Chapter 2

Basic Elements of C++

A Guide to this Instructor's Manual:

We have designed this Instructor's Manual to supplement and enhance your teaching experience through classroom activities and a cohesive chapter summary.

This document is organized chronologically, using the same headings that you see in the textbook. Under the headings you will find: lecture notes that summarize the section, Teacher Tips, Classroom Activities, and Lab Activities. Pay special attention to teaching tips and activities geared towards quizzing your students and enhancing their critical thinking skills.

In addition to this Instructor's Manual, our Instructor's Resources also contain PowerPoint Presentations, Test Banks, and other supplements to aid in your teaching experience.

At a Glance

Instructor's Manual Table of Contents

Lecture Notes

Overview

Chapter 2 teaches your students the basics of C++. Learning a programming language is similar to learning to be a chef or learning to play a musical instrument. All three require direct interaction with the tools; in other words, you cannot become proficient by simply reading books on the topics. In this chapter, your students will begin acquiring a fundamental knowledge of C++ by learning about data types, functions, identifiers, assignment statements, arithmetic operations, and input/output operations. They will then write and test programs using these concepts to verify their knowledge of the material.

Objectives

In this chapter, the student will:

- Become familiar with the basic components of a C++ program, including functions, special symbols, and identifiers
- Explore simple data types
- Discover how to use arithmetic operators
- Examine how a program evaluates arithmetic expressions
- Become familiar with the `string` data type
- Learn what an assignment statement is and what it does
- Learn about variable declaration
- Discover how to input data into memory using input statements
- Become familiar with the use of increment and decrement operators
- Examine ways to output results using output statements
- Learn how to use preprocessor directives and why they are necessary
- Learn how to debug syntax errors
- Explore how to properly structure a program, including using comments to document a program
- Become familiar with compound statements
- Learn how to write a C++ program

Teaching Tips

Introduction

1. Define the terms *computer program* and *programming*.

2. Use the recipe analogy to give students an idea of the process of programming.

A Quick Look at a C++ Program

1. Note that every C++ program must have a function called `main`. Use Example 2-1 to illustrate a basic `main` function. Walk through this example and point out the meaning of each line.

2. Discuss the purpose of an output statement and what it produces.

3. Point out the use of comments.

4. Briefly introduce the `#include` directive.

5. Use Figure 2-1 to describe the various parts of a C++ program.

Teaching Tip	Reassure students that although most of this example probably looks confusing, they will soon understand it and be comfortable with it.

6. Use Figures 2-2 and 2-3 to describe how memory is allocated and used to store values.

The Basics of a C++ Program

1. Explain that a C++ program is essentially a collection of one or more subprograms, called functions. Note that although many functions are predefined in the C++ library, programmers must learn how to write their own functions to accomplish specific tasks.

2. Define the terms *syntax rules* and *semantic rules* as they relate to a programming language and explain the difference between the two.

Teaching Tip	Emphasize that compilers check for syntax but not semantic errors. Give an example of each type of error.

Comments

1. Use the program in Example 2-1 to describe the use and importance of comments. Stress that comments are for the reader, not for the compiler.

2. Describe the two forms of comments shown in the textbook.

Teaching Tip	The importance of documenting a program cannot be underestimated. It is highly important for ensuring that the next programmer to be responsible for maintaining the code will be able to understand what the code is supposed to do.

Special Symbols

1. Explain that the C++ programming language consists of individual units called tokens, and these are divided into special symbols, word symbols, and identifiers.

2. Go over some of the special symbols in C++, including mathematical symbols, punctuation marks, the blank symbol, and double characters that are regarded as a single symbol.

Reserved Words (Keywords)

1. Discuss the word symbols, or keywords, used in C++, using Appendix A as a guide. Emphasize that C++ keywords are reserved and cannot be redefined for any other purpose with a program.

Identifiers

1. Define the term *identifier* as a name for something, such as a variable, constant, or function.

2. Discuss the rules for naming identifiers in C++. Also note that C++ is a case-sensitive language.

3. Use Table 2-1 to review the rules of identifier naming.

Teaching Tip	Discuss the difference between C++ conventions and rules. For example, it is a rule that a mathematical symbol cannot be used in an identifier name. However, it is a convention to begin an identifier with a lowercase letter.

Whitespaces

1. Explain that whitespaces (which include blanks, tabs, and newline characters) are used to separate special symbols, reserved words, and identifiers.

Data Types

1. Explain that C++ categorizes data into different types in order to manipulate the data in a program correctly. Although it may seem cumbersome at first to be so type-conscious, emphasize that C++ has these built-in checks to guard against errors.

Teaching Tip	Explain that C++ is called a strongly typed language because it checks for operations between inconsistent data types. This results in more robust and error-free programs. Demonstrate how C++ checks for data types with a simple program that attempts to add a string and a numeric value.

2. Define the term *data type* as a set of values together with a set of operations.

3. Mention that C++ data types fall into three categories: simple data types, structured data types, and pointers. Only the first type is discussed in this chapter.

Simple Data Types

1. Describe the three categories of simple data types in C++: integral, floating-point, and enumeration.

2. Mention the eleven categories of integral data types. Explain why C++ (and many other languages) has so many categories of the same data type. In addition, discuss the rules involving the use of integral types.

4. Explain the purpose of the `bool` data type.

5. Discuss the `char` data type, including its primary uses. Mention commonly used ASCII characters and their predefined ordering. Explain that a `char` data type is enclosed in single quotation marks, and note that only one symbol may be designated as a character.

6. Use Table 2-2 to summarize the three simple data types. Point out the difference in the amount of memory storage required, but inform students that this is system-dependent.

Floating-Point Data Types

1. Use Table 2-3 to explain how C++ represents real, or floating-point, numbers. Mention the three categories of data types to represent real numbers (`float`, `double`, and `long double`), and explain when to use each type.

2. Define the terms *precision*, *single precision*, and *double precision*.

Teaching Tip	Demonstrate how to find the values of `float` and `double` on a particular system by running the program with the header file `<cfloat>` in Appendix F. Encourage students to try running this program on their own computers and comparing the results.

Quick Quiz 1

1. What is an enumeration type?
   Answer: C++'s method for allowing programmers to create their own simple data types

2. The maximum number of significant digits in a number is called the _____.
   Answer: precision

3. The data type _____ has only two values: `true` and `false`.
   Answer: `bool`

4. The _____ data type, the smallest integral data type, is used to represent characters.
   Answer: `char`

Data Types and Variables, and Assignment Statements

1. Explain that the declaration of a variable requires that the data type be specified.

2. Explain the concept and syntax of an assignment.

Arithmetic Operators, Operator Precedence, and Expressions

1. Discuss the five arithmetic operators in C++ that are used to manipulate integral and floating-type data types.

2. Define the terms *unary* and *binary operators*, and discuss the difference between them.

Order of Precedence

1. Review operator precedence rules, as C++ uses these rules when evaluating expressions. Explain that parentheses can be used to override the order of operator precedence.

Expressions

1. This section discusses integral and floating-point expressions in detail.

2. Describe the three types of arithmetic expressions in C++.

3. Use Examples 2-6 and 2-7 to clarify how C++ processes expressions.

Mixed Expressions

1. Discuss the two rules for evaluating mixed expressions and illustrate these rules in practice using Example 2-8.

Quick Quiz 2

1. A(n) _____ operator has only one operand.
   Answer: unary

2. You can use _____ to override the order of precedence rules.
   Answer: parentheses

3. Describe the associativity of arithmetic operators.
   Answer: Unless there are parentheses, the associativity of arithmetic operators is said to be from left to right.

4. An expression that has operands of different data types is called a(n) _____.
   Answer: mixed expression

Type Conversion (Casting)

1. Explain how C++ avoids the hazards that result from implicit type coercion, which occurs when one data type is automatically changed into another data type.

2. Illustrate the form of the C++ cast operator using Example 2-9.

Teaching Tip	Students may feel a bit overwhelmed after the discussion of the `static_cast` operator. Ask them to run the program from Example 2.9. They should experiment with removing the `static_cast` operator from various statements, as well as changing the variable values. Ask them to report on any unpredictable results.

`string` Type

1. Introduce the C++ data type `string`, which is a programmer-defined data type available in the C++ library. Define a string as a sequence of zero or more characters.

2. Define the terms *null string* or *empty string*.

3. Discuss how to determine the length of a string, as well as the position of each character in a string.

Teaching Tip	Emphasize that the first position of a character in a string is 0, not 1. This will be helpful when manipulating both strings and arrays later on the text.

Variables, Assignment Statements, and Input Statements

1. Explain that data for a C++ program must be input into main memory. Mention the two-step process to store data in the computer's memory.

Allocating Memory with Constants and Variables

1. Emphasize that when allocating memory, the programmer must instruct the computer which names to use for each memory location as well as what type of data to store in those memory locations.

2. Define the term *named constant* and describe the syntax for declaring a named constant. Use Example 2-11 to illustrate the naming conventions for named constants. Explain why named constants are used in programs.

3. Define the term *variable* and use Example 2-12 to illustrate the syntax for declaring single and multiple variables.

4. Give a formal definition of a simple data type.

Putting Data into Variables

1. Mention the two ways you can place data in a variable in C++.

Assignment Statement

1. Discuss the C++ assignment statement, including its syntax, variable initialization, and the associativity rules of the assignment operator.

2. Step through Example 2-13 to illustrate how assignment statements operate in C++.

3. Use Example 2-14 to discuss the importance of doing a walk-through (tracing values through a sequence of steps) when writing code.

Teaching Tip	Building a table showing the values of variables at each step of the program is very helpful for students to understand the nature of variables.

Saving and Using the Value of an Expression

1. Explain the steps involved in saving the value of an expression using Example 2-15.

Declaring and Initializing Variables

1. Explain that when a variable is declared, C++ may not automatically put a meaningful value in it.

2. Emphasize that it is a good practice to initialize variables while they are being declared. Use one or more examples to illustrate how to do this in C++.

Input (Read) Statement

1. This section teaches your students to read data into variables from a standard input device. Define and explain the use of the C++ object `cin` and the stream extraction operator >>.

2. Step through Examples 2-16 through 2-18 to illustrate how to read in numeric and string data.

Variable Initialization

1. Reiterate that a variable can be initialized either through an assignment statement or a read statement. Explain why the read statement option is more versatile. Use Example 2-19 to illustrate both types of initialization.

Teaching Tip	Programmers (and instructors) have various approaches or preferences regarding variable declaration and initialization. Share your views on the topic. Do you think the best approach is to always initialize variables for consistency? Do you prefer initializing variables directly before the block of code that uses them, or initializing them during declaration?

Quick Quiz 3

1. What is a named constant?

Answer: A memory location whose content is not allowed to change during program execution

2. What is the syntax for declaring single or multiple variables?
   Answer: `dataType identifier, identifier, …;`

3. True or False: If you refer to an identifier without declaring it, the compiler will generate an error message.
   Answer: True

4. A variable is said to be _____ the first time a value is placed in it.
   Answer: initialized

Increment and Decrement Operators

1. Explain the purpose of the C++ increment (++) and decrement (--) operators.

2. Discuss how pre and post versions of these operators affect the results in a program. Use Example 2-20 to help explain the difference between these versions.

Teaching Tip	Verify that students are comfortable with using pre- and post-increment/decrement operators correctly, as it will be useful when working with control structures.

Output

1. Review how the C++ output statement is coded with the `cout` object and stream insertion operator (<<). Review the role of the `endl` manipulator in output statements as well.

2. Discuss the use of escape characters (see Table 2-4), such as the newline character, to format output. Demonstrate how to format output with Examples 2-21 through 2-26.

Teaching Tip	Outputting strings can be confusing at first. Talk about the various methods to deal with several lines of string output, and give your opinion as to the best approach. Emphasize that the Enter key cannot be used to break up a long string into two lines.

Preprocessor Directives

1. Explain the role of the preprocessor in C++. Discuss the use of header files and the syntax for including them in a C++ program.

Teaching Tip	Show your students some of the available operations in the `<cmath>` header. Here is one Web site with a description: www.cplusplus.com/ref/cmath/.

`namespace` and Using `cin` and `cout` in a Program

1. Briefly explain the purpose of the `namespace` mechanism in ANSI/ISO Standard C++. Discuss the `std namespace` and how it relates to the `<iostream>` header file.

2. Review the `using namespace std;` statement and its usefulness in programs using `cin` and `cout` statements.

Using the `string` Data Type in a Program

1. Mention that the `<string>` header must be included in C++ programs using the `string` data type.

Creating a C++ Program

1. Discuss the role of the function `main` in a C++ program. Go over the syntax of a `main` function, including declaration, assignment, executable, and return statements. Mention that named constant definitions and preprocessor directives are written before the `main` function.

2. Spend some time stepping through Examples 2-27 through 2-29. Verify that students understand each line of code in Example 2-29.

Debugging: Understanding and Fixing Syntax Errors

1. Review the sample program on Pages 85-86 and the compiler output that is generated when compiling the program. Walk through the various syntax errors and explain how each one should be fixed. Note that a syntax error on one line may be the cause of a compiler error on the following line.

Teaching Tip	Debugging is one of the most important skills a professional programmer acquires. Stress to students the importance of learning how to debug their own programs, including the verification of the results.

Program Style and Form

1. This section covers basic C++ syntax and semantic rules. Review these rules with your students in order for them to feel comfortable writing a complete functioning C++ program.

Syntax

1. Remind students that syntax rules define what is legal and what is not.

2. Discuss some common syntax errors. Emphasize that syntax errors should be corrected in the order in which the compiler lists them.

Use of Blanks

1. Discuss when blanks should and should not be used in a C++ program.

Use of Semicolons, Brackets, and Commas

1. Explain the purpose and meaning of semicolons, brackets, and commas in C++ programs. Define the term *statement terminator*.

Semantics

1. Define the term *semantics*.

2. Reiterate that a program may run without compiler errors and still have semantic errors that generate incorrect results. Use the example in the text to illustrate.

Naming Identifiers

1. Mention the conventions for naming identifiers, including self-documenting identifiers and run-together words.

Prompt Lines

1. Define the term *prompt lines*.

2. Explain why well-written prompt lines are essential for user input.

Documentation

1. Discuss why documenting a program through comments is critical to understanding and modifying the program at a later time.

Form and Style

1. Explain the purpose behind formatting and indentation conventions in source code. Use Example 2-30 to illustrate.

Teaching Tip	As with naming conventions, discuss your own preferences in terms of form and style in programming. Use the programming examples at the end of the chapter to talk about various stylistic elements. Discuss the value of the "art" of programming.

Quick Quiz 4

1. True or False: The semantic rules of a language tell you what is legal and what is not legal.
   Answer: False

2. The semicolon is also called the _____.
   Answer: statement terminator

3. How can you make run-together words easier to understand?
   Answer: Capitalizing the beginning of each new word; or inserting an underscore before each new word

4. Why are comments important in a program?
   Answer: A well-documented program is easier to understand and modify, even a long time after you originally wrote it. You use comments to document programs. Comments should appear in a program to explain the purpose of the program, identify who wrote it, and explain the purpose of particular statements.

More on Assignment Statements

1. Define the terms *simple assignment statement* and *compound assignment statement*.

2. Define the C++ compound operators (+=, -=, *=, /=, and %=) and explain how and why compound assignment statements are used in C++ programs. Use Example 2-31 to illustrate this.

3. Step through the "Convert Length" and "Make Change" Programming Examples to help the students consolidate all the information from this chapter.

Class Discussion Topics

1. As mentioned in this chapter, C++ predefined identifiers such as `cout` and `cin` can be redefined by the programmer. However, why is it not wise to do so?

2. The text mentioned that the `char` data type can be cast into an `int`. What are some possible uses of this functionality?

Additional Projects

1. Learn and report on various compiler errors by modifying one or two of the programs in this chapter. Try to compile the program. What happens when you do not initialize a value for a named constant? What are the error messages when you use a numeric or string constant in an expression without first giving it a value? Finally, what happens when you initialize a `char` data type with a character enclosed in double quotes?

2. Use one of the programs in this chapter to test for invalid user input. The program should compile with no errors. What happens when you enter an unexpected value (such as an incorrect data type) when prompted for user input? Test with several sets of invalid data and document your findings.

Additional Resources

1. C++ Examples:
   https://developers.google.com/edu/c++/getting-started

2. Basic Input/Output:
   www.cplusplus.com/doc/tutorial/basic_io.html

3. C++ Programming Style Guidelines:
   http://geosoft.no/development/cppstyle.html

4. Strong vs. Weak Typing:
   www.artima.com/intv/strongweak.html

Key Terms

- **Arithmetic expression**: an expression constructed using arithmetic operators and numbers
- **Assignment operator**: =; assigns whatever is on the right side to the variable on the left side
- **Associativity**: the associativity of arithmetic operators is said to be from left to right
- **Binary operator**: an operator that has two operands
- **Cast operator (type conversion, type casting)**: used to explicitly convert one data type to another data type
- **Character arithmetic**: arithmetic operation on `char` data
- **Collating sequence**: a predefined ordering for the characters in a set
- **Compound assignment statement**: statements that are used to write simple assignment statements in a more concise notation
- **Computer program**: a sequence of statements whose objective is to accomplish a task
- **Data type**: a set of values together with a set of operations
- **Declaration statements**: statements that are used to declare things, such as variables
- **Decrement operator**: --; decreases the value of a variable by 1
- **Double precision**: values of type `double`
- **Enumeration**: a user-defined data type
- **Executable statements**: statements that perform calculations, manipulate data, create output, accept input, and so on
- **Floating-point**: a data type that deals with decimal numbers
- **Floating-point (decimal) expression**: an expression in which all operands in the expression are floating-point numbers
- **Floating-point notation**: a form of scientific notation used to represent real numbers
- **Function (subprogram)**: a collection of statements; when activated, or executed, it accomplishes something
- **Identifier**: a C++ identifier consists of letters, digits, and the underscore character (_); it must begin with a letter or underscore
- **Implicit type coercion**: when a value of one data type is automatically changed to another data type
- **Increment operator**: ++; increases the value of a variable by 1
- **Initialized**: the first time a value is placed in the variable
- **Input (read) statement**: a statement that places data into variables using `cin` and >>
- **Integral**: a data type that deals with integers, or numbers, without a decimal part
- **Integral expression**: an expression in which all operands are integers
- **Keyword:** a reserved word
- **Mixed expression**: an expression that has operands of different data types
- **Named constant**: a memory location whose content is not allowed to change during program execution
- **Null (empty) string**: a string containing no characters
- **Operands**: numbers appearing in an arithmetic expression
- **Output statement**: an output on the standard output device via `cout` and <<

- **Post-decrement**: has the syntax `variable--`
- **Post-increment**: has the syntax `variable++`
- **Precision**: the maximum number of significant digits
- **Pre-decrement**: has the syntax `--variable`
- **Predefined (standard) function**: a function that is already written and provided as part of the system
- **Pre-increment**: has the syntax `++variable`
- **Preprocessor**: a program that carries out preprocessor directives
- **Programming**: the process of planning and creating a program
- **Programming language**: a set of rules, symbols, and special words
- **Prompt lines**: executable statements that inform the user what to do
- **Reserved words (keywords)**: word symbols in a programming language that cannot be redefined in any program
- **Run-together word**: an identifier that is composed of two or more words that are combined without caps or underscores
- **Self-documenting identifiers**: identifiers that describe the purpose of the identifier through the name
- **Semantic rules**: rules that determine the meaning of the instructions
- **Semantics**: a set of rules that gives meaning to a language
- **Simple assignment statement**: a statement that uses only the assignment operator to assign values to the variable on the left side of the operator
- **Simple data type**: the variable or named constant of that type can store only one value at a time
- **Single precision**: values of type `float`
- **Source code**: consists of the preprocessor directives and program statements
- **Source code file (source file)**: a file containing source code
- **Statement terminator**: the semicolon
- **Stream extraction operator**: >>; takes information from a stream and puts it into a variable
- **Stream insertion operator**: <<; takes information from a variable and puts it into a stream
- **String**: a sequence of zero or more characters
- **Syntax rules**: rules that describe which statements (instructions) are legal, or accepted by the programming language, and which are not legal
- **Token**: the smallest individual unit of a program written in any language
- **Unary operator**: an operator that has only one operand
- **Variable**: a memory location whose content may change during program execution
- **Walk-through**: the process of tracing values through a sequence