<<endl;

}

catch (exception &e)

{

cout<<"Rethrowing exception"<<endl;

throw; //rethrowing an exception instead of handling it

}

}

int main()

{

try

{

displayMenu(); //This function throws an exception

}

catch (exception &e)

{

cerr<<"This exception is handled in Main"<<endl;

cerr<<"Invalid Number"<<endl;

cerr<<"Please choose an appropriate option"<<endl;

cerr<<"Exception handled in main: "<<e.what()<<endl;

}

cout<<endl<<endl;

return 0;

}

**Stack Unwinding**

When an exception is thrown but not caught in a particular scope, the function call stack is unwound, and an attempt is made to catch the exception in the next outer Try... catch block. Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function. If a Try block encloses that statement, an attempt is made to catch the exception. If a try block does not enclose that statement, stack unwinding occurs again. If no catch handler ever catches this exception, function terminate is called to terminate the program.

### Standard Library Exception Hierarchy

Exception hierarchy is headed by base-class exception (defined in header file <exception>), which contains virtual function what, which derived classes can override to issue appropriate error messages.

### Exception hierarchy classes

### Base-class exception

### Contains virtual function what for storing error messages

### Exception classes derived from exception

### bad\_alloc – thrown by new

### bad\_cast – thrown by dynamic\_cast

### bad\_typeid – thrown by typeid

### bad\_exception – thrown by unexpected

### Instead of terminating the program or calling the function specified by set\_unexpected

### Used only if bad\_exception is in the function’s throw list

### Class runtime\_error, derived from exception

### Indicates execution-time errors

### Exception classes derived from runtime\_error

### overflow\_error

### Indicates an arithmetic overflow error – an arithmetic result is larger than the largest storable number

### underflow\_error

### Indicates an arithmetic underflow error – an arithmetic result is smaller than the smallest storable number

### Class logic\_error, derived from exception

### Indicates errors in program logic

### Exception classes derived from logic\_error

### invalid\_argument

### Indicates an invalid argument to a function

### length\_error

### Indicates a length larger than the maximum size for some object was used

### out\_of\_range

### Indicates a value, such as an array subscript, exceeded its allowed range

Programmer-defined exception classes need not be derived from class exception. Thus, writing catch ( exception anyException ) is not guaranteed to catch all exceptions a program could encounter.

To catch all exceptions potentially thrown in a try block, use catch(...). One weakness with catching exceptions in this way is that the type of the caught exception is unknown at compile time. Another weakness is that, without a named parameter, there is no way to refer to the exception object inside the exception handler.

Exception Handling

1. Exception handling may allow a program to:

1. Terminate in a controlled manner.
2. Be more robust and fault-tolerant.
3. Continue executing as if no problem was encountered.
4. All of the above.

2. Which of the following is an advantage of not using exception-handling to deal with errors?

1. Intermixing program logic with error-handling logic can make the program more difficult to read, modify, maintain and debug.
2. Frequent tests for infrequently occurring errors can degrade a program’s performance.
3. Programmers may delay writing error-processing code or sometimes forget to include it.
4. None of the above.

3. The correct order in which an exception is detected and handled is:

1. try, catch, throw
2. throw, catch, try
3. catch, throw, try
4. try, throw, catch

4. Once an exception is thrown, when can control return to the throw point?

1. Never.
2. Only after the exception is caught.
3. Once the stack unwinding process is completed.
4. Immediately after the exception is thrown.

5. The try block cannot:

1. Enclose the code that may throw the exception.
2. Enclose its own catch blocks.
3. Test enclosing try blocks for additional catch statements if this try block’s catch statements can’t match the exception being thrown.
4. Have exceptions explicitly or implicitly thrown in the try block itself.

6. catch blocks are not required to contain:

1. Braces { }.
2. Parentheses ( ).
3. Some form of parameter type indication.
4. A parameter name.

7. An exception:

1. Terminates program execution.
2. Terminates the block where the exception occurred.
3. Will terminate the block where the exception occurred unless a catch command stops it.
4. Will not terminate a block unless explicitly instructed to do so.

8. Exception handling should not be used:

1. As an alternative for program control.
2. To make error handling uniform on large projects.
3. To deal with errors that do not arise very often.
4. To deal with errors for components that will be widely used in other applications, such as classes and libraries.

9. To rethrow an exception, the exception handler must:

1. Use the throw; statement.
2. Use the throw command with the same parameters as the original exception.
3. Return a reference to whatever caused the original exception.
4. Not have attempted to process that exception at all.

10. The proper syntax for a throw list is:

1. int g( double h )  
    throw ( a, b, c )
2. int g( double h )  
    throw ( a b c )
3. int g( double h )  
    throw ( a )  
    throw ( b )  
    throw ( c )
4. int g( double h )  
    throw ( a ),  
    throw ( b ),  
    throw ( c )

11. Placing throw() after a function's parameter list:

1. Guarantees that all exceptions can be thrown in this function.
2. Guarantees that only programmer-defined exceptions can be thrown in this function.
3. Indicates that throwing an exception in this function would call unexpected.
4. Indicates that the compiler will issue an error if the function contains a throw expression.

12. The purpose of stack unwinding is to:

1. Attempt to catch exceptions that are not caught in their scope.
2. Improve catchblocks by allowing them to handle multiple exceptions.
3. Return control to the function that created the exception.
4. Aid the terminate command in shutting down the program.

13. Select the false statement regarding exceptions.

1. The C++ standard has a hierarchy of exception classes.
2. All exception classes are accessible via <exception>.
3. Several classes derive from class exception.
4. The what function can be overridden in each class derived from exception.

14. Which class indicates that an error occurred in which an arithmetic result was larger than the largest number that can be stored in the computer?

1. invalid\_argument.
2. bad\_exception.
3. out\_of\_range.
4. overflow\_error.

15. Both “ignoring the exception” and “aborting the program” are error-handling techniques that:

1. Cannot be used if the error is fatal.
2. Always result in a resource leak.
3. Should not be used for mission-critical applications.
4. Allow program execution to proceed as if no error had occurred.

**Lesson 13: File Processing**

Storage of data in variables and arrays is temporary. **Files** are used for **data persistence** permanent retention of large amounts of data. Computers store files on **secondary storage devices**, such as magnetic disks, optical disks and tapes.

The Data Hierarchy

Ultimately, all data items that digital computers process are reduced to combinations of zeros and ones. This occurs because it is simple and economical to build electronic devices that can assume two stable states one state represents 0 and the other represents 1. It is remarkable that the impressive functions performed by computers involve only the most fundamental manipulations of 0s and 1s.

The smallest data item that computers support is called a **bit** (short for "**binary digit** "a digit that can assume one of two values). Each data item, or bit, can assume either the value 0 or the value 1. Computer circuitry performs various simple bit manipulations, such as examining the value of a bit, setting the value of a bit and reversing a bit (from 1 to 0 or from 0 to 1).

Programming with data in the low-level form of bits is cumbersome. It is preferable to program with data in forms such as **decimal digits** (i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9), letters (i.e., A through Z and a through z) and **special symbols** (i.e., $, @, %, &, \*, (, ), -, +, ", :, ?, / and many others). Digits, letters and special symbols are referred to as **characters**. The set of all characters used to write programs and represent data items on a particular computer is called that computer's **character set**. Because computers can process only 1s and 0s, every character in a computer's character set is represented as a pattern of 1s and 0s. **Bytes** are composed of eight bits. Programmers create programs and data items with characters; computers manipulate and process these characters as patterns of bits. For example, C++ provides data type char. Each char occupies one byte of memory. C++ also provides data type wchar\_t, which can occupy more than one byte (to support larger character sets, such as the Unicode® character set). For more information on Unicode®, visit [www.unicode.org](http://www.unicode.org).

When data needs to be archived or stored in nonvolatile memory areas such as a hard disk, programmers look to data files as a viable answer for storing and retrieving data after the computer’s power has been turned off.

Data files are often text-based and are used for storing and retrieving related information like that stored in a database. Managing the information contained in data files is up to the programmer.

Each file ends either with an **end-of-file marker** or at a specific byte number recorded in a system-maintained, administrative data structure. When a file is opened, an object is created, and a stream is associated with the object.

Just as characters are composed of bits, **fields** are composed of characters. A field is a group of characters that conveys some meaning. For example, a field consisting of uppercase and lowercase letters can represent a person's name.

Data items processed by computers form a **data hierarchy** ([Fig. 17.1](mk:@MSITStore:C:\Documents%20and%20Settings\sowmyag\Desktop\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\Book\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch17lev1sec2.html#ch17fig01)), in which data items become larger and more complex in structure as we progress from bits, to characters, to fields and to larger data aggregates.

Files and Streams

C++ views each file as a sequence of bytes ([Fig. 17.2](mk:@MSITStore:C:\Documents%20and%20Settings\sowmyag\Desktop\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\Book\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch17lev1sec3.html#ch17fig02)). Each file ends either with an **end-of-file marker** or at a specific byte number recorded in a system-maintained, administrative data structure. When a file is opened, an object is created, and a stream is associated with the object.

To perform file processing in C++, header files <iostream> and <fstream> must be included.

Creating a Sequential File

C++ imposes no structure on a file. Thus, a concept like that of a "record" does not exist in a C++ file. Therefore, the programmer must structure files to meet the application's requirements. In the following example, we see how the programmer can impose a simple record structure on a file.

[Figure 17.4](mk:@MSITStore:C:\\Documents%20and%20Settings\\sowmyag\\Desktop\\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\\Book\\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch17lev1sec4.html" \l "ch17fig04) creates a sequential file that might be used in an accounts-receivable system to help manage the money owed by a company's credit clients. For each client, the program obtains the client's account number, name and balance (i.e., the amount the client owes the company for goods and services received in the past). The data obtained for each client constitutes a record for that client. The account number serves as the record key in this application; that is, the program creates and maintains the file in account number order. This program assumes the user enters the records in account number order. In a comprehensive accounts receivable system, a sorting capability would be provided for the user to enter records in any order the records then would be sorted and written to the file.

*Refer to Pages 718 in the textbook for the program*

| Figure 17.5. File open modes. | |
| --- | --- |
| **Mode** | **Description** |
| ios::app | Append all output to the end of the file. |
| ios::ate | Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file. |
| ios::in | Open a file for input. |
| ios::out | Open a file for output. |
| ios::trunc | Discard the file's contents if they exist (this also is the default action for ios::out). |
| ios::binary | Open a file for binary (i.e., nontext) input or output. |

An ofstream object can be created without opening a specific files, file can be attached to the object later. For example, the statement

ofstream outClientFile;

creates an ofstream object named outClientFile. The ofstream member function **open** opens a file and attaches it to an existing ofstream object as follows:

outClientFile.open( "clients.dat", ios::out );

NOTE: Not opening a file before attempting to reference it in a program will result in an error.

NOTE: Closing files explicitly when the program no longer needs to reference them can reduce resource usage (especially if the program continues execution after closing the files).

Reading Data from a Sequential File

Files store data so it may be retrieved for processing when needed. The previous section demonstrated how to create a file for sequential access. In this section, we discuss how to read data sequentially from a file.

*Refer to Pages 721*, *723 – 726* *in the textbook for the program*

Some examples of positioning the "get" file-position pointer are:

// position to the nth byte of fileObject (assumes ios::beg)

fileObject.seekg( n );

// position n bytes forward in fileObject

fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject

fileObject.seekg( n, ios::end );

// position at end of fileObject

fileObject.seekg( 0, ios::end );

The same operations can be performed using ostream member function seekp. Member functions **tellg** and **tellp** are provided to return the current locations of the "get" and "put" pointers, respectively. The following statement assigns the "get" file-position pointer value to variable location of type long:

location = fileObject.tellg();

### Random-Access Files

So far, we have seen how to create sequential files and search them to locate information. Sequential files are inappropriate for **instant-access applications**, in which a particular record must be located immediately. Common instant-access applications are airline reservation systems, banking systems, point-of-sale systems, automated teller machines and other kinds of **transaction-processing systems** that require rapid access to specific data. A bank might have hundreds of thousands (or even millions) of other customers, yet, when a customer uses an automated teller machine, the program checks that customer's account in a few seconds or less for sufficient funds. This kind of instant access is made possible with **random-access files**. Individual records of a random-access file can be accessed directly (and quickly) without having to search other records.

As we have said, C++ does not impose structure on a file. So the application that wants to use random-access files must create them. A variety of techniques can be used. Perhaps the easiest method is to require that all records in a file be of the same fixed length. Using same-size, fixed-length records makes it easy for a program to calculate (as a function of the record size and the record key) the exact location of any record relative to the beginning of the file. We soon will see how this facilitates immediate access to specific records, even in large files.

[Figure 17.9](mk:@MSITStore:C:\\Documents%20and%20Settings\\sowmyag\\Desktop\\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\\Book\\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch17lev1sec7.html" \l "ch17fig09) illustrates C++'s view of a random-access file composed of fixed-length records (each record, in this case, is 100 bytes long). A random-access file is like a railroad train with many same-size carssome empty and some with contents.

Data can be inserted into a random-access file without destroying other data in the file. Data stored previously also can be updated or deleted without rewriting the entire file. In the following sections, we explain how to create a random-access file, enter data into the file, read the data both sequentially and randomly, update the data and delete data that is no longer needed.

### Creating a Random-Access File

*Refer to Pages 730 - 733 in the textbook for the program*

### Writing Data Randomly to a Random-Access File

*Refer to Pages 733 in the textbook for the program*

### Reading from a Random-Access File Sequentially

*Refer to Pages 735 - 736 in the textbook for the program*

Exercises

Fill in the blanks in each of the following:

1. Ultimately, all data items processed by a computer are reduced to combinations of\_\_\_\_\_\_\_\_\_\_and\_\_\_\_\_\_\_\_\_\_.
2. The smallest data item a computer can process is called a\_\_\_\_\_\_\_\_\_\_.
3. A(n)\_\_\_\_\_\_\_\_\_\_is a group of related records.
4. Digits, letters and special symbols are referred to as\_\_\_\_\_\_\_\_\_\_.
5. A group of related files is called a(n)\_\_\_\_\_\_\_\_\_\_.
6. Member function\_\_\_\_\_\_\_\_\_\_of the file streams fstream, ifstream and ofstream closes a file.
7. The istream member function\_\_\_\_\_\_\_\_\_\_reads a character from the specified stream.
8. Member function\_\_\_\_\_\_\_\_\_\_of the file streams fstream, ifstream and ofstream opens a file.
9. The istream member function\_\_\_\_\_\_\_\_\_\_is normally used when reading data from a file in random-access applications.
10. Member functions\_\_\_\_\_\_\_\_\_\_and\_\_\_\_\_\_\_\_\_\_of istream and ostream, set the file-position pointer to a specific location in an input or output stream, respectively.

State which of the following are true and which are false. If false, explain why.

1. Member function read cannot be used to read data from the input object cin.
2. The programmer must create the cin, cout, cerr and clog objects explicitly.
3. A program must call function close explicitly to close a file associated with an ifstream, ofstream or fstream object.
4. If the file-position pointer points to a location in a sequential file other than the beginning of the file, the file must be closed and reopened to read from the beginning of the file.
5. The ostream member function write can write to standard-output stream cout.
6. Data in sequential files always is updated without overwriting nearby data.
7. Searching all records in a random-access file to find a specific record is unnecessary.
8. Records in random-access files must be of uniform length.
9. Member functions seekp and seekg must seek relative to the beginning of a file.

Assume that each of the following statements applies to the same program.

1. Write a statement that opens file oldmast.dat for input; use an ifstream object called inOldMaster.
2. Write a statement that opens file trans.dat for input; use an ifstream object called inTransaction.
3. Write a statement that opens file newmast.dat for output (and creation); use ofstream object outNewMaster.
4. Write a statement that reads a record from the file oldmast.dat. The record consists of integer accountNumber, string name and floating-point currentBalance; use ifstream object inOldMaster.
5. Write a statement that reads a record from the file TRans.dat. The record consists of integer accountNum and floating-point dollarAmount; use ifstream object inTransaction.
6. Write a statement that writes a record to the file newmast.dat. The record consists of integer accountNum, string name, and floating-point currentBalance; use ofstream object outNewMaster.

Find the error(s) and show how to correct it (them) in each of the following.

1. File payables.dat referred to by ofstream object outPayable has not been opened.
2. outPayable << account << company << amount << endl;
3. The following statement should read a record from the file payables.dat. The ifstream object inPayable refers to this file, and istream object inReceivable refers to the file receivables.dat.
4. inReceivable >> account >> company >> amount;
5. The file tools.dat should be opened to add data to the file without discarding the current data.
6. ofstream outTools( "tools.dat", ios::out );

1.Select the false statement.

1. All data items can eventually be reduced to a sequence of 0s and 1s.
2. A binary digit (bit) can store two values simultaneously.
3. It is cumbersome to work with data in bit form.
4. It is easiest to produce electronic devices that can assume one of two stable states.

2. Which statement is not true about 0, z , %, \* and Q?

1. Each can be represented in terms of 0s and 1s.
2. Each is a character.
3. Only two of the elements listed are in typical character sets.
4. Each can be represented as a byte.

3. The data hierarchy, arranged from smallest to largest, is:

1. Bit, byte, record, field, file.
2. Byte, bit, field, record, file.
3. Bit, byte, field, record, file.
4. Byte, bit, field, file, record.

4. In general, which of the following contains the most amount of data?

1. A database.
2. A file.
3. A byte.
4. A field.

5. Which of the following does not have a stream associated with it?

1. cerr.
2. cin.
3. cout.
4. All of the above have streams associated with them.

6. In order to perform file processing in C++, which header files must be included?

1. <cstdio> , <iostream> and <fstream>.
2. <cstdio> and <iostream>.
3. <cstdio> and <fstream>.
4. <iostream> and <fstream>.

7. Which of the following is not true about files?

1. C++ views each file as a sequential stream of bytes.
2. Files are opened by creating objects of stream classes.
3. Member functions of stream objects can be applied to file streams.
4. istream, ostream and iostream are derived from ifstream, ofstream and fstream, respectively.

8. Which file open mode would be used to write data only to the end of an existing file?

1. ios::app
2. ios::in
3. ios::out
4. ios::trunc

9. When used with ofstream objects, operator! is not:

1. Overloaded.
2. Used to determine if the open operation succeeded.
3. Used to close a file explicitly.
4. Used to return a nonzero value if an error occurs.

10. Which of the following will not change the file-position pointer to the same position as the others? Assume a 10-byte file size and a current position at byte # 1.

1. fileObject.seekg( 2 );
2. fileObject.seekg( 1, ios::cur );
3. fileObject.seekg( 2, ios::beg );
4. fileObject.seekg( 8, ios::end );

11. Random access files are more effective than sequential files for:

1. Instant access to data.
2. Updating data easily.
3. Inserting data into the file without destroying other data.
4. All of the above.

12. A random access file is organized most like a(n):

1. Array.
2. Object.
3. Class.
4. Pointer.

13. Select the false statement. The write function:

1. Creates unformatted data.
2. Expects data type const char \* as its first argument.
3. Writes to files in hexadecimal format.
4. Takes an argument of type size\_t.

14. Select the false statement. The reinterpret\_cast operator:

1. Changes the value of the object to which its operand points.
2. Performs its operation at compile time.
3. Is compiler-dependent and can cause programs to behave differently on different platforms.
4. Is easy to use to perform dangerous manipulations that could lead to serious execution-time errors.

15. To write fixed-length records, use file open mode:

1. ios::app
2. ios::ate
3. ios::trunc
4. ios::binary

**Project 4 Hangman POINTS 10**

**Complete the project as per the instructions provided by your instructor.**

**Lesson 14: Data Structures**

We have studied fixed-size **data structures** such as one-dimensional arrays, two-dimensional arrays and structs. This chapter introduces **dynamic data structures** that grow and shrink during execution. **Linked lists** are collections of data items "lined up in a row"insertions and removals are made anywhere in a linked list. **Stacks** are important in compilers and operating systems: Insertions and removals are made only at one end of a stackits **top**. **Queues** represent waiting lines; insertions are made at the back (also referred to as the **tail**) of a queue and removals are made from the front (also referred to as the **head**) of a queue. **Binary trees** facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representation of file system directories and compilation of expressions into machine language. These data structures have many other interesting applications.

We discuss several popular and important data structures and implement programs that create and manipulate them. We use classes, class templates, inheritance and composition to create and package these data structures for reusability and maintainability.

**Self-Referential Classes**

A **self-referential class** contains a pointer member that points to a class object of the same class type. For example, the definition

class Node

{

public:

Node( int ); // constructor

void setData( int ); // set data member

int getData() const; // get data member

void setNextPtr( Node \* ); // set pointer to next Node

Node \*getNextPtr() const; // get pointer to next Node

private:

int data; // data stored in this Node

Node \*nextPtr; // pointer to another object of same type

}; // end class Node

defines a type, Node. Type Node has two private data membersinteger member data and pointer member nextPtr. Member nextPtr points to an object of type Nodeanother object of the same type as the one being declared here, hence the term "self-referential class." Member nextPtr is referred to as a **link**i.e., nextPtr can "tie" an object of type Node to another object of the same type. Type Node also has five member functionsa constructor that receives an integer to initialize member data, a setData function to set the value of member data, a getdata function to return the value of member data, a setNextPtr function to set the value of member nextPtr and a getNextPtr function to return the value of member nextPtr.

Self-referential class objects can be linked together to form useful data structures such as lists, queues, stacks and trees. [Figure 21.1](mk:@MSITStore:C:\Documents%20and%20Settings\sowmyag\Desktop\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\Book\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch21lev1sec2.html#ch21fig01) illustrates two self-referential class objects linked together to form a list. Note that a slashrepresenting a null (0) pointeris placed in the link member of the second self-referential class object to indicate that the link does not point to another object. The slash is only for illustration purposes; it does not correspond to the backslash character in C++. A null pointer normally indicates the end of a data structure just as the null character ('\0') indicates the end of a string.

**Dynamic Memory Allocation and Data Structures**

Creating and maintaining dynamic data structures requires dynamic memory allocation, which enables a program to obtain more memory at execution time to hold new nodes. When that memory is no longer needed by the program, the memory can be released so that it can be reused to allocate other objects in the future. The limit for dynamic memory allocation can be as large as the amount of available physical memory in the computer or the amount of available virtual memory in a virtual memory system. Often, the limits are much smaller, because available memory must be shared among many programs.

The new operator takes as an argument the type of the object being dynamically allocated and returns a pointer to an object of that type. For example, the statement

Node \*newPtr = new Node( 10 ); // create Node with data 10

allocates sizeof( Node ) bytes, runs the Node constructor and assigns the address of the new Node object to newPtr. If no memory is available, new throws a bad\_alloc exception. The value 10 is passed to the Node constructor which initializes the Node's data member to 10.

The delete operator runs the Node destructor and deallocates memory allocated with newthe memory is returned to the system so that the memory can be reallocated in the future. To free memory dynamically allocated by the preceding new, use the statement

delete newPtr;

Note that newPtr itself is not deleted; rather the space newPtr points to is deleted. If pointer newPtr has the null pointer value 0, the preceding statement has no effect. It is not an error to delete a null pointer.

### Linked Lists

A linked list is a linear collection of self-referential class objects, called **nodes**, connected by **pointer links** hence, the term "linked" list. A linked list is accessed via a pointer to the first node of the list. Each subsequent node is accessed via the link-pointer member stored in the previous node. By convention, the link pointer in the last node of a list is set to null (0) to mark the end of the list. Data is stored in a linked list dynamically each node is created as necessary. A node can contain data of any type, including objects of other classes. If nodes contain base-class pointers to base-class and derived-class objects related by inheritance, we can have a linked list of such nodes and use virtual function calls to process these objects polymorphically. Stacks and queues are also **linear data structures** and, as we will see, can be viewed as constrained versions of linked lists. Trees are **nonlinear data structures**.

Lists of data can be stored in arrays, but linked lists provide several advantages. A linked list is appropriate when the number of data elements to be represented at one time is unpredictable. Linked lists are dynamic, so the length of a list can increase or decrease as necessary. The size of a "conventional" C++ array, however, cannot be altered, because the array size is fixed at compile time. "Conventional" arrays can become full. Linked lists become full only when the system has insufficient memory to satisfy dynamic storage allocation requests.

Linked lists can be maintained in sorted order by inserting each new element at the proper point in the list. Existing list elements do not need to be moved.

*NOTE: An array can be declared to contain more elements than the number of items expected, but this can waste memory. Linked lists can provide better memory utilization in these situations. Linked lists allow the program to adapt at runtime.*

Insertion and deletion in a sorted array can be time consuming all the elements following the inserted or deleted element must be shifted appropriately. A linked list allows efficient insertion operations anywhere in the list.

The elements of an array are stored contiguously in memory. This allows immediate access to any array element, because the address of any element can be calculated directly based on its position relative to the beginning of the array. Linked lists do not afford such immediate "direct access" to their elements. So accessing individual elements in a linked list can be considerably more expensive than accessing individual elements in an array. The selection of a data structure is typically based on the performance of specific operations used by a program and the order in which the data items are maintained in the data structure. For example, it is typically more efficient to insert an item in a sorted linked list than a sorted array.

Linked list nodes are normally not stored contiguously in memory. Logically, however, the nodes of a linked list appear to be contiguous. [Figure 21.2](mk:@MSITStore:C:\Documents%20and%20Settings\sowmyag\Desktop\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\Book\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch21lev1sec4.html#ch21fig02) illustrates a linked list with several nodes.

Using dynamic memory allocation (instead of fixed-size arrays) for data structures that grow and shrink at execution time can save memory. Keep in mind, however, that pointers occupy space and that dynamic memory allocation incurs the overhead of function calls.

#### Linked List Implementation

*Refer to Figures 20.6 – 20.12 in the textbook for the program*

#### Linear and Circular Singly Linked and Doubly Linked Lists

The kind of linked list we have been discussing is a **singly linked list** the list begins with a pointer to the first node, and each node contains a pointer to the next node "in sequence." This list terminates with a node whose pointer member has the value 0. A singly linked list may be traversed in only one direction.

A **circular, singly linked list** begins with a pointer to the first node, and each node contains a pointer to the next node. The "last node" does not contain a 0 pointer; rather, the pointer in the last node points back to the first node, thus closing the "circle."

A **doubly linked list** allows traversals both forward and backward. Such a list is often implemented with two "start pointers" one that points to the first element of the list to allow front-to-back traversal of the list and one that points to the last element to allow back-to-front traversal. Each node has both a forward pointer to the next node in the list in the forward direction and a backward pointer to the next node in the list in the backward direction. If your list contains an alphabetized telephone directory, for example, a search for someone whose name begins with a letter near the front of the alphabet might begin from the front of the list. Searching for someone whose name begins with a letter near the end of the alphabet might begin from the back of the list.

In a **circular, doubly linked list** the forward pointer of the last node points to the first node, and the backward pointer of the first node points to the last node, thus closing the "circle."

**Stacks**

A stack data structure allows nodes to be added to the stack and removed from the stack only at the top. For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure. One way to implement a stack is as a constrained version of a linked list. In such an implementation, the link member in the last node of the stack is set to null (zero) to indicate the bottom of the stack.

The primary member functions used to manipulate a stack are push and pop. Function push inserts a new node at the top of the stack. Function pop removes a node from the top of the stack, stores the popped value in a reference variable that is passed to the calling function and returns true if the pop operation was successful (false otherwise).

Stacks have many interesting applications. For example, when a function call is made, the called function must know how to return to its caller, so the return address is pushed onto a stack. If a series of function calls occurs, the successive return values are pushed onto the stack in last-in, first-out order, so that each function can return to its caller. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

Stacks provide the memory for, and store the values of, automatic variables on each invocation of a function. When the function returns to its caller or throws an exception, the destructor (if any) for each local object is called, the space for that function's automatic variables is popped off the stack and those variables are no longer known to the program.

Stacks are used by compilers in the process of evaluating expressions and generating machine-language code. The exercises explore several applications of stacks, including

### Queues

A queue is similar to a supermarket checkout linethe first person in line is serviced first, and other customers enter the line at the end and wait to be serviced. Queue nodes are removed only from the head of the queue and are inserted only at the tail of the queue. For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure. The insert and remove operations are known as **enqueue** and **dequeue**.

Queues have many applications in computer systems. Computers that have a single processor can service only one user at a time. Entries for the other users are placed in a queue. Each entry gradually advances to the front of the queue as users receive service. The entry at the front of the queue is the next to receive service.

Queues are also used to support **print spooling**. For example, a single printer might be shared by all users of a network. Many users can send print jobs to the printer, even when the printer is already busy. These print jobs are placed in a queue until the printer becomes available. A program called a **spooler** managers the queue to ensure that, as each print job completes, the next print job is sent to the printer.

Information packets also wait in queues in computer networks. Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to the packet's final destination. The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.

A file server in a computer network handles file access requests from many clients throughout the network. Servers have a limited capacity to service requests from clients. When that capacity is exceeded, client requests wait in queues.

### Trees

Linked lists, stacks and queues are linear data structures. A tree is a nonlinear, two-dimensional data structure. Tree nodes contain two or more links. This section discusses binary trees trees whose nodes all contain two links (none, one or both of which may be null).

#### Basic Terminology

For the purposes of this discussion, refer to nodes A, B, C and D in [Fig. 20.18](mk:@MSITStore:C:\Documents%20and%20Settings\sowmyag\Desktop\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\Book\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch21lev1sec7.html#ch21fig18). The **root node** (node B) is the first node in a tree. Each link in the root node refers to a **child** (nodes A and D). The **left child** (node A) is the root node of the **left subtree** (which contains only node A), and the **right child** (node D) is the root node of the **right subtree** (which contains nodes D and C). The children of a single node are called **siblings** (e.g., nodes A and D are siblings). A node with no children is called a **leaf node** (e.g., nodes A and C are leaf nodes). Computer scientists normally draw trees from the root node down exactly the opposite of how trees grow in nature.

#### Binary Search Trees

This section discusses a special binary tree called a **binary search tree**. A binary search tree (with no duplicate node values) has the characteristic that the values in any left subtree are less than the value in its **parent node**, and the values in any right subtree are greater than the value in its parent node. [Figure 20.19](mk:@MSITStore:C:\Documents%20and%20Settings\sowmyag\Desktop\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\Book\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch21lev1sec7.html#ch21fig19) illustrates a binary search tree with 9 values. Note that the shape of the binary search tree that corresponds to a set of data can vary, depending on the order in which the values are inserted into the tree.

#### Inorder Traversal Algorithm

Function inOrderTraversal invokes utility function inOrderHelper to perform the inorder traversal of the binary tree. The steps for an inorder traversal are:

1. Traverse the left subtree with an inorder traversal. (This is performed by the call to inOrderHelper at line 102.)
2. Process the value in the nodei.e., print the node value (line 103).
3. Traverse the right subtree with an inorder traversal. (This is performed by the call to inOrderHelper at line 104.)

The value in a node is not processed until the values in its left subtree are processed, because each call to inOrderHelper immediately calls inOrderHelper again with the pointer to the left subtree. The inorder traversal of the tree in [Fig. 20.23](mk:@MSITStore:C:\Documents%20and%20Settings\sowmyag\Desktop\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\Book\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch21lev1sec7.html#ch21fig23) is

6 13 17 27 33 42 48

Note that the inorder traversal of a binary search tree prints the node values in ascending order. The process of creating a binary search tree actually sorts the datathus, this process is called the **binary tree sort**.

#### Preorder Traversal Algorithm

Function preOrderTraversal invokes utility function preOrderHelper to perform the preorder traversal of the binary tree. The steps for an preorder traversal are:

1. Process the value in the node (line 83).
2. Traverse the left subtree with a preorder traversal. (This is performed by the call to preOrderHelper at line 84.)
3. Traverse the right subtree with a preorder traversal. (This is performed by the call to preOrderHelper at line 85.)

The value in each node is processed as the node is visited. After the value in a given node is processed, the values in the left subtree are processed. Then the values in the right subtree are processed. The preorder traversal of the tree in [Fig. 20.23](mk:@MSITStore:C:\Documents%20and%20Settings\sowmyag\Desktop\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\Book\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch21lev1sec7.html#ch21fig23) is

27 13 6 17 42 33 48

#### Postorder Traversal Algorithm

Function postOrderTraversal invokes utility function postOrderHelper to perform the postorder traversal of the binary tree. The steps for an postorder traversal are:

1. Traverse the left subtree with a postorder traversal. (This is performed by the call to postOrderHelper at line 122.)
2. Traverse the right subtree with a postorder traversal. (This is performed by the call to postOrderHelper at line 123.)
3. Process the value in the node (line 124).

The value in each node is not printed until the values of its children are printed. The postOrderTraversal of the tree in [Fig. 20.23](mk:@MSITStore:C:\Documents%20and%20Settings\sowmyag\Desktop\C++%20How%20to%20Program%205th%20+%20Source%20Codes%20-%20Deitel\Book\C++%20How%20to%20Program,%205th%20Edition.chm::/0131857576/ch21lev1sec7.html#ch21fig23) is

6 17 13 33 48 42 27

#### Duplicate Elimination

The binary search tree facilitates **duplicate elimination**. As the tree is being created, an attempt to insert a duplicate value will be recognized, because a duplicate will follow the same "go left" or "go right" decisions on each comparison as the original value did when it was inserted in the tree. Thus, the duplicate will eventually be compared with a node containing the same value. The duplicate value may be discarded at this point.

Searching a binary tree for a value that matches a key value is also fast. If the tree is balanced, then each branch contains about half the number of nodes in the tree. Each comparison of a node to the search key eliminates half the nodes. This is called an O (log n) algorithm. So a binary search tree with n elements would require a maximum of log2 n comparisons either to find a match or to determine that no match exists. This means, for example, that when searching a (balanced) 1000-element binary search tree, no more than 10 comparisons need to be made, because 210 > 1000. When searching a (balanced) 1,000,000-element binary search tree, no more than 20 comparisons need to be made, because 220 > 1,000,000.

Fill in the blanks in each of the following:

1. A self-\_\_\_\_\_ class is used to form dynamic data structures that can grow and shrink at execution time
2. The \_\_\_\_\_ operator is used to dynamically allocate memory and construct an object; this operator returns a pointer to the object.
3. A(n)\_\_\_\_\_ is a constrained version of a linked list in which nodes can be inserted and deleted only from the start of the list and node values are returned in last-in, first-out order.
4. A function that does not alter a linked list, but looks at the list to determine whether it is empty, is an example of a(n)\_\_\_\_\_ function.
5. A queue is referred to as a(n)\_\_\_\_\_ data structure, because the first nodes inserted are the first nodes removed.
6. The pointer to the next node in a linked list is referred to as a(n) \_\_\_\_\_.
7. The \_\_\_\_\_ operator is used to destroy an object and release dynamically allocated memory.
8. A(n)\_\_\_\_\_ is a constrained version of a linked list in which nodes can be inserted only at the end of the list and deleted only from the start of the list.
9. A(n) \_\_\_\_\_ is a nonlinear, two-dimensional data structure that contains nodes with two or more links.
10. A stack is referred to as a(n)\_\_\_\_\_ data structure, because the last node inserted is the first node removed.
11. The nodes of a(n)\_\_\_\_\_ tree contain two link members.
12. The first node of a tree is the \_\_\_\_\_ node.
13. Each link in a tree node points to a(n) \_\_\_\_\_ or \_\_\_\_\_ of that node.
14. A tree node that has no children is called a(n) \_\_\_\_\_ node.
15. The four traversal algorithms we mentioned in the text for binary search trees are \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ and \_\_\_\_\_\_.

What are the differences between a linked list and a stack?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

What are the differences between a stack and a queue?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Which of the following is not a dynamic data structure?

1. Linked list.
2. Stack.
3. Array.
4. Binary tree.

2. In general, linked lists allow:

1. Insertions and removals anywhere.
2. Insertions and removals only at one end.
3. Insertions at the back and removals from the front.
4. None of the above.

3. Which data structure represents a waiting line and limits insertions to be made at the back of the data structure and limits removals to be made from the front?

1. Stack.
2. Queue.
3. Binary tree.
4. Linked list.

4. Which of the following is a self-referential object?

1. class SelfRefer  
   {  
    int \* xPtr;  
   };
2. class selfRefer  
   {  
    selfRefer x;  
   };
3. class SelfRefer  
   {  
    SelfRefer \* xPtr;  
   };
4. class SelfRefer1  
   {  
    SelfRefer2 x;  
   };  
   class SelfRefer2  
   {  
    SelfRefer1 x;  
   };

5. The pointer member in a self-referential class is referred to as a:

1. Link.
2. Connector.
3. Tie.
4. Referrer.

6 . The \_\_\_\_\_\_\_\_\_\_ operator takes as an argument the type of object being allocated and returns a \_\_\_\_\_\_\_\_\_\_.

1. new, pointer to an object of that type.
2. delete, reference to an object of that type.
3. delete, copy of the object of that type.
4. sizeof, reference to an object of that type.

7 . Given that the line

delete newPtr;

just executed, what can you conclude?

1. The memory referenced by newPtr is released only if it is needed by the system.
2. The pointer newPtr is of type void \*.
3. The pointer newPtr only exists if there was an error freeing the memory.
4. The pointer newPtr still exists.

8. \_\_\_\_\_\_\_\_\_\_ is not an advantage of linked lists when compared to arrays.

1. Dynamic memory allocation.
2. Efficient insertion and deletion.
3. Direct access to any list element.
4. No need to allocate extra space, “just in case.”

9. For a non-empty linked list, select the code that should appear in a function that adds a node to the end of the list. newPtr is a pointer to the new node to be added and lastPtr is a pointer to the current last node. Each node contains a pointer nextPtr.

1. lastPtr->nextPtr = newPtr;  
   lastPtr = newPtr
2. lastPtr = newPtr;  
   lastPtr->nextPtr = newPtr
3. newPtr->nextPtr = lastPtr;  
   lastPtr = newPtr
4. lastPtr = newPtr;  
   newPtr->nextPtr = lastPtr

10 . What kind of linked list begins with a pointer to the first node, and each node contains a pointer to the next node, and the pointer in the last node points back to the first node?

1. Circular, singly-linked list.
2. Circular, doubly-linked list.
3. Singly-linked list.
4. Doubly-linked list.

11. Which of the following statements about stacks is incorrect?

1. Stacks can be implemented using linked lists.
2. Stacks are first-in, first-out (FIFO) data structures.
3. New nodes can only be added to the top of the stack.
4. The last node (at the bottom) of a stack has a null (0) link.

12 . A stack is initially empty, then the following commands are performed:

push 5

push 7

pop

push 10

push 5

pop

Which of the following is the correct stack after those commands (assume the top of the stack is on the left)?

1. 5 10 7 5
2. 5 10
3. 7 5
4. 10 5

13. A queue performs the following commands (in pseudo-code):

enqueue 4, 6, 8, 3, 1

dequeue three elements

enqueue 3, 1, 5, 6

dequeue two elements

What number is now at the front of the queue?

1. 3
2. 4
3. 5
4. 6

14. A linked list has the functions insertAtFront, removeFromFront, insertAtBack and removeFromBack, which perform operations on nodes exactly as their names describe. Which two functions would most naturally model the enqueue and dequeue operations, respectively, of a queue?

1. insertAtBack and removeFromBack.
2. insertAtBack and removeFromFront.
3. removeFromFront and insertAtFront.
4. removeFromFront and insertAtBack.

15. If you add the following nodes to a binary search tree in the order they appear (left-to-right):

6 34 17 19 16 10 23 3

what will be the output of a postorder traversal of the resulting tree?

1. 3 10 16 23 19 17 34 6
2. 3 6 17 16 10 19 23 34
3. 6 3 34 17 16 10 19 23
4. 10 16 23 19 17 34 3 6

16. Which of the following tasks would a binary search tree not be well suited for?

1. Duplicate elimination.
2. Searching.
3. Sorting.
4. Reversing an unsorted sequence.

17. If you have a 1000-element balanced binary search tree, what is the maximum number of comparisons that may be needed to find an element in the tree?

1. 500
2. 20
3. 10
4. 8

**Lesson 15: Ogre Library**

**Refer to attachment 1**

27.1 Introduction

Ogre 3D graphics engine was created in 2000 by Steve Streeting, Ogre is an open-source project maintained by the Ogre team at www.ogre3d.org.

Graphics

Graphics are perhaps the most crucial feature of any video game. Once a specialty, graphics programming is becoming more accessible even to novices. There are many 3D graphics engines available—these frameworks hide the often tedious and complex programming required with graphics APIs and allow you to manage graphics more easily.

Ogre (Object-oriented Graphics Rendering Engine), one of the leading graphics engines, has been used in many commercial products including video games. It provides an object-oriented interface for 3D graphics programming. It supports the Direct3D and OpenGL graphics APIs and runs on the Windows, Linux and Mac platforms. Direct3D is Microsoft’s Windows 3D graphics API. OpenGL is a graphics specification implemented by many video card vendors across all major platforms, including Windows.

Ogre is strictly a graphics rendering engine—it does not directly support sound, physics, collision detection, networking or other game-related needs. The Ogre community has produced many add-ons that allow users to integrate other libraries with Ogre to support those features.

3D Models

A 3D model is a computer representation of an object which can be drawn on the screen - a process called rendering. Materials determine an object’s appearance by setting lighting properties, colors and textures. A texture is an image that is wrapped around the model.

Most objects displayed in 3D graphics, everything from the terrain to the characters and the buildings, are 3D models. Many models are created in 3D modeling tools. Some popular 3D modeling tools are Maya (usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7635018), SoftImage XSI (www.softimage.com/) and Blender (www.blender.org/). They’re all available for Windows, Linux and Mac platforms.

Blender is free and Maya offers a free version. SoftImage has a free 30-day trial available.

The Ogre community has also produced several tools to allow users to export 3D models from these and other popular modeling tools into Ogre.

Materials, Textures and Colors

Colors are determined by red, green and blue light intensities, which can range from 0 to

1.0—a color value of (1.0, 0, 0) will create a bright red color, (0, 1.0, 0) will create a bright green, and (0, 0, 1.0) will make a bright blue. The first value is the red intensity, the second is the green intensity and the third is the blue intensity. To create white, use the maximum intensities of all three color values, (1.0, 1.0, 1.0). To create black—the absence of all color—use (0, 0, 0). Color values sometimes include an alpha channel to represent transparency, also ranging from 0 to 1.0, 0 being completely transparent and 1 completely opaque. Figure 27.1 shows common colors and their red, green, and blue intensity values. You can find color charts on the web as well, such as the color chart at [www.tayloredmktg.com/rgb/](http://www.tayloredmktg.com/rgb/).

|  |  |
| --- | --- |
|  |  |

In 3D graphics, materials are used to determine the color of a 3D model. A material determines how the model should reflect different types of light and applies textures to the model. Materials can be set to use different levels of detail (LoD) depending on how far away from the viewer the model is. When close up, the model should be rendered with as much detail as possible. When the object in the scene is far away, there is no point in wasting computing power rendering details that the viewer can’t see. To increase performance, the object can be rendered with much less detail.

Lighting

There are four different types of light in a 3D scene—ambient, diffuse, emissive and specular.

Ambient light is the general lighting in the scene that has been reflected off so many surfaces that it doesn’t appear to have any definite source. Diffuse light appears to come from a particular direction and is reflected evenly off any surfaces it hits. Emissive light appears to come from an object in the scene. Emissive light won’t affect the objects around it but will make the object emitting it seem brighter. Specular light comes from a particular direction and is reflected off an object based on the direction to the viewer. This is used to make an object appear shiny.

Collision Detection and Response

Collision detection is the process of determining whether two objects in a game are touching. You must know which objects to test and must deal with some complex mathematics. Checking whether one square hits another is relatively simple if each is parallel to flat ground. Checking circles and spheres is more difficult—the mathematics of curved surfaces is more complex. Objects need to react appropriately when they collide with other objects. Some objects, such as walls, are stationary, while others move throughout the scene. Modeling the physics of moving objects can be complex. There are collision detection and physics modeling libraries that handle these complexities for you. Such libraries help to create a realistic game-playing experience.

Sound

Sound is crucial to the game-playing experience. Gamers want to hear the lasers on their ships blasting away or the engines of their street racers revving up as they “peel out” at the starting line. Audio libraries help you enrich your games with sound. Many of those libraries support 3D sound. In a 3D scene, objects emitting sound may be at various distances and directions from the user. The sound libraries take these factors into account when playing sounds. A sound from an object close to the listener will be louder than the sound from an object farther away. Also, sounds from one side of the listener will be played differently than sounds from the other.

Text

Games often communicate with the user by displaying text. This can range from giving the user instructions, to simply reporting how many points he or she has scored so far. In many games, text is a crucial form of communication between players. You can find free text fonts to use in your games at [www.1001freefonts.com](http://www.1001freefonts.com).

Timers

The speed at which a game runs can vary between systems due to differences in processor speeds. To solve this problem, game programmers use timers to control animation speed.

If an object moves the same distance every frame (each time the screen is redrawn), then it may move at different speeds on different computers. A game running at 100 frames per second (fps) would be twice as fast as the same game running at 50 frames per second. Timers help keep the game play consistent by regulating the speed.

User Experience

Games should be fun to play and should appeal to the player in as many ways as possible. The basics we’ve discussed contribute to the overall user experience. You can get the player’s attention through graphics and sound. Actions in games often have sounds associated with them. Many web sites offer free sounds you can use in your games. Some popular sound sites are Sound Hunter (www.soundhunter.com), Absolute Sound Effects Archive (www.grsites.com/sounds) and the search engine FindSounds (www.findsounds.com).

You can also play a sound track in the background. Be sure to get permission to use any copyrighted songs if you plan on releasing your game as a product. Players need to interact with games. User input devices include the keyboard, mouse, joystick and game controller. Keep the controls simple—the game should be easy to use, but not easy to beat. You can communicate with the player using text. In the next several sections, we present a complete C++/Ogre implementation of a simple game featuring a play mechanic similar to the classic video game Pong®, originally developed by Atari in 1972. (See the original Pong coin-op game in Fig. 27.2.)

Pong has four major game objects, a ball, two paddles and a rectangular box (Fig. 27.3). The ball will bounce across the screen inside the box while the players control the paddles to keep the ball from hitting the left or right sides. If the ball hits the left or right side of the box, the player “attacking” that side is awarded a point. The score is displayed at the top of the screen.

Play the game for a while, using the A and Z keys to control the left paddle and the up and down arrow keys to control the right paddle. Notice the colors of the objects, the ball interacting with the other objects and the score displayed at the top of the screen. Hit the Esc key to quit—closing the window won’t stop the program.

27.4.1 Ogre Initialization

Root is the base object used in Ogre used to start the engine. No Ogre calls can be made until

the Root object has been created. Call the showConfigDialog function of the Root class to display the dialog. The OGRE Engine Rendering Setup dialog box enables the user to choose the rendering settings. The resolution is defined by two values, width and height, which determine the number of pixels used to draw the scene. A higher resolution will produce more detailed graphics.

A color depth of n bits means that 2n possible colors can be displayed on the screen. The RenderWindow is a window in which Ogre will render graphics.

27.4.2 Creating a Scene

After we’ve initialized Ogre and set up a window to render our graphics in, we’ll add some objects to create our scene—the collection of images that we display on the screen.

SceneManager

To control the scene we use Ogre’s SceneManager object. The SceneManager manages the scene graph, a data structure that contains all the objects in the scene, both visible and nonvisible. The SceneManager is used to create objects that will be added to the scene graph and determines which objects will be rendered. Excluding objects that are not within the visible scene from being rendered, known as culling, decreases rendering time and increases performance. This is done automatically. We’ll keep a pointer to this SceneManager object, as it will be used extensively throughout the game. Several types of SceneManagers have been designed to handle different types of scenes, such as indoor scenes or expansive landscape scenes. An Ogre application can use more than one SceneManager, separately or at the same time. For the purposes of this chapter, we use only one instance of the generic scene type (ST\_GENERIC), a SceneManager that’s not optimized for any particular type of scene.

Camera

Once we have a SceneManager, we can start constructing our scene. First we add a Camera.

A Camera in Ogre is the eye through which you view the scene. A 3D scene is usually too large to be displayed in one window. The Camera looks into the scene and tells Ogre what part you can actually see. Cameras can be placed at any location in the scene or attached to SceneNodes. If attached to a SceneNode, the Camera will follow that node if it moves within the scene. Ogre supports multiple Cameras in a single scene, but we need only one.

We use the SceneManager to create the Camera and then set the position, orientation, clip distances and Viewport. The Viewport is the area of the screen used to display what the Camera can see. We set the Viewport’s background color to white. A Camera can have more than one Viewport, but we’ll use only one for our game.

Light

One of the most important aspects of a 3D scene is lighting. Ogre has three types of Lights—Point, Spot and Directional. Point lights have a position in space and radiate light in all directions. Spot lights have a position in space like Point lights, but radiate light in only one direction; the strength of the light fades as the distance from the source increases. Directional lights do not have a position in space, they have only a direction in which they shine—the light is assumed to come from the same direction no matter where you are in the scene. We use a Point light in our game. Lights are created with the createLight function of class SceneManager.

We set the Light’s position by specifying its x-, y- and z-coordinates.

Section 27.4.3 Adding to the Scene

An Entity is an instance of a mesh within the scene. A mesh is a file that contains the geometry information of a 3D model. Many Entity objects can be based on the same mesh, as long as each

Entity has a unique name. Use the SceneManager to create SceneNodes that hold information about an object and its position in the scene.

The root node is the parent of all other nodes. When you create a child of the root node, its initial position is (0, 0, 0). Attach Entity objects to SceneNodes with the attachObject function of class SceneNode. scale changes the size of the Entity attached to the SceneNode, but it does not affect the size of the actual mesh that the node’s Entity is based on. setScale changes the size based on the original size of the Entity. These functions also scale all children of the SceneNode by the same factor. To change that, call the setInheritScale function and pass it false. setPosition function places the node at the given coordinates in the scene.

Ogre uses a material script to create a material. Save the file with a .material extension. A material file can define multiple materials; every material must have a unique name.

An Overlay is defined by a script saved in an .overlay file. A single .overlay file can hold several

Overlay definitions. Every object in an Overlay has three main attributes—metrics mode, position and size. Overlays are composed of OverlayElements. The first element in an Overlay must be an OverlayContainer. An OverlayContainer can hold any OverlayElement. A TextAreaOverlayElement holds text. Call the show function to display the Overlay on the screen

Use TextAreaOverlayElement to display text. Call setCaption to change text on the screen.

An Overlay with a higher z-order will be rendered on top of an Overlay with a lower z-order.

Fonts are defined by a script in a fontdef file. Use the static member function getSingleton of class OverlayManager to get the OverlayManager object.

Section 27.4.4 Animation and Timers

The translate function moves a SceneNode. SceneNode translations are done in parent space by default. Translations in parent space are done with respect to the parent’s origin. Translations in world space are done with respect to the origin of the scene (0, 0, 0). Translations in local space are done with respect to the node’s origin. A FrameListener processes Ogre::FrameEvents. A FrameEvent occurs when a frame begins or ends.

Section 27.4.5 User Input

Ogre does not directly support user input from devices such as the keyboard, mouse or joystick.

Use the Object Oriented Input System (OIS) for handling user input. The InputManager is used to create the various input devices. To create the InputManager we must provide it with a window in which to collect input. A Keyboard object collects KeyEvents and sends them to a KeyListener. OIS defines an enumeration of all the keys on the keyboard, which we use to determine which key was pressed.

Section 27.4.6 Collision Detection

getPosition returns a Vector3 representing the node’s position relative to its parent node; \_getDerivedPosition returns the position relative to the origin. The SceneManager can retrieve any node within the scene graph by referencing the name given to the node when it was created.

The direction of the Ball is determined by a Vector3. A positive x-value means the Ball will move right along the x-axis, and a negative value will move the Ball left. If the Ball is moving right, multiplying its x-value by –1 will change the sign and reverse the direction. There are whole libraries dedicated to handling collisions and physics.

Section 27.4.7 Sound

OgreAL is a wrapper around the OpenAL audio library. The wrapper allows us to integrate sound functionality into the Ogre code by attaching the sounds to nodes within the scene graph.

We must have the preprocessor directive to include the OgreAL.h header. Sound is the OgreAL object that contains the sound data. Use the createSound function of class SoundManager to create sounds. There can be only one SoundManager. The createSound function takes three parameters. The first is an Ogre::String that will be the name of the Sound within the OgreAL system. The second is the name of the sound file associated with the Sound. The third is a bool that determines whether the Sound should be looped to continue playing. Passing false will play the Sound through once, then stop. Passing true will continuously loop the sound until you stop it. Attach the Sounds to a node with the attachObject function. Each Sound must have a unique name. A Sound must finish playing before it can be played again.

Section 27.4.8 Resources

All of the resources must be loaded before we can use them. Use a ResourceGroupManager to manage the game’s resources. The addResourceLocation function takes three Ogre::String arguments. The first is the location of the resources. The second is the type of file the resources are in. The third is the resource group these files belong to.

Section 27.4.9 Pong Driver

Ogre supports various platforms, so you should try not to write platform-specific code when you

can avoid it.

We presented you with a basic example of Pong. Use it as a foundation for your own version. Go out and find your own sounds to use. Add new features to the game. Explore Ogre’s other capabilities and create some cool visual effects. Really make this game your own. Game programming is all about being creative.

[www.ogre3d.org/](http://www.ogre3d.org/) The Ogre home page. Here you can find the latest Ogre news, download Ogre or Ogre-related tools, browse the documentation or check out projects that use Ogre.

www.ogre3d.org/index.php?option=com\_content&task=view&id=411&Itemid=131

Prebuilt SDK download page. There are SDKs available for Code::Blocks + MingGW C++ Toolbox, Visual C++ .Net 2003 and Visual C++ .Net 2005 (must install Service Pack 1).

Self-Review Exercises

Fill in the blanks in each of the following statements:

a) The \_\_\_\_\_\_\_\_\_\_\_\_\_\_ header includes the most commonly used Ogre header files.

b) The \_\_\_\_\_\_\_\_\_\_\_\_ object must be created before any other Ogre function (other than logging) is called.

c) The main type defined by OgreAL for pointing to sound-file data is \_\_\_\_\_\_\_\_\_\_\_\_\_ .

d) A(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ object is used to represent a color in Ogre.

e) The \_\_\_\_\_\_\_\_\_\_\_\_\_ header includes the most commonly used OgreAL header files.

f) \_\_\_\_\_\_\_\_\_\_\_\_\_\_ are used to define materials and overlays for Ogre programs.

g) The \_\_\_\_\_\_\_\_\_\_\_\_ object is used to load resources for Ogre programs.

h) Ogre uses a(n) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ object to manage the scene.

i) A 3D model is defined in an Ogre \_\_\_\_\_\_\_\_\_\_\_\_\_\_ file.

State whether each of the following is true or false. If false, explain why.

a) The coordinates (0, 0) refer to the bottom-left corner of an OverlayContainer.

b) If Ogre attempts to load an external file that does not exist, a runtime error will occur.

c) Color values in Ogre range from 0 to 255.

d) Passing a value of false to the createSound function will cause the sound file to play continuously.

e) An Overlay that draws text on the screen must specify a font in which that text should be drawn.

f) Every Entity must have a unique name.

Write statements to accomplish each of the following:

a) Attach an Entity pointer named entityPtr to a SceneNode pointer name nodePtr.

b) Scale the Entity from the previous question to half its original size.

c) Create the Sound sample that loops the sound.wav file.

d) If the spacebar is being pressed, set the value of the int number to 0.

e) Set an Overlay Element to position itself relative to the size of its parent Container.

f) Add a folder named sounds in the media folder as a "General" resource location.

g) Move a SceneNode 15 units left, 4 units up and 8 units toward you.

Find the error in each of the following:

a) SceneNode node;

b) ColourValue( 0, 0, 255 );

c) Root \*rootPtr = new Root();

rootPtr->initialize( true, "Window" );

d) viewportPtr = sceneManagerPtr->addViewport( cameraPtr );

**Project 5 Pong POINTS 10**

**Complete the project as per the instructions provided by your instructor.**