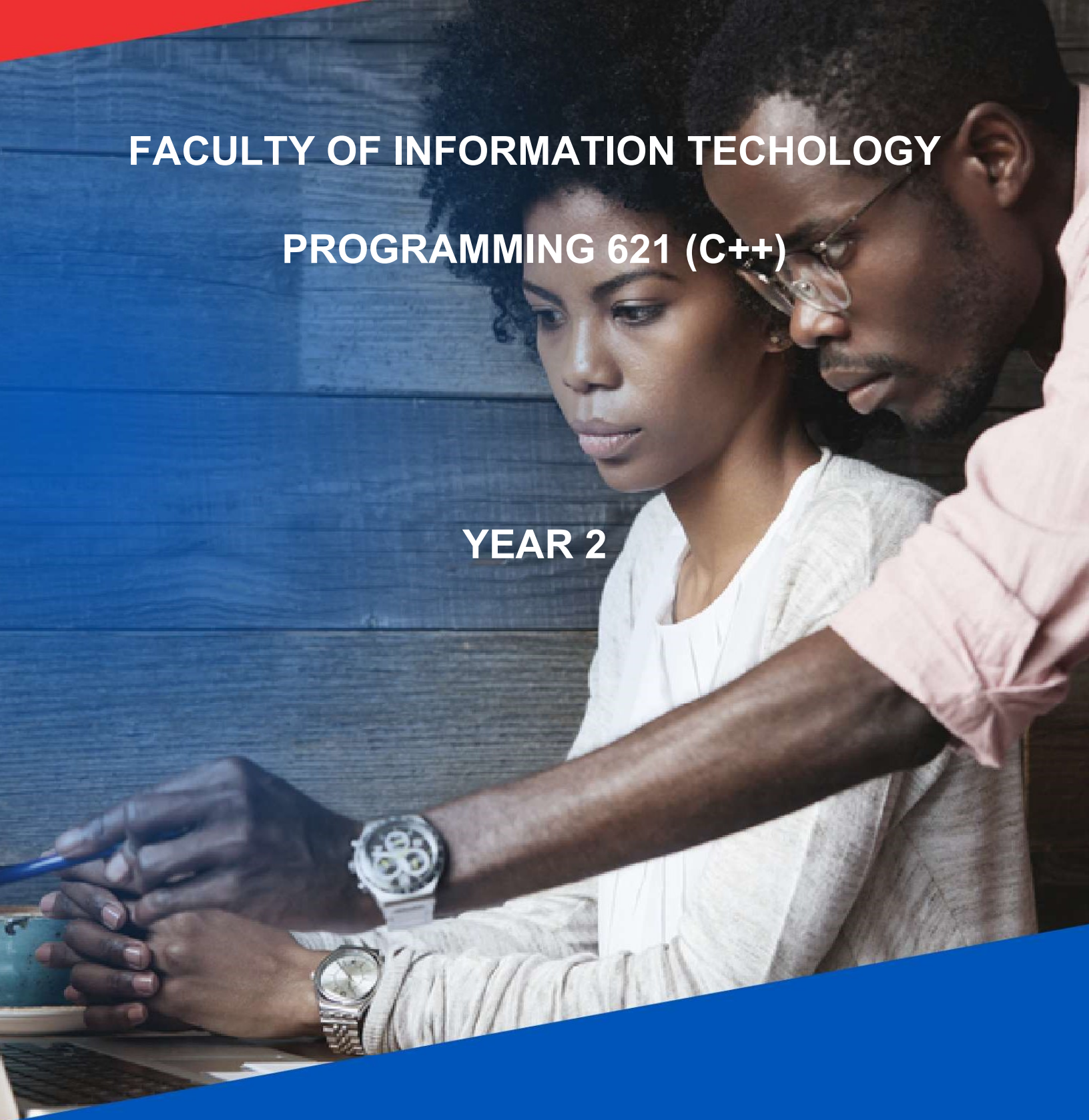
FACULTY OF INFORMATION TECHOLOGY

PROGRAMMING 621 (C++)

YEAR 2



RICHFIELD

richfield.ac.za

FACULTY OF INFORMATION TECHNOLOGY

STUDY GUIDE

MODULE: PROGRAMMING 621

SECTION A: PREFACE

INTRODUCTION

Welcome to the Faculty of Information Technology at Richfield Graduate Institute of Technology.

We trust you will find the contents and learning outcomes of this module both interesting and insightful as you begin your academic journey and eventually, your career in the business world. This section of the study guide is intended to orientate you to the module before the commencement of formal lectures. The following lectures will focus on the study units described

Table 1: Welcome and Orientation

SECTION A: WELCOME & ORIENTATION	
Study unit 1: Orientation Programme Introducing academic staff to the students by the academic head. Introduction of institution policies.	**Lecture 1**
Study unit 2: Orientation of Students to Library and Students Facilities Introducing students to physical structures Issuing of foundation learner guides and necessary learning material	**Lecture 2**
Study unit 3: Orientation of Programming 621, Student Guide, Textbooks and Prescribed Materials	**Lecture 3**
Study unit 4: Discussion of the Objectives and Outcomes of Module	**Lecture 4**
Study unit 5: Orientation and guidelines for completing Assignments Review and Recap of Study units 1-4	**Lecture 5**

MODULE DETAILS

Table 2: Module details

1nd Semester	Details
Title Of Module: **NQF Level:** **Credits:** **Mode of Delivery:**	Programming 621 (C++) NQF 6 10 Contact

PURPOSE OF THE MODULE

The purpose of this module is to give students a thorough understanding of the basics of C++ as a programming Language.When you learn programming, it is not only about the knowledge you acquire, but also (and specially) about the useful transferable skills you get. And the fact is that, in addition to becoming more precise and methodic, when you learn programming, you significantly improve your problem solving and abstraction skills.

PROGRAMMING 621 C++

This module exposes learners to the various aspects of C++. An intermediate-level language; learning this language will give you a much deeper understanding of programming structure. In C++, you have to write, declare, and explain everything in the source code, giving you a deeper knowledge of all the program parts.

LEARNING OUTCOMES

On completion of this module, students should have a basic/fundamental practical and theoretical knowledge of:

- Describe OOPs concepts
- Understand tokens, expressions, and control structures
- Explain arrays and strings and create programs using them
- Use functions and pointers in your C++ program
- Describe and use constructors and destructors

METHOD OF STUDY

Only the key sections that have to be studied are indicated under each topic in this study guide are expected to have a thorough working knowledge of the prescribed textbook. These form the basis for tests, assignments and examinations. To be able to do the activities and assignments for this module, and to achieve the learning outcomes and ultimately to be successful in the tests and exams.

You will need an in-depth understanding of the content of these sections in the learning guide and the prescribed books. To master the learning material, you must accept responsibility for your studies. Learning is not the same as memorising. You are expected to show that you understand and can apply the information. Lectures, tutorials, case studies and group discussions may also be used to present this module.

LECTURES AND TUTORIALS

Students must refer to the notice boards on their respective campuses for details of the lecture and tutorial timetables. The lecturer assigned to the module will also inform you of the number of lecture periods and tutorials allocated to a particular module. Prior preparation is required for each lecture and tutorial. Students are encouraged to actively participate in lectures and tutorials to ensure success in tests, group discussions, assignments and examinations.

NOTICES

All information about this module such as tests dates, lecture and tutorial timetables, assignments, examinations etc. will be displayed on the notice board located at your campus. Students must check the notice board daily. Should you require any clarification, please consult your lecturer, programme manager or administrator of your respective campus.

PRESCRIBED & RECOMMENDED MATERIAL

PRESCRIBED MATERIAL

Malik, D.S 2018. C++ Programming: Program Design including Data Structures. Eighth Edition. United Kingdom: Cengage Learning

RECOMMENDED MATERIAL

Malik, D.S 2018. C++ Programming: Program Design including Data Structures. Eighth Edition. United Kingdom: Cengage Learning

LIBRARY INFRASTRUCTURE

The following services are available to you:

- Each campus keeps a limited quantity of the recommended reading titles and a wider variety of similar titles which you may borrow. Please note that students are required to purchase the prescribed materials.
- Arrangements have been made with municipal, state and other libraries to stock our recommended reading and similar titles. You may use these on their premises or borrow them if available. It is your responsibility to safe keeps all library books.
- RGIT has also allocated one library period per week to assist you with your formal research under professional supervision.
- RGIT has dedicated electronic libraries for use by its students. The computers laboratories, when not in use for academic purposes, may also be used for research purposes. Booking is essential for all electronic library usage.

ASSESSMENT

The assessment for this module will comprise two Continuous Assessment (CA) Tests, an assignment and an examination. Your lecturer will inform you of the dates, times and the venues for each of these. You may also refer to the notice board on your campus or the Academic Calendar, which is displayed in all lecture rooms.

CONTINUOUS ASSESSMENT TESTS

ASSIGNMENT

There is one compulsory assignment for each module in each semester. Your lecturer will inform you of the Assignment questions at the commencement of this module.

EXAMINATION

There is one two-hour examination for each module. Make sure that you diarise the correct date, time and venue. The examinations department will notify you of your results once all administrative matters are cleared, and fees are paid up. The examination may consist of multiple-choice questions, short questions and essay type questions.

This requires you to be thoroughly prepared as all the content matter of lectures, tutorials, all references to the prescribed text and any other additional documentation/reference materials are examinable in both your tests and the examinations. The examination department will make available to you the details of the examination (date, time and venue) in due course. You must be seated in the examination room 15 minutes before the commencement of the examination. If you arrive late, you will not be allowed any extra time. Your learner registration card must always be in your possession.

FINAL ASSESSMENT

There are two compulsory tests for each module (in each semester). The final assessment for this module will be weighted as follows:

CA Test 1	15%
CA Test 2	15%
Assignment	10%
	(Total Mark40%)
Examination	60%
Total	100%

KEY CONCEPTS IN ASSIGNMENTS AND EXAMINATIONS

In assignment and examination questions, you will notice certain vital concepts (i.e. words/verbs) which tell you what is expected of you. For example, you may be asked in a question to list, describe, illustrate, demonstrate, compare, construct, relate, criticise, recommend or design information/aspects/factors/situations. To help you to know what these key concepts or verbs mean so that you will know what is expected of you, we present the following taxonomy by Bloom, explaining the concepts and stating the level of cognitive thinking that theses refer to.

Table 3: Bloom's Taxonomy

Competence	Skills Demonstrated
Knowledge	• Observation and recall of information • Knowledge of dates, events, places • Knowledge of major ideas • Mastery of subject matter ***Question Cues*** list, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc.
Comprehension	• Understanding information • Grasp meaning • Translate knowledge into a new context • Interpret facts, compare, contrast • Order, group, infer causes • Predict consequences ***Question Cues*** summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend

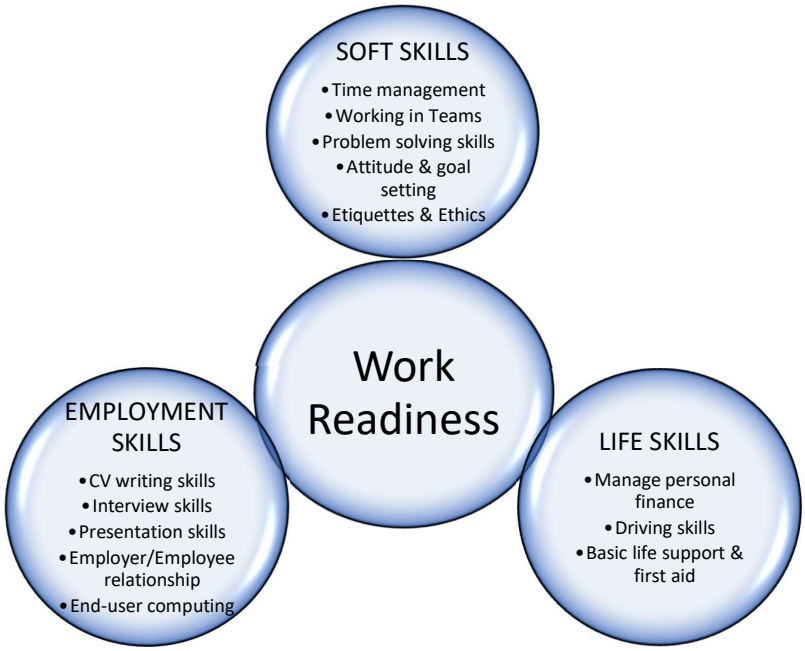
Application	• Use information • Use methods, concepts, theories in new situations • Solve problems using required skills or knowledge ***Questions Cues*** apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover
Analysis	• Seeing patterns • Organisation of parts • Recognition of hidden meanings • Identification of components ***Question Cues*** analyse, separate, order, arrange, divide, compare, select, infer, connect, classify, explain
Synthesis	• Use old ideas to create new ones • Generalise from given facts Relate knowledge from several areas • Predict, draw conclusions ***Question Cues*** combine, integrate, modify, rearrange, substitute, plan, create, design, invent, what if? compose, formulate, prepare, generalise, rewrite
Evaluation	• Compare and discriminate between ideas • Assess the value of theories, presentations • Make choices based on reasoned argument Verify value of evidence recognise subjectivity ***Question Cues*** assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarise

To prepare students for the world of work, a series of interventions over and above the formal curriculum, are concurrently implemented to prepare students. These include:

- Soft skills
- Employment skills
- Life skills
- End-User Computing (if not included in your curriculum)

It is in your interest to attend these workshops, complete the Work Readiness Logbook and prepare for the working world. The illustration below outlines some of the key concepts for Work Readiness that will be included in your timetable.

Figure 1: Work Readiness



WORK INTEGRATED LEARNING

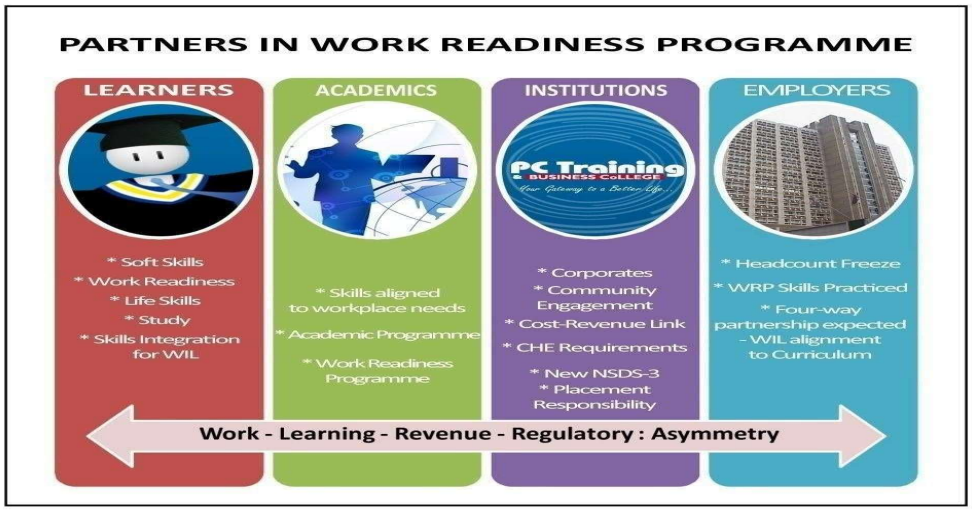
Work Integrated Learning (WIL) forms a core component of the curriculum for the completion of this programme. All modules which form part of this qualification will be

assessed in an integrated manner towards the end of the programme or after completion of all other modules. Prerequisites for placement with employers will include:

- Completion of all tests & assignment
- Success in examination
- Payment of all arrear fees
- Return of library books, etc.
- Completion of the Work Readiness Programme (WRP).

Students will be fully inducted on the Work Integrated Learning Module, the Workbooks & assessment requirements before placement with employers. The partners in the Work Integrated Learning are the same as the Work Readiness Programme:

Figure 1: Work Integrated Learning



Good luck and success in your studies…

TOPIC 1: PROGRAMMING LANGUAGE PARADIGMS	
1.1 Introduction to programming language paradigms	**Lecture 6**
1.2 Structured Programming	
1.3 Object Oriented Programming	**Lecture 7**
1.4 Problem Analysis Cycle	
1.5 Processing a C++ Program	**Lecture 8-9**
1.6 Algorithm Design	
TOPIC 2: INTRODUCTION TO C++	
2.1 Layout of C++ Program	**Lecture 10-15**
2.2Variables, Data Types	
2.3Operators	
2.4Input/Output	
2.5 Strings in C++	
TOPIC 3:CONTROL STRUCTURES	
3.1 Selection	**Lecture 13-15**
3.4 Iteration 3.5 Repetition(cont…)	
3.4 Sentinel controlled Loop	**Lecture 16-18**
3.7 Break and Continue statements	
3.8 Short-circuit evaluation	
TOPIC 4:FUNCTIONS	
4.4 Pre-defined C++ Functions	**Lecture 20-23**
4.2 User-defined Functions	
4.3 Value and reference parameters	
4.5 Function overloading	**Lecture 24-26**
TOPIC 5:COMPOUND DATA TYPES	
5.1 String Manipulation	**Lecture 27-29**
5.2 Array's	
5.3 Dynamic Memory	**Lecture 30-31**
5.4 Pointers	
TOPIC 6:CLASSES (ONE)	

6.1 Class Scope	**Lecture 32-34**
6.2 Constructors & Destructors	
6.3 Inheritance & Friendship	
Assessment questions	**Lecture 35-36**
TOPIC 7: CLASSES(TWO)	**Lecture 37**
7.1 Class Templates	**Lecture 37**
7.2 Virtual Functions	**Lecture 37**
7.3 Polymorphism	
TOPIC 8:ADVANCED CONCEPTS	
8.1 Files	**Lecture 38**
8.2 Exception Handling	**Lecture 39**
TOPIC 9: ADDENDUM 511(C): TYPICAL EXAMINATION QUESTIONS	

TOPIC 1
PROGRAMMING LANGUAGES PARADIGM

LEARNING OUTCOMES:

After studying this topic you should be able to:

- Understand the importance of programming language paradigms
- Become aware of structured-design and object-oriented design programming methodologies
- Learn what an algorithm is and explore problem-solving techniques
- Discover what a compiler is and what it does
- Examine a C++ program
- Explore how a C++ program is processed

1.1 INTRODUCTION TO PROGRAMMING LANGUAGE PARADIGMS

Paradigm can be termed as method to solve some problem or do a specified task. Programming paradigm is an approach to solve problem using some programming language or also it is a method to solve a problem using tools and techniques that are available to us following some approach.

There are lots of programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms.

Two popular approaches to programming design are the structured approach and the object-oriented approach, which are outlined below.

1.1.1 Difference Between Imperative and Declarative Programming

Imperative Programming as the name suggests is a type of programming that describes how the program executes. It comprises with how to get an answer step by step. Comprises the sequence of command imperatives. In this, the order of execution is very important and uses both mutable and immutable data.

Fortran, C, C++ programming languages are examples

Declarative Programming is a type of programming paradigm that describes what programs to be executed. It declares what kind of results we want and leave programming language aside focusing on simply figuring out how to produce them. Mainly focuses on the end result. It expresses the logic of computation. *Miranda, Erlang, Prolog* are examples.

1.2 STRUCTURED PROGRAMMING

Dividing a problem into smaller sub problems is called structured design. Each sub problem is then analysed, and a solution is obtained to solve the sub problem. The solutions to all of the sub problems are then combined to solve the overall problem. This process of implementing a structured design is called structured programming. The structured-design approach is also known as top-down design, bottom-up design, stepwise refinement, and modular programming.

1.3 OBJECTED ORIENTED PROGRAMMING

Object-oriented design (OOD) is a widely used programming methodology. In OOD, the first step in the problem-solving process is to identify the components called objects, which form the basis of the solution, and to determine how these objects interact with one another. For example, suppose you want to write a program that automates the DVD rental process for a local DVD store. The two main objects in this problem are the DVD and the customer. After identifying the objects, the next step is to specify for each object the relevant data and possible operations to be performed on that data. For example, for a DVD object, the data might include:

- movie name
- year released
- producer
- production company
- number of copies in stock

Some of the operations on a DVD object might include:

- checking the name of the movie

- reducing the number of copies in stock by one after a copy is rented
- incrementing the number of copies in stock by one after a customer returns a particular DVD of length.

This illustrates that each object consists of data and operations on that data. An object combines data and operations on the data into a single unit. In OOD, the final program is a collection of interacting objects. A programming language that implements OOD is called an object-oriented programming (OOP) language.

To work with objects, you need to know how to combine data and operations on the data into a single unit. In C11, the mechanism that allows you to combine data and operations on the data into a single unit is called a class.

Finally, or some problems, the structured approach to program design will be very effective. Other problems will be better addressed by OOD. For example, if a problem requires manipulating sets of numbers with mathematical functions, you might use the structured-design approach and outline the steps required to obtain the solution. The C11 library supplies a wealth of functions that you can use effectively to manipulate numbers. On the other hand, if you want to write a program that would make a juice machine operational, the OOD approach is more effective. C11 was designed specially to implement OOD. Furthermore, OOD works well with structured design. Both the structured-design and OOD approaches require that you master the basic components of a programming language to be an effective programmer.

1.4 PROBLEM ANALYSIS-CODING-EXECUTION CYCLE

Programming is a process of problem solving. Different people use different techniques to solve problems. Some techniques are nicely outlined and easy to follow. They not only solve the problem, but also give insight into how the solution is reached. These problem-solving techniques can be easily modified if the domain of the problem changes. To be a good problem solver and a good programmer, you must follow good problem solving techniques. One common problem-solving technique includes analysing a problem, outlining the problem requirements, and designing steps, called an **algorithm**, to solve the problem.

Algorithm: A step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.

In a programming environment, the problem-solving process requires the following three steps:

1. Analyse and outline the problem and its solution requirements, and design an algorithm to solve the problem.
2. Implement the algorithm in a programming language, such as C11, and verify that the algorithm works.
3. Maintain the program by using and modifying it if the problem domain changes.

Figure 1-1 summarizes the first two steps of this programming process.

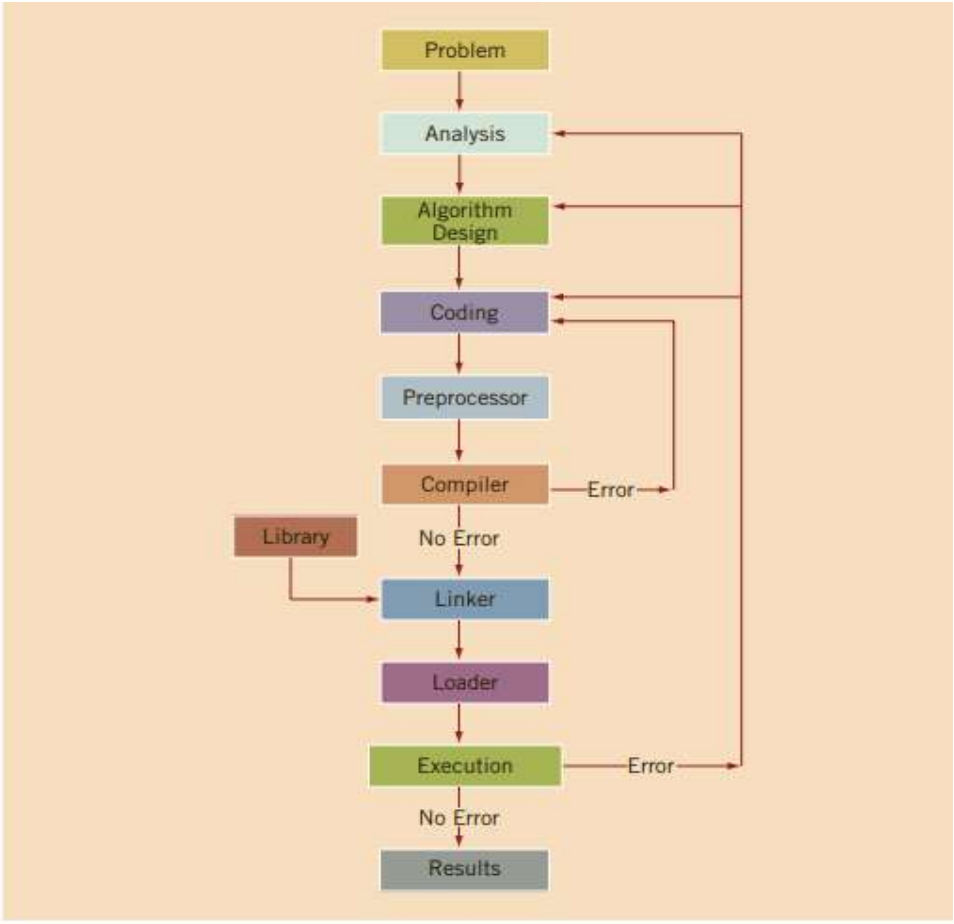


FIGURE 1-1 Problem analysis-coding-execution cycle

To develop a program to solve a problem, you start by analyzing the problem. You then design the algorithm; write the program instructions in a high-level language, or code the program; and enter the program into a computer system. Analyzing the problem is the first and most important step. This step requires you to do the following:

1. Thoroughly understand the problem.

2. Understand the problem requirements. Requirements can include whether the program requires interaction with the user, whether it manipulates data, whether it produces output, and what the output looks like. If the program manipulates data,

the programmer must know what the data is and how it is represented. That is, you need to look at sample data. If the program produces output, you should know how the results should be generated and formatted.

3. If the problem is complex, divide the problem into subproblems and repeat Steps 1 and 2. That is, for complex problems, you need to analyze each subproblem and understand each sub problem's requirements.

After you carefully analyze the problem, the next step is to design an algorithm to solve the problem. If you break the problem into subproblems, you need to design an algorithm for each subproblem. Once you design an algorithm, you need to check it for correctness. You can sometimes test an algorithm's correctness by using sample data. At other times, you might need to perform some mathematical analysis to test the algorithm's correctness.

Once you have designed the algorithm and verified its correctness, the next step is to convert it into an equivalent programming code. You then use a text editor to enter the programming code or the program into a computer. Next, you must make sure that the program follows the language's syntax. To verify the correctness of the syntax, you run the code through a compiler. If the compiler generates error messages, you must identify the errors in the code, remove them, and then run the code through the compiler again. When all the syntax errors are removed, the compiler generates the equivalent machine code, the linker links the machine code with the system's resources, and the loader places the program into main memory so that it can be executed.

The final step is to execute the program. The compiler guarantees only that the program follows the language's syntax. It does not guarantee that the program will run correctly. During execution, the program might terminate abnormally due to logical errors, such as division by zero.

Even if the program terminates normally, it may still generate erroneous results. Under these circumstances, you may have to re-examine the code, the algorithm, or even the problem analysis

EXAMPLE 1-1

In this example, we design an algorithm to find the perimeter and area of a rectangle. To find the perimeter and area of a rectangle, you need to know the rectangle's length and width. The perimeter and area of the rectangle are then given by the following formulas:

perimeter = 2 · (length + width)

area = length · width

The algorithm to find the perimeter and area of the rectangle is as follows:

1. Get the length of the rectangle.
2. Get the width of the rectangle.
3. Find the perimeter using the following equation:

   perimeter = 2 · (length + width)
4. Find the area using the following equation:

   area = length · width

EXAMPLE 1-2

In this example, we design an algorithm that calculates the sales tax and the price of an item sold in a particular state. The sales tax is calculated as follows: The state's portion of the sales tax is 4%, and the city's portion of the sales tax is 1.5%. If the item is a luxury item, such as a car more than $50,000, then there is a 10% luxury tax. To calculate the price of the item, we need to calculate the state's portion of the sales tax, the city's portion of the sales tax, and, if it is a luxury item, the luxury tax.

Suppose **salePrice** denotes the selling price of the item, **stateSalesTax** denotes the state's sales tax, **citySalesTax** denotes the city's sales tax, **luxuryTax** denotes the luxury tax, **salesTax** denotes the total sales tax, and **amountDue** denotes the final

price of the item. To calculate the sales tax, we must know the selling price of the item and whether the item is a luxury item.

The stateSalesTax and citySalesTax can be calculated using the following formulas:

stateSalesTax = salePrice · 0.04

citySalesTax = salePrice · 0.015

Next, you can determine **luxuryTax** as follows:

if (**item is a luxury item**)

luxuryTax = **salePrice · 0.1**

otherwise

luxuryTax = 0

Next, you can determine **salesTax** as follows:

salesTax = stateSalesTax + citySalesTax + luxuryTax

The algorithm to determine **salesTax** and **amountDue** is, therefore:

1. Get the selling price of the item.
2. Determine whether the item is a luxury item.
3. Find the state's portion of the sales tax using the formula:

   stateSalesTax = salePrice · 0.04
4. Find the city's portion of the sales tax using the formula:

   citySalesTax = salePrice · 0.015
5. Find the luxury tax using the following formula:

   if (**item is a luxury item**)

   luxuryTax = salePrice · 0.1

   otherwise

   luxuryTax = 0
6. Find **salesTax** using the formula:

   salesTax = stateSalesTax + citySalesTax + luxuryTax
7. Find **amountDue** using the formula:

   amountDue = salePrice + salesTax

1.5 PROCESSING A C++ PROGRAM

To execute on a computer, these C++ instructions first need to be translated into machine language. A program called a compiler translates instructions written in high-level languages into machine code.

Compiler: A program that translates instructions written in a high-level language into the equivalent machine language.

Consider the following C++ program:

```
#include <iostream>
 using namespace std;
        int main()
         {
         cout << "My first C++ program." << endl;
         return 0;
        }
```

At this point, you need not be too concerned with the details of this program. However, if you run (execute) this program, it will display the following line on the screen:

My first C++ program

Recall that a computer can understand only machine language. Therefore, in order to run this program successfully, the code must first be translated into machine language. In this section, we review the steps required to execute programs written in C++

The following steps, as shown in Figure 1-2, are necessary to process a C++ program.

The following steps, as shown in Figure 1-2, are necessary to process a C++ program.

1. You use a text editor to create a C++ program following the rules, or syntax, of the high-level language. This program is called the source code, or source program. The program must be saved in a text file that has the extension .cpp. For example, if you saved the preceding program in the file named FirstCPPProgram, then its complete name is FirstCPPProgram.cpp.
2. The C++ program given in the preceding section contains the statement #include . In a C++ program, statements that begin with the symbol # are called **preprocessor directives**. These statements are processed by a program called pre-processor.
3. After processing preprocessor directives, the next step is to verify that the program obeys the rules of the programming language—that is, the program is syntactically correct—and translate the program into the equivalent machine language. The compiler checks the source program for syntax errors and, if no error is found, translates the program into the equivalent machine language. The equivalent machine language program is called an object program. Object program: The machine language version of the high-level language program.
4. The programs that you write in a high-level language are developed using an **integrated development environment** (IDE). The IDE contains many programs that are useful in creating your program. For example, it contains the necessary code (program) to display the results of the program and several mathematical functions to make the programmer's job somewhat easier. Therefore, if certain code is already available, you can use this code rather than writing your own code. Once the program is developed and successfully compiled, you must still bring the code for the resources used from the IDE into your program to produce a final program that the computer can execute.

   This prewritten code (program) resides in a place called the library. A program called a linker combines the object program with the programs from libraries.

   Linker: A program that combines the object program with other programs in the library and is used in the program to create the executable code.
5. You must next load the executable program into main memory for execution. A program called a loader accomplishes this task.
6. The final step is to execute the program.

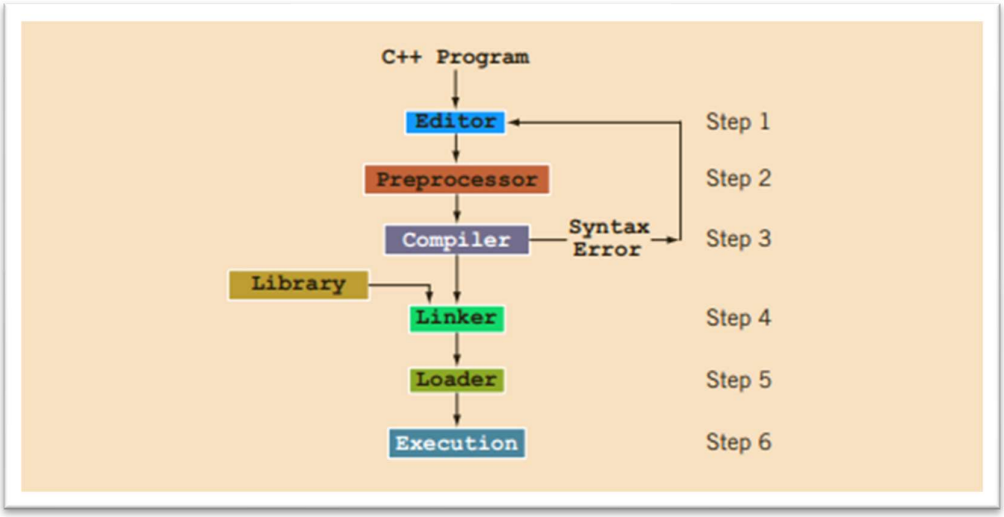


FIGURE 1-2 Processing a C++ program

QUICK REVIEW

1. One kilobyte (KB) is 2 5 1024 bytes 10; one megabyte (MB) is 215,048,576 20 bytes; one gigabyte (GB) is 215,073,741,824 30 bytes; one terabyte (TB) is 215,099,511,627,776 40 bytes; one petabyte (PB) is 21 5,125,899,906,842,624 50 bytes; one exabyte (EB) is 215,152,921,504,606,846,976 60 bytes; and one zettabyte (ZB) is 215,180,591,620,717,411,303,424 70 bytes.
2. An algorithm is a step-by-step problem-solving process in which a solution is arrived at in a finite amount of time.
3. The problem-solving process has three steps: analyze the problem and design an algorithm, implement the algorithm in a programming language, and maintain the program.
4. In structured design, a problem is divided into smaller subproblems. Each subproblem is solved, and the solutions to all of the subproblems are then combined to solve the problem.
5. Linker links the object code with other programs provided by the integrated development environment (IDE) and used in the program to produce executable code.

EXERCISES

1. The dealer's cost of a car is 85% of the listed price. The dealer would accept any offer that is at least $500 over the dealer's cost. Design an algorithm that prompts the user to input the list price of the car and print the least amount that the dealer would accept for the car.

2. The volume of a sphere is $(4.0 / 3.0)\pi r^3$ and the surface area is $4.0\pi r^2$, where r is the radius of the sphere. Given the radius, design an algorithm that computes the volume and surface area of the sphere. Also using the C11 statements provided for Example 1-1, write the C++ statement corresponding to each statement in the algorithm. (You may assume that pi = 3.141592.)

TOPIC 2:
INTRODUCTION TO C++

LEARNING OUTCOMES:

After studying this topic you should be able to:

•Become familiar with the basic components of a C++ program, including functions, special symbols, and identifiers.
•Learn about data types & variable declaration.
•Become familiar with the string data type.
•Become familiar with the use of increment and decrement operators.
•Learn how to debug syntax errors.
•Become familiar with compound statements.

2.1. INTRODUCTION

A computer program, or a program, is a sequence of statements whose objective is to accomplish a task. Programming is a process of planning and creating a program.

You might gain a better grasp of the nature of programming from an analogy, so let us turn to a topic about which almost everyone has some knowledge—cooking. A recipe is also a program, and everyone with some cooking experience can agree on the following:

1. It is usually easier to follow a recipe than to create one.
2. There are good recipes and there are bad recipes.
3. Some recipes are easy to follow and some are not easy to follow.
4. Some recipes produce reliable results and some do not.
5. You must have some knowledge of how to use cooking tools to follow a recipe to completion.
6. To create good new recipes, you must have a lot of knowledge and a good understanding of cooking.

These same six points are also true about programming

2.1.1 BASICS OF A C++ PROGRAM

In this chapter, you will learn the basic elements and concepts of the C++ programming language to create C++ programs. In addition to giving examples to illustrate various concepts, we will also show C++ programs to clarify these concepts. In this section, we provide an example of a C++ program that computes the perimeter and area of a rectangle. At this point you need not be too concerned with the details of this program. You only need to know the effect of an output statement, which is introduced in this program.

In Example 1-1 (Chapter 1), we designed an algorithm to find the perimeter and area of a rectangle. Given the length and width of a rectangle, the C++ program, in Example 2-1, computes and displays the perimeter and area.

EXAMPLE 2-1

```
/*************************************************************
/ Given the length and width of a rectangle, this C++ program
/ computes and outputs the perimeter and area of the rectangle.
/*************************************************************

include <iostream>

sing namespace std;

nt main()

   double length;
   double width;
   double area;
   double perimeter;

   cout << "Program to compute and output the perimeter and "
        << "area of a rectangle." << endl;

   length = 6.0;
   width = 4.0;
   perimeter = 2 * (length + width);
   area = length * width;

   cout << "Length = " << length << endl;
   cout << "Width =  " << width << endl;
   cout << "Perimeter = " << perimeter << endl;
   cout << "Area = " << area << endl;

   return 0;
```

Figure 2-1: C++ Program to calculate Perimeter and area of triangle

Sample Run: (When you compile and execute this program, the following five lines are displayed on the screen.)

```
Program to compute and output the perimeter and area of a rectangle.
Length = 6
Width = 4
Perimeter = 20
Area = 24
```

These lines are displayed by the execution of the following statements:

cout << "Program to compute and output the perimeter and " << "area of a rectangle." << endl;
cout<<"Length="<<length<<endl;
cout<<"Width="<<width<<endl;
cout << "Perimeter = " << perimeter << endl; cout << "Area = " << area << endl;

Next we explain how this happens. Let us first consider the following statement:

cout << "Program to compute and output the perimeter and " << "area of a

rectangle."<<endl;

This is an example of a C++ output statement. It causes the computer to evaluate the expression after the pair of symbols << and display the result on the screen.

A C++ program can contain various types of expressions such as arithmetic and strings. For example, **length + width** is an arithmetic expression. Anything in double quotes is a *string*. For example, ***"Program to compute and output the perimeter and "*** is a string. Similarly***, "area of a rectangle."*** is also a string. Typically, a string evaluates to itself. Arithmetic expressions are evaluated according to rules of arithmetic operations, which you typically learn in an arithmetic course. Later in this chapter, we will explain how arithmetic expressions and strings are formed and evaluated.

Also note that in an output statement, **endl** *causes the insertion point to move to the beginning of the next line.* (Note that in endl, the last letter is lowercase el. Also, on the screen, the insertion point is where the cursor is.) Therefore, the preceding statement causes the system to display the following line on the screen.

2.1.2 Program to compute and output the area and perimeter of a rectangle.

The last statement, that is, ***return 0*;** returns the value 0 to the operating system when the program terminates.

Before ending this section, let us identify certain parts of the C++ program in Figure 2-1.

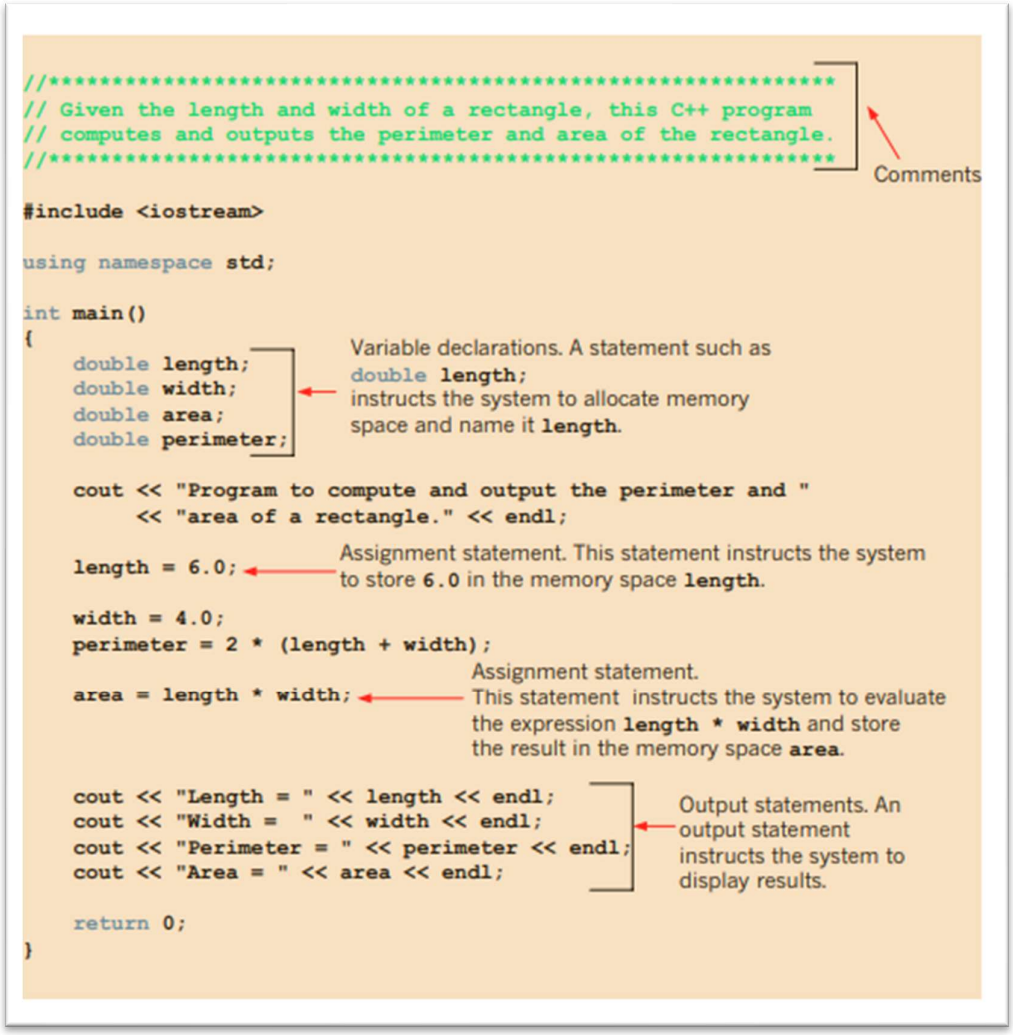


Figure 2-2: Various parts of a C++ Program

2.1.3 Special Symbols & Identifiers

In the previous section, we gave an example of a C++ program and also identified certain parts of the program. In general, a C++ program is a collection of one or more subprograms, called functions. Roughly speaking, a subprogram or a function is a collection of statements, and when it is activated, or executed, it accomplishes something. Some functions, called predefined or standard functions, are already written and are provided as part of the system. But to accomplish most tasks, programmers must learn to write their own functions. Every C++ program has a function called **main**. Thus, if a C++ program has only one function, it must be the function main. Until later Chapters, other than using some of the predefined functions,

you will mainly deal with the function main. By the end of this chapter, you will have learned how to write programs consisting only of the function **main**.

To write meaningful programs, you must learn the programming language's special symbols, words, and syntax rules. The **syntax rules** tell you which statements (instructions) are legal or valid, that is, which are accepted by the programming language and which are not. You must also learn **semantic rules**, which determine the meaning of the instructions. The programming language's rules, symbols, and special words enable you to write programs to solve problems. **Programming language**: A set of rules, symbols, and special words.

Special Symbols

The smallest individual unit of a program written in any language is called a **token**. C++'s tokens are divided into special symbols, word symbols, and identifiers. Following are some of the special symbols:

```
+ - * /
. ; ? ,
<= != == >=
```

The first row includes mathematical symbols for addition, subtraction, multiplication, and division. The second row consists of punctuation marks taken from English grammar. Note that the comma is also a special symbol. In C++, commas are used to separate items in a list. Semicolons are also special symbols and are used to end a C++ statement. Note that a blank, which is not shown above, is also a special symbol. You create a blank symbol by pressing the space bar (only once) on the keyboard. The third row consists of tokens made up of two characters that are regarded as a single symbol. No character can come between the two characters in the token, not even a blank.

Reserved Words (Keywords)

A second category of tokens is reserved word symbols. Some of the reserved word symbols include the following:

int, float, double, char, const, void, return

Reserved words are also called **keywords**. The letters that make up a reserved word are always lowercase. Like the special symbols, each is considered to be a single symbol. Furthermore, reserved words cannot be redefined within any program; that is, they cannot be used for anything other than their intended use.

Identifiers

Identifier: A C++ identifier consists of letters, digits, and the underscore character (_) and must begin with a letter or underscore.

C++ is case sensitive—uppercase and lowercase letters are considered different. Thus, the identifier `NUMBER` is not the same as the identifier `number`. Similarly, the identifiers `X` and `x` are different.

Two predefined identifiers that you will encounter frequently are cout and cin. You have already seen the effect of **cout**. Later in this chapter, you will learn how **cin,** which is used to input data, works. Unlike reserved words, predefined identifiers can be redefined, but it would not be wise to do so.

In C++, identifiers can be of any length.

EXAMPLE 2-2

The following are legal identifiers in C++:

```
first
conversion
payRate
counter1
```

Illegal Identifier	Reason	A Correct Identifier
`employee Salary`	There can be no space between `employee` and `Salary`.	`employeeSalary`
`Hello!`	The exclamation mark cannot be used in an identifier.	`Hello`
`one + two`	The symbol + cannot be used in an identifier.	`onePlusTwo`
`2nd`	An identifier cannot begin with a digit.	`second`

NOTE

Compiler vendors usually begin certain identifiers with an underscore (_). When the linker links the object program with the system resources provided by the integrated development environment (IDE), certain errors could occur. Therefore, it is advisable that you should not begin identifiers in your program with an underscore (_).

Whitespaces

Every C++ program contains whitespaces. Whitespaces include blanks, tabs, and newline characters. In a C++ program, whitespaces are used to separate special symbols, reserved words, and identifiers. Whitespaces are nonprintable in the sense that when they are printed on a white sheet of paper, the space between special symbols, reserved words, and identifiers is white. Proper utilization of whitespaces in a program is important. They can be used to make the program more readable

2.2 DATA TYPES

2.2.1 C++ Data Types

The objective of a C++ program is to manipulate data. Different programs manipulate different data. A program designed to calculate an employee's paycheck will add, subtract, multiply, and divide numbers, and some of the numbers might represent

hours worked and pay rate. Similarly, a program designed to alphabetize a class list will manipulate names.

Data type: A set of values together with a set of allowed operations. C++ data types fall into the following three categories:

- Simple data type
- Structured data type
- Pointers

2.2.1.1 Simple Data Types

The simple data type is the fundamental data type in C++ because it becomes a building block for the structured data type, which you will start learning about in later Chapters. There are three categories of simple data:

- Integral, which is a data type that deals with integers, or numbers without a decimal part
- Floating point, which is a data type that deals with decimal numbers
- Enumeration, which is a user-defined data type

Integral Data Types

Table below gives the range of possible values associated with some integral data types and the size of memory allocated to manipulate these values.

Data Type	Values	Storage (in bytes)
`int`	−2147483648 (= -2^{31}) to 2147483647 (= $2^{31} - 1$)	4
`bool`	`true` and `false`	1
`char`	−128 (= -2^7) to 127 (= $2^7 - 1$)	1
`long long`	−9223372036854775808 (-2^{63}) to 9223372036854775807($2^{63} - 1$)	64

NOTE

Use this table only as a guide. Different compilers may allow different ranges of values. Check your compiler's documentation. To find the exact size of the integral data types on a particular system, you can run a program given in Appendix G (Memory Size of a System). Furthermore, to find the maximum and minimum values of these data types, you can run another program given in Appendix F (Header File **`climits`**). Also, the data type `long long` is not available in C++ standards prior to C++11.

Int Data Types

This section describes the int data type. This discussion also applies to other integral data types. Integers in C++, as in mathematics, are numbers such as the following: -6728, -67, 0, 78, 36782, +763 Note the following two rules from these examples: 1. Positive integers do not need a + sign in front of them. 2. No commas are used within an integer. Recall that in C++, commas are used to separate items in a list. So 36,782 would be interpreted as two integers: 36 and 782.

bool Data Types

The data type bool has only two values: true and false. Also, true and false are called the logical (Boolean) values. The central purpose of this data type is to manipulate logical (Boolean) expressions.

Floating-Point Data Types

To deal with decimal numbers, C++ provides the floating-point data type, which we discuss in this section.

To facilitate the discussion, let us review a concept from a high school or college algebra course. You may be familiar with scientific notation. For example:

43872918 = 4.3872918 * 107 {10 to the power of seven}
.0000265 = 2.65 * 10-5 {10 to the power of minus five}
47.9832 = 4.79832 * 101 {10 to the power of one}

To represent decimal numbers, C++ uses a form of scientific notation called **floating-point notation**. In the C++ floating-point notation, the letter **E** stands for the exponent.

C++ provides three data types to manipulate decimal numbers: float, double, and long double. As in the case of integral data types, the data types float, double, and long double differ in the set of values it can represent.
float: The data type float is used in C++ to represent any decimal number between -3.4 * 1038 and 3.4 * 1038. The memory allocated for a value of the float data type is four bytes. double:
The data type double is used in C++ to represent any decimal number between -1.7 * 10308 and 1.7 * 10308. The memory allocated for a value of the double data type is eight bytes. The maximum and minimum values of the data types float and double are system dependent. To find these values on a particular system, you can check your compiler's documentation or, alternatively, you can run a program given in Appendix F (Header File cfloat). Other than the set of values, there is one more difference between the data types float and double. The maximum number of significant digits—that is, the number of decimal places—in float values is six or seven.

In C++, by default, floating-point numbers are considered type `double`. Therefore, if you use the data type `float` to manipulate floating-point numbers in a program, certain compilers might give you a warning message, such as "truncation from double to float." To avoid such warning messages, you should use the `double` data type. For illustration purposes and to avoid such warning messages in programming examples, this book mostly uses the data type `double` to manipulate floating-point numbers.

The maximum number of significant digits is called the **precision**. Sometimes float values are called single precision, and values of type double are called double

precision. If you are dealing with decimal numbers, for the most part you need only the float type; if you need accuracy to more than six or seven decimal places, you can use the double type.

String Data Type

The data type string is a programmer-defined data type. It is not directly available for use in a program like the simple data types discussed earlier. To use this data type, you need to access program components from the library, which will be discussed later in this chapter. The data type string is a feature of ANSI/ISO Standard C++.

Prior to the ANSI/ISO C++ language standard, the standard C++ library did not provide a **string** data type. Compiler vendors often supplied their own programmer-defined **string** type, and the syntax and semantics of string operations often varied from vendor to vendor.

A string is a sequence of zero or more characters. Strings in C++ are enclosed in double quotation marks. A string containing no characters is called a null or empty string. The following are examples of strings. Note that "" is the empty string. "William Jacob" "Mickey" "" Every character in a string has a relative position in the string. The position of the first character is 0, the position of the second character is 1, and so on. The length of a string is the number of characters in it. When determining the length of a string, you must also count any spaces in the string.

String	Position of a Character in the String	Length of the String
`"William Jacob"`	Position of `'W'` is 0. Position of the first `'i'` is 1. Position of `' '` (the space) is 7. Position of `'J'` is 8. Position of `'b'` is 12.	13
`"Mickey"`	Position of `'M'` is 0. Position of `'i'` is 1. Position of `'c'` is 2. Position of `'k'` is 3. Position of `'e'` is 4. Position of `'y'` is 5.	6

The length of the following string is 22.

```
"It is a beautiful day."
```

2.2.3 Arithmetic Operators

One of the most important uses of a computer is its ability to calculate. You can use the standard arithmetic operators to manipulate integral and floating-point data types. There are five arithmetic operators:

Arithmetic Operators: + (addition), - (subtraction or negation), * (multiplication), / (division), % (mod, (modulus or remainder))

These operators work as follows:

- You can use the operators +, -, *, and / with both integral and floating point data types. These operators work with integral and floating-point data the same way as you learned in a college algebra course.
- When you use / with the integral data type, it gives the quotient in ordinary division. That is, integral division truncates any fractional part; there is no rounding.
- You use % with only the integral data type, to find the remainder in

Example 2-3 shows how the operators / and % work with the integral data types.

EXAMPLE 2-3

Arithmetic Expression	Result	Description
`5 / 2`	2	In the division `5 / 2`, the quotient is `2` and the remainder is `1`. Therefore, `5 / 2` with the integral operands evaluates to the quotient, which is `2`.
`14 / 7`	2	In the division `14 / 7`, the quotient is `2`.
`34 % 5`	4	In the division `34 / 5`, the quotient is `6` and the remainder is `4`. Therefore, `34 % 5` evaluates to the remainder, which is `4`.
`4 % 6`	4	In the division `4 / 6`, the quotient is `0` and the remainder is `4`. Therefore, `4 % 6` evaluates to the remainder, which is `4`.

In the following example, we illustrate how to use the operators / and % with integral data types.

EXAMPLE 2-4

Given length in inches, we write a program that determines and outputs the equivale
ength in feet and (remaining) inches. Now there are 12 inches in a foot. Therefo
100 inches equals 8 feet and 4 inches; similarly, 55 inches equals 4 feet and 7 inch
Note that 100 / 12 = 8 and 100 % 12 = 4; similarly, 55 / 12 = 4 and 55 % 12 =
From these examples, it follows that we can effectively use the operators / and %
ıccomplish our task. The desired program is as follows:

```
// Given length in inches, this program outputs the equivalent
// length in feet and remaining inch(es).

#include <iostream>

using namespace std;

int main()
{
    int inches; //variable to store total inches
    inches = 100; //store 100 in the variable inches
    cout << inches << " inch(es) = ";  //output the value of
                                       //inches and the equal sign
    cout << inches / 12 << " feet (foot) and "; //output maximum
                                        //number of feet (foot)
    cout << inches % 12 << " inch(es)" << endl; //output
                                        //remaining inches
    return 0;
}
```

Sample run:

```
100 inch(es) = 8 feet (foot) and 4 inch(es)
```

Note that each time you run this program, it will output the value of 100 inches. To convert some other value of inches, you need to edit this program and store a different value in the variable inches, which is not very convenient. Later in this chapter we will illustrate how to include statements in a program that will instruct the user to enter different values. However, if you are curious to know at this point, then replace the statement

inches = 100; *//store 100 in the variable inches*

with the following statements and rerun the program:

cout << "Enter total inches and press Enter: "; *//prompt*
//the user to enter total inches

cin >> inches; *//store the value entered by the user //into the variable inches*

cout << endl;

Consider the following expressions, which you have been accustomed to working with since high school: **-5, 8 -7, 3 + 4, 2 + 3 * 5, 5.6 + 6.2 * 3**, and **x + 2 * 5 + 6 / y,** where

x and **y** are unknown numbers. These are examples of **arithmetic expressions**. The numbers appearing in the expressions are called **operands.** The numbers that are used to evaluate an operator are called the operands for that operator. In expression -5, the symbol - specifies that the number 5 is negative. In this expression, - has only one operand. Operators that have only one operand are called **unary operators**. In expression 8 -7, the symbol - is used to subtract 7 from 8. In this expression, - has two operands, 8 and 7. Operators that have two operands are called **binary operators**.
Unary operator: An operator that has only one operand.
Binary operator: An operator that has two operands.

Order of Precedence

When more than one arithmetic operator is used in an expression, C++ uses the operator precedence rules to evaluate the expression. According to the order of precedence rules for arithmetic operators, *, /, and % are at a higher level of precedence than + and - Note that the operators *, /, and % have the same level of precedence.

Similarly, the operators + and - have the same level of precedence. When operators have the same level of precedence, the operations are performed from left to right. To avoid confusion, you can use parentheses to group arithmetic expressions. For example, using the order of precedence rules,

3 * 7 - 6 + 2 * 5 / 4 + 6

means the following:

(((3 * 7) - 6) +((2 * 5) / 4)) + 6
= ((21 - 6) + (10 / 4)) + 6 (Evaluate *)
= ((21 - 6) + 2) + 6 (Evaluate /. Note that this is an integer division.)
= (15 + 2) + 6 (Evaluate -)
= 17 + 6 (Evaluate first +)
= 23 (Evaluate +)

Note that the use of parentheses in the second example clarifies the order of precedence. You can also use parentheses to override the order of precedence rules. Because arithmetic operators, using the precedence rules, are evaluated from left to

right, unless parentheses are present, the associativity of the arithmetic operators is said to be from left to right.

2.3 VARIABLE DECLARATION

Earlier in this chapter, we introduced the term variable and how to declare it. We now review this concept and also give the general syntax to declare variables. In some programs, data needs to be modified during program execution. For example, after each test, the average test score and the number of tests taken changes. Similarly, after each pay increase, the employee's salary changes. This type of data must be stored in those memory cells whose contents can be modified during program execution. In C++, memory cells whose contents can be modified during program execution are called variables.

Variable: A memory location whose content may change during program execution. The syntax for declaring one variable or multiple variables is:

```
dataType identifier, identifier, . . .;
```

2.3.1. Putting Data into Variables

Now that you know how to declare variables, the next question is: How do you put data into those variables? In C++, you can place data into a variable in two ways:

1. Use C++'s assignment statement.
2. Use input (read) statements.

```
variable = expression;
```

In an assignment statement, the value of the expression should match the data type of the variable.

The expression on the right side is evaluated, and its value is assigned to the variable (and thus to a memory location) on the left side. A variable is said to be initialized the first time a value is placed in the variable.

Suppose you have the following variable declarations:

```
int num1, num2;
```

```
double sale;
char first;
string str;
```

Now consider the following assignment statements:

```
num1 = 4;
num2 = 4 * 5 - 11;
sale = 0.02 * 1000;
first = 'D';
str = "It is a sunny day.";
```

For each of these statements, the computer first evaluates the expression on the right and then stores that value in a memory location named by the identifier on the left. The first statement stores the value 4 in num1, the second statement stores 9 in num2, the third statement stores 20.00 in sale, and the fourth statement stores the character D in first. The fifth statement assigns the string "It is a sunny day." to the variable str

EXAMPLE

Suppose that **num1, num2,** and **num3** are int variables and the following statements are executed in sequence.

1. num1 = 18;
2. num1 = num1 + 27;
3. num2 = num1;
4. num3 = num2 / 5;
5. num3 = num3 / 4;

The following table shows the values of the variables after the execution of each statement. (A **?** indicates that the value is unknown. The orange color in a box shows that the value of that variable is changed.

	Values of the Variables/Statement	Explanation
Before Statement 1	num1 = ?, num2 = ?, num3 = ?	
After Statement 1	num1 = 18, num2 = ?, num3 = ? `num1 = 18;`	
After Statement 2	num1 = 45, num2 = ?, num3 = ? `num1 = num1 + 27;`	`num1 + 27 = 18 + 27 = 45`. This value is assigned to `num1`, which replaces the old value of `num1`.
After Statement 3	num1 = 45, num2 = 45, num3 = ? `num2 = num1;`	Copy the value of `num1` into `num2`.
After Statement 4	num1 = 45, num2 = 45, num3 = 9 `num3 = num2 / 5;`	`num2 / 5 = 45 / 5 = 9`. This value is assigned to `num3`. So `num3 = 9`.
After Statement 5	num1 = 45, num2 = 45, num3 = 2 `num3 = num3 / 4;`	`num3 / 4 = 9 / 4 = 2`. This value is assigned to num3, which replaces the old value of `num3`.

Thus, after the execution of the statement in Line 5, num1 = 45, num2 = 45, and num3 = 2.

Tracing values through a sequence, called a **walk-through**, is a valuable tool to learn and practice. Try it in the sequence above.

NOTE

Suppose that `x`, `y`, and `z` are `int` variables. The following is a legal statement in C++:

```
x = y = z;
```

In this statement, first the value of `z` is assigned to `y`, and then the new value of `y` is assigned to `x`. Because the assignment operator, `=`, is evaluated from right to left, the **associativity** of the **assignment operator** is said to be from right to left.

2.3.2. Declaring and Initializing Variables

If you only declare a variable and do not instruct the computer to put data into the variable, the value of that variable is garbage. However, the computer does not warn us, regards whatever values are in memory as legitimate, and performs calculations using those values in memory. Using a variable in an expression without initializing it produces erroneous results. To avoid these pitfalls, C++ allows you to initialize

variables while they are being declared. For example, consider the following C++ statements in which variables are first declared and then initialized:

```
int first, second;
char ch; double x;
first = 13; second = 10;
ch = ' ';
 x = 12.6;
```

You can declare and initialize these variables at the same time using the following C++ statements:

```
int first = 13, second = 10;
 char ch = ' ';
double x = 12.6;
```

The first C++ statement declares two int variables, first and second, and stores 13 in first and 10 in second. The meaning of the other statements is similar. In reality, not all variables are initialized during declaration. It is the nature of the program or the programmer's choice that dictates which variables should be initialized during declaration. The key point is that all variables must be initialized before they are used.

2.3.3. Input (Read) Statement

When the computer gets the data from the keyboard, the user is said to be acting interactively. Putting data into variables from the standard input device is accomplished via the use of cin and the operator >>. The syntax of cin together with >> is:

```
cin >> variable >> variable ...;
```

This is called an **input (read)** statement. In C++, >> is called the **stream extraction operator.**

The C++ program in Example 2-18 illustrates how to read strings and numeric data.

EXAMPLE

```
// This program illustrates how to read strings and numeric data.
#include <iostream>                                    //Line 1
#include <string>                                      //Line 2

using namespace std;                                   //Line 3

int main()                                             //Line 4
{                                                      //Line 5
    string firstName;                                  //Line 6
    string lastName;                                   //Line 7
    int age;                                           //Line 8
    double weight;                                     //Line 9

    cout << "Enter first name, last name, age, "       //Line 10
         << "and weight, separated by spaces."         //Line 11
         << endl;                                      //Line 12

    cin >> firstName >> lastName;                      //Line 13
    cin >> age >> weight;                              //Line 14

    cout << "Name: " << firstName << " "               //Line 15
         << lastName << endl;                          //Line 16

    cout << "Age: " << age << endl;                    //Line 17
```

```
    cout << "Weight: " << weight << endl;              //Line 18

    return 0;                                          //Line 19
}                                                      //Line 20
```

Sample Run: (In this sample run, the user input is shaded.)

```
Enter first name, last name, age, and weight, separated by spaces.
Sheila Mann 23 120.5
Name: Sheila Mann
Age: 23
Weight: 120.5
```

The preceding program works as follows: The statements in Lines 6 to 9 declare the variables firstName and lastName of type string, age of type int, and weight of type double. The statement in Lines 10, 11, and 12 is an output statement and tells the user what to do. (Such output statements are called prompt lines.)

As shown in the sample run, the input to the program is:

Sheila Mann 23 120.5

The statement in Line 13 first reads and stores the string Sheila into the variable firstName and then skips the space after Sheila and reads and stores the string Mann into the variable lastName. Next, the statement in Line 14 first skips the blank after Mann and reads and stores 23 into the variable age and then skips the blank after 23

and reads and stores 120.5 into the variable weight. The statements in Lines 15, 16, 17, and 18 produce the third, fourth, and fifth lines of the sample run.

2.3.4. Type Conversion (Casting)

In the previous section, you learned that when evaluating an arithmetic expression, if the operator has mixed operands, the integer value is changed to a floating-point value with the zero decimal part. When a value of one data type is automatically changed to another data type, an implicit type coercion is said to have occurred. As the examples in the preceding section illustrate, if you are not careful about data types, implicit type coercion can generate unexpected results. To avoid implicit type coercion, C++ provides for explicit type conversion through the use of a cast operator. The cast operator, also called type conversion or type casting, takes the following form: static_cast(expression) First, the expression is evaluated. Its value is then converted to a value of the type specified by dataTypeName. In C++, static_cast is a reserved word.

EXAMPLE

Expression	Evaluates to
`static_cast<int>(7.9)`	`7`
`static_cast<int>(3.3)`	`3`
`static_cast<double>(25)`	`25.0`
`static_cast<double>(5 + 3)`	`= static_cast<double>(8) = 8.0`

`static_cast<double>(15) / 2`	`= 15.0 / 2`
	(because `static_cast<double> (15) = 15.0`)
	`= 15.0 / 2.0 = 7.5`

`static_cast<double>(15/2)`	`= static_cast<double>(7)` (because `15 / 2 = 7`) `= 7.0`

`static_cast<int>(7.8 +`
`static_cast<double>(15)/2)`
	`= static_cast<int>(7.8 + 7.5)`
	`= static_cast<int>(15.3)`
	`= 15`

`static_cast<int>(7.8 +`
`static_cast<double>(15/2))`
	`= static_cast<int>(7.8 + 7.0)`
	`= static_cast<int>(14.8)`
	`= 14`

In C++, the cast operator can also take the form `dataType(expression)`. This form is called C-like casting. For example, `double(5) = 5.0` and `int(17.6) = 17`. However, `static_cast` is more stable than C-like casting.

You can also use cast operators to explicitly convert char data values into int data values and int data values into char data values. To convert char data values into int data values, you use a collating sequence. For example, in the ASCII character set, static_cast('A') is 65 and static_cast('8') is 56. Similarly, static_cast(65) is 'A' and static_cast(56) is '8'.

The following C++ program evaluates the preceding expressions:

```
// This program illustrates how explicit type conversion works.
#include <iostream>
using namespace std;
int main()
{
    cout << "static_cast<int>(7.9) = "
         << static_cast<int>(7.9)
         << endl;
    cout << "static_cast<int>(3.3) = "
         << static_cast<int>(3.3)
         << endl;
    cout << "static_cast<double>(25) = "
         << static_cast<double>(25)
         << endl;
    cout << "static_cast<double>(5 + 3) = "
         << static_cast<double>(5 + 3)
         << endl;
    cout << "static_cast<double>(15) / 2 = "
         << static_cast<double>(15) / 2
         << endl;
    cout << "static_cast<double>(15 / 2) = "
         << static_cast<double>(15 / 2)
         << endl;
    cout << "static_cast<int>(7.8 + static_cast<double>(15) / 2) = "
         << static_cast<int>(7.8 + static_cast<double>(15) / 2)
         << endl;

    cout << "static_cast<int>(7.8 + static_cast<double>(15 / 2)) = "
         << static_cast<int>(7.8 + static_cast<double>(15 / 2))
         << endl;

    return 0;
}
```

Sample Run:

```
static_cast<int>(7.9) = 7
static_cast<int>(3.3) = 3
static_cast<double>(25) = 25
static_cast<double>(5 + 3) = 8
static_cast<double>(15) / 2 = 7.5
static_cast<double>(15 / 2) = 7
static_cast<int>(7.8 + static_cast<double>(15) / 2) = 15
static_cast<int>(7.8 + static_cast<double>(15 / 2)) = 14
```

2.3.5. Debugging: Understanding and Fixing Errors

When you type a program, typos and unintentional syntax errors are likely to occur. Therefore, when you compile a program, the compiler will identify the syntax error. In this section, we show how to identify and fix syntax errors.

Consider the following C++ program:

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int num
8.
9.      num = 18;
10.
11.     tempNum = 2 * num;
12.
13.     cout << "Num = " << num << ", tempNum = " < tempNum << endl;
14.
15.     return ;
16. }
```

(Note that the numbers 1 to 16 on the left side are not part of the program. We have numbered the statements for easy reference.) This program contains syntax errors. When you compile this program, the compiler produces the following errors. (This program is compiled using Microsoft Visual Studio 2015.)

Example_Syntax_Errors.cpp

c:\examplech2_syntax_errors.cpp (9): error C2146: syntax error: missing ';' before identifier 'num' c: examplech2_syntax_errors.cpp (11): error C2065: 'tempNum': undeclared identifier
c:\examplech2_syntax_errors.cpp (13): error C2065: 'tempNum': undeclared identifier
c:\examplech2_syntax_errors.cpp (13): error C2563: mismatch in formal parameter list
c:\examplech2_syntax_errors.cpp (13): error C2568: '<<': unable to resolve function overload
c:\examplech2_syntax_errors.cpp (13): note: could be 'std::basic_ostream &std::endl;(std::basic_ostream &)'
c:\examplech2_syntax_errors.cpp (15): error C2561: 'main': function must return a value c:\examplech2_syntax_errors.cpp (5): note: see declaration of 'main'

It is best to try to correct the errors in top-down fashion because the first error may confuse the compiler and cause it to flag multiple subsequent errors when actually there was only one error on an earlier line.

So, let's first consider the following error: c:\examplech2_syntax_errors.cpp (9): error C2146: syntax error: missing ';' before identifier 'num'

The expression examplech2_syntax_errors.cpp (9) indicates that there is an error in Line 9. The remaining part of this error specifies that there is a missing ; before the identifier num.

If we look at Line 7, we find that there is a missing semicolon at the end of the statement int num. Therefore, we must insert ; at the end of the statement in Line 7.

Next, consider the second error:

c: examplech2_syntax_errors.cpp (11): error C2065: 'tempNum': undeclared identifier

This error occurs in Line 11, and it specifies that the identifier tempNum is undeclared. When we look at the code, we find that this identifier has not been declared. So we must declare tempNum as an int variable.

The error: c:\examplech2_syntax_errors.cpp (13): error C2065: 'tempNum': undeclared identifier occurs in Line 13, and it specifies that the identifier tempNum is undeclared.

As in the previous error, we must declare tempNum. Note that once we declare tempNum and recompile, this and the previous error will disappear.

The next error is: c:\examplech2_syntax_errors.cpp (13): error C2563: mismatch in formal parameter list This error occurs in Line 13, and it indicates that some formal parameter list is mismatched. For a beginner, this error is somewhat hard to understand.

However, as you practice, you will learn how to interpret and correct syntax errors. This error becomes clear if you look at the next error, part of which is: c:\examplech2_syntax_errors.cpp (13): error C2568: '<<': unable to resolve function overload

It tells us that this error has something to do with the operator <<< "Num = " << num << ", tempNum = " < tempNum << endl

Let us look at the last error, which is:

c:\examplech2_syntax_errors.cpp (15): error C2561: 'main': function must return a value c:\examplech2_syntax_errors.cpp (5): note: see declaration of 'main'

This error occurs in Line 15. However, at this point, the explanation given, especially for a beginner, is somewhat unclear. However, if you look at the statement return ; in Line 15 and remember the syntax of the function main as well as all the programs given in this book, we find that the number 0 is missing, that is, this statement must be return 0;

After correcting all of the syntax errors, a correct program is as follows:

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

    num - 18;

    tempNum = 2 * num;

    cout << "Num = " << num << ", tempNum = " << tempNum << endl;

    return 0;
}
```

The output is:

```
Num = 18, tempNum = 36
```

PROGRAMMING EXAMPLE: Convert Length

Write a program that takes as input given lengths expressed in feet and inches. The program should then convert and output the lengths in centimeters. Assume that the given lengths in feet and inches are integers.

Input: Length in feet and inches.
Output: Equivalent length in centimeters.

Problem Analysis and Algorithm Design

Based on this analysis of the problem, you can design an algorithm as follows:
1. Get the length in feet and inches.

2. Convert the length into total inches.
3. Convert total inches into centimeters.
4. Output centimeters.

Complete Program Listing

```
#include <iostream>
using namespace std;

    //named constants
const double CENTIMETERS_PER_INCH = 2.54;
const int INCHES_PER_FOOT = 12;

int main()
{
        //declare variables
    int feet, inches;
    int totalInches;
    double centimeters;

        //Statements: Step 1 - Step 7
    cout << "Enter two integers, one for feet and "
         << "one for inches: ";                          //Step 1
    cin >> feet >> inches;                               //Step 2
    cout << endl;

    cout << "The numbers you entered are " << feet
         << " for feet and " << inches
         << " for inches. " << endl;                     //Step 3

    totalInches = INCHES_PER_FOOT * feet + inches;       //Step 4

    cout << "The total number of inches = "
         << totalInches << endl;                         //Step 5

    centimeters = CENTIMETERS_PER_INCH * totalInches; //Step 6

    cout << "The number of centimeters = "
         << centimeters << endl;                         //Step 7

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded:

```
Enter two integers, one for feet, one for inches: 15 7

The numbers you entered are 15 for feet and 7 for inches.
The total number of inches = 187
The number of centimeters = 474.98
```

QUICK REVIEW

1. C++ data types fall into the following three categories: simple, structured, and pointers.

2. There are three categories of simple data: integral, floating point, and enumeration.
3. Integral data types are classified into the following categories: char, short, int, long, bool, unsigned char, unsigned short, unsigned int, unsigned long, long long, and unsigned long long.
4. The data type float is used in C11 to represent any real number between -3.4 * 1038 and 3.4 * 1038. The memory allocated for a value of the float data type is four bytes.
5. The data type double is used in C11 to represent any real number between -1.7 * 10308 and 1.7 * 10308. The memory allocated for a value of the double data type is eight bytes.
6. Corresponding to the five arithmetic operators +, -, *, /, and %, C++ provides five compound operators: +=, -=, *=, /=, and %=, respectively.
7. The preprocessor command #include instructs the preprocessor to include the header file iostream in the program.
8. Every character in a string has a relative position in the string. The position of the first character is 0, the position of the second character is 1, and so on.
9. The modulus operator, %, takes only integer operands
10. To use cin and cout, the program must include the header file iostream and either include the statement using namespace std; or refer to these identifiers as std::cin and std::cout.

EXERCISES

1. Write a C++ statement that multiplies the value of num1 by 2, adds the value of num2 to it, and then stores the result in newNum. Then, write a C++ statement that outputs the value of newNum.
2. Write a program that prompts the user to enter five test scores and then prints the average test score. (Assume that the test scores are decimal numbers.)
3. A milk carton can hold 3.78 liters of milk. Each morning, a dairy farm ships cartons of milk to a local grocery store. The cost of producing one liter of milk is $0.38, and the profit of each carton of milk is $0.27. Write a program that does the following: a. Prompts the user to enter the total amount of milk produced in the morning. b. Outputs the number of milk cartons needed to hold milk. (Round

your answer to the nearest integer.) c. Outputs the cost of producing milk. d. Outputs the profit for producing milk.

4. For each used car a salesperson sells, the commission is paid as follows: $20 plus 30% of the selling price in excess of the cost of the car. Typically, the minimum selling price of the car is the cost of the car plus $200 and the maximum selling price is the cost of the car and $2,000. Write a program that prompts the user to enter the salesperson's fixed commission, the percentage of the commission, the purchasing cost of the car, the minimum and maximum amount to be added to the purchasing cost to determine the minimum and maximum selling price, and outputs minimum and maximum selling price of the car and the salesperson's commission range.

TOPIC 3
CONTROL STRUCTURES

LEARNING OUTCOMES:

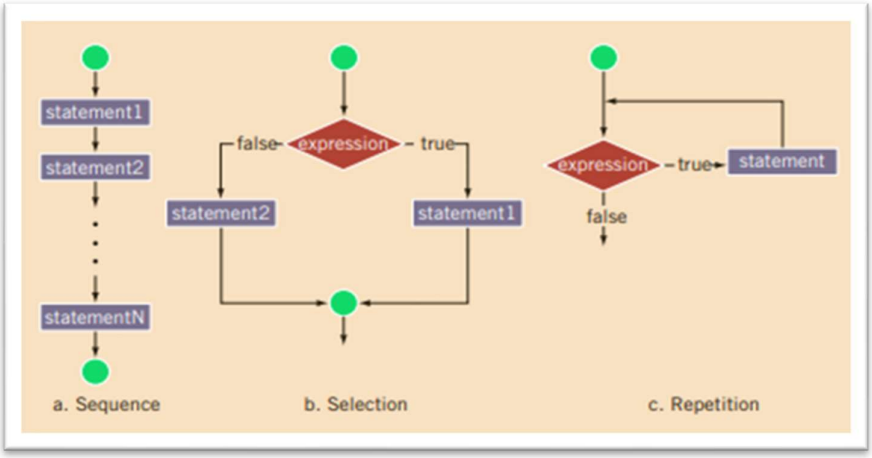
After studying this topic you should be able to:

- Learn about control structures
- Discover how to use the selection control structures if and if ... else
- Become aware of short-circuit evaluation
- Learn how the conditional operator, ?:, works
- Learn how to use pseudocode to develop, test, and debug a program
- Discover how to use a switch statement in a program
- Learn how to avoid bugs by avoiding partially understood concepts
- Learn about repetition (looping) control structures
- Explore how to construct and use count controlled, sentinel controlled, flag controlled, EOF controlled repetition structures
- Learn how to use a for loop in a program
- Examine break and continue statements

3.1 INTRODUCTION

A computer can process a program in one of the following ways: in sequence; selectively, by making a choice, which is also called a branch; repetitively, by executing a statement over and over, using a structure called a loop; or by calling a function.

Control structures provide alternatives to sequential program execution and are used to alter the sequential flow of execution. The two most common control structures are selection and repetition. In selection, the program executes particular statements depending on some condition(s). In repetition, the program repeats particular statements a certain number of times based on some condition(s)



3.2 SELECTION: if AND if….else

Logical expression: An expression that evaluates to true or false is called a logical expression

For example, because “8 is greater than 3” is true, the expression 8 > 3 is a logical expression. Note that > is an operator in C++, called the “greater than” and is an example of a relational operator. The Table below lists the C++ relational operators.

Operator	Description
==	equal to
!=	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

NOTE

In C++, the symbol ==, which consists of two equal signs, is called the equality operator. Recall that the symbol = is called the assignment operator. Remember that the equality operator, ==, determines whether two expressions are equal, whereas the assignment operator, =, assigns the value of an expression to a variable.

Each of the relational operators is a binary operator; that is, it requires two operands. Because the result of a comparison is true or false, expressions using these operators always evaluate to true or false.

Relational Operators and Simple Data Types

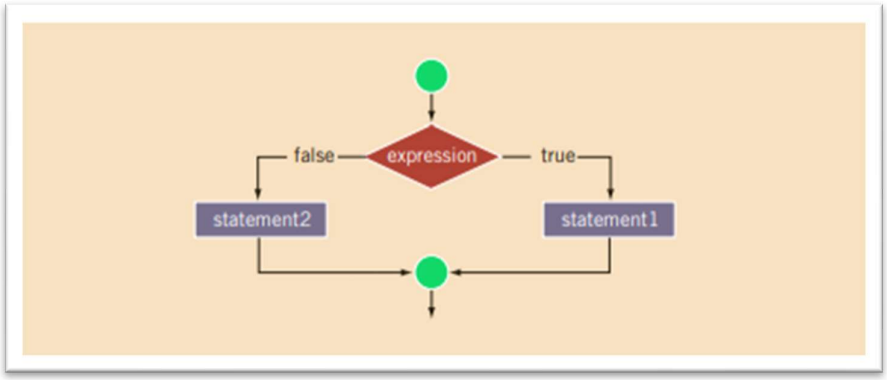
You can use the relational operators with all three simple data types. In the following example, the expressions use both integers and real numbers:

Expression	Meaning	Value
`8 < 15`	`8` is less than `15`	`true`
`6 != 6`	`6` is not equal to `6`	`false`
`2.5 > 5.8`	`2.5` is greater than `5.8`	`false`
`5.9 <= 7.5`	`5.9` is less than or equal to `7.5`	`true`
`7 <= 10.4`	`7` is less than or equal to `10.4`	`true`

3.2.1 Two-Way Selection

There are many programming situations in which you must choose between two alternatives.

Take a moment to examine this syntax. It begins with the reserved word if, followed by a logical expression contained within parentheses, followed by a statement, followed by the reserved word else, followed by a second statement. Statements 1 and 2 are any valid C++ statements. In a two-way selection, if the value of the expression is true, statement1 executes. If the value of the expression is false, statement2 executes.



EXAMPLE

Consider the following statements:

```
if (score >= 60)                    //Line 1
    cout << "Passing" << endl;      //Line 2
    cout << "Failing" << endl;      //Line 3
```

If the expression (`score >= 60`) evaluates to `false`, the output statement in Line 2 does not execute. So the output would be `Failing`. That is, this set of statements performs the same action as an `if...else` statement. It will execute the output statement in Line 3 rather than the output statement in Line 2. For example, if the value of `score` is `50`, these statements will output the following line:

```
Failing
```

However, if the expression (`score >= 60`) evaluates to `true`, the program will execute both of the output statements, giving a very unsatisfactory result. For example, if the value of `score` is `70`, these statements will output the following lines:

```
Passing
Failing
```

The `if` statement controls the execution of only the statement in Line 2. The statement in Line 3 always executes.

Logical(Boolean)Operators and Logical Expressions

The logical expressions used in these examples involve the evaluation of a single relational operator. There are situations when the logical expression is a combination of two or more logical expressions. For example, suppose weight and height are double variables. Consider the following logical expression

weight > 180 and height < 6.0

This logical expression is a combination of the logical expressions weight > 180 and height < 6.0, and these logical expressions are combined using the word "and".

Logical (Boolean) operators enable you to combine logical expressions. C++ has three logical (Boolean)operators.

Operator	Description
!	not
&&	and
||	or

Logical operators take only logical values as operands and yield only logical values as results. The operator ! is unary, so it has only one operand.
The operators && and || are binary operators and there is no space within these operators.

EXAMPLE

Expression	Value	Explanation
`(14 >= 5) || ('A' > 'B')`	`true`	Because `(14 >= 5)` is `true`, `('A' > 'B')` is `false`, and `true || false` is `true`, the expression evaluates to `true`.
`(24 >= 35) || ('A' > 'B')`	`false`	Because `(24 >= 35)` is `false`, `('A' > 'B')` is `false`, and `false || false` is `false`, the expression evaluates to `false`.
`('A' <= 'a') || (7 != 7)`	`true`	Because `('A' <= 'a')` is `true`, `(7 != 7)` is `false`, and `true || false` is `true`, the expression evaluates to `true`.

Comparing if…else Statements with a Series of if Statements

Consider the following C++ program segments, both of which accomplish the same task program segment with if Statements only

```
a.  if (month == 1)                          //Line 1
        cout << "January" << endl;           //Line 2
    else if (month == 2)                     //Line 3
        cout << "February" << endl;          //Line 4
    else if (month == 3)                     //Line 5
        cout << "March" << endl;             //Line 6
    else if (month == 4)                     //Line 7
        cout << "April" << endl;             //Line 8
    else if (month == 5)                     //Line 9
        cout << "May" << endl;               //Line 10
    else if (month == 6)                     //Line 11
        cout << "June" << endl;              //Line 12
```

Program segment (a) is written as a sequence of if. . .else statements; program segment (b) is written as a series of if statements. Both program segments accomplish the same thing. If month is 3, then both program segments output March. If month is 1, then in program segment (a), the expression in the if statement in Line 1 evaluates to true. The statement (in Line 2) associated with this if then executes; the rest of the structure, which is the else of this if statement, is skipped; and the remaining if statements are not evaluated. In program segment (b), the computer has to evaluate the expression in each if statement because there is no else statement. As a consequence, program segment (b) executes more slowly than does program segment (a). In a sequence of if. . .else statements, such as (a), if more than one condition is true, only the statements associated with the first true condition will be executed. On the other hand, in a series of if statements, such as (b), if more than one condition evaluates to true, statements associated with each true condition will execute.

```
b.  if (month == 1)
        cout << "January" << endl;
    if (month == 2)
        cout << "February" << endl;
    if (month == 3)
        cout << "March" << endl;
    if (month == 4)
        cout << "April" << endl;
    if (month = 5)
        cout << "May" << endl;
    if (month == 6)
        cout << "June" << endl;
```

Short-Circuit Evaluation (of a logical expression):

A process in which the computer evaluates a logical expression from left to right and stops as soon as the final value of the expression is known.

EXAMPLE

Consider the following expressions:

```
(age >= 21) || ( x == 5)              //Line 1
(grade == 'A') && (x >= 7)            //Line 2
```

For the expression in Line 1, suppose that the value of `age` is 25. Because `(25 >= 21)` is `true` and the logical operator used in the expression is `||`, the expression evaluates to `true`. Due to short-circuit evaluation, the computer does not evaluate the expression `(x == 5)`. Similarly, for the expression in Line 2, suppose that the value of `grade` is `'B'`. Because `('B' == 'A')` is `false` and the logical operator used in the expression is `&&`, the expression evaluates to `false`. The computer does not evaluate `(x >= 7)`.

Conditional Operator (?)

Certain if. . .else statements can be written in a more concise way by using C++'s conditional operator. The conditional operator, written as ?:, is a ternary operator, which means that it takes three arguments. The syntax for using the conditional operator is:

```
expression1 ? expression2 : expression3
```

This type of expression is called a conditional expression. The conditional expression is evaluated as follows: If expression1 evaluates to a nonzero integer (that is, to true), the result of the conditional expression is expression2. Otherwise, the result of the conditional expression is expression3

Consider the following statements:

```
if (a >= b)
        max = a;
else
        max = b;
```

You can use the conditional operator to simplify the writing of this if. . .else statement as follows: `max = (a >= b) ? a : b;`

3.2.2 Using Pseudocode to Develop, Test and Debug a Program

Sometimes pseudo provides a useful means to outline and refine a program before putting it into formal C++ code. When you are constructing programs that involve complex nested control structures, pseudo can help you quickly develop the correct structure of the program and avoid making common errors. One useful program segment determines the larger of two integers. If x and y are integers, using pseudo, you can quickly write the following:

a. *if (x > y) then*
   x is larger
b. *if (y > x) then*
   y is larger

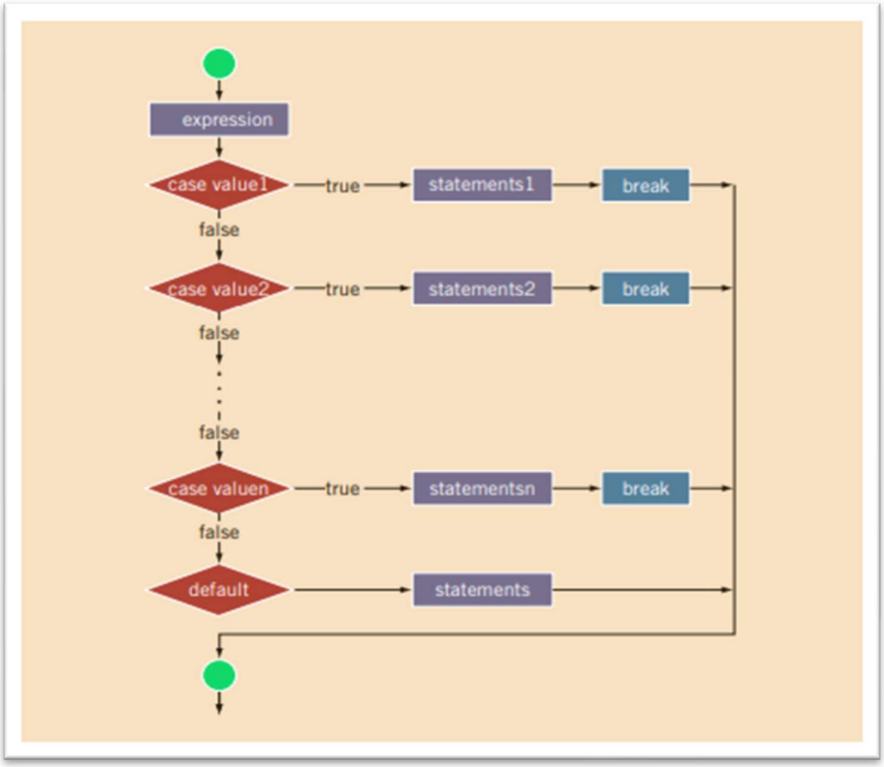
If the statement in (a) is true, then x is larger. If the statement in (b) is true, then y is larger. However, for this code to work in concert to determine the larger of two integers, the computer needs to evaluate both expressions:
(x > y) and (y > x)
even if the first statement is true. Evaluating both expressions when the first one is true is a waste of computer time.

Let's rewrite this pseudo as follows:

if (x > y) then
x is larger
else
y is larger

3.3 switch Structures

Recall that there are two selection, or branch, structures in C++. The first selection structure, which is implemented with if and if. . .else statements, usually requires the evaluation of a (logical) expression. The second selection structure, which does not require the evaluation of a logical expression, is called the switch structure. C++'s switch structure gives the computer the power to choose from among many alternatives.



The switch statement executes according to the following rules:

1. When the value of the expression is matched against a case value (also called a label), the statements execute until either a break statement is found or the end of the switch structure is reached.
2. If the value of the expression does not match any of the case values, the statements following the default label execute. If the switch structure has no default label and if the value of the expression does not match any of the case values, the action of the switch statement is null.
3. A break statement causes an immediate exit from the switch structure.

EXAMPLE

The following program illustrates the effect of the break statement. It asks the user to input a number between 0 and 7.

```
//Program: Effect of break statements in a switch structure

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

int main()                                                   //Line 3
{                                                            //Line 4
    int num;                                                 //Line 5

    cout << "Enter an integer between 0 and 7: ";            //Line 6
    cin >> num;                                              //Line 7


    cout << endl;                                            //Line 8

    switch(num)                                              //Line 9
    {                                                        //Line 10
    case 0:                                                  //Line 11
    case 1:                                                  //Line 12
        cout << "Learning to use ";                          //Line 13
    case 2:                                                  //Line 14
        cout << "C++'s ";                                    //Line 15
    case 3:                                                  //Line 16
        cout << "switch structure." << endl;                 //Line 17
        break;                                               //Line 18
    case 4:                                                  //Line 19
        break;                                               //Line 20
    case 5:                                                  //Line 21
        cout << "This program shows the effect ";            //Line 22
    case 6:                                                  //Line 23
    case 7:                                                  //Line 24
        cout << "of the break statement." << endl;           //Line 25
        break;                                               //Line 26
    default:                                                 //Line 27
        cout << "The number is out of range." << endl;       //Line 28
    }                                                        //Line 29

    cout << "Out of the switch structure." << endl;          //Line 30

    return 0;                                                //Line 31
```

Sample Runs: These outputs were obtained by executing the preceding program several times. In each of these sample runs, the user input is shaded.

Sample Run 1:

```
Enter an integer between 0 and 7: 0

Learning to use C++'s switch structure.
Out of the switch structure.
```

Sample Run 2:

```
Enter an integer between 0 and 7: 2

C++'s switch structure.
Out of the switch structure.
```

PROGRAMMING EXAMPLE: Cable Company Billing

This programming example demonstrates a program that calculates a customer's bill for a local cable company. There are two types of customers: residential and business. There are two rates for calculating a cable bill: one for residential customers and one for business customers. For residential customers, the following rates apply:

- Bill processing fee: $4.50
- Basic service fee: $20.50
- Premium channels: $7.50 per channel

For business customers, the following rates apply:

- Bill processing fee: $15.00
- Basic service fee: $75.00 for first 10 connections, $5.00 for each additional connection
- Premium channels: $50.00 per channel for any number of connections

The program should ask the user for an account number (an integer) and a customer code. Assume that `R` or `r` stands for a residential customer, and `B` or `b` stands for a business customer.

Input The customer's account number, customer code, number of premium channels to which the user subscribes, and, in the case of business customers, number of basic service connections.

Output Customer's account number and the billing amount.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

The purpose of this program is to calculate and print the billing amount. To calculate the billing amount, you need to know the customer for whom the billing amount is calculated (whether the customer is residential or business) and the number of premium channels to which the customer subscribes. In the case of a business customer, you also need to know the number of basic service connections and the number of premium channels. Other data needed to calculate the bill, such as the bill processing fees and the cost of a premium channel, are known quantities. The program should print the billing amount to two decimal places, which is standard for monetary amounts. This problem analysis translates into the following algorithm:

1. Set the precision to two decimal places.
2. Prompt the user for the account number and customer type.
3. Based on the customer type, determine the number of premium channels and basic service connections, compute the bill, and print the bill:
   a. If the customer type is `r` or `R`,
      i. Prompt the user for the number of premium channels.
      ii. Compute the bill.
      iii. Print the bill.
   b. If the customer type is `b` or `B`,
      i. Prompt the user for the number of basic service connections and the number of premium channels.
      ii. Compute the bill.
      iii. Print the bill.

Complete Code in C++

COMPLETE PROGRAM LISTING

```
//*************************************************************
// Author: D. S. Malik
//
// Program: Cable Company Billing
// This program calculates and prints a customer's bill for
// a local cable company. The program processes two types of
// customers: residential and business.
//*************************************************************

#include <iostream>
#include <iomanip>

using namespace std;
```

```
      //Named constants - residential customers
const double RES_BILL_PROC_FEES = 4.50;
const double RES_BASIC_SERV_COST = 20.50;
const double RES_COST_PREM_CHANNEL = 7.50;

      //Named constants - business customers
const double BUS_BILL_PROC_FEES = 15.00;
const double BUS_BASIC_SERV_COST = 75.00;
const double BUS_BASIC_CONN_COST = 5.00;
const double BUS_COST_PREM_CHANNEL = 50.00;

int main()
{
        //Variable declaration
    int accountNumber;
    char customerType;
    int numOfPremChannels;
    int numOfBasicServConn;
    double amountDue;

    cout << fixed << showpoint;                      //Step 1
    cout << setprecision(2);                         //Step 1

    cout << "This program computes a cable "
         << "bill." << endl;

    cout << "Enter account number (an integer): ";  //Step 2
    cin >> accountNumber;                            //Step 3
    cout << endl;

    cout << "Enter customer type: "
         << "R or r (Residential), "
         << "B or b (Business):  ";                  //Step 4
    cin >>  customerType;                            //Step 5
    cout << endl;

    switch (customerType)
    {
    case 'r':                                        //Step 6
    case 'R':
        cout << "Enter the number"
             << " of premium channels: ";            //Step 6a
        cin >> numOfPremChannels;                    //Step 6b
        cout << endl;
```

```
            amountDue = RES_BILL_PROC_FEES                  //Step 6c
                      + RES_BASIC_SERV_COST
                      + numOfPremChannels *
                        RES_COST_PREM_CHANNEL;
```

```
            cout << "Account number: "
                 << accountNumber
                 << endl;                                   //Step 6d
            cout << "Amount due: $"
                 << amountDue
                 << endl;                                   //Step 6d
            break;

        case 'b':                                           //Step 7
        case 'B':
            cout << "Enter the number of basic "
                 << "service connections: ";                //Step 7a
            cin >> numOfBasicServConn;                      //Step 7b
            cout << endl;

            cout << "Enter the number"
                 << " of premium channels: ";               //Step 7c
            cin >> numOfPremChannels;                       //Step 7d
            cout << endl;

            if (numOfBasicServConn <= 10)                   //Step 7e
                amountDue = BUS_BILL_PROC_FEES
                          + BUS_BASIC_SERV_COST
                          + numOfPremChannels *
                            BUS_COST_PREM_CHANNEL;
            else
                amountDue = BUS_BILL_PROC_FEES
                          + BUS_BASIC_SERV_COST
                          + (numOfBasicServConn - 10) *
                             BUS_BASIC_CONN_COST
                          + numOfPremChannels *
                            BUS_COST_PREM_CHANNEL;

            cout << "Account number: "
                 << accountNumber << endl;                  //Step 7f
            cout << "Amount due: $" << amountDue
                 << endl;                                   //Step 7f
            break;

        default:
            cout << "Invalid customer type." << endl;       //Step 8
        }//end switch

        return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

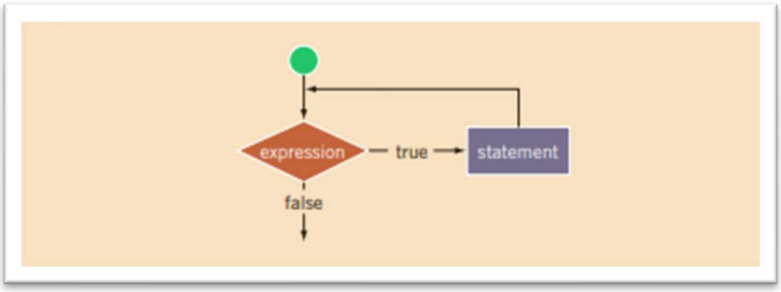
```
This program computes a cable bill.
Enter account number (an integer): 12345

Enter customer type: R or r (Residential), B or b (Business): b

Enter the number of basic service connections: 16

Enter the number of premium channels: 8

Account number: 12345
Amount due: $520.00
```

3.4 REPETITION

There are many other situations in which it is necessary to repeat a set of statements. For example, for each student in a class, the formula for determining the course grade is the same. C++ has three repetition, or looping, structures that let you repeat statements over and over until certain conditions are met. This chapter introduces all three looping (repetition) structures.

3.4.1 while Looping Structure

In C++, while is a reserved word. Of course, the statement can be either a simple or compound statement. The expression acts as a decision maker and is usually a logical expression. The statement is called the body of the loop. Note that the parentheses around the expression are part of the syntax.



The expression provides an entry condition to the loop. If it initially evaluates to true, the statement executes. The loop condition—the expression—is then reevaluated. If it again evaluates to true, the statement executes again. The statement (body of the loop) continues to execute until the expression is no longer true. A loop that continues to execute endlessly is called an infinite loop. To avoid an infinite loop, make sure that the loop's body contains statement(s) that assure that the entry condition—the expression in the while statement—will eventually be false

EXAMPLE

```
#include <iostream>

using namespace std;

int main()
{
    int calBurnedInADay;
    int calBurnedInAWeek;
    int day;

    day = 1;
    calBurnedInAWeek = 0;

    while (day <= 7)
    {
        cout << "Enter calories burned day " << day << ": ";
        cin >> calBurnedInADay;
        cout << endl;

        calBurnedInAWeek = calBurnedInAWeek + calBurnedInADay;
        day = day + 1;
    }

    cout << "Average number of calories burned each day: "
         << calBurnedInAWeek / 7 << endl;

    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter calories burned day 1: 375

Enter calories burned day 2: 425

Enter calories burned day 3: 270

Enter calories burned day 4: 190

Enter calories burned day 5: 350

Enter calories burned day 6: 200

Enter calories burned day 7: 365

Average number of calories burned each day: 310
```

Designing while Loops

The body of a while executes only when the expression, in the while statement, evaluates to true. Typically, the expression checks whether a variable, called the **loop control variable** (LCV), satisfies certain conditions.

The LCV must be properly initialized before the while loop is encountered, and it should eventually make the expression evaluate to false.

We do this by updating or assigning a new value to the LCV in the body of the while loop. Therefore, generally while loops are written in the following form:

```
//initialize the loop control variable(s)
while (expression) //expression tests the LCV
{ .
  .
  . //update the LCV
 }
```

EXAMPLE

Consider the following C++ program segment:

```
i = 20;                    //Line 1
while (i < 20)             //Line 2
{                          //Line 3
    cout << i << " ";      //Line 4
    i = i + 5;             //Line 5
}                          //Line 6
cout << endl;              //Line 7
```

In this example, at Line 1, i is initialized to 20, which makes the expression i < 20 in the while statement (Line 2) evaluate to false. Because initially the loop entry condition, i < 20, is false, the body of the while loop never executes. Hence, no values are output and the value of i remains 20. This example demonstrates the importance of the value to which the LCV is initialized.

3.4.2 Counter Controlled while Loops

Suppose you know exactly how many times certain statements need to be executed. For example, suppose you know exactly how many pieces of data (or entries) need to be read. In such cases, the while loop assumes the form of a counter-controlled while loop. That is, the LCV serves as a "counter." Suppose that a set of statements needs to be executed N times. You can set up a counter (initialized to 0 before the while statement) to track how many items have been read. Before executing the body of the while statement, the counter is compared with N. If counter < N, the body of the while statement executes. The body of the loop continues to execute until the value of counter >= N.

```
counter = 0; //initialize the loop control variable
while (counter < N) //test the loop control variable
 {
      counter++; //update the loop control variable
  }
```

If N represents the number of data items in a file, then the value of N can be determined in several ways. The program can prompt you to enter the number of items in the file and an input statement can read the value you entered; or you can specify the first item in the file as the number of items in the file, so that you need not remember the number of input values (items).

EXAMPLE

Students at a local middle school volunteered to sell fresh baked cookies to raise funds to increase the number of computers for the computer lab. Each student reported the number of boxes he/she sold. We will write a program that will output the total number of boxes of cookies sold, the total revenue generated by selling the cookies, and the average number of boxes sold by each student. The data provided is in the following form: student Name numOfBoxesSold

```
#include <iostream>                                          //Line 1
#include <string>                                            //Line 2
#include <iomanip>                                           //Line 3

using namespace std;                                         //Line 4

int main()                                                   //Line 5
{                                                            //Line 6
    string name;                                             //Line 7
    int numOfVolunteers;                                     //Line 8
    int numOfBoxesSold;                                      //Line 9
    int totalNumOfBoxesSold;                                 //Line 10
    int counter;                                             //Line 11
    double costOfOneBox;                                     //Line 12

    cout << fixed << showpoint << setprecision(2);           //Line 13

    cout << "Line 14: Enter the number of "
         << "volunteers: ";                                  //Line 14
    cin >> numOfVolunteers;                                  //Line 15
    cout << endl;                                            //Line 16

    totalNumOfBoxesSold = 0;                                 //Line 17
    counter = 0;                                             //Line 18

    while (counter < numOfVolunteers)                        //Line 19
    {                                                        //Line 20
        cout << "Line 21: Enter the volunteer's name"
             << " and the number of boxes sold: ";           //Line 21
        cin >> name >> numOfBoxesSold;                       //Line 22
        cout << endl;                                        //Line 23
        totalNumOfBoxesSold = totalNumOfBoxesSold
                            + numOfBoxesSold;                //Line 24
        counter++;                                           //Line 25
    }                                                        //Line 26

    cout << "Line 27: The total number of boxes sold: "
         << totalNumOfBoxesSold << endl;                     //Line 27

    cout << "Line 28: Enter the cost of one box: ";          //Line 28
    cin >> costOfOneBox;                                     //Line 29
    cout << endl;                                            //Line 30

    cout << "Line 31: The total money made by selling "
         << "cookies: $"
         << totalNumOfBoxesSold * costOfOneBox << endl;      //Line 31


  if (counter != 0)                                          //Line 32
      cout << "Line 33: The average number of "
           << "boxes sold by each volunteer: "
           << totalNumOfBoxesSold / counter << endl;         //Line 33
```

```
    else                                                    //Line 34
        cout << "Line 35: No input." << endl;               //Line 35

    return 0;                                               //Line 36
}                                                           //Line 37
```

Sample Run: In this sample run, the user input is shaded.

```
Line 14: Enter the number of volunteers: 5

Line 21: Enter the volunteer's name and the number of boxes sold: Sara 120

Line 21: Enter the volunteer's name and the number of boxes sold: Lisa 128

Line 21: Enter the volunteer's name and the number of boxes sold: Cindy 359

Line 21: Enter the volunteer's name and the number of boxes sold: Nicole 267

Line 21: Enter the volunteer's name and the number of boxes sold: Blair 165

Line 27: The total number of boxes sold: 1039
Line 28: Enter the cost of one box: 3.50

Line 31: The total money made by selling cookies: $3636.50
Line 33: The average number of boxes sold by each volunteer: 207
```

This program works as follows. The statements in Lines 7 to 12 declare the variables used in the program. The statement in Line 14 prompts the user to enter the number of student volunteers. The statement in Line 15 inputs this number into the variable numOfVolunteers. The statements in Lines 17 and 18 initialize the variables totalNumOfBoxesSold and counter. (The variable counter is the loop control variable.) The while statement in Line 19 checks the value of counter to determine how many students' data have been read. If counter is less than numOfVolunteers, the while loop proceeds for the next iteration. The statement in Line 21 prompts the user to input the student's name and the number of boxes sold by the student. The statement in Line 22 inputs the student's name into the variable name and the number of boxes sold by the student into the variable numOfBoxesSold. The statement in Line 24 updates the value of totalNumOfBoxesSold by adding the value of numOfBoxesSold to its current value and the statement in Line 25 increments the value of counter by 1. The statement in Line 27 outputs the total number of boxes sold, the statement in Line 28 prompts the user to input the cost of one box of cookies, and the statement in Line 29 inputs the cost in the variable costOfOneBox.

The statement in Line 31 outputs the total money made by selling cookies, and the statements in Lines 32 through 35 output the average number of boxes sold by each volunteer. Note that totalNumOfBoxesSold is initialized to 0 in Line 17 in this program. In Line 22, after reading the number of boxes sold by a student, the program adds it to the sum of all the boxes sold before the current number of boxes sold.

The first numOfBoxesSold read will be added to zero (because totalNumOfBoxesSold is initialized to 0), giving the correct sum of the first number. To find the average, divide totalNumOfBoxesSold by counter. If counter is 0, then dividing by zero will terminate the program and you will get an error message. Therefore, before dividing totalNumOfBoxesSold by counter, you must check whether or not counter is 0. Notice that in this program, the statement in Line 18 initializes the LCV counter to 0. The expression counter < numOfVolunteers in Line 19 evaluates whether counter is less than numOfVolunteers. The statement in Line 25 updates the value of counter.

3.4.3Sentinel Controlled while Loops

You do not always know how many pieces of data (or entries) need to be read, but you may know that the last entry is a special value, called a **sentinel**, that will tell the loop to stop. In this case, you must read the first item before the while statement so the test expression will have a valid value to test. If this item does not equal the sentinel, the body of the while statement executes. The while loop continues to execute as long as the program has not read the sentinel. Such a while loop is called a **sentinel-controlled** while loop.

In this case, a while loop might look like the following:

```
cin >> variable;                    //initialize the loop control variable
while (variable !5 sentinel)        //test the loop control variable
{ .
  .
   cin >> variable; //update the loop control variable .
  .
 }
```

EXAMPLE

The program computes and outputs the total number of boxes of cookies sold, the total money made, and the average number of boxes sold by each student. However, the program assumes that the programmer knows the exact number of volunteers. Now suppose that the programmer does not know the exact number of volunteers.

Once again, assume that the data is in the following form: student's name followed by a space and the number of boxes sold by the student. Because we do not know the exact number of volunteers, we assume that reading a value of -1 for name will mark the end of the data, since it is a highly unlikely name to run into.

```
#include <iostream>                                          //Line 1
#include <string>                                            //Line 2
#include <iomanip>                                           //Line 3

using namespace std;                                         //Line 4

const string SENTINEL = "-1";                                //Line 5

int main()                                                   //Line 6
{                                                            //Line 7
    string name;                                             //Line 8
    int numOfVolunteers;                                     //Line 9
    int numOfBoxesSold;                                      //Line 10
    int totalNumOfBoxesSold;                                 //Line 11
    double costOfOneBox;                                     //Line 12

    cout << fixed << showpoint << setprecision(2);           //Line 13

    cout << "Line 14: Enter each volunteer's name and "
         << "the number of boxes " << endl
         << "         sold by each volunteer, ending "
         << "with -1: " << endl;                             //Line 14

    totalNumOfBoxesSold = 0;                                 //Line 15
    numOfVolunteers = 0;                                     //Line 16

    cin >> name;                                             //Line 17

    while (name != SENTINEL)                                 //Line 18
    {                                                        //Line 19
        cin >> numOfBoxesSold;                               //Line 20
        totalNumOfBoxesSold = totalNumOfBoxesSold
                            + numOfBoxesSold;                //Line 21
        numOfVolunteers++;                                   //Line 22
        cin >> name;                                         //Line 23
    }                                                        //Line 24

    cout << endl;                                            //Line 25

    cout << "Line 26: The total number of boxes sold: "
         << totalNumOfBoxesSold << endl;                     //Line 26

    cout << "Line 27: Enter the cost of one box: ";          //Line 27
    cin >> costOfOneBox;                                     //Line 28
    cout << endl;                                            //Line 29

    cout << "Line 30: The total money made by selling "
```

```
         << "cookies: $"
         << totalNumOfBoxesSold * costOfOneBox << endl; //Line 30

    if (numOfVolunteers != 0)                           //Line 31
        cout << "Line 32: The average number of "
             << "boxes sold by each volunteer: "
             << totalNumOfBoxesSold / numOfVolunteers
             << endl;                                   //Line 32
    else                                                //Line 33
        cout << "Line 34: No input." << endl;           //Line 34

    return 0;                                           //Line 35
}                                                       //Line 36
```

Sample Run: In this sample run, the user input is shaded.

```
Line 14: Enter each volunteer's name and the number of boxes
         sold by each volunteer, ending with -1:
Sara 120
Lisa 128
Cindy 359
Nicole 267
Blair 165
Abby 290
Amy 190
Megan 450
Elizabeth 280
Meridth 290
Leslie 430
Chelsea 378
-1

Line 26: The total number of boxes sold: 3347
Line 27: Enter the cost of one box: 3.50

Line 30: The total money made by selling cookies: $11714.50
Line 32: The average number of boxes sold by each volunteer: 278
```

This program works as follows. The statements in Lines 8 to 12 declare the variables used in the program. The statement in Line 14 prompts the user to enter the data ending with -1. The statements in Lines 15 and 16 initialize the variables totalNumOfBoxesSold and numOfVolunteers. The statement in Line 17 reads the first name and stores it in name. The while statement in Line 18 checks whether name is not equal to SENTINEL. (The variable name is the loop control variable.) If name is not equal to SENTINEL, the body of the while loop executes. The statement in Line 20 reads and stores the number of boxes sold by the student in the variable numOfBoxesSold and the statement in Line 21 updates the value of totalNumOfBoxesSold by adding numOfBoxesSold to it. The statement in Line 22 increments the value of numOfVolunteers by 1, and the statement in Line 23 reads and stores the next name into name. The statements in Lines 20 through 23 repeat until the program reads the SENTINEL.

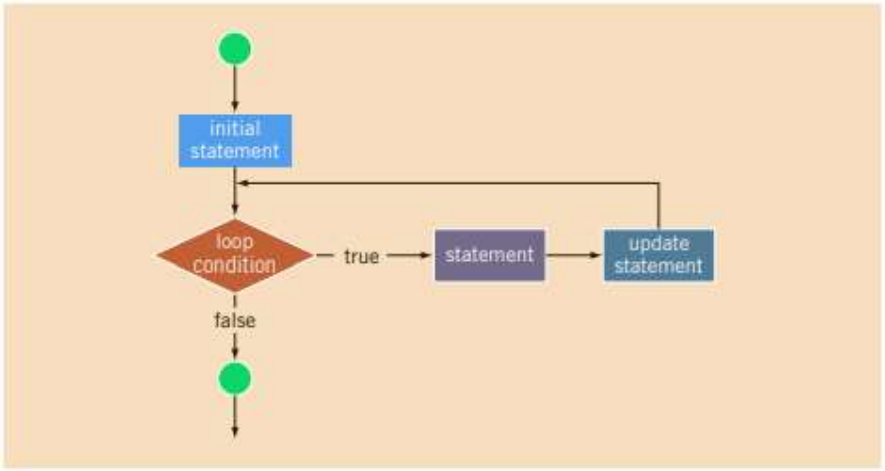
3.4.3 For Looping Structure

The C++ for looping structure discussed here is a specialized form of the while loop. Its primary purpose is to simplify the writing of counter-controlled loops. For this reason, the for loop is typically called a counted or indexed for loop.

The general form of the for statement is:

```
for (initial statement; loop condition; update statement)
    statement
```

The initial statement, loop condition, and update statement (called for loop control statements) enclosed within the parentheses control the body (statement) of the for statement.



The for loop executes as follows:

1. The initial statement executes.
2. The loop condition is evaluated.

If the loop condition evaluates to true:

i. Execute the for loop statement.

ii. Execute the update statement (the third expression in the parentheses).

3. Repeat Step 2 until the loop condition evaluates to false. The initial statement usually initializes a variable (called the for loop control, or for indexed, variable). In C++, for is a reserved word.

EXAMPLE

In this example, a `for` loop reads five numbers and finds their sum and average. Consider the following program code, in which `i`, `newNum`, `sum`, and `average` are `int` variables:

```
sum = 0;

for (i = 1; i <= 5; i++)
{
    cin >> newNum;
    sum = sum + newNum;
}

average = sum / 5;
cout << "The sum is " << sum << endl;
cout << "The average is " << average << endl;
```

In the preceding `for` loop, after reading a `newNum`, this value is added to the previously calculated (partial) `sum` of all the numbers read before the current number. The variable `sum` is initialized to `0` before the `for` loop. Thus, after the program reads the first number and adds it to the value of `sum`, the variable `sum` holds the correct `sum` of the first number.

EXAMPLE

1. The following `for` loop outputs `Hello!` and a star (on separate lines) five times:

   ```
   for (i = 1; i <= 5; i++)
   {
       cout << "Hello!" << endl;
       cout << "*" << endl;
   }
   ```

2. Consider the following `for` loop:

   ```
   for (i = 1; i <= 5; i++)
       cout << "Hello!" << endl;
       cout << "*" << endl;
   ```

   The output of this `for` loop is:

   ```
   Hello!
   Hello!
   Hello!
   Hello!
   Hello!
   *
   ```

   The `for` loop controls only the first output statement because the two output statements are not made into a compound statement using braces. Therefore, the first output statement executes five times because the `for` loop body executes five times. After the `for` loop executes, the second output statement executes only once. The indentation, which is ignored by the compiler, is nevertheless misleading.

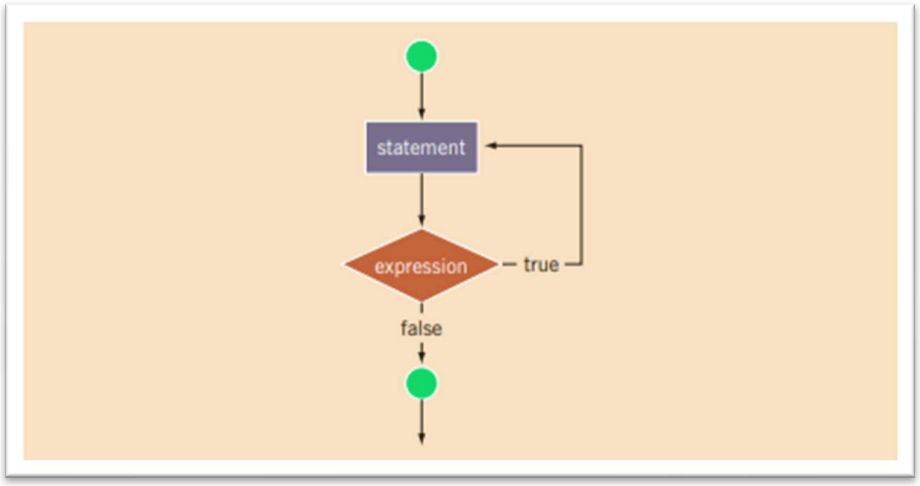
3.4.4 Do….while Looping Structure

This section describes the third type of looping or repetition structure, called a do. . . while loop.

The general form of a do. . .while statement is as follows:

```
do
    statement
while (expression);
```

Of course, statement can be either a simple or compound statement. If it is a compound statement, enclose it between braces. Figure below shows the flow of execution of a do. . .while loop.



In C++, do is a reserved word.

The statement executes first, and then the expression is evaluated. If the expression evaluates to true, the statement executes again. As long as the expression in a do. . .while statement is true, the statement executes. To avoid an infinite loop, you must, once again, make sure that the loop body contains a statement that ultimately makes the expression false and assures that it exits properly

EXAMPLE

```
i = 0;

do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 20);
```

The output of this code is:

```
0 5 10 15 20
```

After `20` is output, the statement:

```
i = i + 5;
```

changes the value of `i` to `25` and so `i <= 20` becomes `false`, which halts the loop.

In a while and for loop, the loop condition is evaluated *before* executing the body of the loop. Therefore, while and for loops are called **pretest loops.** On the other hand, the loop condition in a do. . .while loop is evaluated *after* executing the body of the loop. Therefore, do. . .while loops are called **posttest loops.**

Because the while and for loops both have entry conditions, these loops may never activate. The do...while loop, on the other hand, has an exit condition and therefore always executes the statement at least once.

EXAMPLE

A `do...while` loop can be used for input validation. Suppose that a program prompts a user to enter a test score, which must be greater than or equal to `0` and less than or equal to `50`. If the user enters a score less than `0` or greater than `50`, the user should be prompted to re-enter the score. The following `do...while` loop can be used to accomplish this objective:

```
int score;

do
{
    cout << "Enter a score between 0 and 50: ";
    cin >> score;
    cout << endl;
}
while (score < 0 || score > 50);
```

Break and Continue Statements

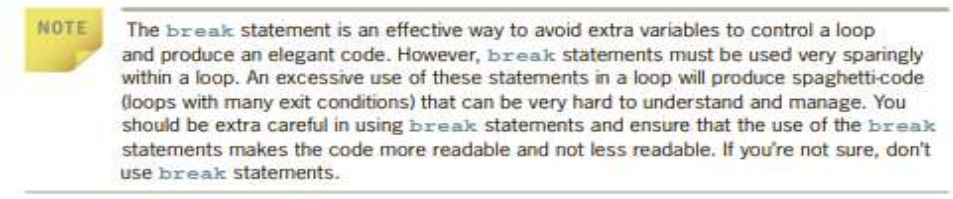
The break statement, when executed in a switch structure, provides an immediate exit from the switch structure. Similarly, you can use the break statement in while, for, and do. . .while loops to immediately exit from the loop structure. The break statement is typically used for two purposes:

- To exit early from a loop.
- To skip the remainder of the switch structure.

After the break statement executes, the program continues to execute with the first statement after the structure. The use of a break statement in a loop can eliminate the use of certain (flag) variables.

```
sum = 0;
isNegative = false;

cin >> num;

while (cin && !isNegative)
{
    if (num < 0)    //if num is negative, terminate the loop
                    //after this iteration
    {
        cout << "Negative number found in the data." << endl;
        isNegative = true;
    }
    else
    {
        sum = sum + num;
        cin >> num;
    }
}
```

This while loop is supposed to find the sum of a set of positive numbers. If the data set contains a negative number, the loop terminates with an appropriate error message. This while loop uses the flag variable isNegative to signal the presence of a negative number. The variable isNegative is initialized to false before the while loop. Before adding num to sum, a check is made to see if num is negative. If num is negative, an error message appears on the screen and isNegative is set to true. In the next iteration, when the



expression in the while statement is evaluated, it evaluates to false because !isNegative is false. (Note that because isNegative is true, !isNegative is false.)

The continue statement is used in while, for, and do. . .while structures. When the continue statement is executed in a loop, it skips the remaining statements in the loop and proceeds with the next iteration of the loop. In a while and do. . .while structure, the expression (that is, the loop-continue test) is evaluated immediately after the continue statement. In a for structure, the update statement is executed after the continue statement, and then the loop condition (that is, the loop-continue test) executes. If the previous program segment encounters a negative number, the while loop terminates. If you want to discard the negative number and read the next number rather than terminate the loop, replace the break statement with the continue statement.

```
sum = 0;
cin >> num;

while (cin)
{
    if (num < 0)
    {
        cout << "Negative number found in the data." << endl;
        cin >> num;
        continue;
    }

    sum = sum + num;
    cin >> num;
}
```

It was stated earlier that all three loops have their place in C++ and that one loop can often replace another. The execution of a continue statement, however, is where the while and do. . .while structures differ from the for structure. When the continue statement is executed in a while or a do. . .while loop, the update statement may not execute. In a for structure, the update statement always executes.

Nested Control Structures

In this section, we give examples that illustrate how to use nested loops to achieve useful results and process data.

Suppose you want to create the following pattern:

```
*
**
***
****
*****
```

Clearly, you want to print five lines of stars. In the first line, you want to print one star, in the second line, two stars, and so on.

Because five lines will be printed, start with the following for statement:

```
for (i = 1; i <= 5; i++)
```

The value of i in the first iteration is 1, in the second iteration it is 2, and so on. You can use the value of i as the limiting condition in another for loop nested within this loop to control the number of stars in a line. A little more thought produces the following code:

```
for (i = 1; i <= 5; i++)                //Line 1
{
    for (j = 1; j <= i; j++)            //Line 2
        cout << "*";                    //Line 3
    cout << endl;                       //Line 4
}                                       //Line 5
```

A walk-through of this code shows that the for loop in Line 1 starts with i = 1. When i is 1, the inner for loop in Line 3 outputs one star and the insertion point moves to the next line. Then i becomes 2, the inner for loop outputs two stars, and the output statement in Line 5 moves the insertion point to the next line, and so on. This process continues until i becomes 6 and the loop stops.

EXAMPLE

Suppose you want to create the following multiplication table:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50

The multiplication table has five lines. Therefore, as in the previous example, we use a for statement to output these lines as follows:

```
for (i = 1; i <= 5; i++)
    //output a line of numbers
```

In the first line, we want to print the multiplication table of 1, in the second line we want to print the multiplication table of 2, and so on. Notice that the first line starts with 1 and when this line is printed, i is 1. Similarly, the second line starts with 2 and when this line is printed, the value of i is 2, and so on. If i is 1, i * 1 is 1; if i is 2, i * 2 is 2; and so on. Therefore, to print a line of numbers, we can use the value of i as the starting number and 10 as the limiting value. That is, consider the following for loop:

```
for (j = 1; j <= 10; j++)
      cout << setw(3) << i * j;
```

Let us take a look at this for loop. Suppose i is 1. Then we are printing the first line of the multiplication table. Also, j goes from 1 to 10 and so this for loop outputs the numbers 1 through 10, which is the first line of the multiplication table.

Similarly, if i is 2, we are printing the second line of the multiplication table. Also, j goes from 1 to 10, and so this for loop outputs the second line of the multiplication table, and so on.

A little more thought produces the following nested loops to output the desired grid:

```
for (i = 1; i <= 5; i++)                    //Line 1
```

```
{
        for (j = 1; j <= 10; j++)                    //Line 2
                cout << setw(3) << i * j;           //Line 3
                 cout << endl; //Line 5 }           //Line 4
        cout << endl;                                //Line 5
}
```

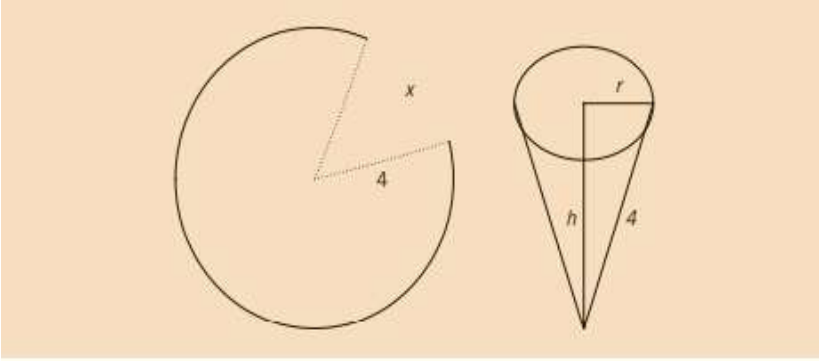
QUICK REVIEW

1. The statement is called the body of the loop.
2. C++ has three looping (repetition) structures: while, for, and do. . .while.
3. The body of the while loop must contain a statement that eventually sets the expression to false.
4. In a counter-controlled while loop, you must initialize the counter before the loop, and the body of the loop must contain a statement that changes the value of the counter variable.
5. A sentinel-controlled while loop uses a sentinel to control the loop. The while loop continues to execute until the sentinel is read.
6. The while and for loop bodies may not execute at all, but the do. . . while loop body always executes at least once.
7. Executing a break statement in the body of a loop immediately terminates the loop.
8. When a continue statement executes in a while or do. . .while loop, the expression update statement in the body of the loop may not execute.
9. Both while and for loops are called pretest loops. A do. . .while loop is called a posttest loop.
10. Putting a semicolon at the end of the for loop (before the body of the for loop) is a semantic error. In this case, the action of the for loop is empty.

PROGRAMMING EXERCISES

1. The program in the Programming Example: Fibonacci Number does not check whether the first number entered by the user is less than or equal to the second number and whether both the numbers are nonnegative. Also, the program does not check whether the user entered a valid value for the position of the desired number in the Fibonacci sequence. Rewrite that program so that it checks for these things.

2. **(The conical paper cup problem)** You have been given the contract for making little conical cups that come with bottled water. These cups are to be made from a circular waxed paper of 4 inches in radius by removing a sector of length x (see Figure below). By closing the remaining part of the circle, a conical cup is made. Your objective is to remove the sector so that the cup is of maximum volume.



Write a program that prompts the user to enter the radius of the circular waxed paper. The program should then output the length of the removed sector so that the resulting cup is of maximum volume. Calculate your answer to two decimal places.

3. **(Apartment problem)** A real estate office handles, say, 50 apartment units. When the rent is, say, $600 per month, all the units are occupied. However, for each, say, $40 increase in rent, one unit becomes vacant. Moreover, each occupied unit requires an average of $27 per month for maintenance. How many units should be rented to maximize the profit?

   Write a program that prompts the user to enter: a. The total number of units. b. The rent to occupy all the units. c. The increase in rent that results in a vacant unit. d. Amount to maintain a rented unit. The program then outputs the number of units to be rented to maximize the profit.

4. Let n be a nonnegative integer. The factorial of n, written n!, is defined by 0! 1 5 , n! = 1·2·3· · ·n if n $ 1. For example, 4! = 1·2·3·4 = 24. Write a program that prompts the user to enter a nonnegative integer and outputs the factorial of the number.

5. The population of town A is less than the population of town B. However, the population of town A is growing faster than the population of town B. Write a program that prompts the user to enter the population and growth rate of each town. The program outputs after how many years the population of town A will be greater than or equal to the population of town B and the populations of both the towns at that time. (A sample input is: Population of town A 5 5,000, growth rate of town A 5 4%, population of town B 5 8,000, and growth rate of town B 5 2%.)

TOPIC 4
FUNCTIONS

LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn about standard (predefined) functions and discover how to use them in a program.
- Learn about user-defined functions.
- Examine value-returning functions, including actual and formal parameters
- Discover the difference between value and reference parameters
- Explore reference parameters and value-returning functions.
- Learn function overloading

4.1 INTRODUCTION

Functions are often called modules. They are like miniature programs; you can put them together to form a larger program. When user-defined functions are discussed, you will see that this is the case. This ability is less apparent with predefined functions because their programming code is not available to us. However, because predefined functions are already written for us, you will learn these first so that you can use them when needed.

4.1.1 Predefined Functions

In algebra, a function can be considered a rule or correspondence between values, called the function's arguments, and the unique values of the function associated with the arguments. Thus, if $f(x) = 2x + 5$, then $f(1) = 7$, $f(2) = 9$, and $f(3) = 11$, where 1, 2, and 3 are the arguments of the function f, and 7, 9, and 11 are the corresponding values of *f*.

In C++, the concept of a function, either predefined or user-defined, is similar to that of a function in algebra. For example, every function has a name and, depending on the values specified by the user, it does some computation.
Some of the predefined mathematical functions are **pow(x, y), sqrt(x),** and **floor(x).**

In C++, predefined functions are organized into separate libraries. For example, the header file iostream contains I/O functions, and the header file cmath contains math functions. Table below lists some of the commonly used predefined functions, the name of the header file in which each function's specification can be found, the data type of the parameters, and the function type. The function type is the data type of the value returned by the function.

Function	Header File	Purpose	Parameter(s) Type	Result
`abs(x)`	`<cmath>`	Returns the absolute value of its argument: `abs(-7) = 7`	`int (double)`	`int (double)`
`ceil(x)`	`<cmath>`	Returns the smallest whole number that is not less than `x`: `ceil(56.34) = 57.0`	`double`	`double`
`cos(x)`	`<cmath>`	Returns the cosine of angle `x`: `cos(0.0) = 1.0`	`double (radians)`	`double`
`exp(x)`	`<cmath>`	Returns e^x, where `e = 2.718`: `exp(1.0) = 2.71828`	`double`	`double`

`islower(x)`	`<cctype>`	Returns `true` if `x` is a lowercase letter; otherwise it returns `false`; `islower('h')` is `true`	`int`	`int`
`isupper(x)`	`<cctype>`	Returns `true` if `x` is a uppercase letter; otherwise it returns `false`; `isupper('K')` is `true`	`int`	`int`
`pow(x, y)`	`<cmath>`	Returns x^y; If x is negative, `y` must be a whole number: `pow(0.16, 0.5) = 0.4`	`double`	`double`
`sqrt(x)`	`<cmath>`	Returns the nonnegative square root of `x`, `x` must be nonnegative: `sqrt(4.0) = 2.0`	`double`	`double`
`tolower(x)`	`<cctype>`	Returns the lowercase value of `x` if `x` is uppercase; otherwise, returns `x`	`int`	`int`
`toupper(x)`	`<cctype>`	Returns the uppercase value of x if x is lowercase; otherwise, returns `x`	`int`	`int`

The square root function, sqrt(x), calculates the nonnegative square root of x for x >= 0.0. For example, sqrt(2.25) is 1.5. The function sqrt is of type double and has only one parameter.

The floor function, floor(x), calculates the largest whole number that is less than or equal to x. For example, floor(48.79) is 48.0. The function floor is of type double and has only one parameter

To use predefined functions in a program, you must include the header file that contains the function's specification via the include statement.

EXAMPLE

```
// How to use predefined functions.
#include <iostream>                                          //Line 1
#include <cmath>                                             //Line 2
#include <cctype>                                            //Line 3
#include <iomanip>                                           //Line 4

using namespace std;                                         //Line 5

int main()                                                   //Line 6
{                                                            //Line 7
    int num;                                                 //Line 8
    double firstNum, secondNum;                              //Line 9
    char ch = 'T';                                           //Line 10

    cout << fixed << showpoint << setprecision (2)
         << endl;                                            //Line 11

    cout << "Line 12: Is " << ch
         << " a lowercase letter? "
         << islower(ch) << endl;                             //Line 12
    cout << "Line 13: Uppercase a is "
         << static_cast<char>(toupper('a')) << endl;         //Line 13

    cout << "Line 14: 4.5 to the power 6.0 = "
         << pow(4.5, 6.0) << endl;                           //Line 14

    cout << "Line 15: Enter two decimal numbers: ";          //Line 15
    cin >> firstNum >> secondNum;                            //Line 16
    cout << endl;                                            //Line 17


     cout << "Line 18: " << firstNum
          << " to the power of " << secondNum
          << " = " << pow(firstNum, secondNum) << endl;      //Line 18

     cout << "Line 19: 5.0 to the power of 4 = "
          << pow(5.0, 4) << endl;                            //Line 19

     firstNum = firstNum + pow(3.5, 7.2);                    //Line 20
     cout << "Line 21: firstNum = " << firstNum << endl;     //Line 21

     num = -32;                                              //Line 22
     cout << "Line 23: Absolute value of " << num
          << " = " << abs(num) << endl;                      //Line 23

     cout << "Line 24: Square root of 28.00 = "
          << sqrt(28.00) << endl;                            //Line 24

      return 0;                                              //Line 25
}                                                            //Line 26
```

Sample Run: In this sample run, the user input is shaded.

```
Line 12: Is T a lowercase letter? 0
Line 13: Uppercase a is A
Line 14: 4.5 to the power 6.0 = 8303.77
Line 15: Enter two decimal numbers: 24.7 3.8

Line 18: 24.70 to the power of 3.80 = 195996.55
Line 19: 5.0 to the power of 4 = 625.00
Line 21: firstNum = 8290.60
Line 23: Absolute value of -32 = 32
Line 24: Square root of 28.00 = 5.29
```

This program works as follows. The statements in Lines 1 to 4 include the header files that are necessary to use the functions used in the program. The statements in Lines 8 to 10 declare the variables used in the program. The statement in Line 11 sets the output of decimal numbers in fixed decimal format with two decimal places. The statement in Line 12 uses the function *islower* to determine and output whether ch is a lowercase letter. The statement in Line 13 uses the function *toupper* to output the uppercase letter that corresponds to 'a', which is A. Note that the function *toupper* returns an int value. Therefore, the value of the expression *toupper*('a') is 65, which is the ASCII value of 'A'. To print the character A rather than the value 65, you need to apply the cast operator as shown in the statement in Line 13. The statement in Line 14 uses the function pow to output 4.56.0. In C++ terminology, it is said that the function pow is called with the parameters 4.5 and 6.0. The statements in Lines 15 to 17 prompt the user to enter two decimal numbers and store the numbers entered by the user in the variables firstNum and secondNum. In the statement in Line 18, the function pow is used to output firstNumsecondNum. In this case, the function pow is called with the parameters firstNum and secondNum and the values of firstNum and secondNum are passed to the function pow. The other statements have similar meanings. Once again, note that the program includes the header files *cctype* and *cmath*, because it uses the functions islower, toupper, pow, abs, and sqrt from these header files.

4.1.2 User- Defined Functions

Because C++ does not provide every function that you will ever need and designers cannot possibly know a user's specific needs, you must learn to write your own functions.

User-defined functions in C++ are classified into two categories:

- Value-returning functions—functions that have a return type. These functions return a value of a specific data type using the return statement, which we will explain shortly. Note that the function main has used a return statement to return the value 0 in every program we've seen so far.
- Void functions—functions that do not have a return type. These functions do not use a return statement to return a value.

Value-Returning Functions

The previous section introduced some predefined C++ functions such as pow, abs, islower, and toupper. These are examples of value-returning functions. To use these functions in your programs, you must know the name of the header file that contains the functions' specification. You need to include this header file in your program using the include statement and know the following item:

1. The name of the function
2. The parameters, if any
3. The data type of each parameter
4. The data type of the value computed (that is, the value returned) by the function, called the type of the function.
5. The code required to accomplish the task

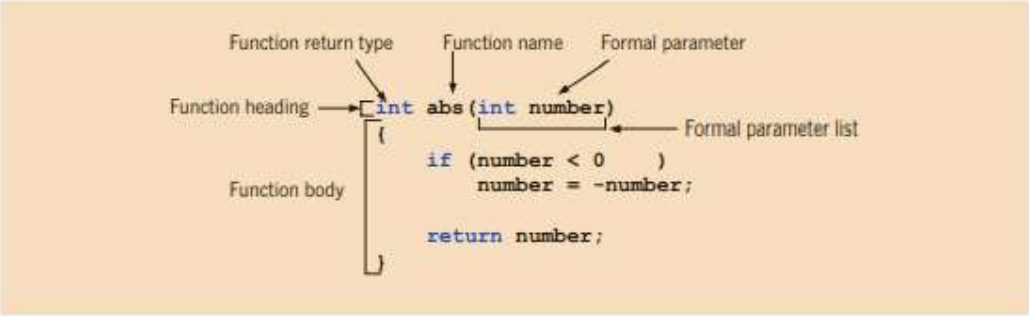
The syntax of a value-returning function is:

```
functionType functionName(formal parameter list)
{
    statements
}
```

in which statements are usually declaration statements and/or executable statements. In this syntax, functionType is the type of the value that the function returns. The functionType is also called the data type or the return type of the value-returning

function. Moreover, statements enclosed between curly braces form the body of the function

Consider the definition of the function abs given earlier in this chapter. Figure below identifies various parts of this function.



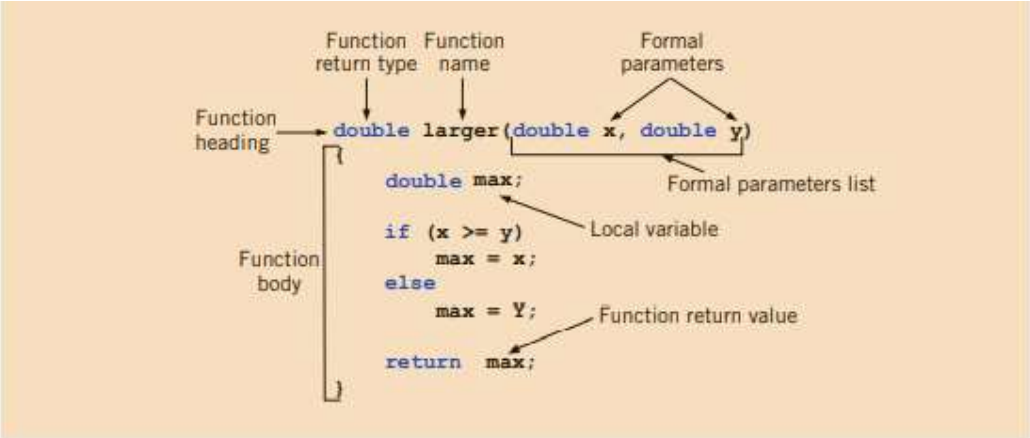
In C++, return is a reserved word. When a return statement executes in a function, the function immediately terminates and the control goes back to the calling function. Moreover, the function call statement is replaced by the value returned by the return statement. When a return statement executes in the function main, the program terminates.

To put the ideas in this discussion to work, let us write a function that determines the larger of two numbers. Because the function compares two numbers, it follows that this function has two parameters and that both parameters are numbers. Let us assume that the data type of these numbers is floating-point (decimal)—say, double. Because the larger number is of type double, the function's data type is also double. Let us name this function larger. The only thing you need to complete this function is the body of the function. Thus, following the syntax of a function, you can write this function as follows:

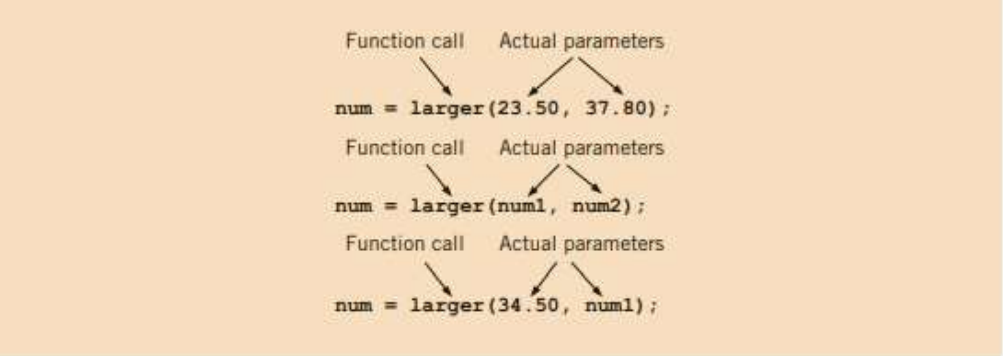
```
double larger(double x, double y)
 {
        double max;
            if (x >= y)
```

```
        max = x;
    else
        max = y;
return max;
}
```

Note that the function larger uses an additional variable max (called a local declaration, in which max is a variable local to the function larger). Figure below describes various parts of the function larger.



Suppose that num, num1, and num2 are double variables. Also suppose that num1 = 45.75 and num2 = 35.50. Figure below shows various ways the function larger can be called.



EXAMPLE (FIBONACCI NUMBER)

In this example, we modify the main program by writing a function that computes and returns the desired number of a Fibonacci sequence.

Given the first number, the second number, and the position of the desired Fibonacci number, the following function returns the Fibonacci number:

```
int nthFibonacciNum(int first, int second, int nthFibNum)
{
    int current;
    int counter;

    if (nthFibNum == 1)
        current = first;
    else if (nthFibNum == 2)
        current = second;
    else
    {
        counter = 3;

        while (counter <= nthFibNum)
        {
            current = second + first;
            first = second;
            second = current;
            counter++;
        }//end while
    }//end else

    return current;
}
    //Program: Fibonacci number

    #include <iostream>

    using namespace std;

    int nthFibonacciNum(int first, int second, int position);

    int main()
    {
        int firstFibonacciNum;
        int secondFibonacciNum;
        int nthFibonacci;

        cout << "Enter the first two Fibonacci "
             << "numbers: ";
        cin >> firstFibonacciNum >> secondFibonacciNum;
        cout << endl;
```

```
    cout << "The first two Fibonacci numbers are "
         << firstFibonacciNum << " and "
         << secondFibonacciNum
         << endl;

    cout << "Enter the position of the desired "
         << "Fibonacci number: ";
    cin >> nthFibonacci;
    cout << endl;

    cout << "The Fibonacci number at position "
         << nthFibonacci << " is "
         << nthFibonacciNum(firstFibonacciNum, secondFibonacciNum,
                            nthFibonacci)
         << endl;

    return 0;
}

int nthFibonacciNum(int first, int second, int nthFibNum)
{
    int current;
    int counter;

    if (nthFibNum == 1)
        current = first;
    else if (nthFibNum == 2)
        current = second;
    else
    {
        counter = 3;

          while (counter <= nthFibNum)
          {
              current = second + first;
              first = second;
              second = current;
              counter++;
          }//end while
      }//end else

      return current;
  }
```

Sample Runs: In these sample runs, the user input is shaded.

Sample Run 1:

```
Enter the first two Fibonacci numbers: 12 16

The first two Fibonacci numbers are 12 and 16
Enter the position of the desired Fibonacci number: 10

The Fibonacci number at position 10 is 796
```

Sample Run 2:

```
Enter the first two Fibonacci numbers: 1 1

The first two Fibonacci numbers are 1 and 1
Enter the position of the desired Fibonacci number: 15

The Fibonacci number at position 15 is 610
```

Function Prototype

In reality, C++ programmers customarily place the function main before all other user-defined functions. However, this organization could produce a compilation error because functions are compiled in the order in which they appear in the program. For example, if the function main is placed before the function larger, the identifier larger will be undefined when the function main is compiled. To work around this problem of undeclared identifiers, we place function prototypes before any function definition (including the definition of main).

The function prototype is not a definition. It gives the program the name of the function, the number and data types of the parameters, and the data type of the returned value: just enough information to let C++ use the function. It is also a promise that the full definition will appear later in the program. If you neglect to write the definition of the function, the program may compile, but it will not execute.

Function Prototype: The function heading, terminated by a semicolon, ;, without the body of the function.

The general syntax of the function prototype of a value-returning function is:

```
functionType functionName(parameter list);
```

(Note that the function prototype ends with a semicolon.)

For the function larger, the prototype is:

```
double larger(double x, double y);                    //function prototype
```

EXAMPLE

```
//Program: Largest of three numbers

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

//Function prototype
double larger(double x, double y);                           //Line 3
double compareThree(double x, double y, double z);          //Line 4

int main()                                                   //Line 5
{                                                            //Line 6
    double one, two;                                         //Line 7

    cout << "Line 8: The larger of 5 and 10 is "
         << larger(5, 10) << endl;                           //Line 8


    cout << "Line 9: Enter two numbers: ";                   //Line 9
    cin >> one >> two;                                       //Line 10
    cout << endl;                                            //Line 11

    cout << "Line 12: The larger of " << one
         << " and " << two << " is "
         << larger(one, two) << endl;                        //Line 12

    cout << "Line 13: The largest of 43.48, 34.00, "
         << "and 12.65 is "
         << compareThree(43.48, 34.00, 12.65)
         << endl;                                            //Line 13

   return 0;                                                 //Line 14
}                                                            //Line 15

double larger(double x, double y)
{
    double max;

    if (x >= y)
        max = x;
    else
        max = y;

    return max;
}


double compareThree (double x, double y, double z)
{
    return larger(x, larger(y, z));
}
```

Sample Run: In this sample run, the user input is shaded.

```
Line 8: The larger of 5 and 10 is 10
Line 9: Enter two numbers: 37.93 26.785

Line 12: The larger of 37.93 and 26.785 is 37.93
Line 13: The largest of 43.48, 34.00, and 12.65 is 43.48
```

NOTE

In the previous program, the function prototypes of the functions **larger** and **compareThree** appear before their function definitions. Therefore, the definition of the functions **larger** and **compareThree** can appear in any order.

Void Functions

In this section, you will explore user-defined functions in general and, in particular, those C++ functions that do not have a data type, called **void functions**.

Void functions and value-returning functions have similar structures. Both have a heading and a body. Like value-returning functions, you can place user-defined void functions either before or after the function main. However, the program execution always begins with the first statement in the function main. If you place a user-defined void function after the function main, you should place the function prototype before the function main. A void function does not have a data type. Therefore, functionType— that is, the return type—in the heading part and the return statement in the body of the void functions are meaningless. However, in a void function, you can use the return statement without any value; it is typically used to exit the function early. Like value-returning functions, void functions may or may not have formal parameters.

The function definition of void functions with parameters has the following syntax:

```
void functionName(formal parameter list)
{
    statements
}
```

EXAMPLE

```
void funExp(int a, double b, char c, int x)
{
    .
    .
    .
}
```

The function `funExp` has four parameters.

Parameter Types

Parameters provide a communication link between the calling function (such as main) and the called function. They enable functions to manipulate different data each time they are called. In general, there are two types of formal parameters: value parameters and reference parameters. Value parameter: A formal parameter that receives a copy of the content of the corresponding actual parameter. Reference parameter: A formal parameter that receives the location (memory address) of the corresponding actual parameter. When you attach & after the dataType in the formal parameter list of a function, the variable following that dataType becomes a reference parameter.

EXAMPLE

Consider the following definition:

```
void averageAndGrade(int testScore, int progScore,
                     double& average, char& grade)
{
    average = (testScore + progScore) / 2.0;

    if (average >= 90.00)
        grade = 'A';
    else if (average >= 80.00)
        grade = 'B';
    else if (average >= 70.00)
        grade = 'C';
    else if (average >= 60.00)
        grade = 'D';
    else
        grade = 'F';
}
```

The function **averageAndGrade** has four parameters: **testScore** and **progScore** are value parameters of type `int`, **average** is a reference parameter of type `double`, and **grade** is a reference parameter of type `char`. Using visual diagrams, Examples 6-13 to 6-15 explicitly show how value and reference parameters work.

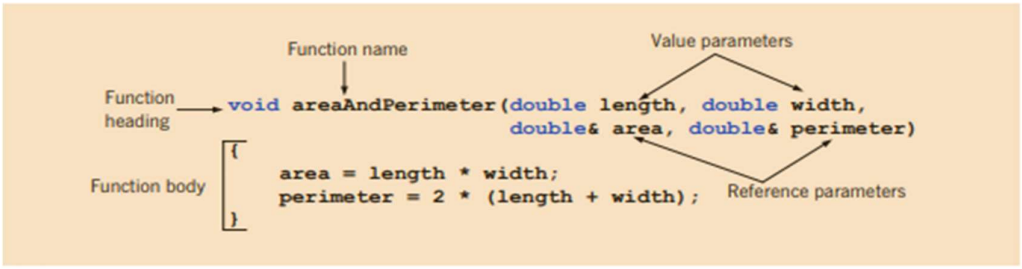
EXAMPLE

Consider the following function definition:

```
void areaAndPerimeter(double length, double width,
                      double& area, double& perimeter)

    area = length * width;
    perimeter = 2 * (length + width);
```

The function **areaAndPerimeter** has four parameters: **length** and **width** are value parameters of type double; and **area** and **perimeter** are reference parameters of type double.

Figure 6-4 describes various parts of the function **areaAndPerimeter**.



EXAMPLE

Consider the following definition:

```
void averageAndGrade(int testScore, int progScore,
                     double& average, char& grade)
{
    average = (testScore + progScore) / 2.0;

    if (average >= 90.00)
        grade = 'A';
    else if (average >= 80.00)
        grade = 'B';
    else if (average >= 70.00)
        grade = 'C';
    else if (average >= 60.00)
        grade = 'D';
    else
        grade = 'F';
}
```

The function **averageAndGrade** has four parameters: **testScore** and **progScore** are value parameters of type `int`, **average** is a reference parameter of type `double`, and **grade** is a reference parameter of type `char`. Using visual diagrams, Examples 6-13 to 6-15 explicitly show how value and reference parameters work.

Value Parameters

When a function is called, for a value parameter, the value of the actual parameter is copied into the corresponding formal parameter. Therefore, if the formal parameter is a value parameter, then after copying the value of the actual parameter, there is no connection between the formal parameter and actual parameter; that is, the formal parameter is a separate variable with its own copy of the data. Therefore, during program execution, the formal parameter manipulates the data stored in its own memory space.

EXAMPLE

The following program shows how a formal parameter of a simple data type works.

```
//Example 6-12: Program illustrating how a value parameter works.

#include <iostream>                                    //Line 1

using namespace std;                                   //Line 2
```

```
void funcValueParam(int num);                                   //Line 3

int main()                                                      //Line 4
{                                                               //Line 5
    int number = 6;                                             //Line 6

    cout << "Line 7: Before calling the function "
         << "funcValueParam, number = " << number
         << endl;                                               //Line 7

    funcValueParam(number);                                     //Line 8

    cout << "Line 9: After calling the function "
         << "funcValueParam, number = " << number
         << endl;                                               //Line 9

    return 0;                                                   //Line 10
}                                                               //Line 11

void funcValueParam(int num)                                    //Line 12
{                                                               //Line 13
    cout << "Line 14: In the function funcValueParam, "
         << "before changing, num = " << num
         << endl;                                               //Line 14

    num = 15;                                                   //Line 15

    cout << "Line 16: In the function funcValueParam, "
         << "after changing, num = " << num
         << endl;                                               //Line 16
}                                                               //Line 17
```

Sample Run:

```
Line 7: Before calling the function funcValueParam, number = 6
Line 14: In the function funcValueParam, before changing, num = 6
Line 16: In the function funcValueParam, after changing, num = 15
Line 9: After calling the function funcValueParam, number = 6
```

This program works as follows. The execution begins at the function main. The statement in Line 6 declares and initializes the int variable number. The statement in Line 7 outputs the value of number before calling the function funcValueParam; the statement in Line 8 calls the function funcValueParam. The value of the variable number is then passed to the formal parameter num. Control now transfers to the function funcValueParam. The statement in Line 14 outputs the value of num before changing its value. The statement in Line 15 changes the value of num to 15; the

statement in Line 16 outputs the value of num. After this statement executes, the function funcValueParam exits and control goes back to the function main

The statement in Line 9 outputs the value of number after calling the function funcValueParam. The sample run shows that the value of number (Lines 7 and 9) remains the same even though the value of its corresponding formal parameter num was changed within the function funcValueParam. The output shows the sequence in which the statements execute.

After copying data, a value parameter has no connection with the actual parameter, so a value parameter cannot pass any result back to the calling function. When the function executes, any changes made to the formal parameters do not in any way affect the actual parameters. The actual parameters have no knowledge of what is happening to the formal parameters. Thus, value parameters cannot pass information outside of the function. Value parameters provide only a one-way link from the actual parameters to the formal parameters. Hence, functions with only value parameters have limitations

Reference Variables as Parameters

Because a reference parameter receives the address (memory location) of the actual parameter, reference parameters can pass one or more values from a function and can change the value of the actual parameter. Reference parameters are useful in three situations:

- When the value of the actual parameter needs to be changed
- When you want to return more than one value from a function (recall that the return statement can return only one value)
- When passing the address would save memory space and time relative to copying a large amount of data

The first two situations are illustrated throughout this book. Later Chapters will discuss the third situation, when arrays and classes are introduced. Recall that when you attach & after the dataType in the formal parameter list of a function, the variable following that dataType becomes a reference parameter.

EXAMPLE

The following program shows how reference and value parameters work.

```
//Example 6-14: Reference and value parameters

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

void funOne(int a, int& b, char v);                          //Line 3
void funTwo(int& x, int y, char& w);                         //Line 4

int main()                                                   //Line 5
{                                                            //Line 6
    int num1, num2;                                          //Line 7
    char ch;                                                 //Line 8

    num1 = 10;                                               //Line 9
    num2 = 15;                                               //Line 10
    ch = 'A';                                                //Line 11

    cout << "Line 12: Inside main: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                      //Line 12

    funOne(num1, num2, ch);                                  //Line 13

    cout << "Line 14: After funOne: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                      //Line 14

    funTwo(num2, 25, ch);                                    //Line 15

    cout << "Line 16: After funTwo: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                      //Line 16

    return 0;                                                //Line 17
}                                                            //Line 18

void funOne(int a, int& b, char v)                           //Line 19
{                                                            //Line 20
    int one;                                                 //Line 21

    one = a;                                                 //Line 22
    a++;                                                     //Line 23
    b = b * 2;                                               //Line 24
    v = 'B';                                                 //Line 25
```

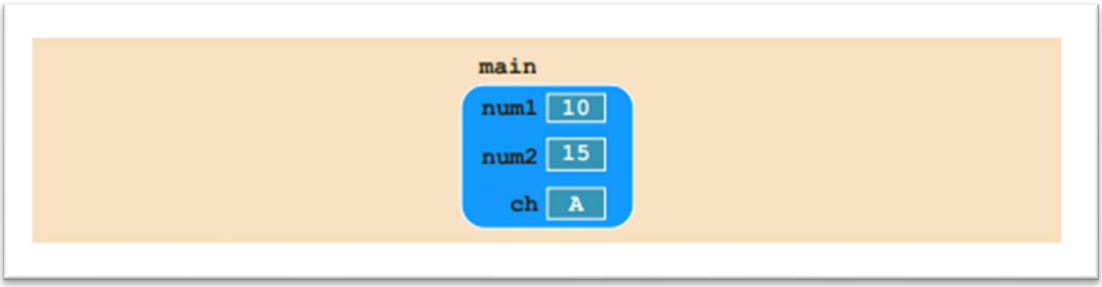
```
void funTwo(int& x, int y, char& w)                    //Line 28
{                                                      //Line 29
    x++;                                               //Line 30
    y = y * 2;                                         //Line 31
    w = 'G';                                           //Line 32

    cout << "Line 33: Inside funTwo: x = " << x
         << ", y = " << y << ", and w = " << w
         << endl;                                      //Line 33
}                                                      //Line 34
```

Sample Run:

```
Line 12: Inside main: num1 = 10, num2 = 15, and ch = A
Line 26: Inside funOne: a = 11, b = 30, v = B, and one = 10
Line 14: After funOne: num1 = 10, num2 = 30, and ch = A
Line 33: Inside funTwo: x = 31, y = 50, and w = G
Line 16: After funTwo: num1 = 10, num2 = 31, and ch = G
```

Let us walk through this program. The values of the variables are shown before and/ or after each statement executes.

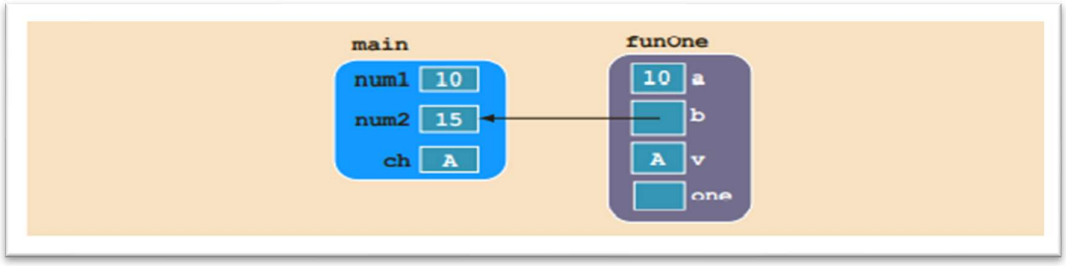
Just before the statement in Line 9 executes, memory is allocated only for the variables of the function main; this memory is not initialized. After the statement in Line 11 executes, the variables are as shown in Figure below.



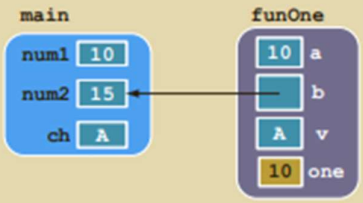
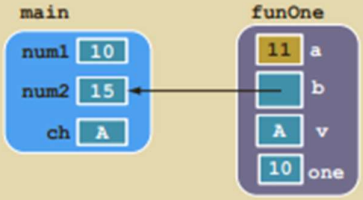
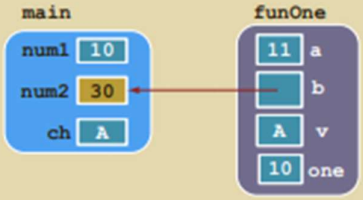
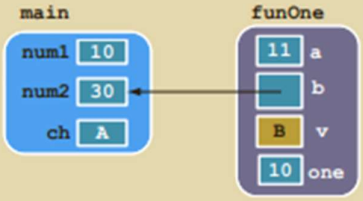
The statement in Line 12 produces the following output:

Line 12: Inside main: num1 = 10, num2 = 15, and ch = A

The statement in Line 13 is a function call to the function funOne. Now function funOne has three parameters (a, b, and v) and one local variable (one). Memory for the parameters and the local variable of function funOne is allocated. Because the formal parameter b is a reference parameter, it receives the address (memory location) of the corresponding actual parameter, which is num2. The other two formal parameters are value parameters, so they copy the values of their corresponding actual parameters. Just before the statement in Line 22 executes, the variables are as shown in Figure below



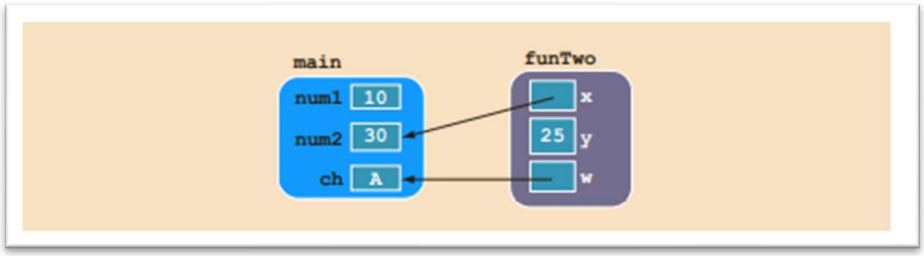
The following shows how the variables are manipulated after each statement from Lines 22 to 25 executes.

St. in Line	Value of the Variables	Statement and Effect
22		`one = a;` Copy the value of **a** into one.
23		`a++;` Increment the value of **a** by **1**.
24		`b = b * 2;` Multiply the value of **b** by **2** and store the result in **b**. Because **b** is the reference parameter and contains the address of **num**, the value of **num** is updated.
25		`v = 'B';` Store `'B'` into **v**.

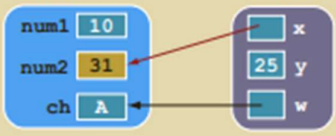
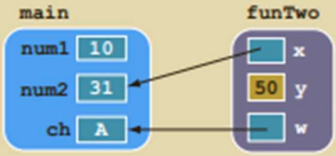
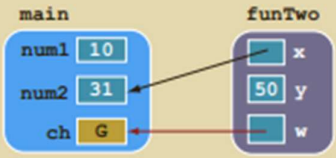
The statement in Line 26 produces the following output:

Line 26: Inside funOne: a = 11, b = 30, v = B, and one = 10

After the statement in Line 26 executes, control goes back to Line 14 in main and the memory allocated for the variables of function funOne is deallocated. Figure below shows the values of the variables of the function main



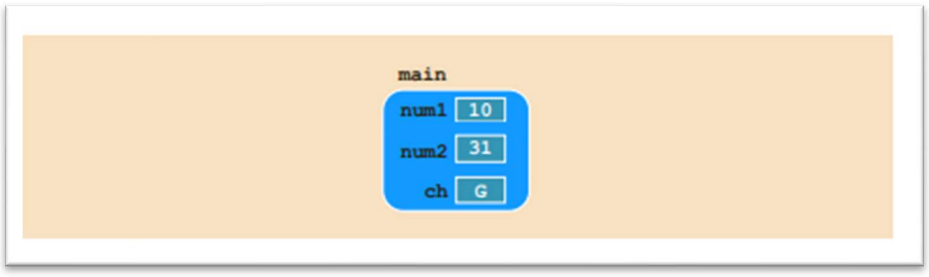
The following shows how the variables are manipulated after each statement from Lines 30 to 32 executes

St. in Line	Value of the Variables	Statement and Effect
30		`x++;` Increment the value of `x` by `1`. Because `x` is the reference parameter and contains the address of `num2`, the value of `num2` is incremented by `1`.
31		`y = y * 2;` Multiply the value of `y` by `2` and store the result in `y`.
32		`w = 'G';` Store `'G'` in `w`. Because `w` is the reference parameter and contains the address of `ch`, the value of `ch` is updated.

Line 33 produces the following output:

Line 33: Inside funTwo: x = 31, y = 50, and w = G

After the statement in Line 33 executes, control goes to Line 16. The memory allocated for the variables of function funTwo is deallocated. The values of the variables of the function main are as shown in Figure below



The statement in Line 16 produces the following output:

Line 16: After funTwo: num1 = 10, num2 = 31, and ch = G

After the statement in Line 16 executes, the program terminates.

Function Overloading

In a C++ program, several functions can have the same name. This is called function overloading, or overloading a function name.

Two functions are said to have different formal parameter lists if both functions have

- A different number of formal parameters or
- The same number of formal parameters and the data types of the formal parameters, in the order listed, differ in at least one position.

For example, consider the following function headings:

```
void functionOne(int x)
void functionTwo(int x, double y)
void functionThree(double y, int x)
int functionFour(char ch, int x, double y)
int functionFive(char ch, int x, string name)
```

Function overloading: Creating several functions with the same name. The signature of a function consists of the function name and its formal parameter list. Two functions have different signatures if they have either different names or different formal parameter lists. (Note that the signature of a function does not include the return type of the function.) If a function's name is overloaded, then

all of the functions in the set have the same name. Therefore, all the functions in the overloaded set must have different formal parameter lists.

Thus, the following function headings correctly overload the function functionXYZ:

```
void functionXYZ()
void functionXYZ(int x, double y)
void functionXYZ(double one, int y)
void functionXYZ(int x, double y, char ch)
```

QUICK REVIEW

1. Functions, also called modules, are like miniature programs.
2. The C++ system provides the standard (predefined) functions.
3. There are two types of user-defined functions: value-returning functions and void functions.
4. Expressions, variables, or constant values used in a function call are called actual parameters.
5. A value-returning function returns a value. Therefore, a value-returning function is used (called) in either an expression or an output statement or as a parameter in a function call.
6. A function can have more than one return statement. However, whenever a return statement executes in a function, the remaining statements are skipped and the function exits.
7. A value parameter receives a copy of its corresponding actual parameter
8. reference parameter receives the address (memory location) of its corresponding actual parameter.
9. A constant value cannot be passed to a reference parameter
10. Variables declared outside of every function definition (and block) are called global variables.
11. C++ does not allow the nesting of function definitions.

12. If a function is overloaded, then in a call to that function the formal parameter list of the function determines which function to execute

PROGRAMMING EXERCISES

1. Write a value-returning function, isVowel, that returns the value true if a given character is a vowel and otherwise returns false.
2. Write a program that uses the function isPalindrome given in Example 6-6 (Palindrome). Test your program on the following strings: "madam", "abba", "22", "67876", "444244", and "trymeuemyrt".
3. Write a program that find the area of a rectangle, the area of a circle, or the volume of a cylinder. Your program must be properly indented. (Note that the program is menu driven and allows the user to run the program as long as the user wishes.)
4. Write a function that takes as a parameter an integer (as a long long value) and returns the number of odd, even, and zero digits. Also write a program to test your function.

TOPIC 5
COMPOUND DATA TYPES

LEARNING OUTCOMES:

After studying this topic you should be able to:

•Explore the string data type, and learn how to use string functions to manipulate strings.
•Learn how to declare and initialize arrays.
• Discover how to pass an array as a parameter to a function
• Learn how to search an array
• Learn how to sort an array
• Learn about parallel arrays
• Learn about multidimensional arrays
• Discover how to manipulate data in a two-dimensional array
• Learn about the pointer data type and pointer variables
• Explore how to declare and manipulate pointer variables
• Learn about the address of the operator and the dereferencing operator

5.1 INTRODUCTION

Recall that prior to the ANSI/ISO C++ language standard, the Standard C++ library did not provide a string data type. Compiler vendors often supplied their own programmer-defined string type, and the syntax and semantics of string operations often varied from vendor to vendor. The data type string is a programmer-defined type and is not part of the C++ language; the C++ standard library supplies it. Before using the data type string, the program must include the header file string, as follows

#include <string>

Recall that in C++, a string is a sequence of zero or more characters, and strings are enclosed in double quotation marks.

The statement:

string name = "William Jacob"; declares name to be a string variable and initializes name to "William Jacob"

Other operators, such as the binary operator + (to allow the string concatenation operation) and the array index (subscript) operator [], have also been defined for the data type string. Let's see how these operators work on the string data type. Suppose you have the following declarations:

string str1, str2, str3;

The statement:

str1 = "Hello There";

stores the string "Hello There" in str1. The statement:

str2 = str1;

copies the value of str1 into str2.

If str1 = "Sunny", the statement:

str2 = str1 + " Day";

stores the string "Sunny Day" into str2.

Suppose str1 = "Hello" and str2 = "There". The statement:

str3 = str1 + " " + str2;

stores "Hello There" into str3. This statement is equivalent to the statement:

str3 = str1 + ' ' + str2;

For the operator + to work with the `string` data type, one of the operands of + must be a `string` variable. For example, the following statements will not work:

```
str1 = "Hello " + "there!";   //illegal
str2 = "Sunny Day" + '!';     //illegal
```

If str1 = "Hello there", the statement:

str1[6] = 'T';

replaces the character t with the character T. Recall that the position of the first character in a string variable is 0. Therefore, because t is the seventh character in str1, its position is 6. In C++, [] is called the array subscript operator.

Additional string Operations

The data type string has a data type, **string::size_type**, and a named constant, **string::npos**, defined as follows:

`string::size_type`	An unsigned integer (data) type
`string::npos`	The maximum value of the (data) type `string::size_type`, a number such as `4294967295` on many machines

The data type string contains several other functions for string manipulation. Table below describes some of these functions. In this table, we assume that strVar is a string variable and str is a string variable, a string constant, or a character array.

Expression	Effect
`strVar.at(index)`	Returns the element at the position specified by **index**.
`strVar[index]`	Returns the element at the position specified by **index**.
`strVar.append(n, ch)`	Appends **n** copies of **ch** to **strVar**, where **ch** is a `char` variable or a `char` constant.
`strVar.append(str)`	Appends **str** to **strVar**.
`strVar.clear()`	Deletes all the characters in **strVar**.
`strVar.compare(str)`	Returns **1** if **strVar > str**; returns **0** if **strVar == str**; returns **-1** if **strVar < str**.
`strVar.empty()`	Returns `true` if **strVar** is empty; otherwise it returns `false`.
`strVar.erase()`	Deletes all the characters in **strVar**.
`strVar.erase(pos, n)`	Deletes **n** characters from **strVar** starting at position **pos**.

EXAMPLE

Consider the following statements:

```
string firstName = "Elizabeth";
string name = firstName + " Jones";
string str1 = "It is sunny.";
string str2 = "";
string str3 = "computer science";
string str4 = "C++ programming.";
string str5 = firstName + " is taking " + str4;
string::size_type len;
```

Next, we show the effect of `clear`, `empty`, `erase`, `length`, and `size` functions.

Statement	Effect
`str3.clear();`	`str3 = "";`
`str1.empty();`	Returns `false`
`str2.empty();`	Returns `true`
`str4.erase(11, 4);`	`str4 = "C++ program.";`
`cout << firstName.length() << endl;`	Outputs **9**
`cout << name.length() << endl;`	Outputs **15**
`cout << str1.length() << endl;`	Outputs **12**
`cout << str5.size() << endl;`	Outputs **36**
`len = name.length();`	The value of `len` is **15**

The following program illustrates the use of the length function. //Example: clear, empty, erase, length, and size functions.

```
#include <iostream>                                    //Line 1
#include <string>                                      //Line 2

using namespace std;                                   //Line 3

int main()                                             //Line 4
{                                                      //Line 5
    string firstName = "Elizabeth";                    //Line 6
    string name = firstName + " Jones";                //Line 7
    string str1 = "It is sunny.";                      //Line 8
    string str2 = "";                                  //Line 9
    string str3 = "computer science";                  //Line 10
    string str4 = "C++ programming.";                  //Line 11
    string str5 = firstName + " is taking " + str4;    //Line 12
```

```
    string::size_type len;                                  //Line 13

    cout << "Line 14: str3: " << str3 << endl;              //Line 14
    str3.clear();                                           //Line 15
    cout << "Line 16: After clear, str3: " << str3
         << endl;                                           //Line 16

    cout << "Line 17: str1.empty(): " << str1.empty()
         << endl;                                           //Line 17
    cout << "Line 18: str2.empty(): " << str2.empty()
         << endl;                                           //Line 18

    cout << "Line 19: str4: " << str4 << endl;              //Line 19
    str4.erase(11, 4);                                      //Line 20
    cout << "Line 21: After erase(11, 4), str4: "
         << str4 << endl;                                   //Line 21

    cout << "Line 22: Length of \"" << firstName << "\" = "
         << static_cast<unsigned int> (firstName.length())
         << endl;                                           //Line 22
    cout << "Line 23: Length of \"" << name << "\" = "
         << static_cast<unsigned int> (name.length())
         << endl;                                           //Line 23
    cout << "Line 24: Length of \"" << str1 << "\" = "
         << static_cast<unsigned int> (str1.length())
         << endl;                                           //Line 24
    cout << "Line 25: Size of \"" << str5 << "\" = "
         << static_cast<unsigned int> (str5.size())
         << endl;                                           //Line 25

    len = name.length();                                    //Line 26
    cout << "Line 27: len = "
         << static_cast<unsigned int> (len) << endl;        //Line 27

    return 0;                                               //Line 28
}                                                           //Line 29
```

Sample Run:

```
Line 14: str3: computer science
Line 16: After clear, str3:
Line 17: str1.empty(): 0
Line 18: str2.empty(): 1
Line 19: str4: C++ programming.
Line 21: After erase(11, 4), str4: C++ program.
Line 22: Length of "Elizabeth" = 9
Line 23: Length of "Elizabeth Jones" = 15
Line 24: Length of "It is sunny." = 12
Line 25: Size of "Elizabeth is taking C++ programming." = 36
Line 27: len = 15
```

Insert AND replace Functions

EXAMPLE

Suppose that you have the following statements:

```
string firstString = "Cloudy and warm.";
string secondString ="Hello there";
string thirdString = "Henry is taking programming I.";
string str1 = " very ";
string str2 = "Lisa";
```

Next, we show the effect of `insert` and `replace` functions.

Statement	Effect
`firstString.insert(10, str1);`	`firstString = "Cloudy and very warm."`
`secondString.insert(11, 5, '!');`	`secondString = "Hello there!!!!!"`
`thirdString.replace(0, 5, str2);`	`thirdString = "Lisa is taking programming I."`

The following program evaluates the previous statements.

```
//Example: insert and replace functions

#include <iostream>                                          //Line 1
#include <string>                                            //Line 2

using namespace std;                                         //Line 3

int main()                                                   //Line 4
{                                                            //Line 5
    string firstString = "Cloudy and warm.";                 //Line 6
    string secondString = "Hello there";                     //Line 7
    string thirdString = "Henry is in programming I.";       //Line 8
    string str1 = " very ";                                  //Line 9
    string str2 = "Lisa";                                    //Line 10

    cout << "Line 11: firstString = " << firstString
         << endl;                                            //Line 11
    firstString.insert(10, str1);                            //Line 12
```

```
    cout << "Line 13: After insert; firstString = "
         << firstString << endl;                          //Line 13

    cout << "Line 14: secondString = " << secondString
         << endl;                                         //Line 14
    secondString.insert(11, 5, '!');                      //Line 15
    cout << "Line 16: After insert; secondString = "
         << secondString << endl;                         //Line 16

    cout << "Line 17: thirdString = " << thirdString
         << endl;                                         //Line 17
    thirdString.replace(0, 5, str2);                      //Line 18
    cout << "Line 19: After replace, thirdString = "
         << thirdString << endl;                          //Line 19

    return 0;                                             //Line 20
}                                                         //Line 21
```

Sample Run:

```
Line 11: firstString = Cloudy and warm.
Line 13: After insert; firstString = Cloudy and very  warm.
Line 14: secondString = Hello there
Line 16: After insert; secondString = Hello there!!!!!
Line 17: thirdString = Henry is in programming I.
Line 19: After replace, thirdString = Lisa is in programming I.
```

The output of this program is self-explanatory. The details are left as an exercise for you.

EXAMPLE

The `swap` function is used to swap—that is, interchange—the contents of two string variables.

Suppose you have the following statements:

```
string str1 = "Warm";
string str2 = "Cold";
```

After the following statement executes, the value of `str1` is `"Cold"` and the value of `str2` is `"Warm"`.

```
str1.swap(str2);
```

PROGRAMMING EXAMPLE: Pig Latin Strings

In this programming example, we write a program that prompts the user to input a string and then outputs the string in the pig Latin form. The rules for converting a string into pig Latin form are as follows:

1. If the string begins with a vowel, add the string `"-way"` at the end of the string. For example, the pig Latin form of the string `"eye"` is `"eye-way"`.
2. If the string does not begin with a vowel, first add `"-"` at the end of the string. Then rotate the string one character at a time; that is, move the first character of the string to the end of the string until the first character of the string becomes a vowel. Then add the string `"ay"` at the end. For example, the pig Latin form of the string `"There"` is `"ere-Thay"`.
3. Strings such as `"by"` contain no vowels. In cases like this, the letter `y` can be considered a vowel. So, for this program, the vowels are `a`, `e`, `i`, `o`, `u`, `y`, `A`, `E`, `I`, `O`, `U`, and `Y`. Therefore, the pig Latin form of `"by"` is `"y-bay"`.
4. Strings such as `"1234"` contain no vowels. The pig Latin form of the string `"1234"` is `"1234-way"`. That is, the pig Latin form of a string that has no vowels in it is the string followed by the string `"-way"`.

Input Input to the program is a string.

Output Output of the program is the string in the pig Latin form.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

Suppose that `str` denotes a string. To convert `str` into pig Latin, check the first character, `str[0]`, of `str`. If `str[0]` is a vowel, add `"-way"` at the end of `str`—that is, `str = str + "-way"`.

Suppose that the first character of `str`, `str[0]`, is not a vowel. First, add `"-"` at the end of the string. Then, remove the first character of `str` from `str` and put it at the end of `str`. Now, the second character of `str` becomes the first character of `str`. This process of checking the first character of `str` and moving it to the end of `str` if the first character of `str` is not a vowel is repeated until either the first character of `str` is a vowel or all the characters of `str` are processed, in which case `str` does not contain any vowels.

In this program, we write a function `isVowel` to determine whether a character is a vowel, a function `rotate` to move the first character of `str` to the end of `str`, and a function `pigLatinString` to find the pig Latin form of `str`. The previous discussion translates into the following algorithm:

1. Get `str`.
2. Find the pig Latin form of `str` by using the function `pigLatinString`.
3. Output the pig Latin form of `str`.

Before writing the main algorithm, each of these functions is described in detail.

The definition of the function pigLatinString is:

```
string pigLatinString(string pStr)
{
    string::size_type len;

    bool foundVowel;

    string::size_type counter;

    if (isVowel(pStr[0]))                          //Step 1
        pStr = pStr + "-way";
    else                                           //Step 2
    {
        pStr = pStr + '-';
        pStr = rotate(pStr);                       //Step 3

        len = pStr.length();                       //Step 3.a
        foundVowel = false;                        //Step 3.b

        for (counter = 1; counter < len - 1;
                          counter++)               //Step 3.d
            if (isVowel(pStr[0]))
            {
                foundVowel = true;
                break;
            }
            else                                   //Step 3.c
                pStr = rotate(pStr);

        if (!foundVowel)                           //Step 4
            pStr = pStr.substr(1, len) + "-way";
        else
            pStr = pStr + "ay";
    }

    return pStr;                                   //Step 5
}
```

PROGRAM LISTING

```
#include <string>

using namespace std;

bool isVowel(char ch);
string rotate(string pStr);
string pigLatinString(string pStr);

int main()
{
    string str;

    cout << "Enter a string: ";
    cin >> str;
    cout << endl;

    cout << "The pig Latin form of " << str << " is: "
         << pigLatinString(str) << endl;

    return 0;
}
```

Sample Runs: In these sample runs, the user input is shaded.

Sample Run 1:

```
Enter a string: eye

The pig Latin form of eye is: eye-way
```

Sample Run 2:

```
Enter a string: There

The pig Latin form of There is: ere-Thay
```

Sample Run 3:

```
Enter a string: why

The pig Latin form of why is: y-whay
```

Sample Run 4:

```
Enter a string: 123456

The pig Latin form of 123456 is: 123456-way
```

5.2 ARRAYS

An array is a collection of a fixed number of components (also called elements) all of the same data type and in contiguous (that is, adjacent) memory space. A one-dimensional array is an array in which the components are arranged in a list

The general form for declaring a one-dimensional array is:

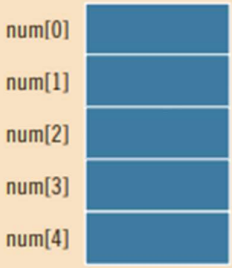
```
dataType arrayName[intExp];
```

in which **intExp** specifies the number of components in the array and can be any constant expression that evaluates to a positive integer.

The statement:

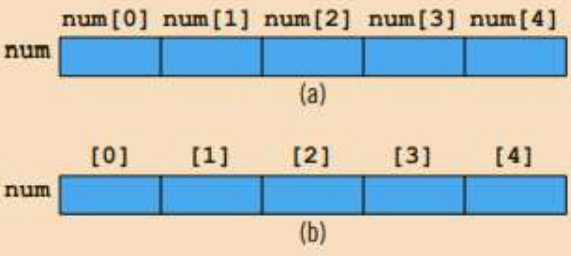
```
int num[5];
```

declares an array num of five components. Each component is of type int. The component names are num[0], num[1], num[2], num[3], and num[4]. Figure 8-1 illustrates the array num.



NOTE

To save space, we also draw an array, as shown in Figure 8-2(a) or 8-2(b).



EXAMPLE

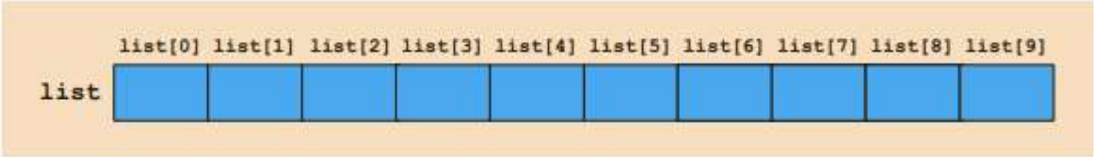
Accessing Array Components

The general form (syntax) used for accessing an array component is:

```
arrayName[indexExp]
```

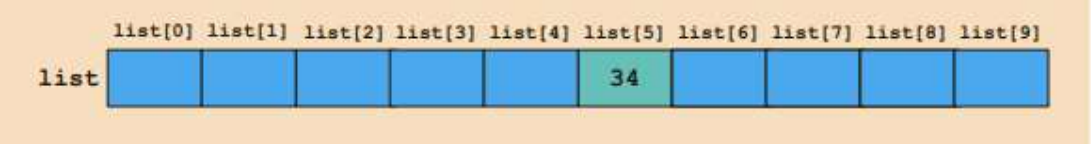
in which indexExp, called the index, is any expression whose value is a nonnegative integer. The index value specifies the position of the component in the array. In C++, [] is an operator called the array subscripting operator. Moreover, in C++, the array index starts at 0.

Consider the following statement: int list[10]; This statement declares an array list of 10 components. The components are list[0], list[1], . . ., list[9]. In other words, we have declared 10 variables



The assignment statement:

list[5] = 34; stores 34 in list[5], which is the sixth component of the array list



EXAMPLE

You can also declare arrays as follows:

```
const int ARRAY_SIZE = 10;
int list[ARRAY_SIZE];
```

That is, you can first declare a named constant and then use the value of the named constant to declare an array and specify its size.

Processing One-Dimensional Arrays

Some of the basic operations performed on a one-dimensional array are initializing, inputting data, outputting data stored in an array, and finding the largest and/or smallest element. Moreover, if the data is numeric, some other basic operations are finding the sum and average of the elements of the array. Each of these operations requires the ability to step through the elements of the array. This is easily accomplished using a loop. For example, suppose that we have the following statements:

```
int list[100]; //list is an array of size 100
int i;
```

The following for loop steps through each element of the array list, starting at the first element of list:

```
for (i = 0; i < 100; i++) //Line 1
        //process list[i] //Line 2
```

If processing the list requires inputting data into list, the statement in Line 2 takes the form of an input statement, such as the cin statement. For example, the following statements read 100 numbers from the keyboard and store the numbers in **list:**

```
for (i = 0; i < 100; i++) //Line 1
            cin >> list[i]; //Line 2
```

Similarly, if processing list requires outputting the data, then the statement in Line 2 takes the form of an output statement. For example, the following statements output the numbers stored in list.

```
for (i = 0; i < 100; i++)         //Line 1
        cout << list[i] << " "; //Line 2
cout << endl
```

EXAMPLE

```
//Program to find the average test score and output the average
//test score and all the test scores that are less than
//the average test score.

#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int test[5];
    int sum = 0;
    double average;
    int index;


       cout << fixed << showpoint << setprecision(2);

       cout << "Enter five test scores: ";

       for (index = 0; index < 5; index++)
       {
           cin >> test[index];
           sum = sum + test[index];
       }

       cout << endl;

       average = sum / 5.0;

       cout << "The average test score = " << average << endl;

       for (index = 0; index < 5; index++)
           if (test[index] < average)
               cout << test[index]
                    << " is less than the average "
                    << "test score." << endl;
    return 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter five test scores: 78 87 65 94 60

The average test score = 76.80
65 is less than the average test score.
60 is less than the average test score.
```

Array Index Out of Bounds

Consider the following declaration:

```
double num[10];
 int i;
```

The component num[i] is valid, that is, i is a valid index if i = 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

The index—say, index—of an array is in bounds if index is between 0 and ARRAY_ SIZE - 1, that is, 0 <= index <= ARRAY_SIZE - 1. If index is negative or index is greater than ARRAY_SIZE - 1, then we say that the index is out of bounds.

Unfortunately, C++ does not check whether the index value is within range—that is, between 0 and ARRAY_SIZE - 1. If the index goes out of bounds and the program tries to access the component specified by the index, then whatever memory location is indicated by the index that location is accessed. This situation can result in altering or accessing the data of a memory location that you never intended to modify or access, or in trying to access protected memory that causes the program to instantly halt. Consequently, several strange things can happen if the index goes out of bounds during execution. It is solely the programmer's responsibility to make sure that the index is within bounds.

Consider the following statement:

```
int list[10];
```

A loop such as the following can set the index of list out of bounds:

```
 for (int i = 0; i <= 10; i++)
        list[i] = 0;
```

When i becomes 10, the loop test condition i <= 10 evaluates to true and the body of the loop executes, which results in storing 0 in list[10]. Logically, list[10] does not exist

Arrays as Parameters to Functions

By reference only: In C++, arrays are passed by reference only. Because arrays are passed by reference only, you do not use the symbol & when declaring an array as a formal parameter. When declaring a one-dimensional array as a formal parameter, the size of the array is usually omitted. If you specify the size of a one-dimensional array when it is declared as a formal parameter, the size is ignored by the compiler.

EXAMPLE

Consider the following function:

```
void funcArrayAsParam(int listOne[], double listTwo[])
{
    .
    .
    .
}
```

The function `funcArrayAsParam` has two formal parameters: (1) `listOne`, a one-dimensional array of type `int` (that is, the component type is `int`) and (2) `listTwo`, a one-dimensional array of type `double`. In this declaration, the size of both arrays is unspecified.

Sometimes, the number of elements in the array might be less than the size of the array. For example, the number of elements in an array storing student data might increase or decrease as students drop or add courses. In such situations, we want to process only the components of the array that hold actual data. To write a function to process such arrays, in addition to declaring an array as a formal parameter, we declare another formal parameter specifying the number of elements in the array, as in the following function:

```
void initialize(int list[], int listSize)
{
    for (int count = 0; count < listSize; count++)
        list[count] = 0;
}
```

The first parameter of the function initialize is an int array of any size. When the function initialize is called, the size of the actual array is passed as the second parameter of the function initialize.

EXAMPLE

Suppose that the distance traveled by an object at time $t = a_1$ is d_1 and at time $t = a_2$ is d_2, where $a_1 < a_2$. Then the average speed of the object from time a_1 to a_2, that is, over the interval $[a_1, a_2]$ is $(d_2 - d_1)/(a_2 - a_1)$. Suppose that the distance traveled by an object at certain times is given by the following table:

Time	0	10	20	30	40	50
Distance traveled	0	18	27	38	52	64

Then the average speed over the interval [0, 10] is $(18 - 0)/(10 - 0) = 1.8$, over the interval [10, 20] is $(27 - 18)/(20 - 10) = 0.9$, and so on.

The following program takes as input the distance traveled by an object at time 0, 10, 20, 30, 40, and 50. The program then outputs the average speed over the intervals $[10 * i, 10 * (i + 1)]$, where $i = 0, 1, 2, 3$, and 4. The program also outputs the maximum and minimum average speed over these intervals. Programming Exercise 17, at the end of this chapter, asks you to modify this program so that the distance traveled by an object recorded is not necessarily after every 10 time units.

```
#include <iostream>
#include <iomanip>

using namespace std;

const int SIZE = 6;

void getData(double list[], int length);
void averageSpeedOverTimeInterval(double list[], int length,
                                  double avgSpeed[]);
double maxAvgSpeed(double avgSpeed[], int length);
double minAvgSpeed(double avgSpeed[], int length);
void print(double list[], int length, double avgSpeed[]);

int main()
{
    double distanceTraveled[SIZE];
    double averageSpeed[SIZE];

    cout << fixed << showpoint << setprecision(2);

    getData(distanceTraveled, SIZE);
    averageSpeedOverTimeInterval(distanceTraveled, SIZE, averageSpeed);
    print(distanceTraveled, SIZE, averageSpeed);

    cout << "Maximum average speed: "
         << maxAvgSpeed(averageSpeed, SIZE) << endl;
    cout << "Minimum average speed: "
         << minAvgSpeed(averageSpeed, SIZE) << endl;

    return 0;
}
```

```
void getData(double list[], int length)
{
    cout << "Enter the total distance traveled after "
         << "every 10 units of time." << endl;



    for (int index = 0; index < length; index++)
    {
        cout << "Enter total distance traveled at time "
             << index * 10 << " units: ";
        cin >> list[index];
        cout << endl;
    }
}

  void averageSpeedOverTimeInterval(double list[], int length,
                                    double avgSpeed[])
  {
      for (int index = 0; index < length - 1; index++)
          avgSpeed[index] = (list[index + 1] - list[index]) / 10;
  }

  double maxAvgSpeed(double avgSpeed[], int length)
  {
      double max = avgSpeed[0];

      for (int index = 1; index < length - 1; index++)
          if (avgSpeed[index] > max)
              max = avgSpeed[index];

      return max;
  }

  double minAvgSpeed(double avgSpeed[], int length)
  {
      double min = avgSpeed[0];

      for (int index = 1; index < length - 1; index++)
          if (avgSpeed[index] < min)
              min = avgSpeed[index];

      return min;
  }
```

```
void print(double list[], int length, double avgSpeed[])
{
    cout << setw(7) << "Time " << setw(20) << "Distance Traveled  "
         << setw(10) << "Average Speed / Time Interval" << endl;

    cout << setw(5) << 0
         << setw(14) << list[0] << setw(6) << " "
         << setw(10) << 0 << "   [0, 0] " << endl;

    for (int index = 1; index < length; index++)
        cout << setw(5) << index * 10
             << setw(14) << list[index] << setw(6) << " "
             << setw(10) << avgSpeed[index - 1]


             << "   [" << (index - 1) * 10 << ", "
             << index * 10 << "]" << endl;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Enter the total distance traveled after every 10 units of time.
Enter total distance traveled at time 0 units: 0

Enter total distance traveled at time 10 units: 30

Enter total distance traveled at time 20 units: 45

Enter total distance traveled at time 30 units: 58

Enter total distance traveled at time 40 units: 67

Enter total distance traveled at time 50 units: 80

  Time  Distance Traveled  Average Speed / Time Interval
    0          0.00              0   [0, 0]
   10         30.00           3.00   [0, 10]
   20         45.00           1.50   [10, 20]
   30         58.00           1.30   [20, 30]
   40         67.00           0.90   [30, 40]
   50         80.00           1.30   [40, 50]
Maximum average speed: 3.00
Minimum average speed: 0.90
```

Other Ways to Declare Arrays

Suppose that a class has 20 students and you need to keep track of their scores. Because the number of students can change from semester to semester, instead of specifying the size of the array while declaring it, you can declare the array as follows:

```
const int NO_OF_STUDENTS = 20;
int testScores[NO_OF_STUDENTS];
```

Other forms used to declare arrays are:

```
const int SIZE = 50;                    //Line 1
typedef double list[SIZE];              //Line 2

list yourList;                          //Line 3
 list myList;                           //Line  4
```

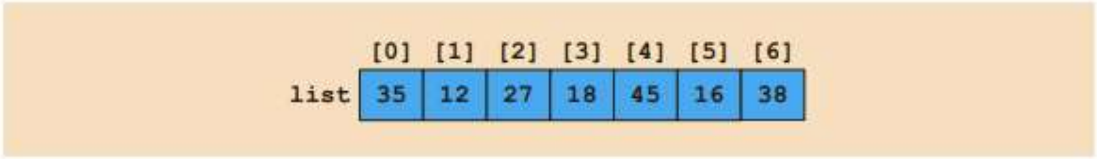
The statement in Line 2 defines a data type list, which is an array of 50 components of type double. The statements in Lines 3 and 4 declare two variables, yourList and myList. Both are arrays of 50 components of type double. Of course, these statements are equivalent to:

```
double yourList[50];
double myList[50];
```

Searching an Array for a Specific Item

Searching a list for a given item is one of the most common operations performed on a list. The search algorithm we describe is called the sequential search or linear search. As the name implies, you search the array sequentially, starting from the first array element. You compare searchItem with the elements in the array (the list) and continue the search until either you find the item or no more data is left in the list to compare with searchItem.

Consider the list of seven elements shown in Figure below



Suppose that you want to determine whether 27 is in the list. A sequential search works as follows: First, you compare 27 with list[0], that is, compare 27 with 35. Because list[0] ≠ 27, you then compare 27 with list[1], that is, with 12, the second item

in the list. Because list[1] ≠ 27, you compare 27 with the next element in the list, that is, compare 27 with list[2]. Because list[2] = 27, the search stops. This search is successful.

Let us now search for 10. As before, the search starts at the first element in the list, that is, at list[0]. Proceeding as before, we see that, this time, the search item, which is 10, is compared with every item in the list. Eventually, no more data is left in the list to compare with the search item. This is an unsuccessful search.

It now follows that, as soon as you find an element in the list that is equal to the search item, you must stop the search and report success. (In this case, you usually also report the location in the list where the search item was found.) Otherwise, after the search item is unsuccessfully compared with every element in the list, you must stop the search and report failure.
Suppose that the name of the array containing the list elements is list. The previous discussion translates into the following algorithm for the sequential search:

```
found is set to false
loc = 0;

while (loc < listLength and not found)
    if (list[loc] is equal to searchItem)
        found is set to true
    else
        increment loc

if (found)
    return loc;
else
    return -1;
```

The following function performs a sequential search on a list. To be specific, and for illustration purposes, we assume that the list elements are of type int

```
int seqSearch(const int list[], int listLength, int searchItem)
{
    int loc;
    bool found = false;

    loc = 0;

    while (loc < listLength && !found)
        if (list[loc] == searchItem)
            found = true;
        else
            loc++;


    if (found)
        return loc;
    else
        return -1;
}
```

If the function seqSearch returns a value greater than or equal to 0, it is a successful search; otherwise, it is an unsuccessful search.

As you can see from this code, you start the search by comparing searchItem with the first element in the list. If searchItem is equal to the first element in the list, you exit the loop; otherwise, loc is incremented by 1 to point to the next element in the list. You then compare searchItem with the next element in the list, and so on.

EXAMPLE

```
#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

const int ARRAY_SIZE = 10;                                   //Line 3

int seqSearch(const int list[], int listLength,
              int searchItem);                               //Line 4

int main()                                                   //Line 5
{                                                            //Line 6
    int intList[ARRAY_SIZE];                                 //Line 7
    int number;                                              //Line 8

    cout << "Line 9: Enter " << ARRAY_SIZE
         << " integers." << endl;                            //Line 9

    for (int index  = 0; index < ARRAY_SIZE; index++)        //Line 10
        cin >> intList[index];                               //Line 11

    cout << endl;                                            //Line 12

    cout << "Line 13: Enter the number to be "
         << "searched: ";                                    //Line 13
    cin >> number;                                           //Line 14
    cout << endl;                                            //Line 15

    int pos = seqSearch(intList, ARRAY_SIZE, number);        //Line 16

    if (pos!= -1)                                            //Line 17
        cout <<"Line 18: " << number

             << " is found at index " << pos
             << endl;                                        //Line 18
      else                                                   //Line 19
          cout << "Line 20: " << number
               << " is not in the list." << endl;            //Line 20

    return 0;                                                //Line 21
}                                                            //Line 22
//Place the definition of the function seqSearch
//given previously here.
```

Sample Run 1: In this sample run, the user input is shaded.

```
Line 9: Enter 10 integers.
18 45 37 29 80 32 67 78 10 30

Line 13: Enter the number to be searched: 29

Line 18: 29 is found at index 3
```

Sample Run 2:

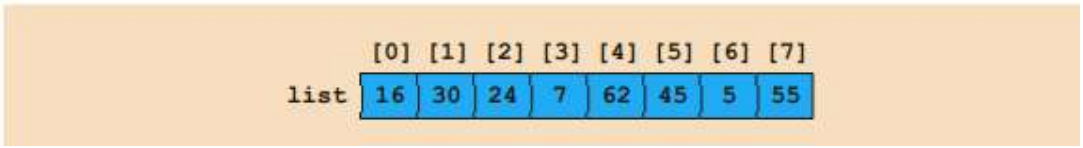
```
Line 9: Enter 10 integers.
18 45 37 29 80 32 67 78 10 30

Line 13: Enter the number to be searched: 12

Line 20: 12 is not in the list.
```

Sorting an Array

In this section, we discuss how to sort an array using the algorithm, called selection sort.

As the name implies, in the selection sort algorithm, we rearrange the list by selecting an element in the list and moving it to its proper position. This algorithm finds the location of the smallest element in the unsorted portion of the list and moves it to the top of the unsorted portion of the list. The first time, we locate the smallest item in the entire list. The second time, we locate the smallest item in the list starting from the second element in the list, and so on.

Suppose you have the list shown in Figure below

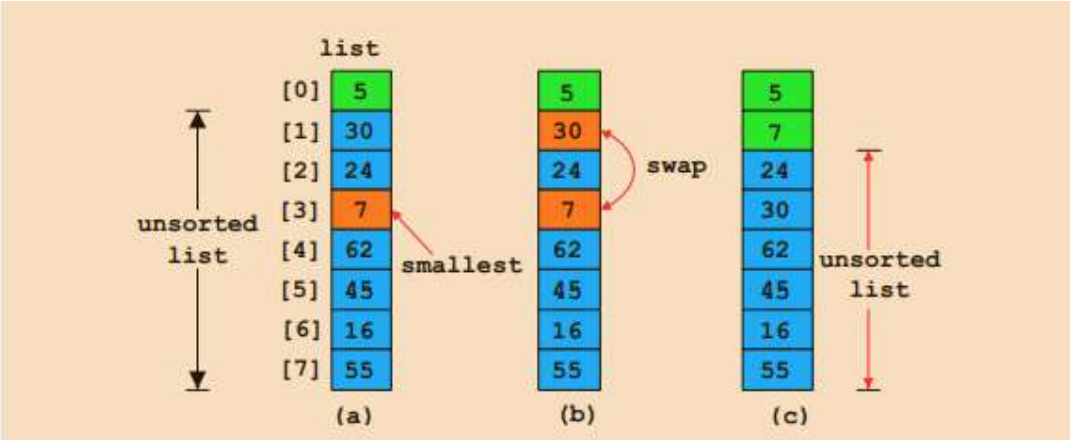


Initially, the entire list is unsorted. So, we find the smallest item in the list. The smallest item is at position 6, as shown in Figure (a). Because this is the smallest item, it must be moved to position 0. So, we swap 16 (that is, list[0]) with 5 (that is, list[6]), as shown in Figure (b). After swapping these elements, the resulting list is as shown in Figure(c).

2nd Iteration

Now the unsorted list is list[1]...list[7]. So, we find the smallest element in the unsorted list. The smallest element is at position 3, as shown in (a). Because the smallest

element in the unsorted list is at position 3, it must be moved to position 1. So, we swap 7 (that is, list[3]) with 30 (that is, list[1]), as shown in Figure (b). After swapping list[1] with list[3], the resulting list is as shown in Figure (c).



Now, the unsorted list is list[2]...list[7]. So, we repeat the preceding process of finding the (position of the) smallest element in the unsorted portion of the list and moving it to the beginning of the unsorted portion of the list. Selection sort thus involves the following steps. In the unsorted portion of the list:

- Find the location of the smallest element.
- Move the smallest element to the beginning of the unsorted list.

Initially, the entire list (that is, list[0]...list[length - 1]) is the unsorted list. After executing Steps a and b once, the unsorted list is list[1]...list [length - 1]. After executing Steps a and b a second time, the unsorted list is list[2]...list[length - 1], and so on. In this way, we can keep track of the unsorted portion of the list and repeat Steps a and b with the help of a for loop.

EXAMPLE

```
//Selection sort

#include <iostream>                                          //Line 1

using namespace std;                                         //Line 2

void selectionSort(int list[],  int length);                 //Line 3

int main()                                                   //Line 4
{                                                            //Line 5
    int list[] = {2, 56, 34, 25, 73, 46, 89,
                  10, 5, 16};                                //Line 6

    int i;                                                   //Line 7

    selectionSort(list, 10);                                 //Line 8

    cout << "After sorting, the list elements are:"
         << endl;                                            //Line 9

    for (i = 0; i < 10; i++)                                 //Line 10
        cout << list[i] << " ";                              //Line 11


    cout << endl;                                            //Line 12

    return 0;                                                //Line 13
}                                                            //Line 14

//Place the definition of the function selectionSort given
//previously here.
```

Sample Run:

```
After sorting, the list elements are:
2 5 10 16 25 34 46 56 73 89
```

The statement in Line 6 declares and initializes list to be an array of 10 components of type int. The statement in Line 8 uses the function selectionSort to sort list. Notice that both list and its length (the number of elements in it, which is 10) are passed as parameters to the function selectionSort. The for loop in Lines 10 and 11 outputs the elements of list. To illustrate the selection sort algorithm in this program, we declared and initialized the array list.

5.3 Multidimensional ARRAYS

In the previous section, you learned how to use one-dimensional arrays to manipulate data. If the data is provided in a list form, you can use one-dimensional arrays.

However, sometimes data is provided in a table form. For example, suppose that you want to track the number of cars in a particular color that are in stock at a local dealership. The dealership sells six types of cars in five different colors. Figure below shows sample data

inStock	[RED]	[BROWN]	[BLACK]	[WHITE]	[GRAY]
[GM]	10	7	12	10	4
[FORD]	18	11	15	17	10
[TOYOTA]	12	10	9	5	12
[BMW]	16	6	13	8	3
[NISSAN]	10	7	12	6	4
[VOLVO]	9	4	7	12	11

You can see that the data is in a table format. The table has 30 entries, and every entry is an integer. Because the table entries are all of the same type, you can declare a one-dimensional array of 30 components of type int. The first five components of the one-dimensional array can store the data of the first row of the table, the next five components of the one-dimensional array can store the data of the second row of the table, and so on. In other words, you can simulate the data given in a table format in a one-dimensional array.

Two-dimensional array: A collection of a fixed number of components arranged in rows and columns (that is, in two dimensions), wherein all components are of the same type.

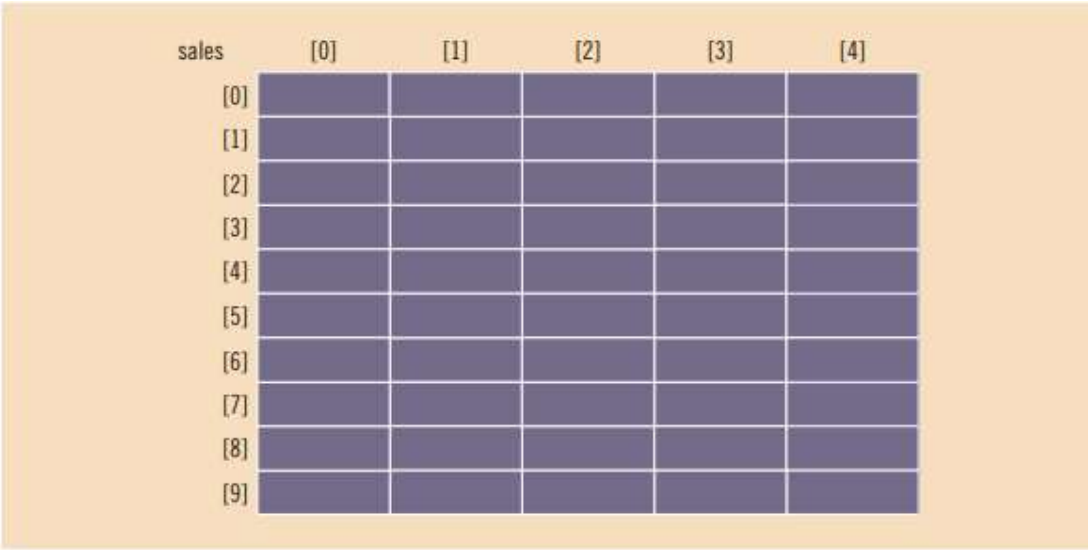
```
dataType arrayName[intExp1][intExp2];
```

wherein intExp1 and intExp2 are constant expressions yielding positive integer values. The two expressions intExp1 and intExp2 specify the number of rows and the number of columns, respectively, in the array.

The statement:

```
double sales[10][5];
```

declares a two-dimensional array **sales** of 10 rows and 5 columns, in which every component is of type double. As in the case of a one-dimensional array, the rows are numbered 0. . .9 and the columns are numbered 0. . .4



Accessing Array Components

To access the components of a two-dimensional array, you need a pair of indices: one for the row position (which occurs first) and one for the column position (which occurs second). The syntax to access a component of a two-dimensional array is:

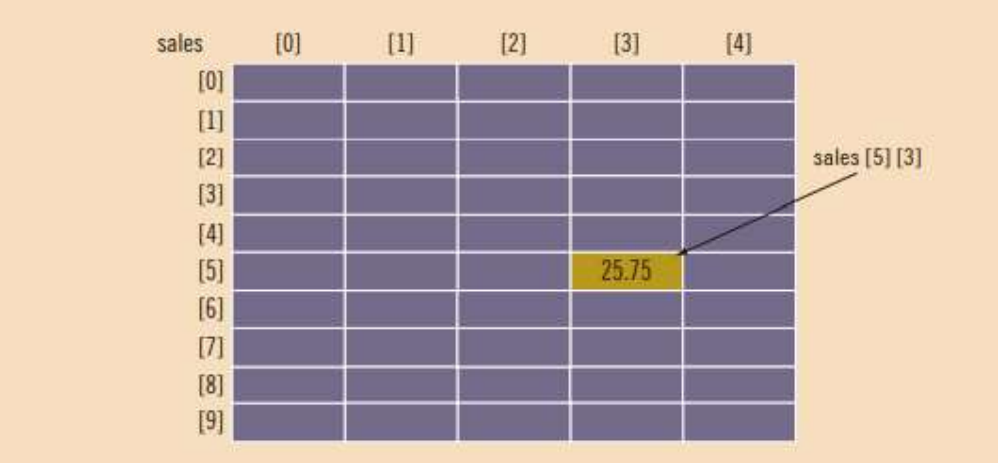
```
arrayName[indexExp1][indexExp2]
```

wherein indexExp1 and indexExp2 are expressions yielding nonnegative integer values. indexExp1 specifies the row position and indexExp2 specifies the column position.

The statement:

sales[5][3] = 25.75;

stores 25.75 into row number 5 and column number 3 (that is, the sixth row and the fourth column) of the array sales



Suppose that:

int i = 5;

int j = 3;

Then, the previous statement:

sales[5][3] = 25.75; is equivalent to:

sales[i][j] = 25.75;

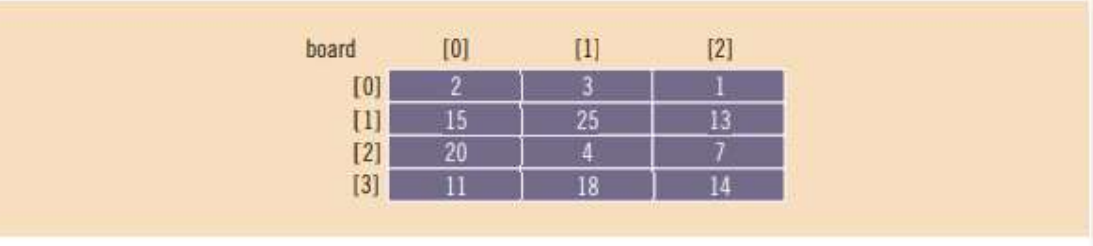
So the indices can also be variables.

Two-Dimensional Array Initialization

Like one-dimensional arrays, two-dimensional arrays can be initialized when they are declared. The following example helps illustrate this concept. Consider the following statement:

```
int board[4][3] = {{2, 3, 1},
                   {15, 25, 13},
                    {20, 4, 7},
                   {11, 18, 14}};
```

This statement declares board to be a two-dimensional array of four rows and three columns. The elements of the first row are 2, 3, and 1; the elements of the second row are 15, 25, and 13; the elements of the third row are 20, 4, and 7; and the elements of the fourth row are 11, 18, and 14, respectively. Figure below shows the array board.



board	[0]	[1]	[2]
[0]	2	3	1
[1]	15	25	13
[2]	20	4	7
[3]	11	18	14

To initialize a two-dimensional array when it is declared:

1. The elements of each row are all enclosed within one set of curly braces and separated by commas.
2. The set of all rows is enclosed within curly braces.
3. For number arrays, if all components of a row are not specified, the unspecified components are initialized to 0. In this case, at least one of the values must be given to initialize all the components of a row.

Two-Dimensional Arrays & Enumeration Types

You can also use the enumeration type for array indices. Consider the following statements:

```
const int NUMBER_OF_ROWS = 6;
const int NUMBER_OF_COLUMNS = 5;
enum carType {GM, FORD, TOYOTA, BMW, NISSAN, VOLVO};
enum colorType {RED, BROWN, BLACK, WHITE, GRAY};
int inStock[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
```

These statements define the carType and colorType enumeration types and define inStock as a two-dimensional array of six rows and five columns.

Suppose that each row in inStock corresponds to a car type, and each column in inStock corresponds to a color type. That is, the first row corresponds to the car type GM, the second row corresponds to the car type FORD, and so on. Similarly, the first column corresponds to the color type RED, the second column corresponds to the color type BROWN, and so on

Processing Two-Dimensional Arrays

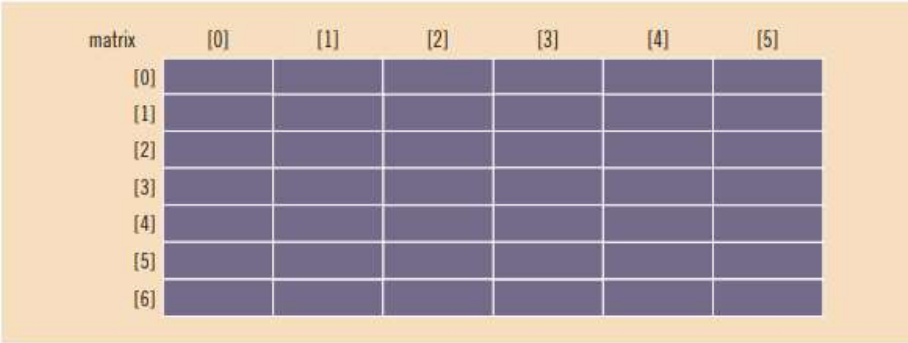
A two-dimensional array can be processed in four ways:

1. Process a single element.
2. Process the entire array.
3. Process a particular row of the array, called row processing.
4. Process a particular column of the array, called column processing.

Processing a single element is like processing a single variable. Initializing and printing the array are examples of processing the entire two-dimensional array. Finding the largest element in a row (column) or finding the sum of a row (column) are examples of row (column) processing.

```
const int NUMBER_OF_ROWS = 7;                //This can be set to any number.
const int NUMBER_OF_COLUMNS = 6;        //This can be set to any number.

int matrix[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS];
int row;
int col;
int sum;
int largest;
int temp;
```

figure below shows the array matrix



All of the components of a two-dimensional array, whether rows or columns, are identical in type. If a row is looked at by itself, it can be seen to be just a one-dimensional array. A column seen by itself is also a one-dimensional array. Therefore, when processing a particular row or column of a two-dimensional array, we use algorithms similar to those that process one-dimensional arrays.

Suppose that we want to process row number 5 of matrix (that is, the sixth row of matrix). The elements of row number 5 of matrix are:

matrix[5][0], matrix[5][1], matrix[5][2], matrix[5][3], matrix[5][4], and matrix[5][5]

We see that in these components, the first index (the row position) is fixed at 5. The second index (the column position) ranges from 0 to 5. Therefore, we can use the following for loop to process row number 5:

```
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        process matrix[5][col]
```

Clearly, this for loop is equivalent to the following for loop:

```
row = 5;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        process matrix[row][col]
```

Initialization

Suppose that you want to initialize row number 4, that is, the fifth row, to 0. As explained earlier, the following for loop does this:

```
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
      matrix[row][col] = 0;
```

If you want to initialize the entire matrix to 0, you can also put the first index (that is, the row position) in a loop. By using the following nested for loops, we can initialize each component of matrix to 0:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
   for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        matrix[row][col] = 0;
```

Print

By using a nested for loop, you can output the elements of matrix. The following nested for loops print the elements of matrix, one row per line:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
   for (col = 0; col < NUMBER_OF_COLUMNS; col++)
```

```
        cout << setw(5) << matrix[row][col] << " ";
         cout << endl;
}
```

Input

The following for loop inputs the data into row number 4, that is, the fifth row of matrix:

```
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
cin >> matrix[row][col];
```

As before, by putting the row number in a loop, you can input data into each component of matrix. The following for loop inputs data into each component of matrix:

```
 for (row = 0; row < NUMBER_OF_ROWS; row++)
      for (col = 0; col < NUMBER_OF_COLUMNS; col++)
          cin >> matrix[row][col];
```

Sum by Row

The following for loop finds the sum of row number 4 of matrix; that is, it adds the components of row number 4:

```
sum = 0;
row = 4; for (col = 0; col < NUMBER_OF_COLUMNS; col++)
     sum = sum + matrix[row][col];
```

Once again, by putting the row number in a loop, we can find the sum of each row separately. The following is the C11 code to find the sum of each individual row:

```
//Sum of each individual row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
   sum = 0;
   for (col = 0; col < NUMBER_OF_COLUMNS; col++)
         sum = sum + matrix[row][col];
```

```
cout << "Sum of row " << row + 1 << " = " << sum << endl;
 }
```

Sum by Column

As in the case of sum by row, the following nested for loop finds the sum of each individual column:

```
      //Sum of each individual column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
{
 sum = 0;
 for (row = 0; row < NUMBER_OF_ROWS; row++)
       sum = sum + matrix[row][col];
 cout << "Sum of column " << col + 1 << " = " << sum
 << endl;
}
```

Largest Element in Each Row and Each Column

As stated earlier, two other operations on a two-dimensional array are finding the largest element in each row and each column.

The following for loop determines the largest element in row number 4:

```
row = 4;
largest = matrix[row][0]; //Assume that the first element of the row is the largest.
for (col = 1; col < NUMBER_OF_COLUMNS; col++)
          if (matrix[row][col] > largest)
          largest = matrix[row][col];
```

The following C++ code determines the largest element in each row and each column:

```
      //Largest element in each row
for (row = 0; row < NUMBER_OF_ROWS; row++)
{
```

```
 largest = matrix[row][0]; //Assume that the first element //of the row is the largest. for
for (col = 1; col < NUMBER_OF_COLUMNS; col++)
    if (matrix[row][col] > largest) largest = matrix[row][col];
 cout << "The largest element in row " << row + 1 << " = " << largest << endl;
 }

    //Largest element in each column
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
 {
largest = matrix[0][col]; //Assume that the first element //of the column is the largest.
for (row = 1; row < NUMBER_OF_ROWS; row++)
    if (matrix[row][col] > largest) largest = matrix[row][col];
cout << "The largest element in column " << col + 1 << " = " << largest << endl;
}
```

Passing Two-Dimensional Arrays as Parameters to Functions

Two-dimensional arrays can be passed as parameters to a function, and they are passed by reference. The base address (that is, the address of the first component of the actual parameter) is passed to the formal parameter. If matrix is the name of a two-dimensional array, then matrix[0][0] is the first component of matrix.

When storing a two-dimensional array in the computer's memory, C++ uses the row order form. That is, the first row is stored first, followed by the second row, followed by the third row, and so on.

In the case of a one-dimensional array, when declaring it as a formal parameter, we usually omit the size of the array.

Because C++ stores two-dimensional arrays in row order form, to compute the address of a component correctly, the compiler must know where one row ends and the next row begins. Thus, when declaring a twodimensional array as a formal parameter, you can omit the size of the first dimension, but not the second; that is, you must specify the number of columns.

Suppose we have the following declaration:

```
const int NUMBER_OF_ROWS = 6;
const int NUMBER_OF_COLUMNS = 5;
```

Consider the following definition of the function printMatrix:

```
void printMatrix(int matrix[][NUMBER_OF_COLUMNS], int noOfRows)
{
        for (int row = 0; row < noOfRows; row++)
        {
                for (int col = 0; col < NUMBER_OF_COLUMNS; col++) cout << setw(5)
                << matrix[row][col] << " ";
                cout << endl;
        }
}
```

This function takes as a parameter a two-dimensional array of an unspecified number of rows and five columns, and outputs the content of the two-dimensional array. During the function call, the number of columns of the actual parameter must match the number of columns of the formal parameter.

Similarly, the following function outputs the sum of the elements of each row of a two-dimensional array whose elements are of type int:

```
void sumRows(int matrix[][NUMBER_OF_COLUMNS], int noOfRows)
{
        int sum;
          //Sum of each individual row
for (int row = 0; row < noOfRows; row++)
        {
        sum = 0;
        for (int col = 0; col < NUMBER_OF_COLUMNS; col++)
                sum = sum + matrix[row][col]; cout << "Sum of row " << (row + ) << "
                = " << sum << endl;
         }
```

}

The following function determines the largest element in each row:

```
void largestInRows(int matrix[][NUMBER_OF_COLUMNS], int noOfRows)
{
 int largest;
     //Largest element in each row
 for (int row = 0; row < noOfRows; row++)
       {
       largest = matrix[row][0]; //Assume that the first element //of the row is the largest
for (int col = 1; col < NUMBER_OF_COLUMNS; col++)
       if (largest < matrix[row][col])
        largest = matrix[row][col]; cout << "The largest element of row " << (row + 1) <<
" = " << largest << endl;
   }
}
```

Likewise, you can write a function to find the sum of the elements of each column, read the data into a two-dimensional array, find the largest and/or smallest element in each row or column, and so on

Example below shows how the functions **printMatrix**, **sumRows**, and **largestInRows** are used in a program.

EXAMPLE

```
// Two-dimensional arrays as parameters to functions.

#include <iostream>                                    //Line 1
#include <iomanip>                                     //Line 2

using namespace std;                                   //Line 3
```

```
const int NUMBER_OF_ROWS = 6;                                    //Line 4
const int NUMBER_OF_COLUMNS = 5;                                 //Line 5

void printMatrix(int matrix[][NUMBER_OF_COLUMNS],
                 int NUMBER_OF_ROWS);                            //Line 6
void sumRows(int matrix[][NUMBER_OF_COLUMNS],
             int NUMBER_OF_ROWS);                                //Line 7
void largestInRows(int matrix[][NUMBER_OF_COLUMNS],
                   int NUMBER_OF_ROWS);                          //Line 8

int main()                                                       //Line 9
{                                                                //Line 10
    int board[NUMBER_OF_ROWS][NUMBER_OF_COLUMNS]
                        = {{17, 8, 24, 10, 28},
                           {9, 20, 16, 55, 90},
                           {25, 45, 35, 8, 78},
                           {5, 0, 96, 45, 38},
                           {76, 30, 8, 14, 28},
                           {9, 60, 55, 62, 10}};                 //Line 11
    printMatrix(board, NUMBER_OF_ROWS);                          //Line 12
    cout << endl;                                                //Line 13
    sumRows(board, NUMBER_OF_ROWS);                              //Line 14
    cout << endl;                                                //Line 15
    largestInRows(board, NUMBER_OF_ROWS);                        //Line 16

    return 0;                                                    //Line 17
}                                                                //Line 18
```

Sample Run:

```
   17     8    24    10    28
    9    20    16    55    90
   25    45    35     8    78
    5     0    96    45    38
   76    30     8    14    28
    9    60    55    62    10

Sum of row 1 = 87
Sum of row 2 = 190
Sum of row 3 = 191
Sum of row 4 = 184
Sum of row 5 = 156
Sum of row 6 = 196

The largest element of row 1 = 28
The largest element of row 2 = 90
The largest element of row 3 = 78
The largest element of row 4 = 96
The largest element of row 5 = 76
The largest element of row 6 = 62
```

In this program, the statement in Line 11 declares and initializes board to be a two dimensional array of six rows and five columns. The statement in Line 12 uses the function printMatrix to output the elements of board (see the first six lines of the Sample Run). The statement in Line 14 uses the function sumRows to calculate and print the sum of each row. The statement in Line 16 uses the function largestInRows to find and print the largest element in each row.

5.3 POINTERS

The values belonging to pointer data types are the memory addresses of your computer. As in many other languages, there is no name associated with the pointer data type in C11. Because the domain—that is, the set of values of a pointer data type—is the addresses (locations) in memory, a pointer variable is a variable whose content is an address, that is, a memory location and the pointer variable is said to point to that memory location.

Pointer variable: A variable whose content is an address (that is, a memory address) and is therefore said to point to a memory address.

Declaring Pointer Variables

The value of a pointer variable is an address or memory space that typically contains some data. Therefore, when you declare a pointer variable, you also specify the data

type of the value to be stored in the memory location pointed to by the pointer variable. For example, if a pointer variable contains the address of a memory location containing an int value, it is said to be an int pointer or a pointer (variable) of type int. As with regular variables, pointers are bound to a data type and they can only contain the addresses of (or point to) variables of the specific data type they were created to hold.

In C++, you declare a pointer variable by using the asterisk symbol (*) between the data type and the variable name. The general syntax to declare a pointer variable is:

```
dataType *identifier;
```

As an example, consider the following statements:

```
int *p;
char *ch;
```

In these statements, both p and ch are pointer variables. The content of p (when properly assigned) points to a memory location of type int, and the content of ch points to a memory location of type char. So, p is a pointer variable of type int, and ch is a pointer variable of type char.

Before discussing how pointers work, let us make the following observations. The statement:

```
int *p;
```

is equivalent to the statement:

```
int* p;
```

which is equivalent to the statement:

```
int * p;
```

Thus, the character * can appear anywhere between the data type name and the variable name

Address of Operator (&)

In C++, the ampersand, &, called the **address of operator**, is a unary operator that returns the address of its operand. For example, given the statements:

```
int x;
```

```
int *p;
```

the statement:

```
p = &x;
```

assigns the address of x to p. That is, x and the value of p refer to the same memory location.

Dereferencing Operator (*)

Every chapter until now has used the asterisk character, *, as the binary multiplication operator. C++ also uses * as a unary operator. When used as a unary operator, *, commonly referred to as the dereferencing operator or indirection operator, refers to the object to which its operand (that is, the pointer) points.

```
int x = 25;
int *p;
p = &x; //store the address of x in p
```

the statement

cout << *p << endl;

prints the value stored in the memory space pointed to by p, which is the value of x. Also, the statement

*p = 55;

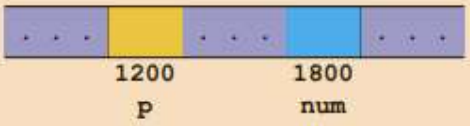
stores 55 in the memory location pointed to by p—that is, in x.

EXAMPLE

Let us consider the following statements:

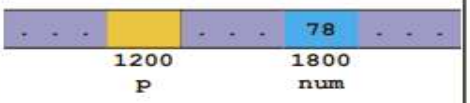
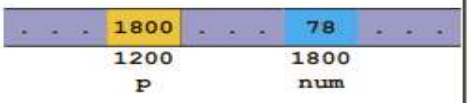
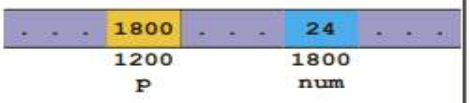
```
int *p;
int num;
```

In these statements, p is a pointer variable of type int, and num is a variable of type int. Let us assume that memory location 1200 is allocated for p, and memory location 1800 is allocated for num.



Consider the following statements:

1. num = 78;
2. p = #
3. *p = 24;

After Statement	Values of the Variables	Explanation
1.		The statement `num = 78;` stores `78` into `num`.
2.		The statement `p = &num;` stores the address of `num`, which is `1800`, into `p`.
3.		The statement `*p = 24;` stores `24` into the memory location to which `p` points. Because the value of `p` is `1800`, statement 3 stores `24` into memory location `1800`. Note that the value of `num` is also changed.

Let us summarize the preceding discussion.

1. A declaration such as int *p; allocates memory for p only, not for *p. Later, you will learn how to allocate memory for *p.

2. The content of p points only to a memory location of type int.

3. &p, p, and *p all have different meanings.

4. &p means the address of p—that is, 1200 (in Figure 12-1).

5. p means the content of p, which is 1800, after the statement p = # executes.

6. *p means the content of the memory location to which p points. Note that after the statement p = # executes, the value of *p is 78; after the statement *p = 24; executes, the value of *p is 24..

EXAMPLE

```
#include <iostream>                                          //Line 1
#include <iomanip>                                           //Line 2

using namespace std;                                         //Line 3

const double PI = 3.1416;                                    //Line 4

int main()                                                   //Line 5
{                                                            //Line 6
    double radius;                                           //Line 7
    double *radiusPtr;                                       //Line 8

    cout << fixed << showpoint << setprecision(2);           //Line 9

    radius = 2.5;                                            //Line 10
    radiusPtr = &radius;                                     //Line 11

    cout << "Line 12: Radius = " << radius
         << ", area = " << PI * radius * radius
         << endl;                                            //Line 12

    cout << "Line 13: Radius = " << *radiusPtr
         << ", area = "
         << PI * (*radiusPtr) * (*radiusPtr)
         << endl;                                            //Line 13

    cout << "Line 14: Enter the radius: ";                   //Line 14
    cin >> *radiusPtr;                                       //Line 15
    cout << endl;                                            //Line 16
```

```
    cout << "Line 17: Radius = " << radius << ", area = "
         << PI * radius * radius << endl;                    //Line 17
    cout << "Line 18: Radius = " << *radiusPtr
         << ", area = "
         << PI * (*radiusPtr) * (*radiusPtr) << endl
         << endl;                                            //Line 18

    cout << "Line 19: Address of radiusPtr: "
         << &radiusPtr << endl;                              //Line 19
    cout << "Line 20: Value stored in radiusPtr: "
         << radiusPtr << endl;                               //Line 20
    cout << "Line 21: Address of radius: "
         << &radius << endl;                                 //Line 21
    cout << "Line 22: Value stored in radius: "
         << radius << endl;                                  //Line 22

    return 0;                                                //Line 23
}                                                            //Line 24
```

Sample Run: In this sample run, the user input is shaded.

```
Line 12: Radius = 2.50, area = 19.64
Line 13: Radius = 2.50, area = 19.64
Line 14: Enter the radius: 4.90

Line 17: Radius = 4.90, area = 75.43
Line 18: Radius = 4.90, area = 75.43

Line 19: Address of radiusPtr: 013EFDA4
Line 20: Value stored in radiusPtr: 013EFDB0
Line 21: Address of radius: 013EFDB0
Line 22: Value stored in radius: 4.90
```

Line 22: Value stored in radius: 4.90 The preceding program works as follows. The statement in Line 7 declares radius to be a variable of type double and the statement in Line 8 declares radiusPtr to be a pointer variable of type double. The statement in Line 10 stores 2.5 in radius and the statement in Line 11 stores the address of radius in radiusPtr. The statement in Line 12 outputs the radius and area of the circle using the value stored in the memory location radius.

The statement in Line 13 outputs the radius and area of the circle using the value stored in the memory location to which radiusPtr is pointing. Note that the output of the statements in Lines 12 and 13 is the same because radiusPtr points to radius. Next, the statement in Line 14 prompts the user to input the radius and the statement in Line 15 stores the radius in the memory location to which radiusPtr is pointing. Next, similar to the statements in Lines 12 and 13, the statements in Lines 17 and 18 output the radius and area using the variables radius and radiusPtr.

The statements in Lines 19 to 22 output the address of radiusPtr, the value stored in radiusPtr, the address of radius, and the value stored in radius.

From the output of the statements in Lines 20 and 21, it follows that radiusPtr stores the address of the variable radius. (Note that the address of radiusPtr, the value of radiusPtr, and the address of radius as shown by the output of Lines 19, 20, and 21, respectively, are machine dependent. When you run this program on your machine, you are likely to get different values. Furthermore, the pointer values, that is, the addresses, are printed in hexadecimal by default.

Initializing Pointer Variables

Because C++ does not automatically initialize variables, pointer variables must be initialized if you do not want them to point to anything. Pointer variables are initialized using the constant value 0, called the **null pointer**. Thus, the statement p = 0; stores the null pointer in p, that is, p points to nothing. Some programmers use the named constant NULL to initialize pointer variables. The named constant NULL is defined in the header file **cstddef**. The following two statements are equivalent:

```
p = NULL;
p = 0;
```

The number 0 is the only number that can be directly assigned to a pointer variable.

Initializing Pointer Variables

C++11 Standard provides the null pointer nullptr to initialize pointer variables. A pointer with the value nullptr points to nothing, and is called the **null pointer**. nullptr has a special value type that can be converted to any pointer type. The following statement declares p to be a pointer of type int and it also initializes it to the null pointer: int *p = nullptr;

Dynamic Variables

In the previous sections, you learned how to declare pointer variables, how to store the address of a variable into a pointer variable of the same type as the variable, and how to manipulate data using pointers. However, you learned how to use pointers to manipulate data only into memory spaces that were created using other variables. In other words, the pointers manipulated data into already existing memory spaces. But you could have accessed these memory spaces through the variables that were used to create them. So what is the benefit of using pointers? In this section, you will learn about the power behind pointers. In particular, you will learn how to allocate and deallocate memory during program execution using pointers.

Variables that are created during program execution are called **dynamic variables**. With the help of pointers, C++ creates dynamic variables. C++ provides two operators, new and delete, to create and destroy dynamic variables, respectively. When a program requires a new variable, the operator new is used. When a program no longer needs a dynamic variable, the operator delete is used. In C++, new and delete are reserved words.

Operator new

The operator new has two forms: one to allocate a single variable and another to allocate an array of variables. The syntax to use the operator new is:

```
new dataType;          //to allocate a single variable
new dataType[intExp];  //to allocate an array of variables
```

in which **intExp** is any expression evaluating to a positive integer.

The operator new allocates memory (as a variable) of the designated type and returns a pointer to it—that is, the address of this allocated memory. Moreover, the allocated memory is uninitialized.

Consider the following declaration:

int *p;

```
char *q;
int x;
```

The statement

```
p = &x;
```

stores the address of x in p. However, no new memory is allocated. On the other hand, consider the following statement:

```
p = new int;
```

This statement creates a variable during program execution somewhere in memory and stores the address of the allocated memory in p. The allocated memory is accessed via pointer dereferencing—namely, *p.

Because a dynamic variable is unnamed, it cannot be accessed directly. It is accessed indirectly by the pointer returned by new. The following statements illustrate this concept:

```
int *p;             //p is a pointer of type int
char *name;         //name is a pointer of type char
string *str;        //str is a pointer of type string

p = new int;        //allocates memory of type int and stores
                    //the address of the allocated memory in p
*p = 28;            //stores 28 in the allocated memory

name = new char[5];     //allocates memory for an array of five
                        //components of type char and stores the
                        //base address of the array in name
strcpy(name, "John");   //stores John in name

str = new string;  //allocates memory of type string
                   //and stores the address of the
                   //allocated memory in str

*str = "Sunny Day";     //stores the string "Sunny Day" in
                        //the memory pointed to by str
```

Operator delete

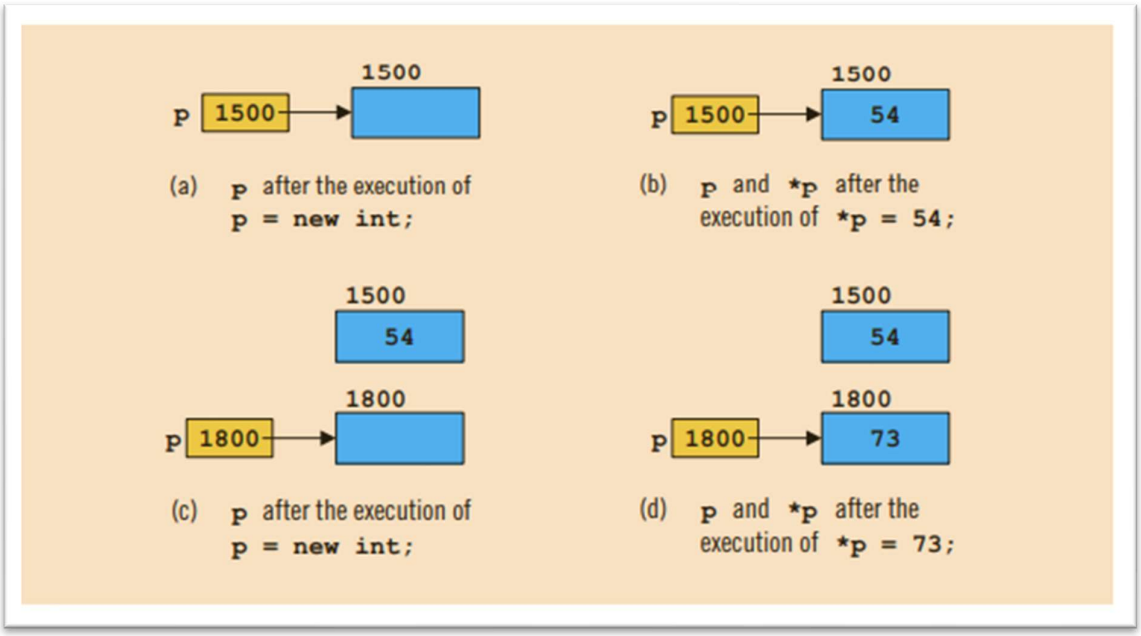
Suppose you have the following declaration:

```
int *p;
```

This statement declares p to be a pointer variable of type int. Next, consider the following statements:

```
 p = new int;       //Line 1
*p = 54;            //Line 2
 p = new int;       //Line 3
*p = 73;            //Line 4
```

Figure below shows the effect of these statements.



(The number 1500 on top of the box indicates the address of the memory space.) The statement in Line 1 allocates memory space of type int and stores the address of the allocated memory space into p. Suppose that the address of allocated memory space is 1500. Then, the value of p after the execution of this statement is 1500 (see Figure (a)). The statement in Line 2 stores 54 into the memory space that p points to, which is 1500 (see Figure (b)).

Now the obvious question is what happened to the memory space 1500 that p was pointing to after execution of the statement in Line 1. After execution of the statement in Line 3, p points to the new memory space at location 1800. The previous memory space at location 1500 is now inaccessible. In addition, the memory space 1500 remains as marked allocated. In other words, it cannot be freed or reallocated. This is

called **memory leak**. That is, there is an unused memory space that cannot be allocated.

Imagine what would happen if you executed statements, such as Line 3, a few thousand or a few million times. There would be a good amount of memory leak. The program might then run out of memory spaces for data manipulation, which would result in an abnormal termination of the program. The question at hand is how to avoid memory leak. When a dynamic variable is no longer needed, it can be destroyed; that is, its memory can be deallocated. The C++ operator delete is used to destroy dynamic variables. The syntax to use the operator delete has two forms:

```
delete pointerVariable;      //to deallocate a single
                             //dynamic variable
delete [] pointerVariable;   //to deallocate a dynamically
                             //created array
```

Thus, given the declarations of the previous section, the statements

delete p;

delete [] name;

delete str;

deallocate the memory spaces that the pointers p, name, and str point to.

Suppose p and name are pointer variables, as declared previously. Notice that an expression such as

delete p;

or

delete [] name;

only marks the memory spaces that these pointer variables point to as deallocated. Depending on the particular operating system, after these statements execute, these pointer variables may still contain the addresses of the deallocated memory spaces. In this case, we say that these pointers are **dangling**.

Therefore, if later you access the memory spaces via these pointers without properly initializing them, depending on the particular system, either the program will access a wrong memory space, which may result in corrupting data now stored in those spaces, or the program may terminate with an error message. One way to avoid this pitfall is

to set these pointers to nullptr after the delete operation. Also note that for the operator delete to work properly, the pointer must point to a valid memory space

Operations on Pointer Variables

The operations that are allowed on pointer variables are the assignment and relational operations and some limited arithmetic operations. The value of one pointer variable can be assigned to another pointer variable of the same type. For example, suppose that we have the following statements:

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

Integer values can be added and subtracted from a pointer variable. The value of one pointer variable can be subtracted from another pointer variable. The arithmetic operations that are allowed differ from the arithmetic operations on numbers. First, let us use the following statements to explain the increment and decrement operations on pointer variables:

```
int *p;
double *q;
char *chPtr;
studentType *stdPtr;                //studentType is as defined before
```

Recall that the size of the memory allocated for an int variable is 4 bytes, a double variable is 8 bytes, and a char variable is 1 byte. The memory allocated for a variable of type studentType is then 40 bytes.

The statement

p++; or p = p + 1;

ncrements the value of p by 4 bytes because p is a pointer of type int.

Similarly, the statements

```
q++;
chPtr++;
```

increment the value of q by 8 bytes and the value of chPtr by 1 byte, respectively. The statement

```
stdPtr++;
```

increments the value of stdPtr by 40 bytes.

The increment operator increments the value of a pointer variable by the size of the data type or structure to which it is pointing. Similarly, the decrement operator decrements the value of a pointer variable by the size of the data type or structure to which it is pointing.

Moreover, the statement p = p + 2; increments the value of p (an int pointer) by 8 bytes. Thus, when an integer is added to a pointer variable, the value of the pointer variable is incremented by the integer times the size of the data type or structure to which the pointer is pointing. Similarly, when an integer is subtracted from a pointer variable, the value of the pointer variable is decremented by the integer times the size of the data type or structure to which the pointer is pointing.

Pointer arithmetic can be very dangerous. Using pointer arithmetic, the program can accidentally access the memory locations of other variables and change their content without warning, leaving the programmer trying to find out what went wrong. If a pointer variable tries to access either the memory spaces of other variables or an illegal memory space, some systems might terminate the program with an appropriate error message. Always exercise extra care when doing pointer arithmetic.

Functions and Pointers

A pointer variable can be passed as a parameter to a function either by value or by reference. To declare a pointer as a value parameter in a function heading, you use the same mechanism as you use to declare a variable. To make a formal parameter be a reference parameter, you use & when you declare the formal parameter in the function heading. Therefore, to declare a formal parameter as a reference pointer parameter, between the data type name and the identifier name, you must include * to make the identifier a pointer and & to make it a reference parameter. The obvious

question is: In what order should & and * appear between the data type name and the identifier to declare a pointer as a reference parameter?

In C++, to make a pointer a reference parameter in a function heading, * appears before the & between the data type name and the identifier. The following example illustrates this concept:

```
void pointerParameters(int* &p, double *q)
{
        …….
        …….
}
```

In the function pointerParameters, both p and q are pointers. The parameter p is a reference parameter; the parameter q is a value parameter. Furthermore, the function pointerParameters can change the value of *q, but not the value of q. However, the function pointerParameters can change the value of both p and *p.

QUICK REVIEW

1. An array is a structured data type with a fixed number of components. Every component is of the same type, and components are accessed using their relative positions in the array.
2. In C++, an array index starts with 0.
3. Arrays can be initialized during their declaration. If there are fewer initial values than the array size, the remaining elements are initialized to 0

4. The base address of an array is the address of the first array component. For example, if list is a one-dimensional array, the base address of list is the address of list[0].
5. Although as parameters, arrays are passed by reference, when declaring an array as a formal parameter, using the reserved word const before the data type prevents the function from modifying the array.
6. The sequential search algorithm searches a list for a given item, starting with the first element in the list. It continues to compare the search item with the other elements in the list until either the item is found or the list has no more elements left to be compared with the search item.
7. Selection sort sorts the list by finding the smallest (or equivalently largest) element in the list and moving it to the beginning (or end) of the list.
8. For a list of length n, selection sort makes exactly $\frac{n(n-1)}{2}$ key comparisons and 3(n - 1) item assignments.
9. In a two-dimensional array, the elements are arranged in a table form.

PROGRAMMING EXERCISES

1. Write a C++ function, smallestIndex, that takes as parameters an int array and its size and returns the index of the first occurrence of the smallest element in the array. Also, write a program to test your function.

2. Write a C++ function, lastLargestIndex, that takes as parameters an int array and its size and returns the index of the last occurrence of the largest element in the array. Also, write a program to test your function.

3. Write a program that allows the user to enter the last names of five candidates in a local election and the number of votes received by each candidate. The program should then output each candidate's name, the number of votes received, and the percentage of the total votes received by the candidate. Your program should also output the winner of the election. A sample output is:

Candidate	Votes Received	% of Total Votes
Johson	5000	25.91
Miller	4000	20.73
Duffy	6000	31.09
Robinson	2500	12.95
Ashtony	1800	9.33
Total	19300	

The Winner of the Election is Duffy.

4. Write a program that uses a two-dimensional array to store the highest and lowest temperatures for each month of the year. The program should output the average high, average low, and the highest and lowest temperatures for the year. Your program must consist of the following functions:

   a. Function getData: This function reads and stores data in the two dimensional array.

b. Function averageHigh: This function calculates and returns the average high temperature for the year

c. Function averageLow: This function calculates and returns the average low temperature for the year

d. Function indexHighTemp: This function returns the index of the highest high temperature in the array.

e. Function indexLowTemp: This function returns the index of the lowest low temperature in the array

TOPIC 6
CLASSES (PART ONE)

LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn about classes
- Learn about private, protected, and public members of a class
- Explore how classes are implemented.
- Examine constructors and destructors
- Learn about the abstract data type (ADT)
- Learn about inheritance
- Learn about derived and base classes
- Explore three types of inheritance: public, protected, and private
- Examine how the constructors of base and derived classes work
- Learn how to construct the header file of a derived class

6.1 INTRODUCTION

Chapter 1 introduced the problem-solving methodology called object-oriented design (OOD). In OOD, the first step is to identify the components, called objects. An object combines data and the operations on that data in a single unit. In C11, the mechanism that allows you to combine data and the operations on that data in a single unit is called a class. Now that you know how to store and manipulate data in computer memory and how to construct your own functions, you are ready to learn how objects are constructed. This and subsequent chapters develop and implement programs using OOD. This chapter first explains how to define a class and use it in a program.

A **class** is a collection of a fixed number of components. The components of a class are called the **members** of the class.

The general syntax for defining a class is:

```
class classIdentifier
{
    classMembersList
};
```

in which classMembersList consists of variable declarations and/or functions. That is, a member of a class can be either a variable (to store data) or a function (to manipulate data).

For example, the following statements define the class courseType, with variables and functions, to implement the basic properties of a course.

```
class courseType
{
        public:
        void setCourseInfo(string cName, string cNo, int credits);
        void print() const;
        int getCredits();
        string getCourseNumber();
        string getCourseName();
        private:
         string courseName;
         string courseNo;
         int courseCredits;
};
```

Let us note the following:

- If a member of a class is a variable, you declare it just like any other variable. Also, in C++ versions prior to C++11, in the definition of the class, you cannot initialize a variable when you declare it.
- If a member of a class is a function, you typically use the function prototype to declare that member.
- If a member of a class is a function, it can (directly) access any member of the class—member variables and member functions. That is, when you write the definition of a member function, you can directly access any member variable of the class without passing it as a parameter. The only condition is that you must declare an identifier before you can use it.

In C++, class is a reserved word, and it defines only a data type; no memory is allocated. It announces the declaration of a class. Moreover, note the semicolon (;) after the right brace. The semicolon is part of the syntax. A missing semicolon, therefore, will result in a syntax error.

The members of a class are classified into three categories: private, public, and protected. This chapter mainly discusses the first two types, private and public.

In C++, private, protected, and public are reserved words and are called member access specifiers.

Following are some facts about public and private members of a class:

- By default, all members of a class are private.
- If a member of a class is private, you cannot access it directly from outside of the class.
- A public member is accessible outside of the class. (Example 10-1 illustrates this concept.)
- To make a member of a class public, you use the member access specifier public with a colon.

Suppose that we want to define a class to implement the time of day in a program. Because a clock gives the time of day, let us call this class clockType. Furthermore, to represent time in computer memory, we use three int variables: one to represent the hours, one to represent the minutes, and one to represent the seconds.

EXAMPLE Suppose these three variables are:

```
int hr;
int min;
int sec;
```

We also want to perform the following operations on the time:

1. Set the time.
2. Retrieve the time.
3. Print the time.
4. Increment the time by one second.
5. Increment the time by one minute.
6. Increment the time by one hour.
7. Compare the two times for equality.

To implement these seven operations, we will write seven functions—**setTime**, **getTime**, **printTime**, i**ncrementSeconds, incrementMinutes**, **incrementHours**, and **equalTime.**

Some members of the class clockType will be private; others will be public. Deciding which member to make public and which to make private depends on the nature of the member. The general rule is that any member that needs to be directly accessed outside of the class is declared public; any member that should not be accessed directly by the user should be declared private. For example, the user should be able to set the time and print the time. Therefore, the members that set the time and print the time should be declared public.

Similarly, the members to increment the time and compare the time for equality should be declared public. On the other hand, to prevent the direct manipulation of the member variables hr, min, and sec, we will declare them private. Furthermore, note that if the user has direct access to the member variables, member functions such as setTime are not needed.

The following statements define the class clockType:

```
class clockType {
public:
        void setTime(int, int, int);
        void getTime(int&, int&, int&) const;
        void printTime() const;
        void incrementSeconds();
        void incrementMinutes();
        void incrementHours();
        bool equalTime(const clockType&) const;


private:
        int hr;
        int min;
        int sec;
};
```

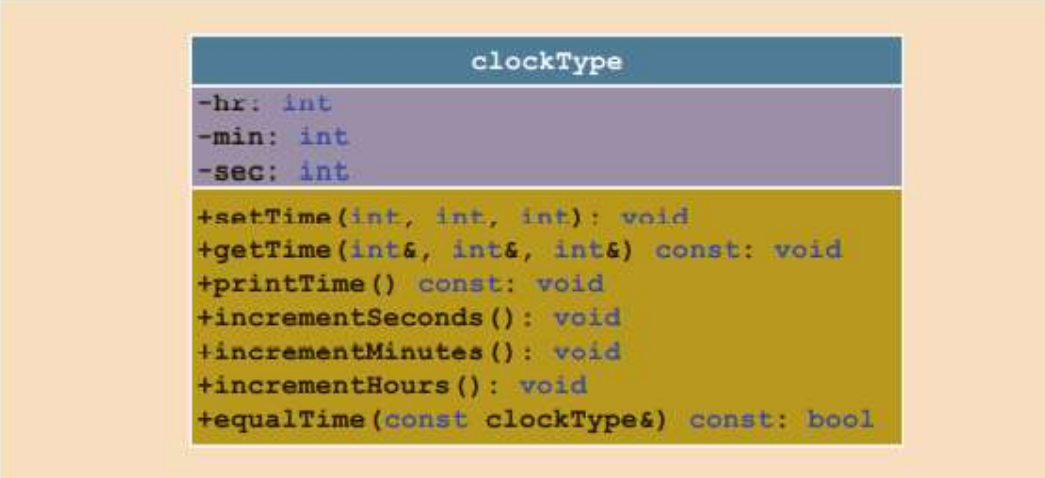
In this definition:

- The class clockType has seven member functions: setTime, getTime, printTime, incrementSeconds, incrementMinutes, incrementHours, and equalTime. It has three member variables: hr, min, and sec.
- The three member variables—hr, min, and sec—are private to the class and cannot be accessed outside of the class.? The seven member functions—setTime, getTime, printTime, incrementSeconds, incrementMinutes, incrementHours, and equalTime—are public and can be directly accessed outside the class. They can also directly access the member variables (hr, min, and sec). In other words, when we write the definitions of these functions, we do not pass these member variables as parameters to the member functions.
- In the function equalTime, the formal parameter is a constant reference parameter. That is, in a call to the function equalTime, the formal parameter receives the address of the actual parameter, but the formal parameter cannot modify the value of the actual parameter. You could have declared the formal parameter as a value parameter, but that would require the formal parameter to copy the value of the actual parameter, which could result in poor performance.
- The word const at the end of the member functions getTime, printTime, and equalTime specifies that these functions cannot modify the member variables of a variable of type clockType.

In the definition of the `class clockType`, all member variables are `private` and all member functions are `public`. However, a member function can also be `private`. For example, if a member function is used only to implement other member functions of the class and the user does not need to access this function, you make it `private`. Similarly, a member variable of a class can also be `public`.

Unified Modelling Language Class Diagrams

A class and its members can be described graphically using a notation known as the Unified Modeling Language (UML) notation.



The top box contains the name of the class. The middle box contains the member variables and their data types. The last box contains the member function name, parameter list, and the return type of the function. A + (plus) sign in front of a member name indicates that the member is a public member; a - (minus) sign indicates that the member is a private member. The symbol # before the member name indicates that the member is a protected member.

Variable (Object) Declaration

Once a class is defined, you can declare variables of that type. In C++ terminology, a class variable is called a **class object** or **class instance**. To help you become familiar with this terminology, from now on we will use the term class object, or simply **object**, for a class variable.

The syntax for declaring a class object is the same as that for declaring any other variable. The following statements declare two objects of type clockType: clockType myClock; clockType yourClock;

Each object has 10 members: seven member functions and three member variables. Each object has separate memory allocated for hr, min, and sec.

In actuality, memory is allocated only for the member variables of each class object. The C++ compiler generates only one physical copy of a member function of a class, and each class object executes the same copy of the member function. Therefore,

whenever we draw the figure of a class object, we will show only the member variables. As an example, Figure below shows the objects myClock and yourClock with values in their member variables.

Accessing Class Members

Once an object of a class is declared, it can access the members of the class. The general syntax for an object to access a member of a class is:

```
classObjectName.memberName
```

The class members that a class object can access depend on where the object is declared.

- If the object is declared in the definition of a member function of the class, then the object can access both the public and private members. (We will elaborate on this when we write the definition of the member function **equalTime** of the class clockType in the section "Implementation of Member Functions," later in this chapter.)
- If the object is declared elsewhere (for example, in a user's program), then the object can access only the public members of the class. Recall that in C++, the dot, . (period), is an operator called the **member access operator.**

EXAMPLE

Suppose we have the following declaration (say, in a user's program):

```
clockType myClock;
clockType yourClock;
```

Consider the following statements:

```
myClock.setTime(5, 2, 30);
myClock.printTime();
yourClock.setTime(x, y, z);    //assume x, y, and z are
                               //variables of type int
if (myClock.equalTime(yourClock))
.
.
.
```

In the first statement, myClock.setTime(5, 2, 30);, the member function setTime is executed. The values 5, 2, and 30 are passed as parameters to the function setTime, and the function uses these values to set the values of the three member variables hr, min, and sec of myClock to 5, 2, and 30, respectively. Similarly, the second statement executes the member function printTime and outputs the contents of the three member variables of myClock. In the third statement, the values of the variables x, y, and z are used to set the values of the three member variables of yourClock. In the fourth statement, the member function equalTime executes and compares the three member variables of myClock to the corresponding member variables of yourClock. Because in this statement equalTime is a member of the object myClock, it has direct access to the three member variables of myClock. So it needs one more object, which in this case is yourClock, to compare. In essence, equalTime needs two objects to compare. The object to which it is dotted, myClock, is one and the argument, yourClock, is the other. This explains why the function equalTime has only one parameter. The objects myClock and yourClock can access only public members of the class clockType. Thus, the following statements are illegal because hr and min are declared as private members of the class clockType and, therefore, cannot be accessed by the objects myClock and yourClock:

```
myClock.hr = 10;                    //illegal
myClock.min = yourClock.min;        //illegal
```

Reference Parameters and Class Objects (Variables)

Recall that when a variable is passed by value, the formal parameter copies the value of the actual parameter. That is, memory space to copy the value of the actual

parameter is allocated for the formal parameter. As a parameter, a class object can be passed by value.

Suppose that a class has several member variables requiring a large amount of memory to store data, and you need to pass a variable by value. The corresponding formal parameter then receives a copy of the data of the variable. That is, the compiler must allocate memory for the formal parameter, so as to copy the value of the member variables of the actual parameter. This operation might require, in addition to a large amount of storage space, a considerable amount of computer time to copy the value of the actual parameter into the formal parameter.

On the other hand, if a variable is passed by reference, the formal parameter receives only the address of the actual parameter. Therefore, an efficient way to pass a variable as a parameter is by reference. If a variable is passed by reference, then when the formal parameter changes, the actual parameter also changes. Sometimes, however, you do not want the function to be able to change the values of the member variables. In C++, you can pass a variable by reference and still prevent the function from changing its value by using the keyword const in the formal parameter declaration. As an example, consider the following function definition:

```
void testTime(const clockType& otherClock)
{
    clockType                                                    dClock;
    .                          .                                 .
}
```

The function testTime contains a reference parameter, otherClock. The parameter otherClock is declared using the keyword const. Thus, in a call to the function testTime, the formal parameter otherClock receives the address of the actual parameter, but otherClock cannot modify the contents of the actual parameter.

For example, after the following statement executes, the value of myClock will not be altered:

testTime(myClock);

Recall that if a formal parameter is a value parameter, within the function definition, you can change the value of the formal parameter. That is, you can use an assignment statement to change the value of the formal parameter (which, of course, would have no effect on the actual parameter). However, if a formal parameter is a constant reference parameter, you cannot use an assignment statement to change its value within the function, nor can you use any other function to change its value. Therefore, within the definition of the function testTime, you cannot alter the value of otherClock. . For example, the following would be illegal in the definition of the function testTime:

```
otherClock.setTime(5, 34, 56);          //illegal
otherClock = dClock;                    //illegal
```

Implementation of Member Functions

One way to implement these functions is to provide the function definition rather than the function prototype in the class itself. Unfortunately, the class definition would then be very long and difficult to comprehend. Another reason for providing function prototypes instead of function definitions relates to information hiding; that is, we want to hide the details of the operations on the data.

That is, we will write the definitions of the functions **setTime**, **getTime**, **printTime**, **incrementSeconds**, **equalTime**, and so on. Because the identifiers **setTime**, **printTime**, and so forth are local to the class, we cannot reference them (directly) outside of the class. In order to reference these identifiers, we use the **scope resolution operator, ::** (double colon). In the function definition's heading, the name of the function is the name of the class, followed by the scope resolution operator,

```
void clockType::setTime(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;
```

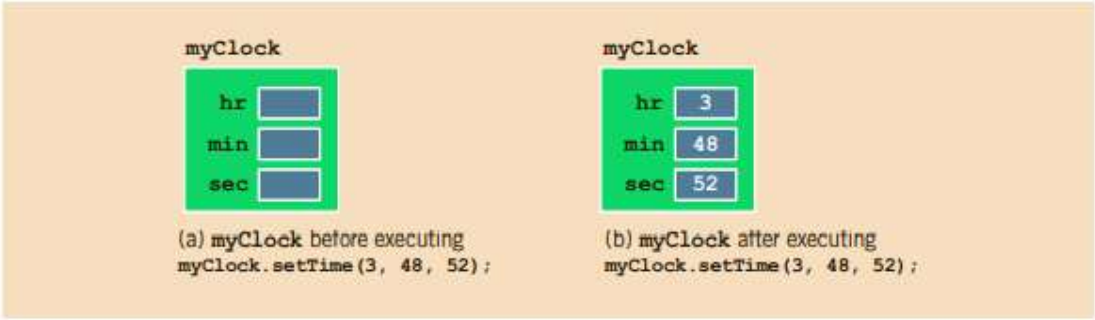
```
    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;
  if (0 <= seconds && seconds < 60)
      sec = seconds;
    else
     sec = 0;
}
```

Note that the definition of the function setTime checks for the valid values of hours, minutes, and seconds. If these values are out of range, the member variables hr, min, and sec are initialized to 0. Let us now explain how the member function setTime works when accessed by an object of type clockType. The member function setTime is a void function and has three parameters. Therefore:

- A call to this function is a stand-alone statement.
- We must use three parameters in a call to this function.

Furthermore, recall that because setTime is a member of the class clockType, it can directly access the member variables hr, min, and sec, as shown in the definition of setTime.

The object myClock has three member variables, as shown in Figure (a) below



Consider the following statement:

myClock.setTime(3, 48, 52);

Now the function **setTime** is called with parameters 3, 48, and 52. So the values of the formal parameters hours, minutes, and seconds of the function setTime are 3, 48, and 52, respectively.

Next, in the statement myClock.setTime(3, 48, 52);, setTime is accessed by the object myClock. Therefore, the three variables—hr, min, and sec—referred to in the body of the function **setTime** are the three member variables of myClock. When the body of the function **setTime** executes, the value of hours is copied into myClock.hr, the value of minutes is copied into **myClock.min,** and the value of seconds is copied into myClock.sec. In essence, the values, **3**, **48**, and **52**, which are passed as parameters in the preceding statement, are assigned to the three member variables of **myClock** by the function **setTime** (see the body of the function setTime). After the previous statement executes, the object myClock is as shown in Figure (b)

The definitions of these functions are simple and easy to follow:

```
void clockType::getTime(int& hours, int& minutes, int& seconds) const
{
        hours = hr;
        minutes = min;
        seconds = sec;
 }

void clockType::printTime() const
```

```
{
        if (hr < 10)
                cout << "0";
                cout << hr << ":";
       if (min < 10) cout << "0";
                cout << min << ":";
        if (sec < 10)
                cout << "0";
                cout << sec;
}


void clockType::incrementHours()
{
        hr++;
                if (hr > 23)
                hr = 0;
}









void clockType::incrementMinutes()
{
        min++;
        If (min > 59)
         {
                min = 0;
                incrementHours();
        }
}
```

```
void clockType::incrementSeconds()
{
        sec++;
        if (sec > 59)
                {
                 sec = 0;
                incrementMinutes();
                 }
 }
```

From the definitions of the functions incrementMinutes and incrementSeconds, it is clear that a member function of a class can call other member functions of the class.

The function equalTime has the following definition: bool clockType::equalTime(const clockType& otherClock) const

```
 {
        return (hr == otherClock.hr
        && min == otherClock.min
        && sec == otherClock.sec);
}
```

Suppose that myClock and yourClock are objects of type clockType, as declared previously. Further suppose that we have myClock and yourClock,

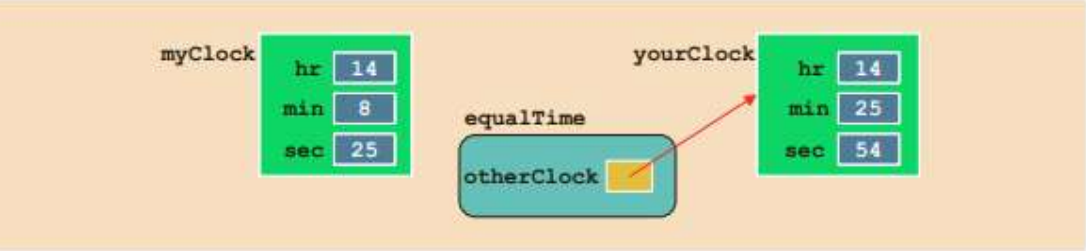
Consider the following statement:

```
if (myClock.equalTime(yourClock))
. . .
```

In the expression:

myClock.equalTime(yourClock)

the object **myClock** accesses the member function **equalTime**. Because **otherClock** is a reference parameter, the address of the actual parameter **yourClock** is passed to the formal parameter otherClock,as in Figure below



The member variables hr, min, and sec of otherClock have the values 14, 25, and 54, respectively. In other words, when the body of the function equalTime executes, the value of otherClock.hr is 14, the value of otherClock.min is 25, and the value of otherClock.sec is 54. The function equalTime is a member of myClock. When the function equalTime executes, the variables hr, min, and sec in the body of the function equalTime are the member variables of the object myClock.

Let us again take a look at the definition of the function equalTime. Notice that within the definition of this function, the object otherClock accesses the member variables hr, min, and sec. However, these member variables are private. So is there any violation? The answer is no. The function equalTime is a member of the class clockType, and hr, min, and sec are the member variables. Moreover, otherClock is an object of type clockType. Therefore, the object otherClock can access its private member variables within the definition of the function equalTime.

The same is true for any member function of a class. In general, when you write the definition of a member function, say, dummyFunction, of a class, say, dummyClass, and the function uses an object, dummyObject of the class dummyClass, then within the definition of dummyFunction, the object dummyObject can access its private

member variables (in fact, any private member of the class). Once a class is properly defined and implemented, it can be used in a program.

A program or software that uses and manipulates the objects of a class is called a **client** of that class.

Accessor and Mutator Functions

Let us look at the member functions of the class clockType. The function setTime sets the values of the member variables to the values specified by the user. In other words, it alters or modifies the values of the member variables. Similarly, the functions incrementSeconds, incrementMinutes, and incrementHours also modify the member variables. On the other hand, functions such as getTime, printTime, and equalTime only access the values of the member variables. They do not modify the member variables. We can, therefore, categorize the member functions of the class clockType into two categories: member functions that modify the member variables and member functions that only access, but do not modify, the member variables.

This is typically true for any class. That is, every class has member functions that only access but do not modify the member variables, called accessor functions, and member functions that modify the member variables, called mutator functions.

Accessor function: A member function of a class that only accesses (that is, does not modify) the value(s) of the member variable(s).
Mutator function: A member function of a class that modifies the value(s) of the member variable(s).

Because an accessor function only accesses the values of the member variables, as a safeguard, we typically include the reserved word const at the end of the headings of these functions. Moreover, a constant member function of a class cannot modify the member variables of that class. For example, see the headings of the member functions getTime, printTime, and equalTime of the class clockType.

Full Program – ClockType

EXAMPLE

```
int hours;                                              //Line 5
int minutes;                                            //Line 6
int seconds;                                            //Line 7

    //Set the time of myClock
myClock.setTime(5, 4, 30);                              //Line 8

cout << "Line 9: myClock: ";                            //Line 9
myClock.printTime();  //print the time of myClock       Line 10
cout << endl;                                           //Line 11

cout << "Line 12: yourClock: ";                         //Line 12
yourClock.printTime(); //print the time of yourClock    Line 13
cout << endl;                                           //Line 14

    //Set the time of yourClock
yourClock.setTime(5, 45, 16);                           //Line 15

cout << "Line 16: After setting, yourClock: ";          //Line 16
yourClock.printTime(); //print the time of yourClock    Line 17
cout << endl;                                           //Line 18

    //Compare myClock and yourClock
if (myClock.equalTime(yourClock))                       //Line 19
    cout << "Line 20: Both times are equal."
         << endl;                                       //Line 20
else                                                    //Line 21
    cout << "Line 22: The two times are not equal."
         << endl;                                       //Line 22

cout << "Line 23: Enter the hours, minutes, and "
     << "seconds: ";                                    //Line 23
cin >> hours >> minutes >> seconds;                     //Line 24
```

```
cout << endl;                                          //Line 25

    //Set the time of myClock using the value of the
    //variables hours, minutes, and seconds
myClock.setTime(hours, minutes, seconds);              //Line 26

cout << "Line 27: New myClock: ";                      //Line 27
myClock.printTime();   //print the time of myClock      Line 28
cout << endl;                                          //Line 29

    //Increment the time of myClock by one second
myClock.incrementSeconds();                            //Line 30

cout << "Line 31: After incrementing myClock by "
     << "one second, myClock: ";                       //Line 31
myClock.printTime();   //print the time of myClock      Line 32
cout << endl;                                          //Line 33
```

```
        //Retrieve the hours, minutes, and seconds of the
        //object myClock
    myClock.getTime(hours, minutes, seconds);              //Line 34

        //Output the value of hours, minutes, and seconds
    cout << "Line 35: hours = " << hours
         << ", minutes = " << minutes
         << ", seconds = " << seconds << endl;             //Line 35

    return 0;                                              //Line 36
}//end main                                                  Line 37

void clockType::setTime(int hours, int minutes, int seconds)
{
    if (0 <= hours && hours < 24)
        hr = hours;
    else
        hr = 0;

    if (0 <= minutes && minutes < 60)
        min = minutes;
    else
        min = 0;

    if (0 <= seconds && seconds < 60)
        sec = seconds;
    else
        sec = 0;
}
```

Sample Run: In this sample run, the user input is shaded.

```
Line 9: myClock: 05:04:30
Line 12: yourClock: 0-858993460:0-858993460:0-858993460
Line 16: After setting, yourClock: 05:45:16
Line 22: The two times are not equal.
Line 23: Enter the hours, minutes, and seconds: 8 45 59

Line 27: New myClock: 08:45:59
Line 31: After incrementing myClock by one second, myClock: 08:46:00
Line 35: hours = 8, minutes = 46, seconds = 0
```

The value of `yourClock`, as printed in the second line of the output (Line 12), is machine dependent; you might get different values.

Constructors

To guarantee that the member variables of a class are initialized, you use constructors. There are two types of constructors: with parameters and without parameters. The constructor without parameters is called the **default constructor.**

Constructors have the following properties:

- The name of a constructor is the same as the name of the class.
- A constructor is a function and it has no type. That is, it is neither a value-returning function nor a void function.
- A class can have more than one constructor. However, all constructors of a class have the same name.
- If a class has more than one constructor, the constructors must have different formal parameter lists. That is, either they have a different number of formal parameters or, if the number of formal parameters is the same, then the data type of the formal parameters, in the order you list, must differ in at least one position. In other words, like function overloading, a constructor's name is overloaded.
- Constructors execute automatically when a class object is declared and enters its scope. Because they have no types, they cannot be called like other functions.
- Which constructor executes depends on the types of values passed to the class object when the class object is declared.

Let us extend the definition of the class **clockType** by including two constructors:

```
class clockType
{
        public:
```

```
        void setTime(int, int, int);
        void getTime(int&, int&, int&) const;
        void printTime() const;
        void incrementSeconds();
        void incrementMinutes();
        void incrementHours();
        bool equalTime(const clockType&) const;
        clockType(int, int, int);          //constructor with parameters
        clockType();                       //default constructor

        private:
                int hr;
                int min;
                int sec;
      };
```

This definition of the class clockType includes two constructors: one with three parameters and one without any parameters. Let us now write the definitions of these constructors:

```
clockType::clockType(int hours, int minutes, int seconds)
{
        if (0 <= hours && hours < 24)
                hr = hours;
        else
                hr = 0;
        if (0 <= minutes && minutes < 60)
                min = minutes;
        else
                min = 0;
        if (0 <= seconds && seconds < 60)
                sec = seconds;
      else
                 sec = 0;
}

clockType::clockType()              //default constructor
 {
        hr = 0;
        min = 0;
        sec = 0;
 }
```

Invoking Constructors

Invoking the Default Constructor Suppose that a class contains the default constructor. The syntax to invoke the default constructor is:

```
className classObjectName;
```

For example, the statement:

clockType yourClock;

declares yourClock to be an object of type clockType.

If you declare an object and want the default constructor to be executed, the empty parentheses after the object name are not required in the object declaration statement. In fact, if you accidentally include the empty parentheses, the compiler generates a syntax error message. For example, the following statement to declare the object **yourClock** is illegal:

```
clockType yourClock(); //illegal object declaration
```

Invoking a Constructor with Parameters

Suppose a class contains constructors with parameters. The syntax to invoke a constructor with a parameter is:

```
className classObjectName(argument1, argument2, ...);
```

in which argument1, argument2, and so on are either a variable or an expression. Note the following:

- The number of arguments and their type should match the formal parameters (in the order given) of one of the constructors.
- If the type of the arguments does not match the formal parameters of any constructor (in the order given), C++ uses type conversion and looks for the best match. For example, an integer value might be converted to a floating-point value with a zero decimal part. Any ambiguity will result in a compile-time error.

EXAMPLE

Consider the following class definition:

```
class inventory
{
public:
    inventory();                            //Line 1
    inventory(string);                      //Line 2
    inventory(string, int, double);         //Line 3
    inventory(string, int, double, int); //Line 4

    //Add additional functions

private:
    string name;
    int itemNum;
    double price;
    int unitsInStock;
};
```

This class has four constructors and four member variables. Suppose that the definitions of the constructors are as follows:

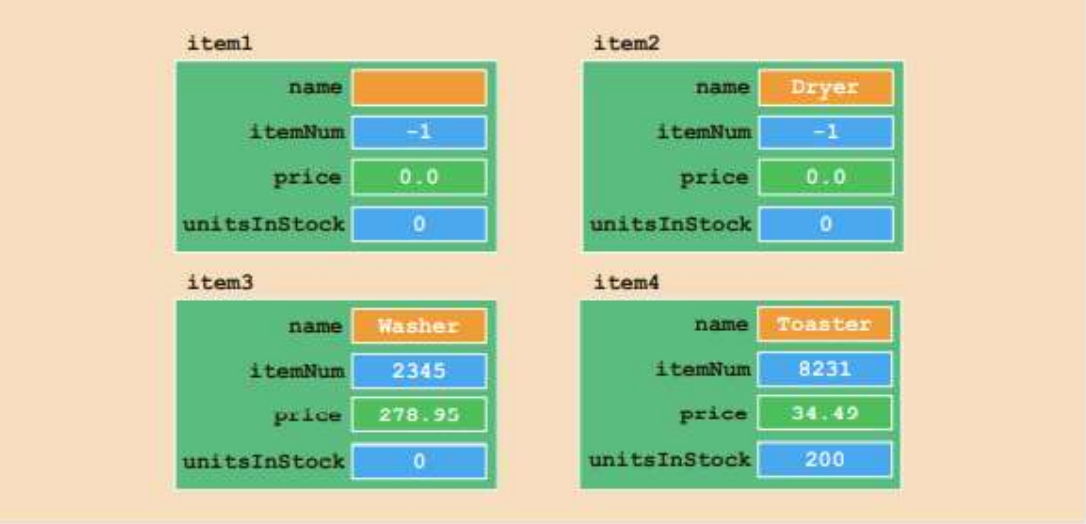
```
inventory::inventory() //default constructor
{
    name = "";
    itemNum = -1;
    price = 0.0;
    unitsInStock = 0;
}

inventory::inventory(string n)
{
    name = n;
    itemNum = -1;
    price = 0.0;
    unitsInStock = 0;
}

inventory::inventory(string n, int iNum, double cost)
{
    name = n;
    itemNum = iNum;
    price = cost;
    unitsInStock = 0;
}
```

```
inventory::inventory(string n, int iNum, double cost, int inStock)
{
    name = n;
    itemNum = iNum;
    price = cost;
    unitsInStock = inStock;
}
```

Consider the following declarations:

```
inventory item1;
inventory item2("Dryer");
inventory item3("Washer", 2345, 278.95);
inventory item4("Toaster", 8231, 34.49, 200);
```

For item1, the default constructor in Line 1 executes because no value is passed to this variable. For item2, the constructor in Line 2 executes because only one parameter, which is of type string, is passed, and it matches with the constructor in Line 2. For item3, the constructor in Line 3 executes because three parameters are passed to item3, and they match with the constructor in Line 3. Similarly, for item4, the constructor in Line 4 executes (see Figure below).



NOTE

If the values passed to a class object do not match the parameters of any constructor and if no type conversion is possible, a compile-time error will be generated.

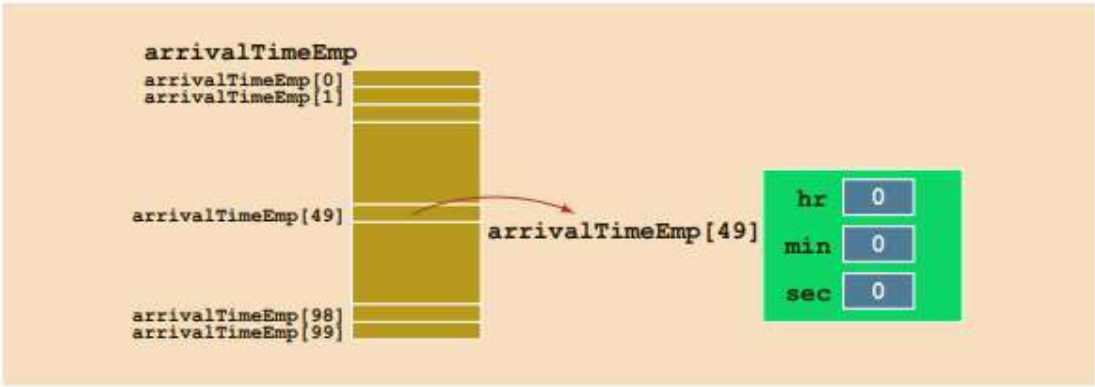
Arrays of Objects (Variables) and Constructors

If a class has constructors and you declare an array of that class's objects, the class should have the default constructor. The default constructor is typically used to initialize each (array) class object. For example, if you declare an array of 100 class objects, then it is impractical (if not impossible) to specify different constructors for each component.

Suppose that you have 100 employees who are paid on an hourly basis, and you need to keep track of their arrival and departure times. You can declare two arrays—arrivalTimeEmp and departureTimeEmp—of 100 components each, wherein each component is an object of type clockType

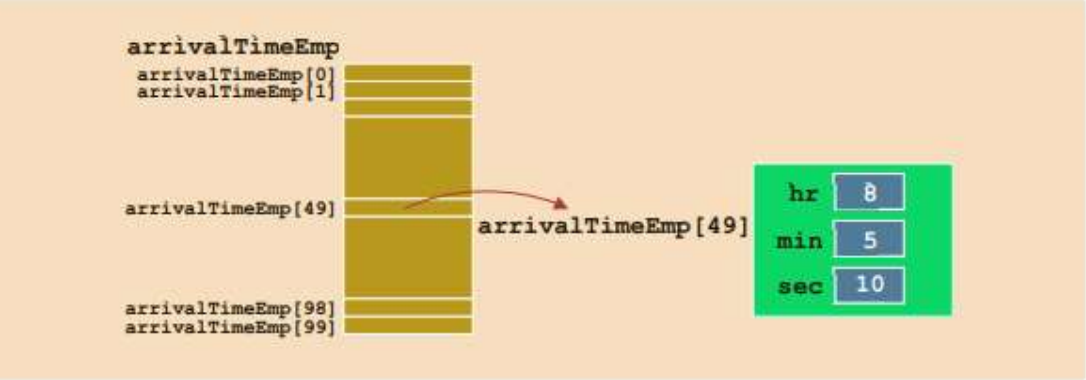
Consider the following statement:
clockType arrivalTimeEmp[100]; //Line 1

The statement in Line 1 creates the array of objects arrivalTimeEmp[0], arrivalTimeEmp[1], ..., arrivalTimeEmp[99], as shown in Figure below.



You can now use the functions of the class clockType to manipulate the time for each employee. For example, the following statement sets the arrival time, that is, hr, min, and sec, of the 50th employee to 8, 5, and 10, respectively see Figure below

arrivalTimeEmp[49].setTime(8, 5, 10); //Line 2



To output the arrival time of each employee, you can use a loop, such as the following:

```
for (int j = 0; j < 100; j++)                                    //Line 3
 {
              cout << "Employee " << (j + 1) << " arrival time: ";
        arrivalTimeEmp[j].printTime();                           //Line 4
        cout << endl;
}
```

The statement in Line 4 outputs the arrival time of an employee in the form **hr:min:sec**.

To keep track of the departure time of each employee, you can use the array **departureTimeEmp**.

Similarly, you can use arrays to manage a list of names or other objects.

Destructors

Like constructors, destructors are also functions. Moreover, like constructors, a destructor does not have a type. That is, it is neither a value-returning function nor a void function. However, a class can have only one destructor, and the destructor has no parameters. The name of a destructor is the tilde character (~), followed by the name of the class. For example, the name of the destructor for the class clockType is:

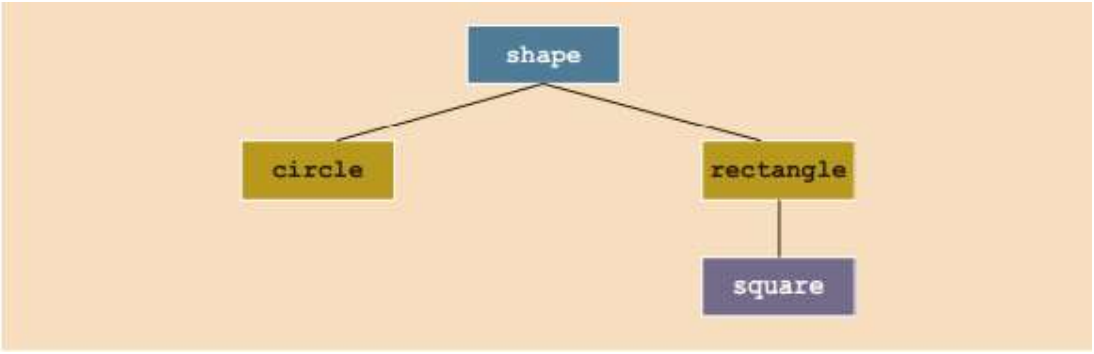
~clockType();

6.2 INHERITANCE

Inheritance lets us create new classes from existing classes. The new classes that we create from the existing classes are called the **derived classes**; the existing classes

are called the **base classes**. The derived classes inherit the properties of the base classes. So rather than create completely new classes from scratch, we can take advantage of inheritance and reduce software development complexity.

Each derived class, in turn, can become a base class for a future derived class. Inheritance can be either single inheritance or multiple inheritance. In **single inheritance**, the derived class is derived from a single base class; in **multiple inheritance**, the derived class is derived from more than one base class. This chapter concentrates on single inheritance.



Inheritance can be viewed as a treelike, or hierarchical, structure wherein a base class is shown with its derived classes. Consider the tree diagram shown in Figure above. In this diagram, shape is the base class. The classes circle and rectangle are derived from shape, and the class square is derived from rectangle. Every circle and every rectangle is a shape. Every square is a rectangle.

The general syntax of a derived class is:

```
class className: memberAccessSpecifier baseClassName
{
    member list
};
```

in which **memberAccessSpecifier** is public, protected, or private. When no memberAccessSpecifier is specified, it is assumed to be a private inheritance.

EXAMPLE

Suppose that we have defined a class called **shape**. The following statements specify that the class **circle** is derived from **shape**, and it is a public inheritance.

```
class circle: public shape
{
    .
    .
    .
};
```

On the other hand, consider the following definition of the class **circle**:

```
class circle: private shape
{
    .
    .
    .
};
```

This is a private inheritance. In this definition, the public members of **shape** become private members of the class **circle**. So any object of type **circle** cannot directly access these members. The previous definition of **circle** is equivalent to:

```
class circle: shape
{
    .
    .
    .
};
```

That is, if we do not use either the **memberAccessSpecifier** public or private, the public members of a base class are inherited as private members by default.

The following facts about the base and the derived classes should be kept in mind.

1. The private members of a base class remain private to the base class; hence, the members of the derived class cannot directly access them. In other words, when you write the definitions of the member functions of the derived class, even though the private members of the base class are members of the derived class, the derived class cannot directly access them.
2. The public members of a base class can be inherited either as public members or as private members by the derived class. That is, the public members of the base class can become either public or private members of the derived class. This means that what were public members in the base class can either remain public members or become private members in the derived class.

3. The derived class can include additional members—data and/or functions.
4. The derived class can redefine the `public` member functions of the base class. That is, in the derived class, you can have a member function with the same name, number, and types of parameters as a function in the base class, but with different code in the function body. However, this redefinition applies only to the objects of the derived class, not to the objects of the base class.
5. All member variables of the base class are also member variables of the derived class. Similarly, the member functions of the base class (unless redefined) are also member functions of the derived class. (Remember Rule 1 when accessing a member of the base class in the derived class.)

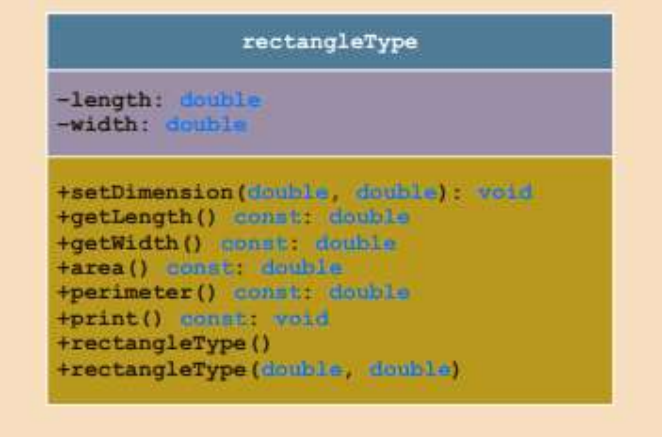
Redefining(Overiding) Member Functions of the Base Class

Consider the definition of the following class:

```
class rectangleType
{
public:
    void setDimension(double l, double w);
      //Function to set the length and width of the rectangle.
      //Postcondition: length = l; width = w;

    double getLength() const;
      //Function to return the length of the rectangle.
      //Postcondition: The value of length is returned.

    double getWidth() const;
      //Function to return the width of the rectangle.
      //Postcondition: The value of width is returned.

    double area() const;
      //Function to return the area of the rectangle.
      //Postcondition: The area of the rectangle is
      //               calculated and returned.

    double perimeter() const;
      //Function to return the perimeter of the rectangle.
      //Postcondition: The perimeter of the rectangle is
      //               calculated and returned.

    void print() const;
      //Function to output the length and width of
      //the rectangle.

    rectangleType();
      //Default constructor
      //Postcondition: length = 0; width = 0;
```

```
        rectangleType(double l, double w);
          //Constructor with parameters
          //Postcondition: length = l; width = w;

    private:
        double length;
        double width;
};
```

Figure below shows the UML class diagram of the class **rectangleType**



The class rectangleType has 10 members. Suppose that the definitions of the member functions of the class rectangleType are as follows:

```
void rectangleType::setDimension(double l, double w)
{
    if (l >= 0)
        length = l;
    else
        length = 0;

    if (w >= 0)
        width = w;
    else
        width = 0;
}

double rectangleType::getLength() const
{
    return length;
}

double rectangleType::getWidth()const
{
    return width;
}

double rectangleType::area() const
{
    return length * width;
}
```

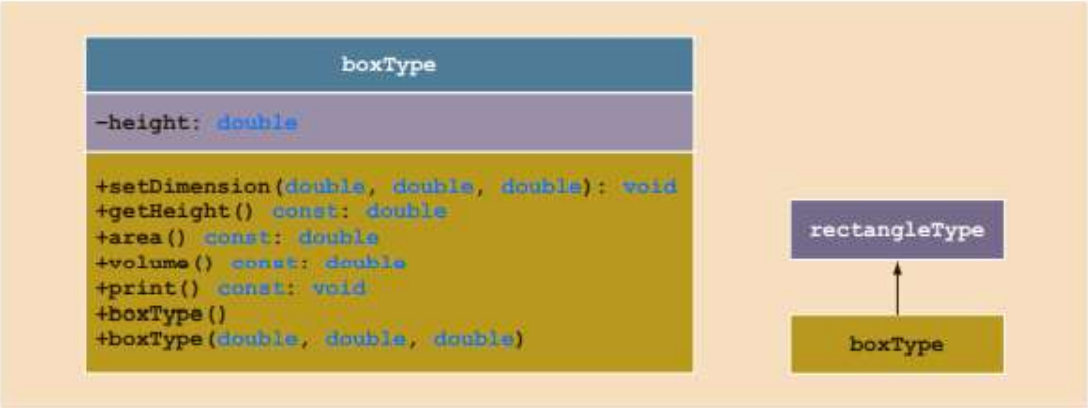
```
double rectangleType::perimeter() const
{
     return 2 * (length + width);
}

void rectangleType::print() const
{
    cout << "Length = "  << length
         << "; Width = " << width;
}

rectangleType::rectangleType(double l, double w)
{
    setDimension(l, w);
}

rectangleType::rectangleType()
{
    length = 0;
    width = 0;
}
```

Now consider the definition of the following class boxType, derived from the class rectangleType:

```
class boxType: public rectangleType
{
public:
    void setDimension(double l, double w, double h);
      //Function to set the length, width, and height
      //of the box.
      //Postcondition: length = l; width = w; height = h;

    double getHeight() const;
      //Function to return the height of the box.
      //Postcondition: The value of height is returned.

    double area() const;
      //Function to return the surface area of the box.
      //Postcondition: The surface area of the box is
      //               calculated and returned.

    double volume() const;
      //Function to return the volume of the box.
      //Postcondition: The volume of the box is
      //               calculated and returned.

    void print() const;
      //Function to output the length, width, and height of a box.

    boxType();
      //Default constructor
      //Postcondition: length = 0; width = 0; height = 0;
```

```
    boxType(double l, double w, double h);
      //Constructor with parameters
      //Postcondition: length = l; width = w; height = h;

private:
    double height;
};
```

Figure below shows the UML class diagram of the class boxType and the inheritance hierarchy.



From the definition of the class boxType, it is clear that the class boxType is derived from the class rectangleType, and it is a public inheritance. Therefore, all public members of the class rectangleType are public members of the class boxType. The class boxType also overrides (redefines) the functions print and area.

In general, while writing the definitions of the member functions of a derived class to specify a call to a public member function of the base class, we do the following: ? If the derived class overrides a public member function of the base class, then to specify a call to that public member function of the base class, you use the name of the base class, followed by the scope resolution operator, ::, followed by the function name with the appropriate parameter list. For example, to call the function area of the class rectangleType the statement is: rectangleType::area().

To call the member function print of rectangleType in the definition of the member function print of boxType, we must use the following statement:

rectangleType::print();

This statement ensures that we call the member function print of the base class rectangleType, not of the class boxType.

The definition of the member function print of the class boxType is:

```
void boxType::print() const
{
        rectangleType::print();
         cout << "; Height = " << height;
}
```

The definition of the function setDimension is:

```
void boxType::setDimension(double l, double w, double h)
{
        rectangleType::setDimension(l, w);
                if (h >= 0)
                        height = h;
                else height = 0;
}
```

Notice that in the preceding definition of the function setDimension, a call to the member function setDimension of the class rectangleType is preceded by the name of the class and the scope resolution operator, even though the class boxType overloads—not overrides—the function setDimension.

The definition of the function getHeight is:

```
double boxType::getHeight() const
{
```

```
    return height;
}
```

The member function area of the class boxType determines the surface area of a box. To determine the surface area of a box, we need to access the length and width of the box, which are declared as private members of the class rectangleType. Therefore, we use the member functions getLength and getWidth of the class rectangleType to retrieve the length and width, respectively. Because the class boxType does not contain any member functions that have the names getLength or getWidth, we can call these member functions of the class rectangleType without coupling them to the name of the base class.

```
double boxType::area() const
{
    return 2 * (getLength() * getWidth()
            + getLength() * height
            + getWidth() * height);
}
```

The member function volume of the class boxType determines the volume of a box. To determine the volume of a box, you multiply the length, width, and height of the box or multiply the area of the base of the box by its height.

Let us write the definition of the member function volume by using the second alternative. To do this, you can use the member function area of the class rectangleType to determine the area of the base. Because the class boxType overrides the member function area, to specify a call to the member function area of the class rectangleType, we use the name of the base class and the scope resolution operator, as shown in the following definition:

```
double boxType::volume() const
{
    return rectangleType::area() * height;
}
```

Constructors of Derived and Base Classes

A derived class can have its own private member variables, so a derived class can explicitly include its own constructors to initialize them. When we declare a derived class object, this object inherits the members of the base class, but the derived class object cannot directly access the private (data) members of the base class. The same is true for the member functions of a derived class. That is, the member functions of a derived class cannot directly access the private members of the base class.

As a consequence, the constructors of a derived class can (directly) initialize only the (public data) members inherited from the base class of the derived class. Thus, when a derived class object is declared, it must also trigger the execution of one of the base class's constructors. Furthermore, this triggering of the base class's constructor is specified in the *heading of the definition* of a derived class constructor.

First, let us write the definition of the default constructor of the class boxType. Recall that, if a class contains a default constructor and no values are specified when the object is declared, the default constructor executes and initializes the object. Because the class rectangleType contains the default constructor, we do not specify any constructor of the base class when writing the definition of the default constructor of the class boxType.

```
boxType::boxType()
 {
        height = 0.0;
}
```

To write the definition of class boxType constructor with parameters, we first write the class boxType constructor heading *including all of the parameters needed for both the base class and derived class constructors;* that is, all the parameters needed for both **boxType** and **rectangleType**. Then, to trigger the execution of the base class constructor with parameters, we add a colon (:) to the heading followed by the name of the constructor of the base class with its parameters in the heading of the definition

of the constructor of the derived class. In effect, we “tack on” the base class constructor to the derived class constructor via a colon. The derived class constructor gets all of the parameters needed for itself and the base class constructor, then passes on the base class parameters to its constructor.

Consider the following definition of the constructor with parameters of the class boxType:

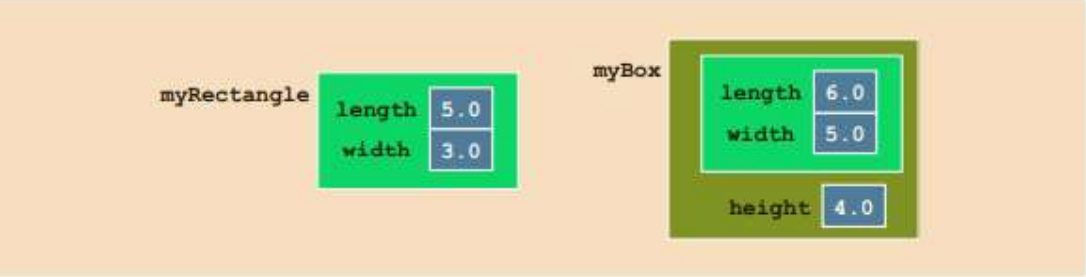
```
boxType::boxType(double l, double w, double h) : rectangleType(l, w)
{
        if (h >= 0)
                height = h;
        else
                height = 0;
}
```

In this definition, we specify the constructor of **rectangleType** with two parameters. When this constructor of **boxType** executes, it triggers the execution of the constructor of the class rectangleType with two parameters of type double.

Consider the following statements:

```
rectangleType myRectangle(5.0, 3.0);          //Line 1
boxType myBox(6.0, 5.0, 4.0);                 //Line 2
```

The statement in Line 1 creates the **rectangleType** object **myRectangle**. Thus, the object **myRectangle** has two member variables: length and width. The statement in Line 2 creates the boxType object **myBox**. Thus, the object myBox has three member variables: **length**, **width**, and height (see Figure below)



Consider the following statements:

```
myRectangle.print();                    //Line 3
cout << endl;                           //Line 4
myBox.print();                          //Line 5
 cout << endl;                          //Line 6
```

In the statement in Line 3, the member function print of the class rectangleType is executed. In the statement in Line 5, the function print associated with the class boxType is executed. Recall that, if a derived class overrides a member function of the base class, the redefinition applies only to the objects of the derived class. Thus, the output of the statement in Line 3 is (as defined in the class rectangleType):

Length = 5.0; Width = 3.0

The output of the statement in Line 5 is (as defined in the class boxType):

Length = 6.0; Width = 5.0; Height = 4.0

When the object myBox enters its scope, the constructors of both the classes rectangleType and boxType execute. Note that the constructors of a base class are not inherited by a derived class. So if a base class contains private data members, only base class constructors can construct the base class data members, including the base class part of a derived class. In this case, derived class constructors can only construct the additional members specified in its definition. This means that a call to a

base class constructor must be included in the definition of a constructor of the derived class.

When a derived class constructor executes, first a constructor of the base class executes to initialize the data members inherited from the base class, and then the constructor of the derived class executes to initialize the data members declared by the derived class. So first, the constructor of the class rectangleType executes to initialize the instance variables length and width, and then the constructor of the class boxType executes to initialize the instance variable height.

EXAMPLE

In this example, we write a program to solve the following problems:

1. Jim's lawn care store specializes in putting up fences around small farms and home lawns and fertilizing the farms and lawns. For simplicity, we assume that the yards and farms are rectangular. In order to put up the fence, the program needs to know the perimeter, and to fertilize, the program needs to know the area. We will write a program that uses the `class` **`rectangle`** to store the dimensions of a yard or a farm. The program will also prompt the user to input the dimensions (in feet) of a yard or farm, the cost (per foot) to put up the fence, and the cost (per square foot) to fertilize the area. The program will then output the cost of putting up the fence and fertilizing the area.
2. Linda's gift store specializes in wrapping small packages. For simplicity, we assume that a package is in the shape of a box with a specific length, width, and height. We will write a program that uses the `class` **`boxType`** to store the dimensions of a package. The program will ask the user to input the dimensions of the package and the cost (per square foot) to wrap the package. The program will then output the cost of wrapping the package. (The program assumes that the minimum cost of wrapping a package is $1.00.)

Consider the following C++ program:

```
#include <iostream>                                          //Line 1
#include <iomanip>                                           //Line 2
#include "rectangleType.h"                                   //Line 3
#include "boxType.h"                                         //Line 4

using namespace std;                                         //Line 5


int main()                                                   //Line 6
{                                                            //Line 7
    rectangleType yard;                                      //Line 8
    double fenceCostPerFoot;                                 //Line 9
    double fertilizerCostPerSquareFoot;                      //Line 10
    double length, width;                                    //Line 11
    double billingAmount;                                    //Line 12

    cout << fixed << showpoint << setprecision(2);           //Line 13

    cout << "Line 14: Enter the length and width of the "
         << "yard (in feet): ";                              //Line 14
    cin >> length >> width;                                  //Line 15
    cout << endl;                                            //Line 16
```

```
    yard.setDimension(length, width);                          //Line 17

    cout << "Line 18: Enter the cost of fence "
         << "(per foot): $";                                   //Line 18
    cin >> fenceCostPerFoot;                                   //Line 19
    cout << endl;                                              //Line 20

    cout << "Line 21: Enter the cost of fertilizer "
         << "(per square foot): $";                            //Line 21
    cin >> fertilizerCostPerSquareFoot;                        //Line 22
    cout << endl;                                              //Line 23

    billingAmount = yard.perimeter() * fenceCostPerFoot
              + yard.area() * fertilizerCostPerSquareFoot;     //Line 24

    cout << "Line 25: Amount due: $" << billingAmount
         << endl;                                              //Line 25

    boxType package;                                           //Line 26
    double height;                                             //Line 27
    double wrappingCostPerSquareFeet;                          //Line 28

    cout << "Line 29: Enter the length, width, and height "
         << "of the package (in feet): ";                      //Line 29
    cin >> length >> width >> height;                          //Line 30
    cout << endl;                                              //Line 31

    package.setDimension(length, width, height);               //Line 32

    cout << "Line 33: Enter the cost (25 to 50 cents) of "
         << "wrapping per square foot: ";                      //Line 33
    cin >> wrappingCostPerSquareFeet;                          //Line 34
    cout << endl;                                              //Line 35


      billingAmount = wrappingCostPerSquareFeet
                    * package.area() / 100;                    //Line 36



      if (billingAmount < 1.00)                                //Line 37
          billingAmount = 1.00;                                //Line 38

      cout << "Line 39: Amount due: $" << billingAmount
           << endl;                                            //Line 39

      return 0;                                                //Line 40
}                                                              //Line 41
```

Sample Run: In this sample run, the user input is shaded.

```
Line 14: Enter the length and width of the yard (in feet): 70 50

Line 18: Enter the cost of fence (per foot): $10.00

Line 21: Enter the cost of fertilizer (per square foot): $0.25

Line 25: Amount due: $3275.00

Line  29: Enter the length, width, and height of the package (in feet): 3 2 0.25

Line 33: Enter the cost (25 to 50 cents) of wrapping per square foot: 25

Line 39: Amount due: $3.63
```

NOTE

Suppose that a base `class`, **`baseClass`**, has *private* member variables and constructors. Further suppose that the `class` **`derivedClass`** is derived from **`baseClass`**, and **`derivedClass`** has no member variables. Therefore, the member variables of **`derivedClass`** are the ones inherited from **`baseClass`**. A constructor cannot be called like other functions, and the member variables of **`baseClass`** cannot be directly accessed by the member functions of **`derivedClass`**. To guarantee the initialization of the inherited member variables of an object of type **`derivedClass`**, even though **`derivedClass`** has no member variables, it must have the appropriate constructors. A constructor (with parameters) of **`derivedClass`** merely issues a call to a constructor (with parameters) of **`baseClass`**. Therefore, when you write the definition of the constructor (with parameters) of **`derivedClass`**, the heading of the definition of the constructor contains a call to an appropriate constructor (with parameters) of **`baseClass`**, and the body of the constructor is empty—that is, it contains only the opening and closing braces.

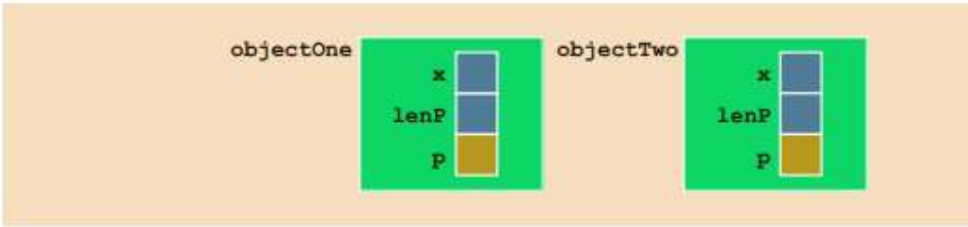
Classes and Pointers

Because a class can have pointer member variables, this section discusses some peculiarities of such classes. To facilitate the discussion, we will use the following class:

```
class ptrMemberVarType
 {
        public:
                . . .
         private:
                int x;
                int lenP;
                int *p;
};
```

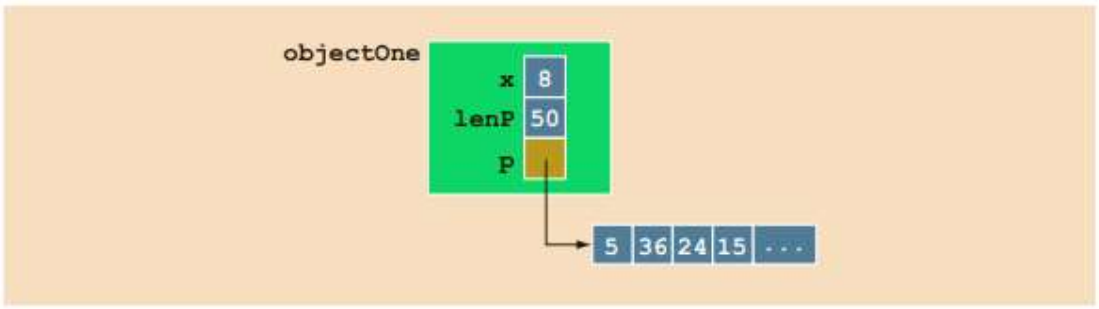
Also, consider the following statements

ptrMemberVarType objectOne;
ptrMemberVarType objectTwo;



Destructor

The object objectOne has a pointer member variable p. Suppose that during program execution, the pointer p creates a dynamic array. When objectOne goes out of scope, all of the member variables of objectOne are destroyed. However, p created a dynamic array, and dynamic memory must be deallocated using the operator delete. Thus, if the pointer p does not use the delete operator to deallocate the dynamic array, the memory space of the dynamic array will stay marked as allocated, even though it cannot be accessed. How do we ensure that when p is destroyed, the dynamic memory created by p is also destroyed? Suppose that objectOne is as shown in Figure below.



Therefore, we can put the necessary code in the destructor to ensure that when objectOne goes out of scope, the memory created by the pointer p is deallocated. For example, the definition of the destructor for the class **ptrMemberVarType** is:

```
ptrMemberVarType::~ptrMemberVarType()
{
```

```
        delete [] p;
}
```

Of course, you must include the destructor as a member of the class in its definition. Let us extend the definition of the class ptrMemberVarType by including the destructor. Moreover, the remainder of this section assumes that the definition of the destructor is as given previously—that is, the destructor deallocates the memory space pointed to by p.

```
class ptrMemberVarType
{
        public:
                ~ptrMemberVarType();
                . . .
        private:
                int x;
                int lenP;
                int *p;
};
```

For the destructor to work properly, the pointer `p` must have a valid value. If `p` is not properly initialized (that is, if the value of `p` is garbage) and the destructor executes, either the program terminates with an error message or the destructor deallocates an unrelated memory space. For this reason, you should exercise extra caution while working with pointers.

Inline Functions

The definiton of the class clockType contains the declarations of the data members and the function prototypes of the member functions. The definitions of the member

functions are placed in the implementations file. However, in the definition of a class you can give the complete definition of a member function. Such member functions definitions are called inline function definitions. Suppose that you want to include a function to return the hours of a clock. You can write the definition of the class clockType as follows:

```
class clockType
{
public:
    void setTime(int hours, int minutes, int seconds);
    void getTime(int& hours, int& minutes, int& seconds) const;
    void printTime() const;

    void incrementSeconds();
    void incrementMinutes();
    void incrementHours();

    bool equalTime(const clockType& otherClock) const;

    int getHours() const
    {
        return hr;
    }

    clockType(int hours = 0, int minutes = 0,
              int seconds = 0);

private:
    int hr;
    int min;
    int sec;
};
```

In this definition of the class, the function getHours is inline. Inline function definitions are typically used for very short function definitions. The compiler treats inline functions in a special way. It typically inserts the code of an inline function at every location the function is called. When a function is called, memory for its parameters and local variables is allocated and when the function exits the memory is deallocated. So there is an overhead when calling a function. In the case of an inline function the overhead of a function invocation is saved.

PROGRAMMING EXAMPLE: Juice Machine

A common place to buy juice is from a machine. A new juice machine has been purchased for the gym, but it is not working properly. The machine sells the following types of juices: orange, apple, mango, and strawberry–banana. You have been asked to write a program for this juice machine so that it can be put into operation.

The program should do the following:

1. Show the customer the different products sold by the juice machine.
2. Let the customer make the selection.
3. Show the customer the cost of the item selected.
4. Accept money from the customer.
5. Release the item.

Input The item selection and the cost of the item.

Output The selected item.

COMPLETE PROGRAM LISTING

In the previous sections, we designed the classes to implement cash registers and dispensers to implement a juice machine. In this section, for the sake of completeness, we give complete definitions of the classes, the implementation file, and the user program to implement a juice machine.

```
class cashRegister
{
public:
    int getCurrentBalance() const;
        //Function to show the current amount in the cash
        //register.
        //Postcondition: The value of cashOnHand is returned.
```

```
        void acceptAmount(int amountIn);
           //Function to receive the amount deposited by
           //the customer and update the amount in the register.
           //Postcondition: cashOnHand = cashOnHand + amountIn;

        cashRegister(int cashIn = 500);
           //Constructor
           //Sets the cash in the register to a specific amount.
           //Postcondition: cashOnHand = cashIn;
           //               If no value is specified when the
           //               object is declared, the default value
           //               assigned to cashOnHand is 500.

     private:
          int cashOnHand;      //variable to store the cash
                               //in the register
     };
```

```
class dispenserType
{
public:
    int getNoOfItems() const;
      //Function to show the number of items in the machine.
      //Postcondition: The value of numberOfItems is returned.

    int getCost() const;
      //Function to show the cost of the item.
      //Postcondition: The value of cost is returned.

    void makeSale();
      //Function to reduce the number of items by 1.
      //Postcondition: numberOfItems--;

    dispenserType(int setNoOfItems = 50, int setCost = 50);
      //Constructor
      //Sets the cost and number of items in the dispenser
      //to the values specified by the user.
      //Postcondition: numberOfItems = setNoOfItems;
      //               cost = setCost;
      //               If no value is specified for a
      //               parameter, then its default value is
      //               assigned to the corresponding member
      //               variable.
```

```
private:
    int numberOfItems;      //variable to store the number of
                            //items in the dispenser
    int cost;  //variable to store the cost of an item
};

//**********************************************************
// Author: D.S. Malik
//
// Implementation file juiceMachineImp.cpp
// This file contains the definitions of the functions to
// implement the operations of the classes cashRegister and
// dispenserType.
//**********************************************************

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

using namespace std;

int cashRegister::getCurrentBalance() const
{
    return cashOnHand;
}

void cashRegister::acceptAmount(int amountIn)
{
    cashOnHand = cashOnHand + amountIn;
}

cashRegister::cashRegister(int cashIn)
{
    if (cashIn >= 0)
        cashOnHand = cashIn;
    else
        cashOnHand = 500;
}
```

```
int dispenserType::getNoOfItems() const
{
    return numberOfItems;
}

int dispenserType::getCost() const
{
    return cost;
}

void dispenserType::makeSale()
{
    numberOfItems--;
}
```

```
dispenserType::dispenserType(int setNoOfItems, int setCost)
{
    if (setNoOfItems >= 0)
         numberOfItems = setNoOfItems;
    else
        numberOfItems = 50;

    if (setCost >= 0)
        cost = setCost;
    else
        cost = 50;
}
```

```
#include <iostream>
#include "juiceMachine.h"

using namespace std;

void showSelection();
void sellProduct(dispenserType& product,
                 cashRegister& pCounter);

int main()
{
    cashRegister counter;
    dispenserType orange(100, 50);
    dispenserType apple(100, 65);
    dispenserType mango(75, 80);
    dispenserType strawberryBanana(100, 85);

    int choice;  //variable to hold the selection

    showSelection();
    cin >> choice;

    while (choice != 9)
    {
        switch (choice)
        {
        case 1:
            sellProduct(orange, counter);
            break;
```

```
            case 2:
                sellProduct(apple, counter);
                break;
            case 3:
                sellProduct(mango, counter);
                break;
            case 4:
                sellProduct(strawberryBanana, counter);
                break;
            default:
                cout << "Invalid selection." << endl;
            }//end switch

            showSelection();
            cin >> choice;
        }//end while

        return 0;
}//end main

void showSelection()
{
    cout << "*** Welcome to Shelly's Juice Shop ***" << endl;
    cout << "To select an item, enter " << endl;
    cout << "1 for orange juice (50 cents)" << endl;
    cout << "2 for apple juice (65 cents)" << endl;
    cout << "3 for mango juice (80 cents)" << endl;
    cout << "4 for strawberry banana juice (85 cents)" << endl;
    cout << "9 to exit" << endl;
}//end showSelection
```

```
void sellProduct(dispenserType& product,
                 cashRegister& pCounter)
{
    int amount;  //variable to hold the amount entered
    int amount2; //variable to hold the extra amount needed

    if (product.getNoOfItems() > 0) //if the dispenser is not
                                    //empty
    {
        cout << "Please deposit " << product.getCost()
             << " cents" << endl;
        cin >> amount;

        if (amount < product.getCost())
        {
            cout << "Please deposit another "
                 << product.getCost()- amount
                 << " cents" << endl;
            cin >> amount2;
            amount = amount + amount2;
       }
```

```
        if (amount >= product.getCost())
        {
            pCounter.acceptAmount(amount);
            product.makeSale();
            cout << "Collect your item at the bottom and "
                 << "enjoy." << endl;
        }
        else
            cout << "The amount is not enough. "
                 << "Collect what you deposited." << endl;

        cout << "*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*"
             << endl << endl;
    }
    else
        cout << "Sorry, this item is sold out." << endl;
}//end sellProduct
```

Sample Run: In this sample run, the user input is shaded.

```
*** Welcome to Shelly's Juice Shop ***
To select an item, enter
1 for orange juice (50 cents)
2 for apple juice (65 cents)
3 for mango juice (80 cents)
4 for strawberry banana juice (85 cents)
9 to exit
1
Please deposit 50 cents
50
Collect your item at the bottom and enjoy.
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

*** Welcome to Shelly's Juice Shop ***
To select an item, enter
1 for orange juice (50 cents)
2 for apple juice (65 cents)
3 for mango juice (80 cents)
4 for strawberry banana juice (85 cents)
9 to exit
9
```

QUICK REVIEW

1. A class is a collection of a fixed number of components.
2. Components of a class are called the members of the class.
3. The private members of a class are not directly accessible outside of the class.
4. The public members of a class are directly accessible outside of the class.
5. By default, all members of a class are private.
6. A member function of a class is called a constant function if its heading contains the reserved word const at the end. Moreover, a constant member function of a class cannot modify the member variables of the class.
7. In the Unified Modeling Language (UML) diagram of a class, the top box contains the name of the class. The middle box contains the member variables and their data types. The last box contains the member function name, parameter list, and the return type of the function. A + (plus) sign in front of a member name indicates that the member is a public member. A - (minus) sign preceding a member name indicates that the member a private member. The symbol # before the member name indicates that the member is a protected member.
8. Constructors guarantee that the member variables are initialized when an object is declared.
9. A class can have only one destructor, and the destructor has no parameters.
10. A data type that separates the logical properties from the implementation details is called an abstract data type (ADT).
11. Classes were specifically designed in C++ to handle ADTs.
12. A precondition is a statement specifying the condition(s) that must be true before the function is called.

PROGRAMMING EXERCISES

1. Define a class counterType to implement a counter. Your class must have a private data member counter of type int and functions to set counter to the value specified by the user, initialize counter to 0, retrieve the value of counter, and increment and decrement counter by one. The value of counter must be nonnegative.

2. Write the definition of a class, swimmingPool, to implement the properties of a swimming pool. Your class should have the instance variables to store the length (in feet), width (in feet), depth (in feet), the rate (in gallons per minute) at which the water is filling the pool, and the rate (in gallons per minute) at which the water is draining from the pool. Add appropriate constructors to initialize the instance variables. Also add member functions to do the following: determine the amount of water needed to fill an empty or partially filled pool, determine the time needed to completely or partially fill or empty the pool, and add or drain water for a specific amount of time.

3. Define the class bankAccount to implement the basic properties of a bank account. An object of this class should store the following data: Account holder's name (string), account number (int), account type (string, checking/saving), balance (double), and interest rate (double). (Store interest rate as a decimal number.) Add appropriate member functions to manipulate an object. Use a static member in the class to automatically assign account numbers. Also declare an array of 10 components of type bankAccount to process up to 10 customers and write a program to illustrate how to use your class

TOPIC 7
CLASSES (PART TWO)

LEARNING OUTCOMES:

After studying this topic you should be able to:

- Learn about templates
- Explore how to construct function templates and class templates
- Learn about virtual functions
- Learn about overloading
- Become familiar with the restrictions on operator overloading
- Examine the pointer this

7.1 TEMPLATES

Templates are a very powerful feature of C++. They allow you to write a single code segment for a set of related functions, called a function template, and for a set of related classes, called a class template. The syntax we use for templates is

```
template <class Type>
declaration;
```

in which Type is the name of a data type, built-in or user-defined, and declaration is either a function declaration or a class declaration. In C++, template is a reserved word. The word class in the heading refers to any user-defined type or built-in type. Type is referred to as a formal parameter to the template. (Note that in the first line, template , the keyword class can be replaced with the keyword typename.)

Using class templates, we can create a generic class listType, and the compiler can generate the appropriate source code for a specific implementation.

Class Templates

Class templates are called parameterized types because, based on the parameter type, a specific class is generated.

The following statements define listType to be a class template:

```
template<class elemType>
class listType
{
public:
    bool isEmpty() const;
      //Function to determine whether the list is empty.
      //Postcondition: Returns true if the list is empty,
      //               otherwise it returns false.

    bool isFull() const;
      //Function to determine whether the list is full.
      //Postcondition: Returns true if the list is full,
      //               otherwise it returns false.

    bool search(const elemType& searchItem) const;
      //Function to search the list for searchItem.
      //Postcondition: Returns true if searchItem
      //               is found in the list, and
      //               false otherwise.

    void insert(const elemType& newElement);
      //Function to insert newElement in the list.
      //Precondition: Prior to insertion, the list must
      //              not be full.
      //Postcondition: The list is the old list plus
      //               newElement.

    void remove(const elemType& removeElement);
      //Function to remove removeElement from the list.
      //Postcondition: If removeElement is found in the list,
      //               it is deleted from the list, and the
      //               list is the old list minus removeElement.
      //               If the list is empty, output the message
      //               "Cannot delete from the empty list."
```

```
    void destroyList();
      //Function to destroy the list.
      //Postcondition: length = 0;

    void printList();
      //Function to output the elements of the list.

    listType();
      //default constructor
      //Sets the length of the list to 0.
      //Postcondition: length = 0;
```

```
protected:
    elemType list[100];    //array to hold the list elements
    int length;            //variable to store the number of
                           //elements in the list
};
```

This definition of the class template **listType** is a generic definition and includes only the basic operations on a list. To derive a specific list from this list and to add or rewrite the operations, we declare the array containing the list elements and the length of the list as protected.

Next, we describe a specific list. Suppose that you want to create a list to process integer data. The statement

```
listType <int>intList;                    //Line 1
```

declares intList to be an object of listType. The protected member list is an array of 100 components, with each component being of type int. Similarly, the statement

```
listType<int> stringList;                 //Line 2
```

declares **stringList** to be an object of **listType.** The protected member list is an array of 100 components, with each component being of type **newString.**

In the statements in Lines 1 and 2, listType<int> and listType<newString> are referred to as *template instantiations or instantiations of the class template*

listType<elemType>, in which **elemType** is the class parameter in the template header. A template instantiation can be created with either a built-in or user-defined type.

The function members of a class template are considered function templates. Thus, when giving the definitions of the function members of a class template, we must follow the definition of the function template. For example, the definition of the member insert of the class listType is:

```
Template <class elemType>
void listType<eleType>::insert(elemType newElement)
{
        . . .
}
```

In the heading of the member function's definition, the name of the class is specified with the parameter elemType.

The statement in Line 1 declares intList to be a list of 100 components. When the compiler generates the code for intList, it replaces the word elemType with int in the definition of the class listType.

The template parameter in the definitions of the member functions (for example, elemType in the definition of insert) of the class listType is also replaced by int.

.

Header file and Implementation of a Class Template

Until now, we have placed the definition of the class (in the header file) and the definitions of the member functions (in the implementation file) in separate files. The object code was generated from the implementation file and linked with the user code. However, this mechanism of separating the class definition and the definitions of the member functions does not work with class templates.

Passing parameters to a function has an effect at run time, whereas passing a parameter to a class template has an effect at compile time. Because the actual parameter to a class is specified in the user code and because the compiler cannot instantiate a function template without the actual parameter to the template, we can no longer compile the implementation file independently of the user code

This problem has several possible solutions. We could put the class definition and the definitions of the function templates directly in the client code, or we could put the class definition and the definitions of the function templates together in the same header file. Another alternative is to put the class definition and the definitions of the functions in separate files (as usual) but include a directive to the implementation file at the end of the header file. In either case, the function definitions and the client code are compiled together. For illustrative purposes, we will put the class definition and the function definitions in the same header file.

Function Templates

To implement the function larger, we need to write four function definitions for the data type: one for int, one for char, one for double, and one for string. However, the body of each function is similar. C++ simplifies the process of overloading functions in cases such as this by providing function templates.

The syntax of the function template is

```
template <class Type>
function definition;
```

in which Type is referred to as a formal parameter of the template. It is used to specify the type of parameters to the function and the return type of the function and to declare variables within the function.

The statements:

```
Template
 Type larger(Type x, Type y)
{
        if (x >= y)
                return x;
        else
                return y;
 }
```

define a function template larger, which returns the larger of two items. In the function heading, the type of the formal parameters x and y is Type, which will be specified by the type of the actual parameters when the function is called.

The statement cout << larger(5, 6) << endl;

is a call to the function template larger. Because 5 and 6 are of type int, the data type int is substituted for Type, and the compiler generates the appropriate code.

Note that the function template larger will work only for those data types for which the operator >= has been defined. If we omit the body of the function in the function template definition, the function template, as usual, is the prototype.

EXAMPLE

The following program uses the function template larger to determine the larger of the two items.

```
#include <iostream>                                    //Line 1
#include "myString.h"                                  //Line 2

using namespace std;                                   //Line 3

template <class Type>                                  //Line 4
Type larger(Type x, Type y);                           //Line 5


int main()                                             //Line 6
{                                                      //Line 7
    cout << "Line 8: Larger of 5 and 6 = "
         << larger(5, 6) << endl;                      //Line 8
    cout << "Line 9: Larger of A and B = "
         << larger('A', 'B') << endl;                  //Line 9
    cout << "Line 10: Larger of 5.6 and 3.2 = "
         << larger(5.6, 3.2) << endl;                  //Line 10

    newString str1 = "Hello";                          //Line 11
    newString str2 = "Happy";                          //Line 12

    cout << "Line 13: Larger of " << str1 << " and "
         << str2 << " = " << larger(str1, str2)
         << endl;                                      //Line 13

    return 0;                                          //Line 14
}                                                      //Line 15

template <class Type>
Type larger(Type x, Type y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

Sample Run:

```
Line 8: Larger of 5 and 6 = 6
Line 9: Larger of A and B = B
Line 10: Larger of 5.6 and 3.2 = 5.6
Line 13: Larger of Hello and Happy = Hello
```

Virtual Functions

So, during program execution, how does C++ correct this problem of making the call to the appropriate function? C++ corrects this problem by providing the mechanism of virtual functions. The binding of virtual functions occurs at program execution time, not at compile time. This kind of binding is called **run-time binding**, **late binding**, or **dynamic binding**.

More formally, in run-time binding he compiler does not generate the code to call a specific function. Instead, it generates enough information to enable the run-time system to generate the specific code for the appropriate function call. Run-time binding is also known as **dynamic binding**.

In C++, virtual functions are declared using the reserved word virtual.

```
class petType
{
        public:
                virtual void print() const;             //virtual function
                petType(string n = "");
        private:
                 string name;
```

```
};

class dogType: public petType {
        public:
                void print() const;
                dogType(string n = "", string b = "");
         private:
                string breed;
}
```

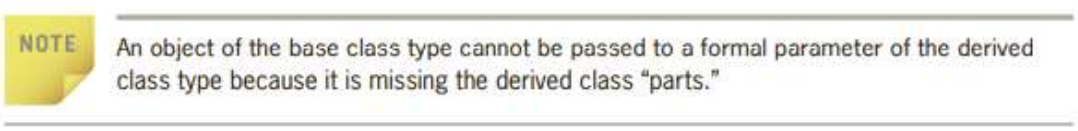
Note that we need to declare a virtual function only in the base class.

The definition of the **dogType** member function **print** is the same as before. Because we have placed a virtual function declaration in the base class, a base class object can use the derived class's definition. For example, if we execute the previous program with these modifications, the output is as follows:

Sample Run:

```
Name: Lucky
Name: Tommy, Breed: German Shepherd
*** Calling the function callPrint ***
Name: Lucky
Name: Tommy, Breed: German Shepherd
```

This output shows that for the statement in Line 11, the print function of dogType is executed (see the last two lines of the output).



Before closing this section, we discuss another issue related to virtual functions.

Suppose that the definition of the class petType is as before, and the definition of the class dogType is modified slightly as follows:

```
class dogType: public petType
{
        public:
                void print() const;
                void setBreed(string b = "");
               dogType(string n = "", string b = "");

       private:
               string breed;
};
```

Consider the following statements:

```
petType pet("Lucky");
dogType dog("Tommy", "German Shepherd");

pet = dog;
```

C++ allows this type of assignment, that is, the values of a derived class object can be copied into a base class object. (Note that the reverse statement, that is, dog = pet; is not allowed.) Now, because the object pet has only one data member (name), only the value of the data member name of dog is copied into the data member name of pet. This is called the **slicing problem**.

Abstract Classes and Pure Virtual Functions

The derived classes, in addition to inheriting the existing members of the base class, can add their own members and also redefine or override public and protected

member functions of the base class. The base class can contain functions that you would want each derived class to implement. There are many scenarios when it is desirable for a class to serve as a base class for a number of derived classes; however, the derived classes may contain certain functions that may not have meaningful definitions in the base class.

For example, you could have the definition of the class shape similar to the following:

```
class shape
{
public:
    virtual void draw();
      //Function to draw the shape.

    virtual void move(double x, double y);
      //Function to move the shape at the position (x, y).

    .
    .
    .
};
```

Because the definitions of the functions **draw** and **move** are specific to a particular shape, each derived class can provide an appropriate definition of these functions. Note that we have made the functions draw and move virtual to enforce run-time binding of these functions.

This definition of the class shape requires you to write the definitions of the functions **draw** and **move**. However, at this point, there is no shape to draw or move. Therefore, these function bodies have no code. One way to handle this is to make the body of these functions empty. This solution would work, but it has a drawback. Once we write the definitions of the functions of the class shape, then we could create an object of this class and invoke the empty draw and move functions. Because there is no shape to work with, we would like to prevent the user from creating objects of the class shape. It follows that we would like to do the following two things—to not include the definitions

of the functions draw and move and to prevent the user from creating objects of the class shape.

Because we do not want to include the definitions of the functions **draw** and **move** of the class shape, we must convert these functions to **pure virtual functions**. In this case, the prototypes of these functions are:

```
virtual void draw () = 0;
virtual void move (double x, double y) = 0;
```

Note the expression = 0 before the semicolon. Once you make these functions pure virtual functions in the class shape, you no longer need to provide the definitions of these functions for the class shape.

Once a class contains one or more pure virtual functions, then that class is called an abstract class. Thus, the abstract definition of the class shape is similar to the following:

```
class shape
{
public:
    virtual void draw() = 0;
      //Function to draw the shape. Note that this is a
      //pure virtual function.

    virtual void move(double x, double y) = 0;
      //Function to move the shape at the position (x, y).
      //Note that this is a pure virtual function.
    .
    .
    .

};
```

Because an abstract class (or its implementation file) does not contain the definitions of certain functions, it is not a complete class and you cannot create objects of that class.

Now suppose that we derive the class rectangle from the class shape. To make rectangle a nonabstract class so that we can create objects of this class, the class (or its implementation file) must provide the definitions of the pure virtual functions of its base class, which is the class shape

Note that in addition to the pure virtual functions, an abstract class can contain instance variables, constructors, and functions that are not pure virtual. However, the abstract class must provide the definitions of the constructor and functions that are not pure virtual. The following example further illustrates how abstract classes work.

EXAMPLE

```
class employeeType: public personType
{
public:
    virtual void print() const = 0;
      //Function to output employee's data.

    virtual double calculatePay() const = 0;
      //Function to calculate and return the wages.
      //Postcondition: Pay is calculated and returned

    void setId(long id);
      //Function to set the ID.
      //Postcondition: personId = id

    long getId() const;
      //Function to retrieve the id.
      //Postcondition: returns personId

    employeeType(string first = "", string last = "",
                 long id = 0);
```

```
private:
    long personId;    //stores the id
};
```

The definitions of the constructor and functions of the class employeeType that are not pure virtual are:

```
long employeeType::getId() const
{
    return personId;
}

employeeType::employeeType(string first, string last, long id)
              : personType(first, last)
{
      personId = id;
}

void employeeType::setId(long id)
{
    personId = id;
}
```

The definition of the class fullTimeEmployee is:

```
class fullTimeEmployee: public employeeType
{
public:
    void set(string first, string last, long id,
             double salary, double bonus);
      //Function to set the first name, last name,
      //id, and salary according to the parameters.
      //Postcondition: firstName = first; lastName = last;
      //               personId = id; empSalary = salary;
      //               empBonus = bonus

    void setSalary(double salary);
      //Function to set the salary.
      //Postcondition: empSalary = salary

    double getSalary();
      //Function to retrieve the salary.
      //Postcondition: returns empSalary

    void setBonus(double bonus);
      //Function to set the bonus.
      //Postcondition: empBonus = bonus

    double getBonus();
      //Function to retrieve the bonus.
      //Postcondition: returns empBonus

    void print() const;
      //Function to output the id, first name, last name
double calculatePay() const;
  //Function to calculate and return the wages.
  //Postcondition: Pay is calculated and returned

fullTimeEmployee(string first = "", string last = "",
                 long id = 0, double salary = 0,
                 double bonus = 0);


  private:
      double empSalary;
      double empBonus;
};
```

The definitions of the constructor and functions of the class fullTimeEmployee are:

```
void fullTimeEmployee::set(string first, string last,
                           long id,
                           double salary, double bonus)
{
    setName(first, last);
    setId(id);
    empSalary = salary;
    empBonus = bonus;
}

void fullTimeEmployee::setSalary(double salary)
{
    empSalary = salary;
}

double fullTimeEmployee::getSalary()
{
    return empSalary;
}


  void fullTimeEmployee::setBonus(double bonus)
  {
      empBonus = bonus;
  }

  double fullTimeEmployee::getBonus()
  {
      return empBonus;
  }

  void fullTimeEmployee::print() const
  {
      cout << "Id: " << getId() << endl;
      cout << "Name: ";
      personType::print();
      cout << endl;
      cout << "Wages: $" << calculatePay() << endl;
  }
```

```
double fullTimeEmployee::calculatePay() const
{
    return empSalary + empBonus;
}

      //constructor
fullTimeEmployee::fullTimeEmployee(string first, string last,
                                   long id, double salary,
                                   double bonus)
                : employeeType(first, last, id)
{
    empSalary = salary;
    empBonus = bonus;
}
```

The definition of the class partTimeEmployee is:

```
class partTimeEmployee: public employeeType
{
public:
    void set(string first, string last, long id, double rate,
             double hours);
      //Function to set the first name, last name, id,
      //payRate, and hoursWorked according to the
      //parameters.
      //Postcondition: firstName = first; lastName = last;
      //               personId = id;
      //               payRate = rate; hoursWorked = hours

    double calculatePay() const;
      //Function to calculate and return the wages.
      //Postcondition: Pay is calculated and returned

    void setPayRate(double rate);
      //Function to set the pay rate.
      //Postcondition: payRate = rate

    double getPayRate();
      //Function to retrieve the pay rate.
      //Postcondition: returns payRate

    void setHoursWorked(double hours);
      //Function to set the hours worked.
      //Postcondition: hoursWorked = hours

    //Postcondition: returns hoursWorked

  void print() const;
    //Function to output the Id, first name, last name,
    //and the wages.
    //Postcondition: Outputs
```

```
        //         Id:
        //         Name: firstName lastName
        //         Wages: $$$$.$$

    partTimeEmployee(string first = "", string last = "",
                     long id = 0,
                     double rate = 0, double hours = 0);
      //Constructor with parameters
      //Sets the first name, last name, payRate, and
      //hoursWorked according to the parameters. If
      //no value is specified, the default values are
      //assumed.
      //Postcondition: firstName = first; lastName = last;
      //               personId = id, payRate = rate;
      //               hoursWorked = hours

private:
    double payRate;      //stores the pay rate
    double hoursWorked; //stores the hours worked
};
```

The definitions of the constructor and functions of the class partTimeEmployee are:

```
void partTimeEmployee::set(string first, string last, long id,
                           double rate, double hours)
{
    setName(first, last);
    setId(id);
    payRate = rate;
    hoursWorked = hours;
}

void partTimeEmployee::setPayRate(double rate)
{
    payRate = rate;
}

double partTimeEmployee::getPayRate()
{
    return payRate;
}

void partTimeEmployee::setHoursWorked(double hours)
{
    hoursWorked = hours;
}

double partTimeEmployee::getHoursWorked()
{
    return hoursWorked;
}
```

```
void partTimeEmployee::print() const
{
    cout << "Id: " << getId() << endl;
    cout << "Name: ";
    personType::print();
    cout << endl;
    cout << "Wages: $" << calculatePay() << endl;
}

double partTimeEmployee::calculatePay() const
{
    return (payRate * hoursWorked);
}

      //constructor
partTimeEmployee::partTimeEmployee(string first, string last,
                                   long id,
                                   double rate, double hours)
                : employeeType(first, last, id)
{
    payRate = rate;
    hoursWorked = hours;
}
```

The following function **main** tests these classes:

```
#include <iostream>
#include "partTimeEmployee.h"
#include "fullTimeEmployee.h"

int main()
{
    fullTimeEmployee newEmp("John", "Smith", 75, 56000, 5700);
    partTimeEmployee tempEmp("Andy", "Turner", 275, 15.50, 57);

    newEmp.print();
    cout << endl;
    tempEmp.print();

    return 0;
}
```

Sample Run:

```
Id: 75
Name: John Smith
Wages: $61700

Id: 275
Name: Andy Turner
Wages: $883.5
```

OVERLOADING

Recall that the only built-in operations on classes are the assignment operator and the member selection operator. Therefore, other operators cannot be directly applied to class objects by default. However, C++ allows the programmer to extend the definitions of operators such as relational operators, arithmetic operators, the insertion operator for data output, and the extraction operator for data input—so they can be applied to classes. In C++ terminology, this is called operator overloading.

OPERATOR OVERLOADING

Recall how the arithmetic operator / works. If both operands of / are integers, the result is an integer; otherwise, the result is a floating-point number. This means that the / operator has one definition when both operands are integers and another when an operand is a floating-point number. Which definition is used depends on the data types of the operand it is used with. Similarly, the stream insertion operator, <>, are overloaded. The operator >> is used as both a stream extraction operator and a right shift operator. The operator << is used as both a stream insertion operator and a left shift operator. These are examples of operator overloading. (Note that the operators << and >> have also been overloaded for various data types, such as int, double, and string.)
Other examples of overloaded operators are + and -. The results of + and - are different for integer arithmetic, floating-point arithmetic, and pointer arithmetic.

C++ allows the user to overload most of the operators so that the operators can work effectively in a specific application. It does not allow the user to create new operators. Most of the existing operators can be overloaded to manipulate class objects. In order to overload an operator, you must write function(s) (that is, the header and the body) to define what operation the overloaded operator indicates should be performed. The

name of the function that overloads an operator is the reserved word operator followed by the operator to be overloaded. For example, the name of the function to overload the operator >= is: operator>= Operator function: The function that overloads an operator.

Syntax for Operator Functions

The result of an operation is a value. Therefore, the operator function is a value returning function. The syntax of the heading for an operator function is:

```
returnType operator operatorSymbol(formal parameter list)
```

In C++, operator is a reserved word. Recall that the only built-in operations on classes are assignment (=) and member selection. To use other operators on class objects, they must be explicitly overloaded. Operator overloading provides the same concise expressions for user-defined data types as it does for built-in data types.

Overloading an Operator: Rules

1. When overloading an operator, keep the following in mind:
2. You cannot change the precedence of an operator.
3. The associativity cannot be changed. (For example, the associativity of the arithmetic operator addition is from left to right, and it cannot be changed.)
4. Default parameters cannot be used with an overloaded operator.
5. You cannot change the number of parameters an operator takes.
6. You cannot create new operators. Only existing operators can be overloaded.
7. The meaning of how an operator works with built-in types, such as int, remains the same. That is, you cannot redefine how operators work with built-in data types.
8. Operators can be overloaded either for objects of the user-defined types, or for a combination of objects of the user-defined type and objects of the built-in type

PROGRAMMING EXAMPLE: Complex Numbers

A number of the form $a + ib$, in which $i^2 = -1$ and a and b are real numbers, is called a **complex number**. We call a the real part and b the imaginary part of $a + ib$. Complex numbers can also be represented as ordered pairs (a, b). The addition and multiplication of complex numbers are defined by the following rules:

$$(a + ib) + (c + id) = (a + c) + i(b + d)$$

$$(a + ib) * (c + id) = (ac - bd) + i(ad + bc)$$

Using the ordered pair notation, these rules are written as:

$$(a,b) + (c,d) = ((a + c), (b + d))$$

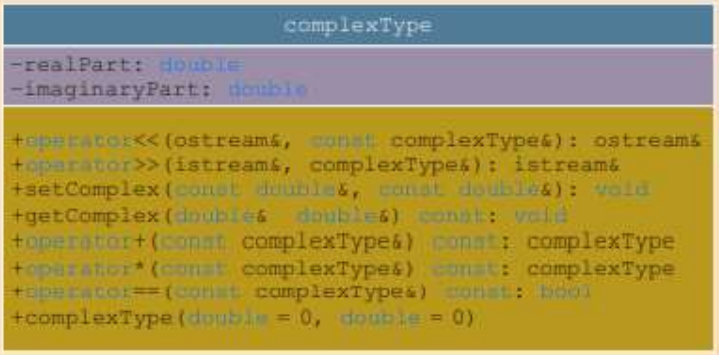
$$(a,b) * (c,d) = ((ac - bd), (ad + bc))$$

C++ has no built-in data type that allows us to manipulate complex numbers. In this example, we will construct a data type, `complexType`, that can be used to process complex numbers. We will overload the stream insertion and stream extraction operators for easy input and output. We will also overload the operators + and * to perform addition and multiplication of complex numbers. If x and y are complex numbers, we can evaluate expressions such as $x + y$ and $x * y$.

The definition of the class complexType is:

```
#include <iostream>
using namespace std;

class complexType
{
      //Overload the stream insertion and extraction operators
    friend ostream& operator<<(ostream&, const complexType&);
    friend istream& operator>>(istream&, complexType&);

public:
    void setComplex(const double& real, const double& imag);
      //Function to set the complex numbers according to
      //the parameters.
      //Postcondition: realPart = real; imaginaryPart = imag;

    void getComplex(double& real, double& imag) const;
      //Function to retrieve the complex number.
      //Postcondition: real = realPart; imag = imaginaryPart;

    complexType(double real = 0, double imag = 0);
      //Constructor
      //Initializes the complex number according to
       //the parameters.
       //Postcondition: realPart = real; imaginaryPart = imag;

     complexType operator+
                     (const complexType& otherComplex) const;
       //Overload the operator +

     complexType operator*
                     (const complexType& otherComplex) const;
       //Overload the operator *

     bool operator== (const complexType& otherComplex) const;
       //Overload the operator ==

 private:
     double realPart;         //variable to store the real part
     double imaginaryPart;    //variable to store the
                              //imaginary part
 };
```

Figure below shows a UML class diagram of the class complexType.



Next, we write the definitions of the functions to implement various operations of the `class` **`complexType`**.

The definitions of most of these functions are straightforward. We will discuss only the definitions of the functions to overload the stream insertion operator, `<<`, and the stream extraction operator, `>>`.

To output a complex number in the form:

`(a, b)`

in which `a` is the real part and `b` is the imaginary part, the algorithm is as follows:

a. Output the left parenthesis, `(`.
b. Output the real part.
c. Output the comma and a space.
d. Output the imaginary part.
e. Output the right parenthesis, `)`.

Therefore, the definition of the function operator<< is as follows:

```
ostream& operator<<(ostream& osObject,
                    const complexType& complex)

{
    osObject << "(";                        //Step a
    osObject << complex.realPart;           //Step b
    osObject << ", ";                       //Step c
    osObject << complex.imaginaryPart;      //Step d
    osObject << ")";                        //Step e

    return osObject;          //return the ostream object
}
```

Next, we discuss the definition of the function to overload the stream extraction operator, >>

The input is of the form:

```
(3, 5)
```

In this input, the real part of the complex number is 3, and the imaginary part is 5. The algorithm to read this complex number is as follows:

a. Read and discard the left parenthesis.
b. Read and store the real part.
c. Read and discard the comma.
d. Read and store the imaginary part.
e. Read and discard the right parenthesis.

Following these steps, the definition of the function operator>> is as follows:

```
istream& operator>>(istream& isObject, complexType& complex)
{
    char ch;

    isObject >> ch;                    //Step a
    isObject >> complex.realPart;      //Step b
    isObject >> ch;                    //Step c
```

7.4.1 Overloading the Stream Extractoin Operator(>>)

The general syntax to overload the stream extraction operator, >>, for a class is described next**. Function Prototype** (to be included in the definition of the class):

```
friend istream& operator>>(istream&, className&);
```

Function Definition:

```
istream& operator>>(istream& isObject, className& cObject)
{
      //local declaration, if any
      //Read the data into cObject.
      //isObject >> . . .

      //Return the stream object.
    return isObject;
}
```

In this function definition: ? Both parameters are reference parameters.

- The first parameter—that is, isObject—is a reference to an istream object.
- The second parameter is usually a reference to a particular class. The data read will be stored in the object.
- The function return type is a reference to an istream object.

EXAMPLE

```
#include <iostream>

using namespace std;

class rectangleType
{
      //Overload the stream insertion and extraction operators
    friend ostream& operator<< (ostream&, const rectangleType &);
    friend istream& operator>> (istream&, rectangleType &);

public:
    void setDimension(double l, double w);
    double getLength() const;
    double getWidth() const;
    double area() const;
    double perimeter() const;
    void print() const;

  rectangleType operator+(const rectangleType&) const;
    //Overload the operator +
  rectangleType operator*(const rectangleType&) const;
    //Overload the operator *

  bool operator==(const rectangleType&) const;
    //Overload the operator ==
```

```
    bool operator!=(const rectangleType&) const;
      //Overload the operator !=

    rectangleType();
    rectangleType(double l, double w);

private:
    double length;
    double width;
};
```

Notice that we have removed the member function print because we are overloading the stream insertion operator <<.

```
ostream& operator<< (ostream& osObject,
                     const rectangleType& rectangle)
{
    osObject << "Length = "  << rectangle.length
             << "; Width = " << rectangle.width;

    return osObject;
}

istream& operator>> (istream& isObject,
                     rectangleType& rectangle)
{
    isObject >> rectangle.length >> rectangle.width;

    return isObject;
}
```

Consider the following program. (We assume that the definition of the class rectangleType is in the header file rectangleType.h.)

```
#include <iostream>                                          //Line 1

#include "rectangleType.h"                                   //Line 2

using namespace std;                                         //Line 3

int main()                                                   //Line 4
{                                                            //Line 5
    rectangleType myRectangle(23, 45);                       //Line 6
    rectangleType yourRectangle;                             //Line 7

    cout << "Line 8: myRectangle: " << myRectangle
         << endl;                                            //Line 8

    cout << "Line 9: Enter the length and width "
         << "of a rectangle: ";                              //Line 9


    cin >> yourRectangle;                                    //Line 10
    cout << endl;                                            //Line 11

    cout << "Line 12: yourRectangle: "
         << yourRectangle << endl;                           //Line 12

    cout << "Line 13: myRectangle + yourRectangle: "
         << myRectangle + yourRectangle << endl;             //Line 13
    cout << "Line 14: myRectangle * yourRectangle: "
         << myRectangle * yourRectangle << endl;             //Line 14

    return 0;                                                //Line 15
}                                                            //Line 16
```

Sample Run: In this sample run, the user input is shaded.

```
Line 8: myRectangle: Length = 23; Width = 45
Line 9: Enter the length and width of a rectangle: 28 17

Line 12: yourRectangle: Length = 28; Width = 17
Line 13: myRectangle + yourRectangle: Length = 51; Width = 62
Line 14: myRectangle * yourRectangle: Length = 644; Width = 765
```

Pointer this

A member function of a class can (directly) access the member variables of a given object of that class. Sometimes, it is necessary for a member function to refer to the object as a whole, rather than the object's individual member variables. How do you refer to the object as a whole (that is, as a single unit) in the definition of the member function, especially when the object is not passed as a parameter? Every object of a class maintains a (hidden) pointer to itself, and the name of this pointer is this. In C++, this is a reserved word.

The pointer this (in a member function) is available for you to use. When an object invokes a member function, the member function references the pointer this of the object. For example, suppose that test is a class and has a member function called one.

Further suppose that the definition of one looks like the following:

```
test test::one()
{
        . . .
        return *this;
 }
```

If x and y are objects of type test, then the statement:

```
        y = x.one();
```

Copies the value of object x into object y. That is, the member variables of x are copied into the corresponding member variables of y. When object x invokes function one, the pointer this in the definition of member function one refers to object x, so this means the address of x and *this means the contents of x. On the other hand, in the statement:

```
x = y.one();
```

the pointer this in the definition of member function one refers to object y, and so this means the address of y and *this means the contents of y. So the statement copies

the contents of object y into object x. The following example illustrates how the pointer this works.

QUICK REVIEW

1. An operator that has different meanings with different data types is said to be overloaded.
2. In C++, template is a reserved word.
3. Using templates, you can write a single code segment for a set of related functions—called the function template.
4. Using templates, you can write a single code segment for a set of related classes—called the class template
5. Class templates are called parameterized types
6. The parameter Type is mentioned in every class header and member function definition
7. To overload the pre-increment (++) operator for a class if the operator function is a member of that class, it must have no parameters. Similarly, to overload the pre-decrement (--) operator for a class if the operator function is a member of that class, it must have no parameters.
8. To overload the post-increment (++) operator for a class if the operator function is a member of that class, it must have one parameter, of type int. The user does not specify any value for the parameter. The dummy parameter in the function heading helps the compiler generate the correct code. The post-decrement operator has similar conventions.
9. The operator functions that overload the stream insertion operator, <<or>>, for a class must be friend functions of that class.
10. Using templates, you can write a single code segment for a set of related classes—called the class template.

PROGRAMMING EXERCISES

1. Write a program that uses C++ random number generators to generate 25 real numbers between 10 and 100.

TOPIC 8: ADVANCED CONCEPTS

LEARNING OUTCOMES:
After studying this topic you should be able to:

- Learn what a stream is and examine input and output streams
- Learn what an exception is
- Learn how to handle exceptions within a program
- Learn how a try/catch block is used to handle exceptions
- Learn how to throw an exception
- Become familiar with C++ exception classes and how to use them in a program

FILE INPUT/OUTPUT

The previous sections discussed in some detail how to get input from the keyboard (standard input device) and send output to the screen (standard output device). However, getting input from the keyboard and sending output to the screen have several limitations. Inputting data in a program from the keyboard is comfortable as long as the amount of input is very small. Sending output to the screen works well if the amount of data is small (no larger than the size of the screen) and you do not want to distribute the output in a printed format to others.

This section discusses how to obtain data from other input devices, such as a flash drive (that is, secondary storage), and how to save the output to a flash drive. C++ allows a program to get data directly from and save output directly to secondary storage. A program can use the file I/O and read data from or write data to a file. Formally, a file is defined as follows:

File: An area in secondary storage used to hold information.

The standard I/O header file, iostream, contains data types and variables that are used only for input from the standard input device and output to the standard output device. In addition, C++ provides a header file called fstream, which is used for file I/O. Among other things, the fstream header file contains the definitions of two data types: ifstream, which means input file stream and is similar to istream, and ofstream, which means output file stream and is similar to ostream.

File I/O is a five-step process:

1. Include the header file fstream in the program.
2. Declare file stream variables.
3. Associate the file stream variables with the I/O sources.
4. Use the file stream variables with >>, <
5.Close the files.

The stream member function open is used to open files. The syntax for opening a file is:

```
fileStreamVariable.open(sourceName);
```

Here, fileStreamVariable is a file stream variable, and sourceName is the name of the I/O file.

NOTE

Suppose that a program reads data from a file. Because different computers have drives labeled differently, for simplicity, throughout the book, we assume that the file containing the data and the program reading data from the file are in the same directory (subdirectory).

NOTE

We typically use `.dat`, `.out`, or `.txt` as an extension for the input and output files and use Notepad, Wordpad, or TextPad to create and open these files. You can also use your IDE's editor, if any, to create `.txt` (text) files. (To be absolutely sure about it, check your IDE's documentation.)

Step 1

Requires that the header file fstream be included in the program. The following statement accomplishes this task: #include

Step 2

Requires you to declare file stream variables. Consider the following statements:

ifstream inData;

ofstream outData;

Step 3

Requires you to associate file stream variables with the I/O sources. This step is called opening the files. The stream member function open is used to open files.

Suppose you include the declaration from Step 2 in a program. Further suppose that the input data is stored in a file called prog.dat. The following statements associate inData with prog.dat and outData with prog.out. That is, the file prog.dat is opened for inputting data, and the file prog.out is opened for outputting data.

```
inData.open("prog.dat");                    //open the input file; Line 1
outData.open("prog.out");                   //open the output file; Line 2
```

Step 4

You use the file stream variables with >>, <> or << with file stream variables is exactly the same as the syntax for using cin and cout. Instead of using cin and cout, however, you use the file stream variable names that were declared. For example, the statement:

inData >> payRate;

reads the data from the file prog.dat and stores it in the variable payRate. The statement:

outData << "The paycheck is: $" << pay << endl;

stores the output—The paycheck is: $565.78—in the file prog.out. This statement assumes that the pay was calculated as 56

Once the I/O is complete, Step 5 requires closing the files. Closing a file means that the file stream variables are disassociated from the storage area and are freed. Once these variables are freed, they can be reused for other file I/O. Moreover, closing an output file ensures that the entire output is sent to the file; that is, the buffer is emptied. You close files by using the stream function close. For example, assuming the program includes the declarations listed in Steps 2 and 3, the statements for closing the files are:

```
inData.close();
outData.close();
```

In skeleton form, a program that uses file I/O usually takes the following form:

```
#include <fstream>

//Add additional header files you use

using namespace std;

int main()
{
        //Declare file stream variables such as the following
    ifstream inData;
    ofstream outData;
    .
    .
    .

        //Open the files
    inData.open("prog.dat");  //open the input file
    outData.open("prog.out"); //open the output file

        //Code for data manipulation

        //Close files
    inData.close();
    outData.close();

    return 0;
}
```

Opening a file associates a file stream variable declared in the program with a physical file at the source, such as a flash drive. In the case of an input file, the file must exist before the open statement executes. If the file does not exist, the open statement fails and the input stream enters the fail state. An output file does not have to exist before it is opened; if the output file does not exist, the computer prepares an empty file for output. If the designated output file already exists, by default, the old contents are erased when the file is opened.

NOTE

To add the output at the end of an existing file, you can use the option **ios::app** as follows. Suppose that **outData** is declared as before and you want to add the output at the end of the existing file, say, **firstProg.out**. The statement to open this file is:

```
outData.open("firstProg.out", ios::app);
```

If the file **firstProg.out** does not exist, then the system creates an empty file.

PROGRAMMING EXAMPLE: Student Grade

Write a program that reads a student name followed by five test scores. The program should output the student name, the five test scores, and the average test score. Output the average test score with two decimal places.

The data to be read is stored in a file called `test.txt`. The output should be stored in a file called `testavg.out`.

Input A file containing the student name and the five test scores. A sample input is:

```
Andrew Miller 87.50 89 65.75 37 98.50
```

Output The student name, the five test scores, and the average of the five test scores, saved to a file.

PROBLEM ANALYSIS AND ALGORITHM DESIGN

To find the average of the five test scores, you add the five test scores and divide the sum by 5. The input data is in the following form: the student name followed by the five test scores. Therefore, you must read the student name first and then read the five test scores. This problem analysis translates into the following algorithm:

1. Read the student name and the five test scores.
2. Output the student name and the five test scores.
3. Calculate the average.
4. Output the average.

You output the average test score in the fixed decimal format with two decimal places.

COMPLETE PROGRAM LISTING

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>

using namespace std;

int main()
{
        //Declare variables;                              Step 1
    ifstream inFile; //input file stream variable
    ofstream outFile; //output file stream variable

    double test1, test2, test3, test4, test5;
    double average;

    string firstName;
    string lastName;

    inFile.open("test.txt");                             //Step 2
    outFile.open("testavg.out");                         //Step 3

    outFile << fixed << showpoint;                       //Step 4
    outFile << setprecision(2);                          //Step 4
```

```
    cout << "Processing data" << endl;

    inFile >> firstName >> lastName;                    //Step 5
    outFile << "Student name: " << firstName
            << " " << lastName << endl;                 //Step 6

    inFile >> test1 >> test2 >> test3
           >> test4 >> test5;                           //Step 7
    outFile << "Test scores: " << setw(6) << test1
            << setw(6) << test2 << setw(6) << test3
            << setw(6) << test4 << setw(6) << test5
            << endl;                                    //Step 8

    average = (test1 + test2 + test3 + test4
              + test5) / 5.0;                           //Step 9

    outFile << "Average test score: " << setw(6)
            << average << endl;                         //Step 10

    inFile.close();                                     //Step 11
    outFile.close();                                    //Step 11

    return 0;
}
```

Sample Run:

Input File (contents of the file `test.txt`):

```
Andrew Miller 87.50 89 65.75 37 98.50
```

Output File (contents of the file `testavg.out`):

```
Student name: Andrew Miller
Test scores: 87.50 89.00 65.75 37.00 98.50
Average test score: 75.55
```

NOTE

The preceding program uses five variables—`test1`, `test2`, `test3`, `test4`, and `test5`—to read the five test scores and then find the average test score. The website accompanying this book contains a modified version of this program that uses only one variable, `testScore`, to read the test scores and another variable, `sum`, to find the sum of the test scores. The program is named `Ch3_AverageTestScoreVersion2.cpp`.

8.2 EXCEPTION HANDLING

An exception is an occurrence of an undesirable situation that can be detected during program execution. For example, division by zero is an exception. Similarly, trying to open an input file that does not exist is an exception, as is an array index that goes out of bounds.

The code to handle exceptions depends on the type of application you develop. One common way to provide exception-handling code is to add exception-handling code at the point where an error can occur. This technique allows the programmer reading the code to see the exception-handling code together with the actual code and to determine whether the error-checking code is properly implemented. The disadvantage of this approach is that the program can become cluttered with exception-handling code, which can make understanding and maintaining the program difficult. This can distract the programmer from ensuring that the program functions correctly.

EXAMPLE

```
// Division by zero.

#include <iostream>                                    //Line 1

using namespace std;                                   //Line 2

int main()                                             //Line 3
{                                                      //Line 4
    int dividend, divisor, quotient;                   //Line 5

    cout << "Line 6: Enter the dividend: ";            //Line 6
    cin >> dividend;                                   //Line 7
    cout << endl;                                      //Line 8

    cout << "Line 9: Enter the divisor: ";             //Line 9
    cin >> divisor;                                    //Line 10
    cout << endl;                                      //Line 11

    quotient = dividend / divisor;                     //Line 12
    cout << "Line 13: Quotient = " << quotient
         << endl;                                      //Line 13

    return 0;                                          //Line 14
}                                                      //Line 15
```

Sample Run 1:

```
Line 6: Enter the dividend: 27

Line 9: Enter the divisor: 4

Line 13: Quotient = 6
```

Sample Run 2:

```
Line 6: Enter the dividend: 15

Line 9: Enter the divisor: 0

CPP_Proj.exe has stopped working
A problem caused the program to stop working correctly.
Windows will close the program and notify you if a solution is
available.
```

In Sample Run 1, the value of divisor is nonzero, so no exception occurs. The program calculates and outputs the quotient and terminates normally. In Sample Run 2, the value entered for divisor is 0. The statement in Line 12 divides dividend by the divisor.

However, the program does not check whether divisor is 0 before dividing dividend by divisor. So the program crashes with an error message shown. Notice that the error message is platform independent, that is, IDE dependent. Some IDEs might not give this error message and might simply hang.

Next, consider Example below. This is the same program as in Example above, except that in Line 8, the program checks whether divisor is zero.

EXAMPLE

```
#include <iostream>                                          //Line 1

using namespace std;                                        //Line 2

int main()                                                  //Line 3
{                                                           //Line 4
    int dividend, divisor, quotient;                        //Line 5

    cout << "Line 6: Enter the dividend: ";                 //Line 6
    cin >> dividend;                                        //Line 7
    cout << endl;                                           //Line 8

    cout << "Line 9: Enter the divisor: ";                  //Line 9
    cin >> divisor;                                         //Line 10
    cout << endl;                                           //Line 11

    if (divisor != 0)                                       //Line 12
    {                                                       //Line 13
        quotient = dividend / divisor;                      //Line 14
        cout << "Line 15: Quotient = " << quotient
             << endl;                                       //Line 15
    }                                                       //Line 16
    else                                                    //Line 17
        cout << "Line 18: Cannot divide by zero."
             << endl;                                       //Line 18

    return 0;                                               //Line 19
}                                                           //Line 20
```

Sample Run 1:

```
Line 6: Enter the dividend: 17
Line 9: Enter the divisor: 5
Line 15: Quotient = 3
```

Sample Run 2:

```
Line 6: Enter the dividend: 45
Line 9: Enter the divisor: 0
Line 18: Cannot divide by zero.
```

In Sample Run 1, the value of divisor is nonzero, so no exception occurs. The program calculates and outputs the quotient and terminates normally.

In Sample Run 2, the value entered for divisor is 0. In Line 12, the program checks whether divisor is 0. Because divisor is 0,

try/catch Block

The statements that may generate an exception are placed in a try block. The try block also contains statements that should not be executed if an exception occurs. The try block is followed by one or more catch blocks. A catch block specifies the type of exception it can catch and contains an exception handler.

The general syntax of the try/catch block is:

```
try
{
    //statements
}
catch (dataType1 identifier)
{
    //exception-handling code
}
.
.
.
catch (dataTypen identifier)
{
    //exception-handling code
}
.
.
.
catch (...)
{
    //exception-handling code
}
```

Suppose there is a statement that can generate an exception, for example, division by 0. Usually, before executing such a statement, we check whether certain conditions are met. For example, before performing the division, we check whether the divisor is nonzero. If the conditions are not met, we typically generate an exception, which in C++ terminology is called throwing an exception.

Consider the following catch block:

```
catch (int x) {
            //exception-handling code
    } In this catch block:
```

- The identifier x acts as a parameter. In fact, it is called a catch block parameter.

- The data type int specifies that this catch block can catch an exception of type int.
- A catch block can have at most one catch block parameter.

Essentially, the catch block parameter becomes a placeholder for the value thrown. In this case, x becomes a placeholder for any thrown value that is of type int. In other words, if the thrown value is caught by this catch block, then the thrown value is stored in the catch block parameter. This way, if the exception-handling code wants to do something with that value, it can be accessed via the catch block parameter.

Throwing An Exception

In order for an exception to occur in a try block and be caught by a catch block, the exception must be thrown in the try block. The general syntax to throw an exception is

```
throw expression;
```

in which expression is a constant value, variable, or object. The object being thrown can be either a specific object or an anonymous object. It follows that in C++, *an exception is a value*.

In C++, throw is a reserved word

EXAMPLE

```
#include <iostream>                                          //Line 1

using namespace std;                                        //Line 2

int main()                                                  //Line 3
{                                                           //Line 4
    int dividend, divisor, quotient;                        //Line 5

    try                                                     //Line 6
    {                                                       //Line 7
        cout << "Line 8: Enter the dividend: ";             //Line 8
        cin >> dividend;                                    //Line 9
        cout << endl;                                       //Line 10

        cout << "Line 11: Enter the divisor: ";             //Line 11
        cin >> divisor;                                     //Line 12
        cout << endl;                                       //Line 13

        if (divisor == 0)                                   //Line 14
            throw 0;                                        //Line 15

        quotient = dividend / divisor;                      //Line 16


        cout << "Line 17: Quotient = " << quotient
             << endl;                                       //Line 17
    }                                                       //Line 18
    catch (int)                                             //Line 19
    {                                                       //Line 20
        cout << "Line 21: Division by 0." << endl;          //Line 21
    }                                                       //Line 22

    return 0;                                               //Line 23
}                                                           //Line 24
```

Sample Run 1: In this sample run, the user input is shaded.

```
Line 8: Enter the dividend: 26
Line 11: Enter the divisor: 7
Line 17: Quotient = 3
```

Sample Run 2: In this sample run, the user input is shaded.

```
Line 8: Enter the dividend: 52
Line 11: Enter the divisor: 0
Line 21: Division by 0.
```

This program works as follows. The statement in Line 5 declares the int variables dividend, divisor, and quotient. The try block starts at Line 6. The statement in Line 8 prompts the user to enter the value for the dividend; the statement in Line 9 stores this number in the variable dividend.

The statement in Line 11 prompts the user to enter the value for the divisor, and the statement in Line 12 stores this number in the variable divisor. The statement in Line 14 checks whether the value of divisor is 0.

If the value of divisor is 0, the statement in Line 15 throws the constant value 0. The statement in Line 16 calculates the quotient and stores it in quotient. The statement in Line 17 outputs the value of *quotient.*

The catch block starts in Line 19 and catches an exception of type int.

In Sample Run 1, the program does not throw any exception.

In Sample Run 2, the entered value of divisor is 0. Therefore, the statement in Line 15 throws 0, which is caught by the catch block starting in Line 19. The statement in Line 21 outputs the appropriate message

Throwing An Exception

C++ provides support to handle exceptions via a hierarchy of classes. The class exception is the base of the classes designed to handle exceptions. Among others, this class contains the function what. The function what returns a string containing an appropriate message. All derived classes of the class exception override the function what to issue their own error messages.

Two classes are immediately derived from the class exception: logic_error and runtime_error. Both of these classes are defined in the header file stdexcept.

To deal with logical errors in a program, such as a string subscript out of range or an invalid argument to a function call, several classes are derived from the class logic_error. For example, the class invalid_argument is designed to deal with illegal arguments used in a function call. The class out_of_range deals with the string subscript out of range error. If a length greater than the maximum allowed for a string object is used, the class length_error deals with this error. For example, recall that every string object has a maximum length. If a length

larger than the maximum length allowed for a string is used, then the length_error exception is generated. If the operator new cannot allocate memory space, this operator throws a bad_alloc exception.

The class runtime_error is designed to deal with errors that can be detected only during program execution. For example, to deal with arithmetic overflow and underflow exceptions, the classes overflow_error and underflow_error are derived from the class runtime_error.

The program in Example below shows how to handle the exceptions out_of_range and length_error. Notice that in this program, these exceptions are thrown by the string functions substr and the string concatenation operator +. Because the exceptions are thrown by these functions, we do not include any throw statement in the try block.

EXAMPLE

```
#include <iostream>                                          //Line 1
#include <string>                                            //Line 2

using namespace std;                                         //Line 3

int main()                                                   //Line 4
{                                                            //Line 5
    string sentence;                                         //Line 6
    string str1, str2, str3;                                 //Line 7

    try                                                      //Line 8
    {                                                        //Line 9
        sentence = "Testing string exceptions!";             //Line 10
        cout << "Line 11: sentence = " << sentence
             << endl;                                        //Line 11
```

```
        cout << "Line 12: sentence.length() = "
             << static_cast<int>(sentence.length())
             << endl;                                    //Line 12

        str1 = sentence.substr(8, 20);                   //Line 13
        cout << "Line 14: str1 = " << str1 << endl;      //Line 14

        str2 = sentence.substr(28, 10);                  //Line 15
        cout << "Line 16: str2 = " << str2 << endl;      //Line 16

        str3 = "Exception handling. " + sentence;        //Line 17
        cout << "Line 18: str3 = " << str3 << endl;      //Line 18

    }                                                    //Line 19
    catch (out_of_range re)                              //Line 20
    {                                                    //Line 21
        cout << "Line 22: In the out_of_range catch"
             << " block: " << re.what() << endl;         //Line 22
    }                                                    //Line 23
    catch (length_error le)                              //Line 24
    {                                                    //Line 25
        cout << "Line 26: In the length_error catch"
             << " block: " << le.what() << endl;         //Line 26
    }                                                    //Line 27

    return 0;                                            //Line 28
}                                                        //Line 29
```

Sample Run:

```
Line 11: sentence = Testing string exceptions!
Line 12: sentence.length() = 26
Line 14: str1 = string exceptions!
Line 22: In the out_of_range catch block: invalid string position
```

In this program, the statement in Line 13 uses the function **substr** to determine a substring in the string object sentence. The length of the string sentence is 26. Because the starting position of the substring is 8, which is less than 26, no exception is thrown. However, in the statement in Line 15, the starting position of the substring is 28, which is greater than 26 (the length of sentence). Therefore, the function substr throws an out_of_range exception, which is caught and processed by the catch block in Line 20. Notice that in the statement in Line 22, the object re uses the function what to return the error message, invalid string position.

QUICK REVIEW

1. An exception is an occurrence of an undesirable situation that can be detected during program execution.
2. The try/catch block is used to handle exceptions within a program.
3. Statements that may generate an exception are placed in a try block. The try block also contains statements that should not be executed if an exception occurs.
4. If no exceptions are thrown in a try block, all catch blocks associated with that try block are ignored and program execution resumes after the last catch block.
5. If an exception is thrown in a try block, the remaining statements in the try block are ignored. The program searches the catch blocks, in the order they appear after the try block, and looks for an appropriate exception handler. If the type of the thrown exception matches the parameter type in one of the catch blocks, then the code in that catch block executes and the remaining catch blocks after this catch block are ignored.
6. In order for an exception to occur in a try block and be caught by a catch block, the exception must be thrown in the try block.
7. To close a file as indicated by the ifstream variable inFile, you use the statement inFile.close();. To close a file as indicated by the ofstream variable outFile, you use the statement outFile.close();