C++ Programming: From Problem Analysis to Program Design, Fifth Edition

Chapter 1: An Overview of Computers and Programming Languages

The Evolution of Programming Languages (cont'd.)

- High-level languages include Basic, FORTRAN, COBOL, Pascal, C, C++, C#, and Java
- Compiler: translates a program written in a high-level language machine language

Processing a C++ Program

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

Sample Run:

```
My first C++ program.
```

Processing a C++ Program (cont'd.)

- To execute a C++ program:
  - Use an editor to create a source program in C++
  - Preprocessor directives begin with # and are processed by a the preprocessor
  - Use the compiler to:
    - Check that the program obeys the rules
    - Translate into machine language (object program)

Processing a C++ Program (cont'd.)

- To execute a C++ program (cont'd.):
  - Linker:
    - Combines object program with other programs provided by the SDK to create executable code
  - Loader:
    - Loads executable program into main memory
  - The last step is to execute the program

Processing a C++ Program (cont'd.)

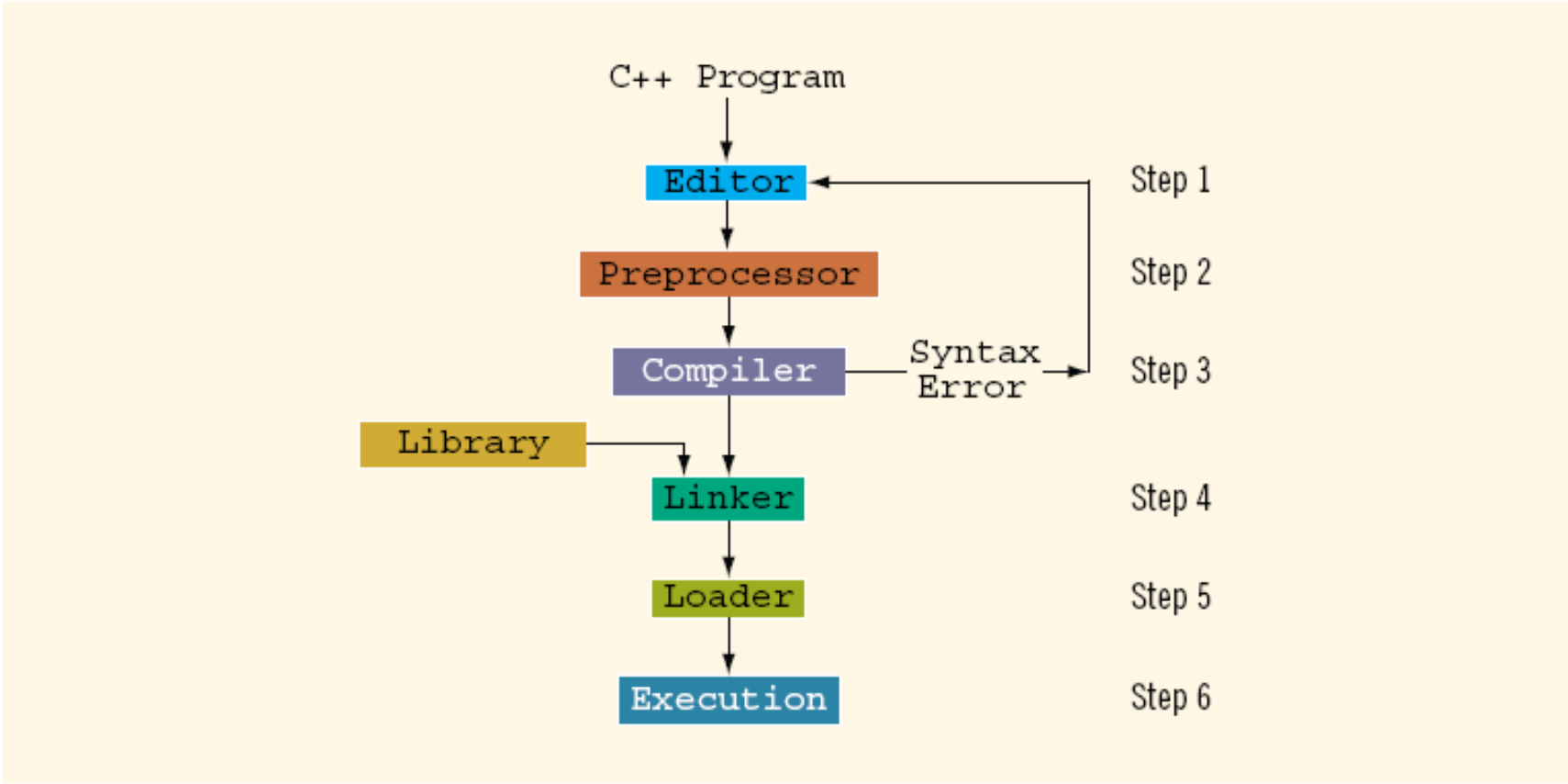


FIGURE 1-3 Processing a C++ program

Programming with the Problem Analysis–Coding–Execution Cycle

- Programming is a process of problem solving
- One problem-solving technique:
  - Analyze the problem
  - Outline the problem requirements
  - Design steps (algorithm) to solve the problem
- Algorithm:
  - Step-by-step problem-solving process
  - Solution achieved in finite amount of time

The Problem Analysis–Coding–Execution Cycle (cont'd.)

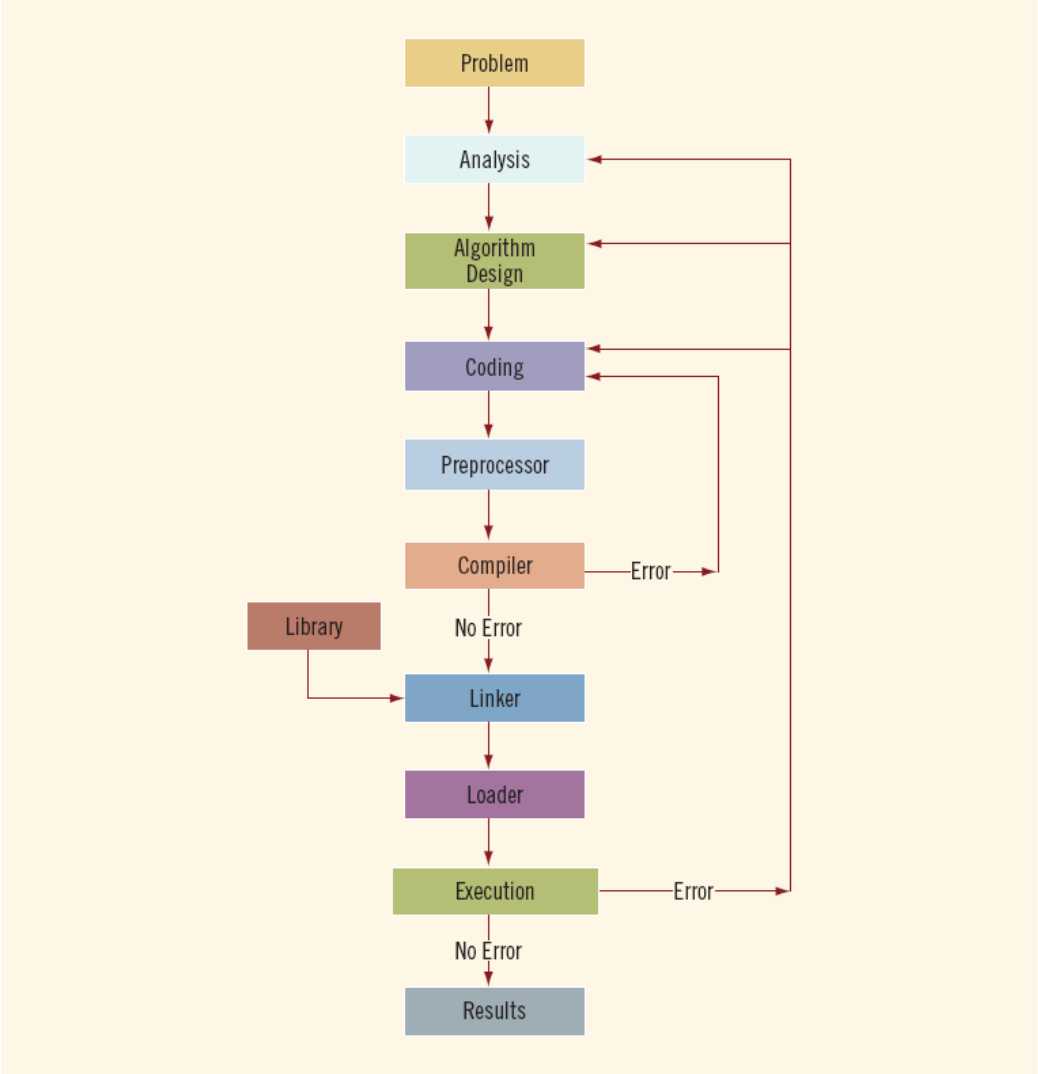


FIGURE 1-4 Problem analysis–coding–execution cycle

The Problem Analysis–Coding–Execution Cycle (cont'd.)

- Run code through compiler
- If compiler generates errors
  - Look at code and remove errors
  - Run code again through compiler
- If there are no syntax errors
  - Compiler generates equivalent machine code
- Linker links machine code with system resources

The Problem Analysis–Coding–Execution Cycle (cont'd.)

- Once compiled and linked, loader can place program into main memory for execution
- The final step is to execute the program
- Compiler guarantees that the program follows the rules of the language
  - Does not guarantee that the program will run correctly

Example 1-1

- Design an algorithm to find the perimeter and area of a rectangle
- The perimeter and area of the rectangle are given by the following formulas:

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

Example 1-1 (cont'd.)

- Algorithm:
  - Get length of the rectangle
  - Get width of the rectangle
  - Find the perimeter using the following equation:

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

  - Find the area using the following equation:

```
area = length * width
```

C++ Programming: From Problem Analysis to Program Design, Fourth Edition

Chapter 2: Basic Elements of C++

Objectives

In this chapter, you 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

Objectives (continued)

- Learn what an assignment statement is and what it does
- Become familiar with the `string` data type
- 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

Objectives (continued)

- Learn how to use preprocessor directives and why they are necessary
- Explore how to properly structure a program, including using comments to document a program
- Learn how to write a C++ program

The Basics of a C++ Program

- Function: collection of statements; when executed, accomplishes something
  - May be predefined or standard
- Syntax: rules that specify which statements (instructions) are legal
- Programming language: a set of rules, symbols, and special words
- Semantic rule: meaning of the instruction

Comments

- Comments are for the reader, not the compiler
- Two types:
  - Single line

```
// This is a C++ program. It prints the sentence:
// Welcome to C++ Programming.
```

  - Multiple line

```
/*
   You can include comments that can
   occupy several lines.
*/
```

Special Symbols

- Special symbols

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

Reserved Words (Keywords)

- Reserved words, keywords, or word symbols
  - Include:
    - `int`
    - `float`
    - `double`
    - `char`
    - `const`
    - `void`
    - `return`

Identifiers

- Consist of letters, digits, and the underscore character (_)
- Must begin with a letter or underscore
- C++ is case sensitive
  - `NUMBER` is not the same as `number`
- Two predefined identifiers are `cout` and `cin`
- Unlike reserved words, predefined identifiers may be redefined, but it is not a good idea

Identifiers (continued)

- The following are legal identifiers in C++:
  - `first`
  - `conversion`
  - `payRate`

TABLE 2-1 Examples of Illegal Identifiers

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

Whitespaces

- Every C++ program contains whitespaces
  - Include blanks, tabs, and newline characters
- Used to separate special symbols, reserved words, and identifiers
- Proper utilization of whitespaces is important
  - Can be used to make the program readable

Data Types

- Data type: set of values together with a set of operations
- C++ data types fall into three categories:

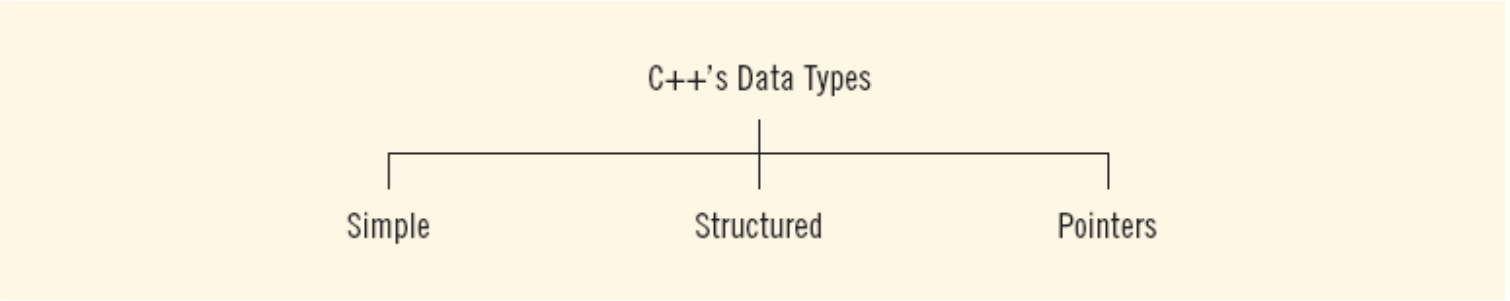


FIGURE 2-1 C++ data types

Simple Data Types

- Three categories of simple data
  - Integral: integers (numbers without a decimal)
  - Floating-point: decimal numbers
  - Enumeration type: user-defined data type

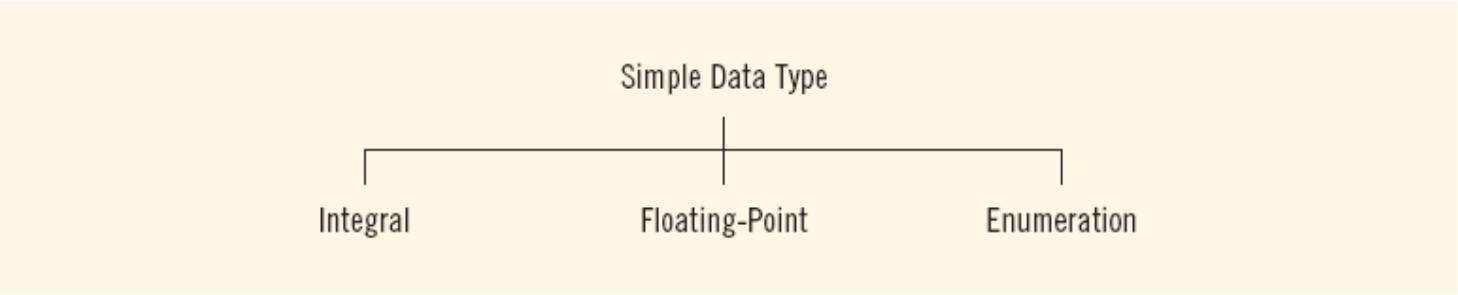


FIGURE 2-2 Simple data types

Simple Data Types (continued)

- Integral data types are further classified into nine categories:

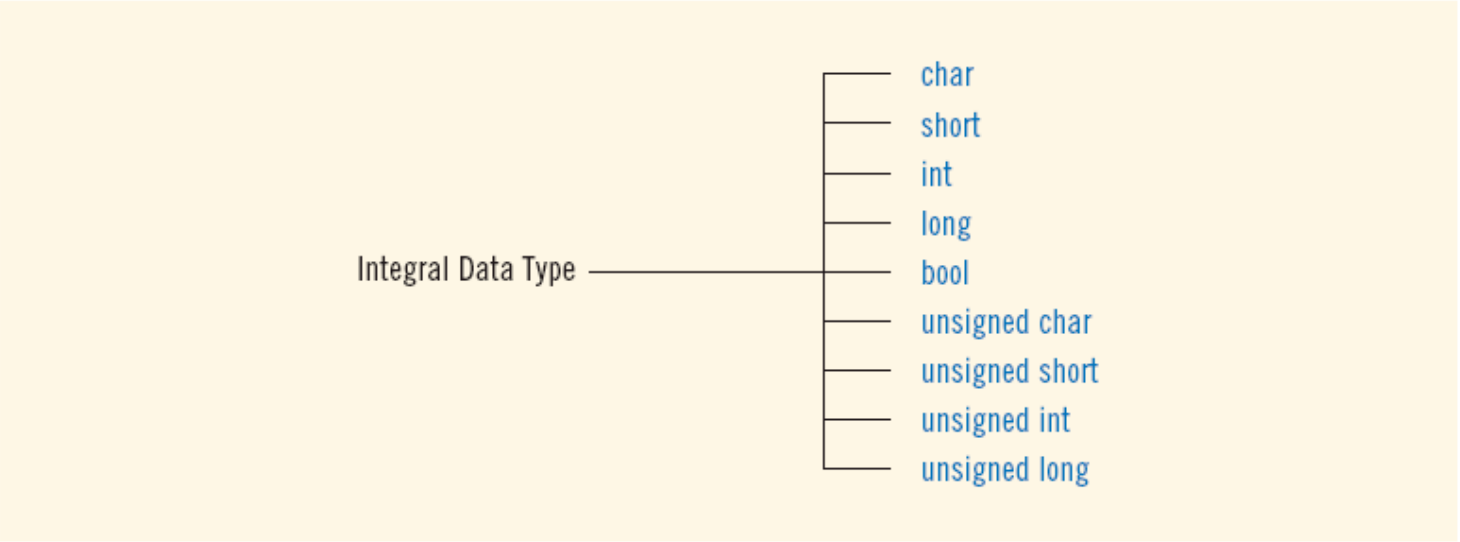


FIGURE 2-3 Integral data types

Simple Data Types (continued)

TABLE 2-2 Values and Memory Allocation for Three Simple Data Types

Data Type	Values	Storage (in bytes)
`int`	-2147483648 to 2147483647	4
`bool`	`true` and `false`	1
`char`	-128 to 127	1

- Different compilers may allow different ranges of values

`int` Data Type

- Examples:

  `-6728`

  `0`

  `78`

  `+763`

- Positive integers do not need a `+` sign
- No commas are used within an integer
  - Commas are used for separating items in a list

`bool` Data Type

- `bool` type
  - Two values: `true` and `false`
  - Manipulate logical (Boolean) expressions
- `true` and `false` are called logical values
- `bool`, `true`, and `false` are reserved words

`char` Data Type

- The smallest integral data type
- Used for characters: letters, digits, and special symbols
- Each character is enclosed in single quotes
  - `'A', 'a', '0', '*', '+', '$', '&'`
- A blank space is a character and is written `' '`, with a space left between the single quotes

Floating-Point Data Types

- C++ uses scientific notation to represent real numbers (floating-point notation)

TABLE 2-3 Examples of Real Numbers Printed in C++ Floating-Point Notation

Real Number	C++ Floating-Point Notation
75.924	7.592400E1
0.18	1.800000E-1
0.0000453	4.530000E-5
-1.482	-1.482000E0
7800.0	7.800000E3

Floating-Point Data Types (continued)

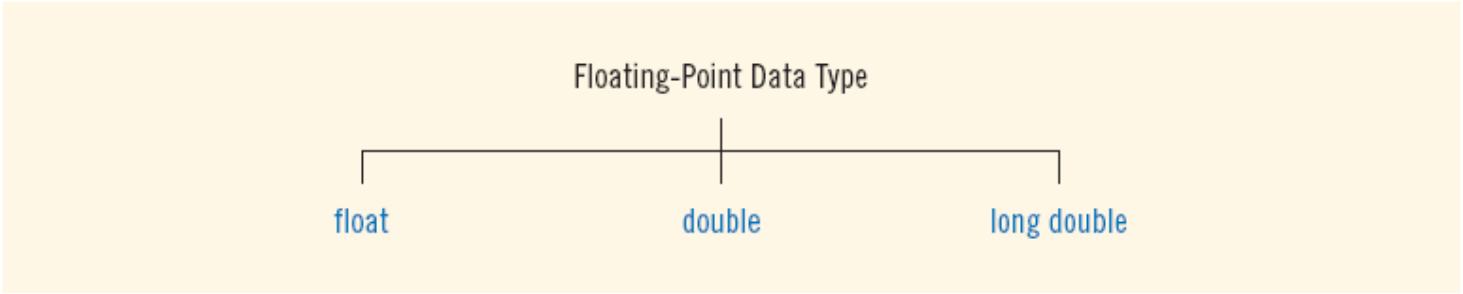


FIGURE 2-4 Floating-point data types

- `float`: represents any real number
  - Range: -3.4E+38 to 3.4E+38 (four bytes)
- `double`: represents any real number
  - Range: -1.7E+308 to 1.7E+308 (eight bytes)
- On most newer compilers, data types `double` and `long double` are same

Floating-Point Data Types (continued)

- Maximum number of significant digits (decimal places) for float values is 6 or 7
- Maximum number of significant digits for double is 15
- Precision: maximum number of significant digits
  - Float values are called single precision
  - Double values are called double precision

Arithmetic Operators and Operator Precedence

- C++ arithmetic operators:
  - + addition
  - - subtraction
  - * multiplication
  - / division
  - % modulus operator
- +, -, *, and / can be used with integral and floating-point data types
- Operators can be unary or binary

Order of Precedence

- All operations inside of () are evaluated first
- *, /, and % are at the same level of precedence and are evaluated next
- + and – have the same level of precedence and are evaluated last
- When operators are on the same level
  - Performed from left to right (associativity)
- `3 * 7 – 6 + 2 * 5 / 4 + 6` means

```
(((3 * 7) – 6) + ((2 * 5) / 4 )) + 6
```

Expressions

- If all operands are integers
  - Expression is called an integral expression
    - Yields an integral result
    - Example: `2 + 3 * 5`
- If all operands are floating-point
  - Expression is called a floating-point expression
    - Yields a floating-point result
    - Example: `12.8 * 17.5 - 34.50`

Mixed Expressions

- Mixed expression:
  - Has operands of different data types
  - Contains integers and floating-point
- Examples of mixed expressions:

```
2 + 3.5
6  /  4 + 3.9
5.4  *  2 – 13.6 + 18  /  2
```

Mixed Expressions (continued)

- Evaluation rules:
  - If operator has same types of operands
    - Evaluated according to the type of the operands
  - If operator has both types of operands
    - Integer is changed to floating-point
    - Operator is evaluated
    - Result is floating-point
  - Entire expression is evaluated according to precedence rules

Type Conversion (Casting)

- Implicit type coercion: when value of one type is automatically changed to another type
- Cast operator: provides explicit type conversion

```
static_cast<dataTypeName>(expression)
```

Type Conversion (continued)

EXAMPLE 2-9

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`

`string` Type

- Programmer-defined type supplied in ANSI/ISO Standard C++ library
- Sequence of zero or more characters
- Enclosed in double quotation marks
- Null: a string with no characters
- Each character has relative position in string
  - Position of first character is 0
- Length of a string is number of characters in it
  - Example: length of `"William Jacob"` is 13

Input

- Data must be loaded into main memory before it can be manipulated
- Storing data in memory is a two-step process:
  - Instruct computer to allocate memory
  - Include statements to put data into memory

Allocating Memory with Constants and Variables

- Named constant: memory location whose content can't change during execution
- The syntax to declare a named constant is:

```
const dataType identifier = value;
```

- In C++, `const` is a reserved word

EXAMPLE 2-11

Consider the following C++ statements:

```
const double CONVERSION = 2.54;
const int NO_OF_STUDENTS = 20;
const char BLANK = ' ';
const double PAY_RATE = 15.75;
```

Allocating Memory with Constants and Variables (continued)

- Variable: memory location whose content may change during execution
- The syntax to declare a named constant is:

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

EXAMPLE 2-12

Consider the following statements:

```
double amountDue;
int counter;
char ch;
int x, y;
string name;
```

Putting Data into Variables

- Ways to place data into a variable:
  - Use C++'s assignment statement
  - Use input (read) statements

Assignment Statement

- The assignment statement takes the form:

```
variable = expression;
```

- Expression is evaluated and its value is assigned to the variable on the left side
- In C++, = is called the assignment operator

Assignment Statement (continued)

EXAMPLE 2-13

```
int num1, num2;
double sale;
char first;
string str;

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

EXAMPLE 2-14

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

Saving and Using the Value of an Expression

- To save the value of an expression:
  - Declare a variable of the appropriate data type
  - Assign the value of the expression to the variable that was declared
    - Use the assignment statement
- Wherever the value of the expression is needed, use the variable holding the value

Declaring & Initializing Variables

- Variables can be initialized when declared:

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

- All variables must be initialized before they are used
  - But not necessarily during declaration

Input (Read) Statement

- `cin` is used with `>>` to gather input

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

- The stream extraction operator is >>
- For example, if miles is a double variable

```
cin >> miles;
```

  - Causes computer to get a value of type `double`
  - Places it in the variable `miles`

Input (Read) Statement (continued)

- Using more than one variable in `cin` allows more than one value to be read at a time
- For example, if `feet` and `inches` are variables of type `int`, a statement such as:

```
cin >> feet >> inches;
```

  - Inputs two integers from the keyboard
  - Places them in variables `feet` and `inches` respectively

Input (Read) Statement (continued)

EXAMPLE 2-17

```
#include <iostream>

using namespace std;

int main()
{
    int feet;
    int inches;

    cout << "Enter two integers separated by spaces: ";
    cin >> feet >> inches;
    cout << endl;

    cout << "Feet = " << feet << endl;
    cout << "Inches = " << inches << endl;

    return 0;
}
```

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

```
Enter two integers separated by spaces: 23 7

Feet = 23
Inches = 7
```

Variable Initialization

- There are two ways to initialize a variable:

```
int feet;
```

  - By using the assignment statement

```
feet = 35;
```

  - By using a read statement

```
cin >> feet;
```

Increment & Decrement Operators

- Increment operator: increment variable by 1
  - Pre-increment: `++variable`
  - Post-increment: `variable++`
- Decrement operator: decrement variable by 1
  - Pre-decrement: `--variable`
  - Post-decrement: `variable–`
- What is the difference between the following?

```
x = 5;
y = ++x;
```

```
x = 5;
y = x++;
```

Output

- The syntax of `cout` and `<<` is:

```
cout << expression or manipulator << expression or manipulator...;
```

  - Called an output statement
- The stream insertion operator is <<
- Expression evaluated and its value is printed at the current cursor position on the screen

Output (continued)

- A manipulator is used to format the output
  - Example: `endl` causes insertion point to move to beginning of next line

EXAMPLE 2-21

	Statement	Output
1	`cout << 29 / 4 << endl;`	`7`
2	`cout << "Hello there." << endl;`	`Hello there.`
3	`cout << 12 << endl;`	`12`
4	`cout << "4 + 7" << endl;`	`4 + 7`
5	`cout << 4 + 7 << endl;`	`11`
6	`cout << 'A' << endl;`	`A`
7	`cout << "4 + 7 = " << 4 + 7 << endl;`	`4 + 7 = 11`
8	`cout << 2 + 3 * 5 << endl;`	`17`
9	`cout << "Hello \nthere." << endl;`	`Hello` `there.`

Output (continued)

- The new line character is '`\n`'
  - May appear anywhere in the string

```
cout << "Hello there.";
cout << "My name is James.";
```

- Output:

```
Hello there.My name is James.
```

```
cout << "Hello there.\n";
cout << "My name is James.";
```

- Output :

```
Hello there.
My name is James.
```

Output (continued)

TABLE 2-4 Commonly Used Escape Sequences

	Escape Sequence	Description
`\n`	Newline	Cursor moves to the beginning of the next line
`\t`	Tab	Cursor moves to the next tab stop
`\b`	Backspace	Cursor moves one space to the left
`\r`	Return	Cursor moves to the beginning of the current line (not the next line)
`\\`	Backslash	Backslash is printed
`\'`	Single quotation	Single quotation mark is printed
`\"`	Double quotation	Double quotation mark is printed

Preprocessor Directives

- C++ has a small number of operations
- Many functions and symbols needed to run a C++ program are provided as collection of libraries
- Every library has a name and is referred to by a header file
- Preprocessor directives are commands supplied to the preprocessor
- All preprocessor commands begin with #
- No semicolon at the end of these commands

Preprocessor Directives (continued)

- Syntax to include a header file:

```
#include <headerFileName>
```

- For example:

```
#include <iostream>
```

  - Causes the preprocessor to include the header file `iostream` in the program

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

- `cin` and `cout` are declared in the header file `iostream`, but within `std` namespace
- To use `cin` and `cout` in a program, use the following two statements:

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

Using the `string` Data Type in a Program

- To use the `string` type, you need to access its definition from the header file `string`
- Include the following preprocessor directive:

```
#include  <string>
```

Creating a C++ Program

- C++ program has two parts:
  - Preprocessor directives
  - The program
- Preprocessor directives and program statements constitute C++ source code (.cpp)
- Compiler generates object code (.obj)
- Executable code is produced and saved in a file with the file extension .exe

Creating a C++ Program (continued)

- A C++ program is a collection of functions, one of which is the function `main`
- The first line of the function `main` is called the heading of the function:

```
int main()
```

- The statements enclosed between the curly braces ({ and }) form the body of the function
  - Contains two types of statements:
    - Declaration statements
    - Executable statements

EXAMPLE 2-29

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

using namespace std;                                        //Line 2

const int NUMBER = 12;                                      //Line 3

int main()                                                  //Line 4
{                                                           //Line 5
    int firstNum;                                           //Line 6
    int secondNum;                                          //Line 7

    firstNum = 18;                                          //Line 8
    cout << "Line 9: firstNum = " << firstNum
         << endl;                                           //Line 9

    cout << "Line 10: Enter an integer: ";                  //Line 10
    cin >> secondNum;                                       //Line 11
    cout << endl;                                           //Line 12

    cout << "Line 13: secondNum = " << secondNum
         << endl;                                           //Line 13

    firstNum = firstNum + NUMBER + 2 * secondNum;           //Line 14

    cout << "Line 15: The new value of "
         << "firstNum = " << firstNum << endl;              //Line 15

    return 0;                                               //Line 16
}                                                           //Line 17
```

Creating a C++ Program (continued)

Sample Run:

```
Line 9: firstNum = 18
Line 10: Enter an integer: 15

Line 13: secondNum = 15
Line 15: The new value of firstNum = 60
```

Program Style and Form

- Every C++ program has a function `main`
- It must also follow the syntax rules
- Other rules serve the purpose of giving precise meaning to the language

Syntax

- Errors in syntax are found in compilation

```
int x;          //Line 1
int y           //Line 2: error
double z;       //Line 3

y = w + x;      //Line 4: error
```

Use of Blanks

- In C++, you use one or more blanks to separate numbers when data is input
- Used to separate reserved words and identifiers from each other and from other symbols
- Must never appear within a reserved word or identifier

Use of Semicolons, Brackets, and Commas

- All C++ statements end with a semicolon
  - Also called a statement terminator
- { and } are not C++ statements
- Commas separate items in a list

Semantics

- Possible to remove all syntax errors in a program and still not have it run
- Even if it runs, it may still not do what you meant it to do
- For example,

  `2 + 3 * 5` and `(2 + 3) * 5`

  are both syntactically correct expressions, but have different meanings

Naming Identifiers

- Identifiers can be self-documenting:
  - `CENTIMETERS_PER_INCH`
- Avoid run-together words :
  - `annualsale`
  - Solution:
    - Capitalize the beginning of each new word
      - `annualSale`
    - Inserting an underscore just before a new word
      - `annual_sale`

Prompt Lines

- Prompt lines: executable statements that inform the user what to do

```
cout << "Please enter a number between 1 and 10 and "
     << "press the return key" << endl;
cin >> num;
```

Documentation

- A well-documented program is easier to understand and modify
- You use comments to document programs
- Comments should appear in a program to:
  - Explain the purpose of the program
  - Identify who wrote it
  - Explain the purpose of particular statements

Form and Style

- Consider two ways of declaring variables:
  - Method 1

```
int feet, inch;
double x, y;
```

  - Method 2

```
int a,b;double x,y;
```

- Both are correct; however, the second is hard to read

More on Assignment Statements

- C++ has special assignment statements called compound assignments

  `+=`, `-=`, `*=`, `/=`, and `%=`

- Example:

```
x *= y;
```

Programming Example: Convert Length

- Write a program that takes as input a given length expressed in feet and inches
  - Convert and output the length in centimeters
- Input: length in feet and inches
- Output: equivalent length in centimeters
- Lengths are given in feet and inches
- Program computes the equivalent length in centimeters
- One inch is equal to `2.54` centimeters

Programming Example: Convert Length (continued)

- Convert the length in feet and inches to all inches:
  - Multiply the number of feet by 12
  - Add given inches
- Use the conversion formula (1 inch = 2.54 centimeters) to find the equivalent length in centimeters

Programming Example: Convert Length (continued)

- The algorithm is as follows:
  - Get the length in feet and inches
  - Convert the length into total inches
  - Convert total inches into centimeters
  - Output centimeters

Programming Example: Variables and Constants

- Variables

```
int feet;       //variable to hold given feet
int inches;     //variable to hold given inches
int totalInches;  //variable to hold total inches
double centimeters;  //variable to hold length in
                     //centimeters
```

- Named Constant

```
const double CENTIMETERS_PER_INCH = 2.54;
const int INCHES_PER_FOOT = 12;
```

Programming Example: Main Algorithm

- Prompt user for input
- Get data
- Echo the input (output the input)
- Find length in inches
- Output length in inches
- Convert length to centimeters
- Output length in centimeters

Programming Example: Putting It Together

- Program begins with comments
- System resources will be used for I/O
- Use input statements to get data and output statements to print results
- Data comes from keyboard and the output will display on the screen
- The first statement of the program, after comments, is preprocessor directive to include header file `iostream`

Programming Example: Putting It Together (continued)

- Two types of memory locations for data manipulation:
  - Named constants
    - Usually put before `main`
  - Variables
- This program has only one function (`main`), which will contain all the code
- The program needs variables to manipulate data, which are declared in `main`

Programming Example: Body of the Function

- The body of the function `main` has the following form:

```
int main ()
{
   declare variables
   statements
   return 0;
}
```

Programming Example: Writing a Complete Program

- Begin the program with comments for documentation
- Include header files
- Declare named constants, if any
- Write the definition of the function `main`

```
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 centimeter;

        //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

    centimeter = CENTIMETERS_PER_INCH * totalInches;     //Step 6

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

    return 0;
}
```

Programming Example: Sample Run

```
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
```

Summary

- C++ program: collection of functions where each program has a function called `main`
- Identifier consists of letters, digits, and underscores, and begins with letter or underscore
- The arithmetic operators in C++ are addition (+), subtraction (-),multiplication (*), division (/), and modulus (%)
- Arithmetic expressions are evaluated using the precedence associativity rules

Summary (continued)

- All operands in an integral expression are integers and all operands in a floating-point expression are decimal numbers
- Mixed expression: contains both integers and decimal numbers
- Use the cast operator to explicitly convert values from one data type to another
- A named constant is initialized when declared
- All variables must be declared before used

Summary (continued)

- Use `cin` and stream extraction operator `>>` to input from the standard input device
- Use `cout` and stream insertion operator `<<` to output to the standard output device
- Preprocessor commands are processed before the program goes through the compiler
- A file containing a C++ program usually ends with the extension `.cpp`

C++ Programming: From Problem Analysis to Program Design, Fourth Edition

Chapter 4: Control Structures I *(Selection)*

Objectives

In this chapter, you will:

- Learn about control structures
- Examine relational and logical operators
- Explore how to form and evaluate logical (Boolean) expressions
- Discover how to use the selection control structures `if`, `if...else`, and `switch` in a program
- Learn to use the `assert` function to terminate a program

Control Structures

- A computer can proceed:
  - In sequence
  - Selectively (branch) - making a choice
  - Repetitively (iteratively) - looping
- Some statements are executed only if certain conditions are met
- A condition is met if it evaluates to `true`

Control Structures (continued)

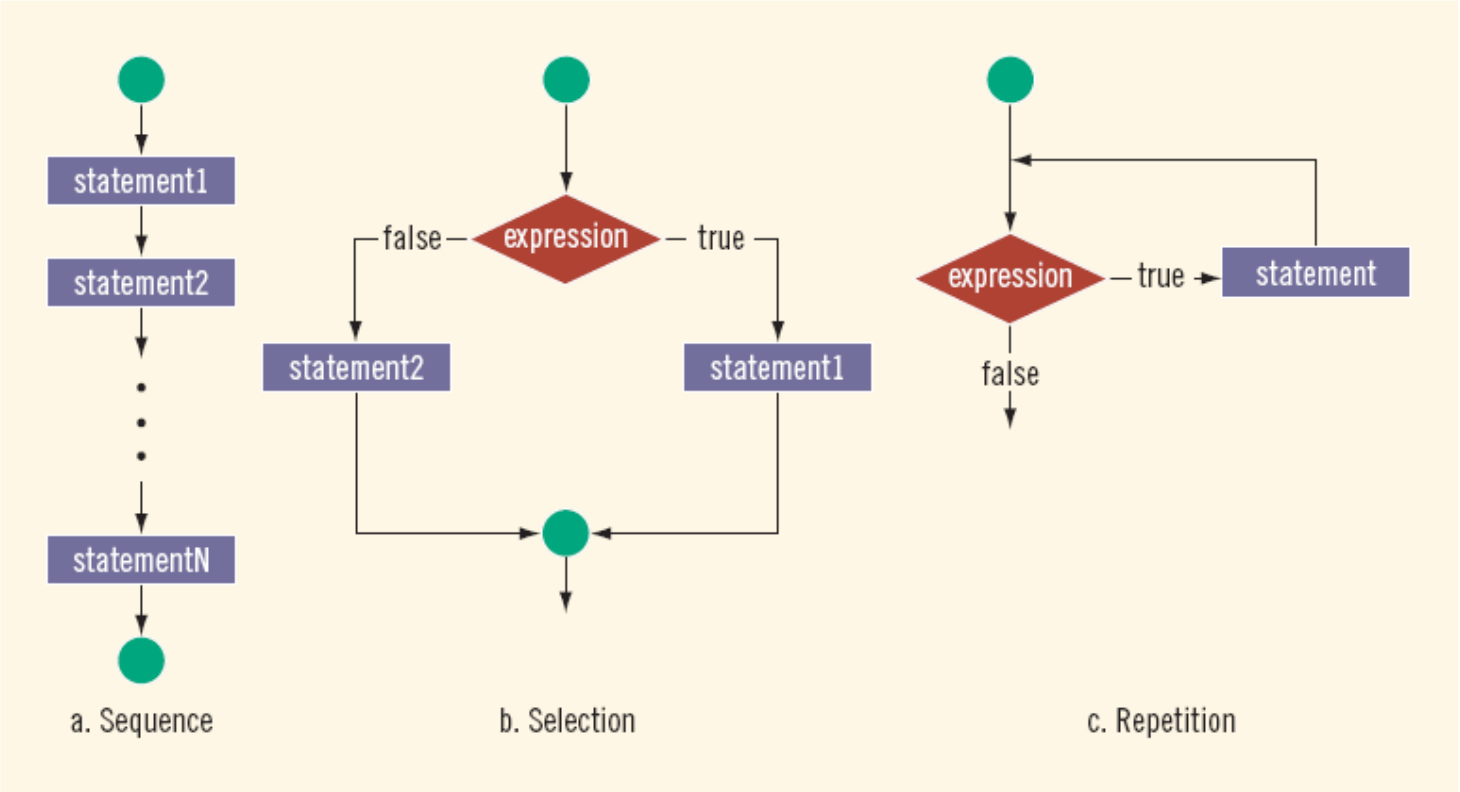


FIGURE 4-1 Flow of execution

Relational Operators

- A condition is represented by a logical (Boolean) expression that can be `true` or `false`
- Relational operators:
  - Allow comparisons
  - Require two operands (binary)
  - Evaluate to `true` or `false`

Relational Operators (continued)

TABLE 4-1 Relational Operators in C++

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

Relational Operators and Simple Data Types

- You can use the relational operators with all three simple data types:
  - `8 < 15` evaluates to `true`
  - `6 != 6` evaluates to `false`
  - `2.5 > 5.8` evaluates to `false`
  - `5.9 <= 7.5` evaluates to `true`

Comparing Floating-Point Numbers for Equality

- Comparison of floating-point numbers for equality may not behave as you would expect
  - Example:
    - `1.0 == 3.0/7.0 + 2.0/7.0 + 2.0/7.0` evaluates to `false`
    - Why? `3.0/7.0 + 2.0/7.0 + 2.0/7.0 = 0.9999999999999989`
- Solution: use a tolerance value
  - Example: `fabs(x - y) < 0.000001`

Comparing Characters

TABLE 4-2 Evaluating Expressions Using Relational Operators and the ASCII Collating Sequence

Expression	Value of Expression	Explanation
`' ' < 'a'`	`true`	The ASCII value of `' '` is 32, and the ASCII value of `'a'` is 97. Because `32 < 97` is `true`, it follows that `' ' < 'a'` is `true`.
`'R' > 'T'`	`false`	The ASCII value of `'R'` is 82, and the ASCII value of `'T'` is 84. Because `82 > 84` is `false`, it follows that `'R' > 'T'` is `false`.
`'+' < '*'`	`false`	The ASCII value of `'+'` is 43, and the ASCII value of `'*'` is 42. Because `43 < 42` is `false`, it follows that `'+' < '*'` is `false`.
`'6' <= '>'`	`true`	The ASCII value of `'6'` is 54, and the ASCII value of `'>'` is 62. Because `54 <= 62` is `true`, it follows that `'6' <= '>'` is `true`.

Relational Operators and the `string` Type

- Relational operators can be applied to strings
- Strings are compared character by character, starting with the first character
- Comparison continues until either a mismatch is found or all characters are found equal
- If two strings of different lengths are compared and the comparison is equal to the last character of the shorter string
  - The shorter string is less than the larger string

Relational Operators and the `string` Type (continued)

- Suppose we have the following declarations:

```
string str1 = "Hello";
string str2 = "Hi";
string str3 = "Air";
string str4 = "Bill";
string str4 = "Big";
```

Relational Operators and the `string` Type (continued)

TABLE 4-3 Evaluating Logical Expressions with `string` Variables

Expression	Value	Explanation
`str1 < str2`	`true`	`str1 = "Hello"` and `str2 = "Hi"`. The first characters of `str1` and `str2` are the same, but the second character `'e'` of `str1` is less than the second character `'i'` of `str2`. Therefore, `str1 < str2` is `true`.
`str1 > "Hen"`	`false`	`str1 = "Hello"`. The first two characters of `str1` and `"Hen"` are the same, but the third character `'l'` of `str1` is less than the third character `'n'` of `"Hen"`. Therefore, `str1 > "Hen"` is `false`.
`str3 < "An"`	`true`	`str3 = "Air"`. The first characters of `str3` and `"An"` are the same, but the second character `'i'` of `"Air"` is less than the second character `'n'` of `"An"`. Therefore, `str3 < "An"` is `true`.

Relational Operators and the `string` Type (continued)

TABLE 4-3 Evaluating Logical Expressions with `string` Variables (continued)

Expression	Value	Explanation
`str1 == "hello"`	`false`	`str1 = "Hello"`. The first character `'H'` of `str1` is less than the first character `'h'` of `"hello"` because the ASCII value of `'H'` is 72, and the ASCII value of `'h'` is 104. Therefore, `str1 == "hello"` is `false`.
`str3 <= str4`	`true`	`str3 = "Air"` and `str4 = "Bill"`. The first character `'A'` of `str3` is less than the first character `'B'` of `str4`. Therefore, `str3 <= str4` is `true`.
`str2 > str4`	`true`	`str2 = "Hi"` and `str4 = "Bill"`. The first character `'H'` of `str2` is greater than the first character `'B'` of `str4`. Therefore, `str2 > str4` is `true`.

Relational Operators and the `string` Type (continued)

TABLE 4-4 Evaluating Logical Expressions with `string` Variables

Expression	Value	Explanation
`str4 >= "Billy"`	`false`	`str4 = "Bill"`. It has four characters and `"Billy"` has five characters. Therefore, `str4` is the shorter string. All four characters of `str4` are the same as the corresponding first four characters of `"Billy"`, and `"Billy"` is the larger string. Therefore, `str4 >= "Billy"` is `false`.
`str5 <= "Bigger"`	`true`	`str5 = "Big"`. It has three characters and `"Bigger"` has six characters. Therefore, `str5` is the shorter string. All three characters of `str5` are the same as the corresponding first three characters of `"Bigger"`, and `"Bigger"` is the larger string. Therefore, `str5 <= "Bigger"` is `true`.

Logical (Boolean) Operators and Logical Expressions

- Logical (Boolean) operators enable you to combine logical expressions

TABLE 4-5 Logical (Boolean) Operators in C++

Operator	Description
! ← unary	not
&& ← binary	and
|| ← binary	or

Logical (Boolean) Operators and Logical Expressions (continued)

TABLE 4-6 The ! (Not) Operator

Expression	!(Expression)
`true` (nonzero)	`false` (0)
`false` (0)	`true` (1)

EXAMPLE 4-2

Expression	Value	Explanation
`!('A' > 'B')`	`true`	Because `'A' > 'B'` is `false`, `!('A' > 'B')` is `true`.
`!(6 <= 7)`	`false`	Because `6 <= 7` is `true`, `!(6 <= 7)` is `false`.

TABLE 4-7 The && (And) Operator

Expression1	Expression2	Expression1 && Expression2
`true` (nonzero)	`true` (nonzero)	`true` (1)
`true` (nonzero)	`false` (0)	`false` (0)
`false` (0)	`true` (nonzero)	`false` (0)
`false` (0)	`false` (0)	`false` (0)

EXAMPLE 4-3

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

TABLE 4-8 The || (Or) Operator

Expression1	Expression2	Expression1 || Expression2
`true` (nonzero)	`true` (nonzero)	`true` (1)
`true` (nonzero)	`false` (0)	`true` (1)
`false` (0)	`true` (nonzero)	`true` (1)
`false` (0)	`false` (0)	`false` (0)

EXAMPLE 4-4

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`.

Order of Precedence

- Relational and logical operators are evaluated from left to right
- The associativity is left to right
- Parentheses can override precedence

Order of Precedence (continued)

TABLE 4-9 Precedence of Operators

Operators	Precedence
!, +, – (unary operators)	first
*, /, %	second
+, –	third
<, <=, >=, >	fourth
==, !=	fifth
&&	sixth
||	seventh
= (assignment operator)	last

Order of Precedence (continued)

EXAMPLE 4-5

Suppose you have the following declarations:

```
bool found = true;
bool flag = false;
int num = 1;
double x = 5.2;
double y = 3.4;
int a = 5, b = 8;
int n = 20;
char ch = 'B';
```

Order of Precedence (continued)

Expression	Value	Explanation
`!found`	`false`	Because `found` is `true`, `!found` is `false`.
`x > 4.0`	`true`	Because `x` is `5.2` and `5.2 > 4.0` is `true`, the expression `x > 4.0` evaluates to `true`.
`!num`	`false`	Because `num` is `1`, which is nonzero, `num` is `true` and so `!num` is `false`.
`!found && (x >= 0)`	`false`	In this expression, `!found` is `false`. Also, because `x` is `5.2` and `5.2 >= 0` is `true`, `x >= 0` is `true`. Therefore, the value of the expression `!found && (x >= 0)` is `false && true`, which evaluates to `false`.
`!(found && (x >= 0))`	`false`	In this expression, `found && (x >= 0)` is `true && true`, which evaluates to `true`. Therefore, the value of the expression `!(found && (x >= 0))` is `!true`, which evaluates to `false`.
`x + y <= 20.5`	`true`	Because `x + y = 5.2 + 3.4 = 8.6` and `8.6 <= 20.5`, it follows that `x + y <= 20.5` evaluates to `true`.

Order of Precedence (continued)

Expression	Value	Explanation
`(n >= 0) && (n <= 100)`	`true`	Here `n` is `20`. Because `20 >= 0` is `true`, `n >= 0` is `true`. Also, because `20 <= 100` is `true`, `n <= 100` is `true`. Therefore, the value of the expression `(n >= 0) && (n <= 100)` is `true && true`, which evaluates to `true`.
`('A' <= ch && ch <= 'Z')`	`true`	In this expression, the value of `ch` is `'B'`. Because `'A' <= 'B'` is `true`, `'A' <= ch` evaluates to `true`. Also, because `'B' <= 'Z'` is `true`, `ch <= 'Z'` evaluates to `true`. Therefore, the value of the expression `('A' <= ch && ch <= 'Z')` is `true && true`, which evaluates to `true`.
`(a + 2 <= b) && !flag`	`true`	Now `a + 2 = 5 + 2 = 7` and `b` is `8`. Because `7 <= 8` is `true`, the expression `a + 2 <= b` evaluates to `true`. Also, because `flag` is `false`, `!flag` is `true`. Therefore, the value of the expression `(a + 2 <= b) && !flag` is `true && true`, which evaluates to `true`.

Short-Circuit Evaluation

- Short-circuit evaluation: evaluation of a logical expression stops as soon as the value of the expression is known
- Example:

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

`int` Data Type and Logical (Boolean) Expressions

- Earlier versions of C++ did not provide built-in data types that had Boolean values
- Logical expressions evaluate to either 1 or 0
  - The value of a logical expression was stored in a variable of the data type `int`
- You can use the `int` data type to manipulate logical (Boolean) expressions

The `bool` Data Type and Logical (Boolean) Expressions

- The data type `bool` has logical (Boolean) values `true` and `false`
- `bool`, `true`, and `false` are reserved words
- The identifier `true` has the value `1`
- The identifier `false` has the value `0`

Logical (Boolean) Expressions

- Logical expressions can be unpredictable
- The following expression appears to represent a comparison of `0`, `num`, and `10`:

```
0 <= num <= 10
```

- It always evaluates to `true` because `0 <= num` evaluates to either `0` or `1`, and `0 <= 10` is `true` and `1 <= 10` is `true`
- A correct way to write this expression is:

```
0 <= num && num <= 10
```

Selection: `if` and `if...else`

- One-Way Selection
- Two-Way Selection
- Compound (Block of) Statements
- Multiple Selections: Nested `if`
- Comparing `if...else` Statements with a Series of `if` Statements

Selection: `if` and `if...else` (continued)

- Using Pseudocode to Develop, Test, and Debug a Program
- Input Failure and the `if` Statement
- Confusion Between the Equality Operator (==) and the Assignment Operator (=)
- Conditional Operator (?:)

One-Way Selection

- The syntax of one-way selection is:

```
if (expression)
    statement
```

- The statement is executed if the value of the expression is `true`
- The statement is bypassed if the value is `false`; program goes to the next statement
- `if` is a reserved word

One-Way Selection (continued)

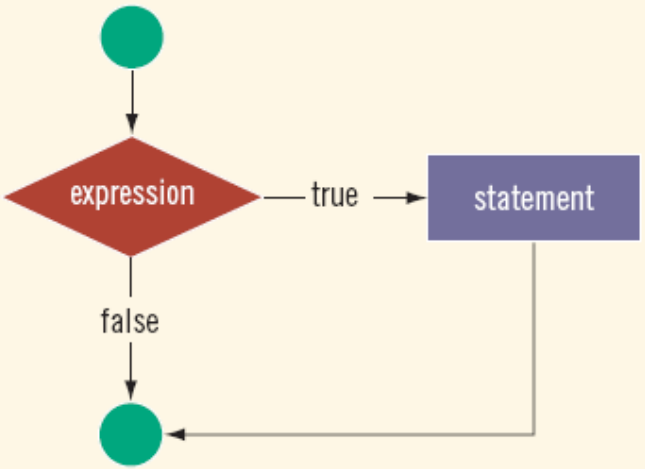


FIGURE 4-2 One-way selection

One-Way Selection (continued)

EXAMPLE 4-9

```
if (score >= 60)
    grade = 'P';
```

In this code, if the expression (`score >= 60`) evaluates to `true`, the assignment statement, `grade = 'P';`, executes. If the expression evaluates to `false`, the statements (if any) following the `if` structure execute. For example, if the value of `score` is 65, the value assigned to the variable `grade` is `'P'`.

EXAMPLE 4-10

The following C++ program finds the absolute value of an integer:

```
//Program: Absolute value of an integer

#include <iostream>

using namespace std;

int main()
{
    int number, temp;

    cout << "Line 1: Enter an integer: ";          //Line 1
    cin >> number;                                 //Line 2
    cout << endl;                                  //Line 3

    temp = number;                                 //Line 4

    if (number < 0)                                //Line 5
        number = -number;                          //Line 6

    cout << "Line 7: The absolute value of "
         << temp << " is " << number << endl;      //Line 7

    return 0;
}
```

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

```
Line 1: Enter an integer: -6734
Line 7: The absolute value of -6734 is 6734
```

One-Way Selection (continued)

EXAMPLE 4-11

Consider the following statement:

```
if score >= 60      //syntax error
   grade = 'P';
```

This statement illustrates an incorrect version of an `if` statement. The parentheses around the logical expression are missing, which is a syntax error.

EXAMPLE 4-12

Consider the following C++ statements:

```
if (score >= 60);          //Line 1
    grade = 'P';           //Line 2
```

Because there is a semicolon at the end of the expression (see Line 1), the `if` statement in Line 1 terminates. The action of this `if` statement is null, and the statement in Line 2 is not part of the `if` statement in Line 1. Hence, the statement in Line 2 executes regardless of how the `if` statement evaluates.

Two-Way Selection

- Two-way selection takes the form:

```
if (expression)
    statement1
else
    statement2
```

- If expression is `true`, `statement1` is executed; otherwise, `statement2` is executed
  - `statement1` and `statement2` are any C++ statements
- `else` is a reserved word

Two-Way Selection (continued)

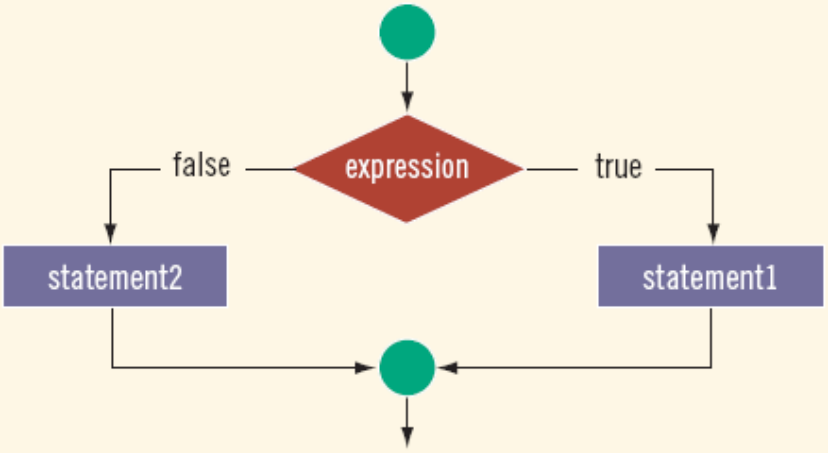


FIGURE 4-3 Two-way selection

Two-Way Selection (continued)

EXAMPLE 4-13

Consider the following statements:

```
if (hours > 40.0)                                  //Line 1
    wages = 40.0 * rate +
            1.5 * rate * (hours - 40.0);           //Line 2
else                                               //Line 3
    wages = hours * rate;                          //Line 4
```

If the value of the variable `hours` is greater than `40.0`, then the `wages` include overtime payment. Suppose that `hours` is `50`. The expression in the `if` statement, in Line 1, evaluates to `true`, so the statement in Line 2 executes. On the other hand, if `hours` is `30`, or any number less than or equal to `40`, the expression in the `if` statement, in Line 1, evaluates to `false`. In this case, the program skips the statement in Line 2 and executes the statement in Line 4—that is, the statement following the reserved word `else` executes.

Two-Way Selection (continued)

EXAMPLE 4-14

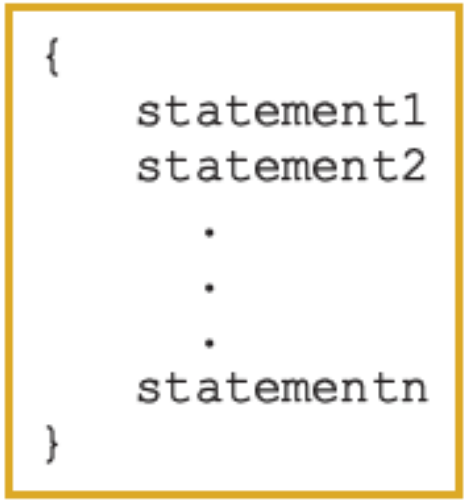
The following statements show an example of a syntax error:

```
if (hours > 40.0);                                  //Line 1
    wages = 40.0 * rate +
            1.5 * rate * (hours - 40.0);            //Line 2
else                                                //Line 3
    wages = hours * rate;                           //Line 4
```

The semicolon at the end of the `if` statement (see Line 1) ends the `if` statement, so the statement in Line 2 separates the `else` clause from the `if` statement. That is, `else` is all by itself. Because there is no stand-alone `else` statement in C++, this code generates a syntax error.

Compound (Block of) Statement

- Compound statement (block of statements):



- A compound statement is a single statement

Compound (Block of) Statement (continued)

```
if (age > 18)
{
   cout << "Eligible to vote." << endl;
   cout << "No longer a minor." << endl;
}
else
{
   cout << "Not eligible to vote." << endl;
   cout << "Still a minor." << endl;
}
```

Multiple Selections: Nested `if`

- Nesting: one control statement in another
- An `else` is associated with the most recent `if` that has not been paired with an `else`

EXAMPLE 4-18

Suppose that `balance` and `interestRate` are variables of type `double`. The following statements determine the `interestRate` depending on the value of the `balance`:

```
if (balance > 50000.00)                    //Line 1
    interestRate = 0.07;                   //Line 2
else                                       //Line 3
    if (balance >= 25000.00)               //Line 4
        interestRate = 0.05;               //Line 5
    else                                   //Line 6
        if (balance >= 1000.00)            //Line 7
            interestRate = 0.03;           //Line 8
        else                               //Line 9
            interestRate = 0.00;           //Line 10
```

To avoid excessive indentation, the code in Example 4-18 can be rewritten as follows:

```
if (balance > 50000.00)                    //Line 1
    interestRate = 0.07;                   //Line 2
else if (balance >= 25000.00)              //Line 3
    interestRate = 0.05;                   //Line 4
else if (balance >= 1000.00)               //Line 5
    interestRate = 0.03;                   //Line 6
else                                       //Line 7
    interestRate = 0.00;                   //Line 8
```

Multiple Selections: Nested `if` (continued)

EXAMPLE 4-19

Assume that `score` is a variable of type `int`. Based on the value of `score`, the following code outputs the grade:

```
if (score >= 90)
    cout << "The grade is A." << endl;
else if (score >= 80)
    cout << "The grade is B." << endl;
else if (score >= 70)
    cout << "The grade is C." << endl;
else if (score >= 60)
    cout << "The grade is D." << endl;
else
    cout << "The grade is F." << endl;
```

Comparing `if...else` Statements with a Series of `if` Statements

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
```

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;
```

Confusion Between == and =

- C++ allows you to use any expression that can be evaluated to either `true` or `false` as an expression in the `if` statement:

```
if (x = 5)
    cout << "The value is five." << endl;
```

- The appearance of = in place of == resembles a *silent killer*
  - It is not a syntax error
  - It is a logical error

Conditional Operator (?:)

- Conditional operator (`?:`) takes three arguments
  - Ternary operator
- Syntax for using the conditional operator:
  `expression1 ? expression2 : expression3`
- If `expression1` is `true`, the result of the conditional expression is `expression2`
  - Otherwise, the result is `expression3`

`switch` Structures

- `switch` structure: alternate to if-else
- `switch` (integral) expression is evaluated first
- Value of the expression determines which corresponding action is taken
- Expression is sometimes called the selector

```
switch (expression)
{
case value1:
    statements1
    break;
case value2:
    statements2
    break;
    .
    .
    .
case valuen:
    statementsn
    break;
default:
    statements
}
```

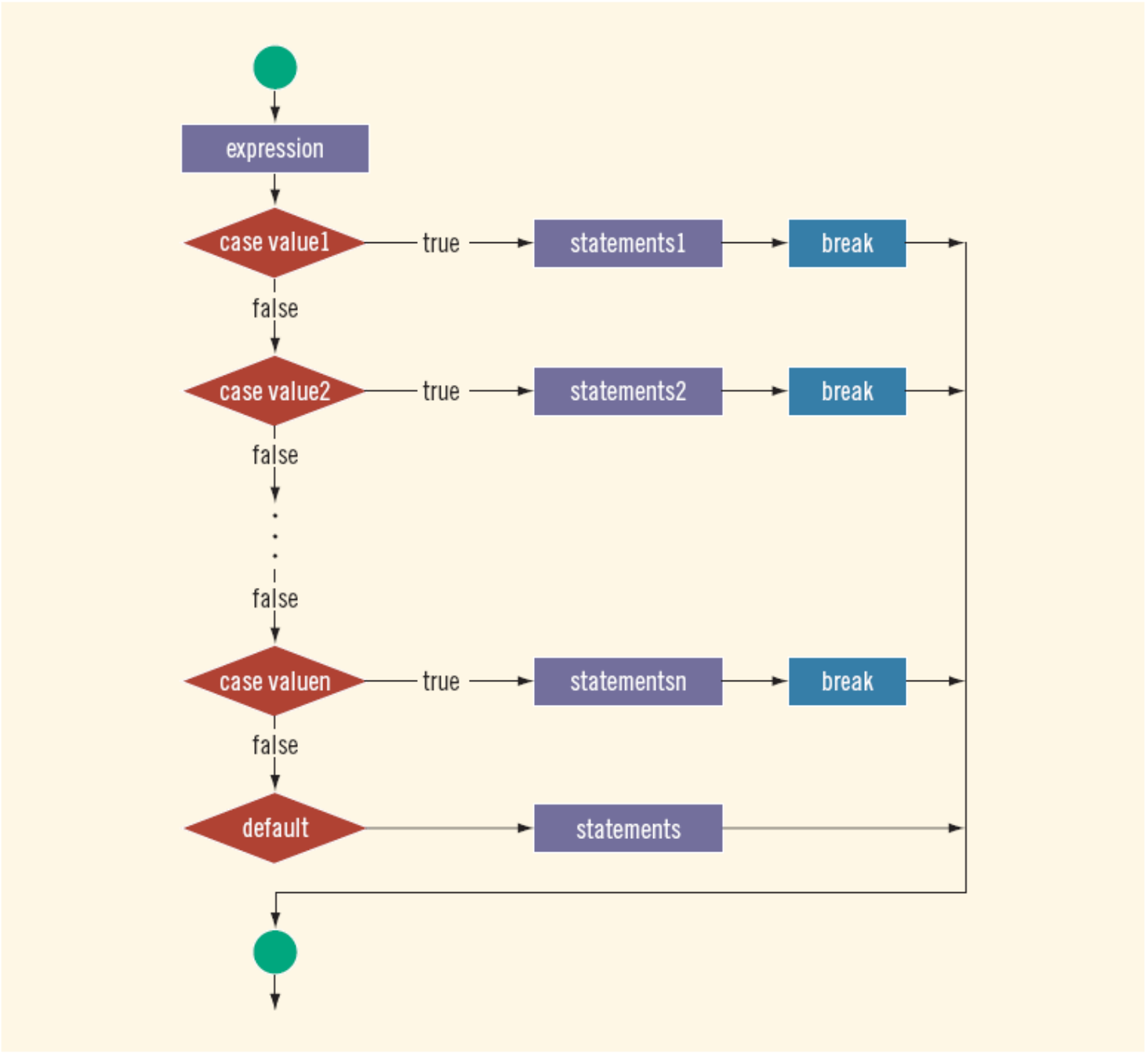


FIGURE 4-4 `switch` statement

`switch` Structures (continued)

- One or more statements may follow a case label
- Braces are not needed to turn multiple statements into a single compound statement
- The `break` statement may or may not appear after each statement
- `switch`, `case`, `break`, and `default` are reserved words

EXAMPLE 4-24

Consider the following statements, where grade is a variable of type `char`:

```
switch (grade)
{
case 'A':
    cout << "The grade is 4.0.";
    break;
case 'B':
    cout << "The grade is 3.0.";
    break;
case 'C':
    cout << "The grade is 2.0.";
    break;
case 'D':
    cout << "The grade is 1.0.";
    break;
case 'F':
    cout << "The grade is 0.0.";
    break;
default:
    cout << "The grade is invalid.";
}
```

In this example, the expression in the `switch` statement is a variable identifier. The variable `grade` is of type `char`, which is an integral type. The possible values of `grade` are `'A'`, `'B'`, `'C'`, `'D'`, and `'F'`. Each `case` label specifies a different action to take, depending on the value of `grade`. If the value of `grade` is `'A'`, the output is:

```
The grade is 4.0.
```

Summary

- Control structures alter normal control flow
- Most common control structures are selection and repetition
- Relational operators: `==`, `<`, `<=`, `>`, `>=`, `!=`
- Logical expressions evaluate to `1` (`true`) or `0` (`false`)
- Logical operators: `!` (not), `&&` (and), `||` (or)

Summary (continued)

- Two selection structures: one-way selection and two-way selection
- The expression in an `if` or `if...else` structure is usually a logical expression
- No stand-alone `else` statement in C++
  - Every `else` has a related `if`
- A sequence of statements enclosed between braces, `{` and `}`, is called a compound statement or block of statements

Summary (continued)

- Using assignment in place of the equality operator creates a semantic error
- `switch` structure handles multiway selection
- `break` statement ends `switch` statement
- Use `assert` to terminate a program if certain conditions are not met

C++ Programming: From Problem Analysis to Program Design, Fourth Edition

Chapter 5: Control Structures II
(Repetition)

Objectives

In this chapter, you will:

- Learn about repetition (looping) control structures
- Explore how to construct and use count-controlled, sentinel-controlled, flag-controlled, and EOF-controlled repetition structures
- Examine `break` and `continue` statements
- Discover how to form and use nested control structures

Why Is Repetition Needed?

- Repetition allows you to efficiently use variables
- Can input, add, and average multiple numbers using a limited number of variables
- For example, to add five numbers:
  - Declare a variable for each number, input the numbers and add the variables together
  - Create a loop that reads a number into a variable and adds it to a variable that contains the sum of the numbers

`while` Looping (Repetition) Structure

- The general form of the `while` statement is:

```
while (expression)
    statement
```

  `while` is a reserved word
- Statement can be simple or compound
- Expression acts as a decision maker and is usually a logical expression
- Statement is called the body of the loop
- The parentheses are part of the syntax

`while` Looping (Repetition) Structure (continued)

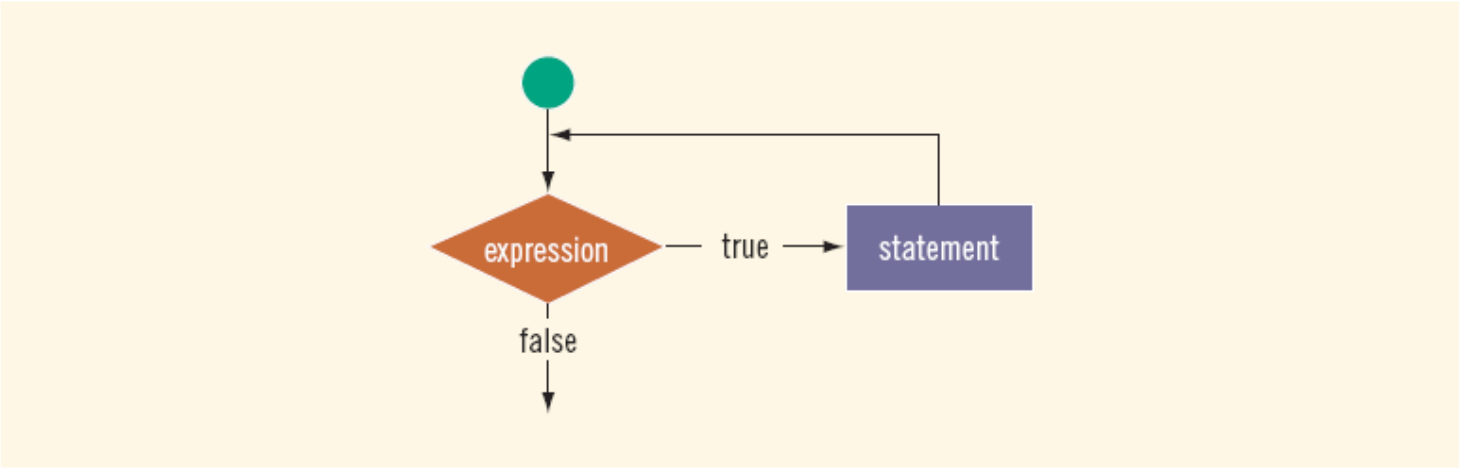


FIGURE 5-1 `while` loop

- Infinite loop: continues to execute endlessly
  - Avoided by including statements in loop body that assure exit condition is eventually `false`

`while` Looping (Repetition) Structure (continued)

EXAMPLE 5-1

Consider the following C++ program segment:

```
i = 0;                          //Line 1

while (i <= 20)                 //Line 2
{
    cout << i << " ";           //Line 3
    i = i + 5;                  //Line 4
}

cout << endl;
```

Sample Run:

```
0 5 10 15 20
```

Designing `while` Loops

EXAMPLE 5-2

Consider the following C++ program segment:

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

It is easy to overlook the difference between this example and Example 5-1. In this example, in Line 1, `i` is set to 20. Because `i` is 20, the expression `i < 20` in the `while` statement (Line 2) evaluates 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.

Case 1: Counter-Controlled `while` Loops

- If you know exactly how many pieces of data need to be read, the `while` loop becomes a counter-controlled loop

```
counter = 0;          //initialize the loop control variable

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

Case 2: Sentinel-Controlled `while` Loops

- Sentinel variable is tested in the condition and loop ends when sentinel is encountered

```
cin >> variable;                //initialize the loop control variable

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

Case 3: Flag-Controlled `while` Loops

- A flag-controlled `while` loop uses a `bool` variable to control the loop
- The flag-controlled `while` loop takes the form:

```
found = false;       //initialize the loop control variable

while (!found)       //test the loop control variable
{
    .
    .
    .
    if (expression)
        found = true; //update the loop control variable
    .
    .
    .
}
```

More on Expressions in `while` Statements

- The expression in a `while` statement can be complex
  - For example:

```
while ((noOfGuesses < 5) && (!isGuessed))
{
    …
}
```

`for` Looping (Repetition) Structure

- The general form of the `for` statement is:

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

- The `initial statement`, `loop condition`, and `update statement` are called `for` loop control statements
  - `initial statement` usually initializes a variable (called the `for` **loop control**, or `for` **indexed**, **variable**)
- In C++, `for` is a reserved word

`for` Looping (Repetition) Structure (continued)

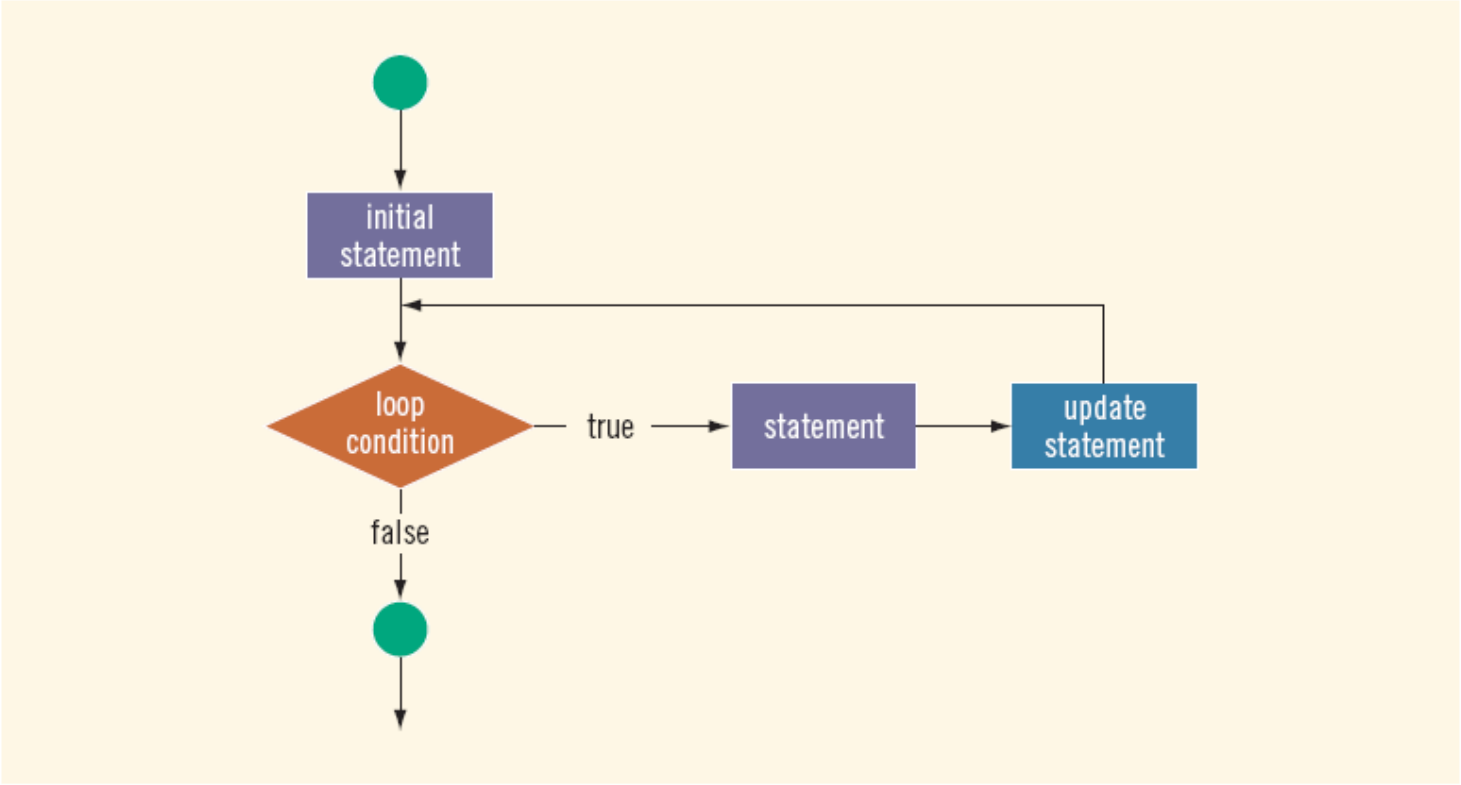


FIGURE 5-2 `for` loop

`for` Looping (Repetition) Structure (continued)

EXAMPLE 5-7

The following `for` loop prints the first 10 non negative integers:

```
for (i = 0; i < 10; i++)
    cout << i << " ";
cout << endl;
```

EXAMPLE 5-8

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;
```

`for` Looping (Repetition) Structure (continued)

- C++ allows you to use fractional values for loop control variables of the `double` type
  - Results may differ
- The following is a semantic error:

EXAMPLE 5-9

The following `for` loop executes five empty statements:

```
for (i = 0; i < 5; i++);      //Line 1
    cout << "*" << endl;      //Line 2
```

- The following is a legal `for` loop:

```
for (;;)
    cout << "Hello" << endl;
```

`for` Looping (Repetition) Structure (continued)

EXAMPLE 5-10

You can count backward using a `for` loop if the `for` loop control expressions are set correctly.

For example, consider the following `for` loop:

```
for (i = 10; i >= 1; i--)
    cout << " " << i;
cout << endl;
```

The output is:

```
10 9 8 7 6 5 4 3 2 1
```

EXAMPLE 5-11

You can increment (or decrement) the loop control variable by any fixed number. In the following `for` loop, the variable is initialized to `1`; at the end of the `for` loop, `i` is incremented by `2`. This `for` loop outputs the first `10` positive odd integers.

```
for (i = 1; i <= 20; i = i + 2)
    cout << " " << i;
cout << endl;
```

do...while Looping (Repetition) Structure

- General form of a do...while:

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

- The statement executes first, and then the expression is evaluated
- To avoid an infinite loop, body must contain a statement that makes the expression false
- The statement can be simple or compound
- Loop always iterates at least once

`do...while` Looping (Repetition) Structure (continued)

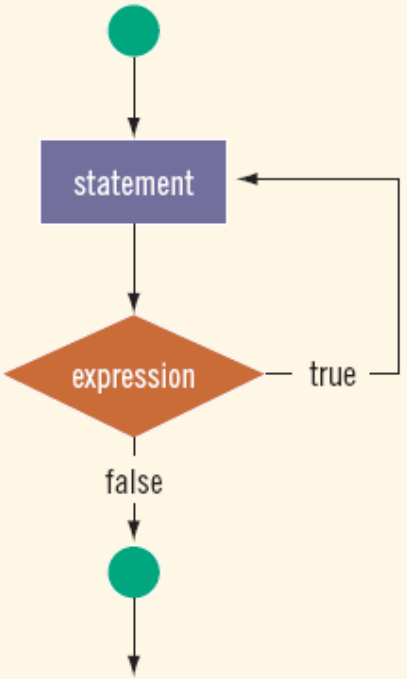


FIGURE 5-3 `do...while` loop

do...while Looping (Repetition) Structure (continued)

EXAMPLE 5-15

```
i = 0;

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

The output of this code is:

```
0 5 10 15 20
```

EXAMPLE 5-16

Consider the following two loops:

a.
```
i = 11;
while (i <= 10)
{
    cout << i << " ";
    i = i + 5;
}
cout << endl;
```

b.
```
i = 11;
do
{
    cout << i << " ";
    i = i + 5;
}
while (i <= 10);

cout << endl;
```

In (a), the `while` loop produces nothing. In (b), the `do...while` loop outputs the number `11` and also changes the value of `i` to 16.

Divisibility Test by 3 and 9

```
sum = 0;

do
{
    sum = sum + num % 10; //extract the last digit
                          //and add it to sum
    num = num / 10;       //remove the last digit
}
while (num > 0);

cout << "The sum of the digits = " << sum << endl;

if (sum % 3 == 0)
    cout << temp << " is divisible by 3" << endl;
else
    cout << temp << " is not divisible by 3" << endl;

if (sum % 9 == 0)
    cout << temp << " is divisible by 9" << endl;
else
    cout << temp << " is not divisible by 9" << endl;
```

Choosing the Right Looping Structure

- All three loops have their place in C++
  - If you know or can determine in advance the number of repetitions needed, the `for` loop is the correct choice
  - If you do not know and cannot determine in advance the number of repetitions needed, and it could be zero, use a `while` loop
  - If you do not know and cannot determine in advance the number of repetitions needed, and it is at least one, use a `do...while` loop

`break` and `continue` Statements

- `break` and `continue` alter the flow of control
- `break` statement is used for two purposes:
  - To exit early from a loop
    - Can eliminate the use of certain (flag) variables
  - To skip the remainder of the `switch` structure
- After the `break` statement executes, the program continues with the first statement after the structure

`break` & `continue` Statements (continued)

- `continue` is used in `while`, `for`, and `do`…`while` structures
- When executed in a loop
  - It skips remaining statements and proceeds with the next iteration of the loop

Nested Control Structures

- To create the following pattern:

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

- We can use the following code:

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

Nested Control Structures (continued)

- What is the result if we replace the first `for` statement with the following?

```
for (i = 5; i >= 1; i--)
```

- Answer:

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

Summary

- C++ has three looping (repetition) structures:
  - `while`, `for`, and `do`…`while`
- `while`, `for`, and `do` are reserved words
- `while` and `for` loops are called pretest loops
- `do`...`while` loop is called a posttest loop
- `while` and `for` may not execute at all, but `do`...`while` always executes at least once

Summary (continued)

- `while`: expression is the decision maker, and the statement is the body of the loop
- A `while` loop can be:
  - Counter-controlled
  - Sentinel-controlled
  - EOF-controlled
- In the Windows console environment, the end-of-file marker is entered using `Ctrl+z`

Summary (continued)

- `for` loop: simplifies the writing of a counter-controlled while loop
  - Putting a semicolon at the end of the `for` loop is a semantic error
- Executing a `break` statement in the body of a loop immediately terminates the loop
- Executing a `continue` statement in the body of a loop skips to the next iteration

C++ Programming: From Problem Analysis to Program Design, Fourth Edition

Chapter 6: User-Defined Functions I

Objectives

In this chapter, you will:

- 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
- Explore how to construct and use a value-returning, user-defined function in a program

Introduction

- Functions are like building blocks
- They allow complicated programs to be divided into manageable pieces
- Some advantages of functions:
  - A programmer can focus on just that part of the program and construct it, debug it, and perfect it
  - Different people can work on different functions simultaneously
  - Can be re-used (even in different programs)
  - Enhance program readability

Introduction (continued)

- Functions
  - Called modules
  - Like miniature programs
  - Can be put together to form a larger program

Predefined Functions

- In algebra, a function is defined as a rule or correspondence between values, called the function's arguments, and the unique value of the function associated with the arguments
  - If `f(x) = 2x + 5`, then `f(1) = 7`, `f(2) = 9`, and `f(3) = 11`
    - `1`, `2`, and `3` are arguments
    - `7`, `9`, and `11` are the corresponding values

Predefined Functions (continued)

- Some of the predefined mathematical functions are:

```
sqrt(x)
pow(x, y)
floor(x)
```

- Predefined functions are organized into separate libraries
- I/O functions are in `iostream` header
- Math functions are in `cmath` header

Predefined Functions (continued)

- `pow(x,y)` calculates x^y
  - `pow(2, 3) = 8.0`
  - Returns a value of type `double`
  - `x` and `y` are the parameters (or arguments)
    - The function has two parameters
- `sqrt(x)` calculates the nonnegative square root of `x`, for `x >= 0.0`
  - `sqrt(2.25)` is `1.5`
  - Type `double`

Predefined Functions (continued)

- The `floor` function `floor(x)` calculates largest whole number not greater than `x`
  - `floor(48.79)` is `48.0`
  - Type `double`
  - Has only one parameter

Predefined Functions (continued)

TABLE 6-1 Predefined Functions

Function	Header File	Purpose	Parameter(s) Type	Result
`abs(x)`	`<cstdlib>`	Returns the absolute value of its argument: `abs(-7) = 7`	`int`	`int`
`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`
`fabs(x)`	`<cmath>`	Returns the absolute value of its argument: `fabs(-5.67) = 5.67`	`double`	`double`

Predefined Functions (continued)

TABLE 6-1 Predefined Functions (continued)

Function	Header File	Purpose	Parameter(s) Type	Result
`floor(x)`	`<cmath>`	Returns the largest whole number that is not greater than `x`: `floor(45.67) = 45.00`	`double`	`double`
`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`
`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`

EXAMPLE 6-1

```
//How to use predefined functions.

#include <iostream>
#include <cmath>
#include <cctype>
#include <cstdlib>

using namespace std;

int main()
{
    int    x;
    double u, v;

    cout << "Line 1: Uppercase a is "
         << static_cast<char>(toupper('a'))
         << endl;                                          //Line 1

    u = 4.2;                                               //Line 2
    v = 3.0;                                               //Line 3
    cout << "Line 4: " << u << " to the power of "
         << v << " = " << pow(u, v) << endl;               //Line 4

    cout << "Line 5: 5.0 to the power of 4 = "
         << pow(5.0, 4) << endl;                           //Line 5

    u = u + pow(3.0, 3);                                   //Line 6
    cout << "Line 7: u = " << u << endl;                   //Line 7

    x = -15;                                               //Line 8
    cout << "Line 9: Absolute value of " << x
         << " = " << abs(x) << endl;                       //Line 9

    return 0;
}
```

Predefined Functions (continued)

- Example 6-1 sample run:

```
Line 1: Uppercase a is A
Line 4: 4.2 to the power of 3 = 74.088
Line 5: 5.0 to the power of 4 = 625
Line 7: u = 31.2
Line 9: Absolute value of -15 = 15
```

- To use these functions you must:
  - Include the appropriate header file in your program using the include statement
  - Know the following items:
    - Name of the function
    - Number of parameters, if any
    - Data type of each parameter
    - Data type of the value returned: called the type of the function

User-Defined Functions

- Value-returning functions: have a return type
  - Return a value of a specific data type using the `return` statement
- Void functions: do not have a return type
  - *Do not* use a `return` statement to return a value

Value-Returning Functions (continued)

- Because the value returned by a value-returning function is unique, we must:
  - Save the value for further calculation
  - Use the value in some calculation
  - Print the value
- A value-returning function is used in an assignment or in an output statement
- One more thing is associated with functions:
  - The code required to accomplish the task

Value-Returning Functions (continued)

```
int abs(int number)
int abs(int number)
{
    if (number < 0)
        number = -number;

    return number;
}

double pow(double base, double exponent)

double u = 2.5;
double v = 3.0;
double x, y, w;

x = pow(u, v);              //Line 1
y = pow(2.0, 3.2);          //Line 2
w = pow(u, 7);              //Line 3
```

Value-Returning Functions (continued)

- Heading: first four properties above
  - Example: `int abs(int number)`
- Formal Parameter: variable declared in the heading
  - Example: `number`
- Actual Parameter: variable or expression listed in a call to a function
  - Example: `x = pow(u, v)`

Syntax: Value-Returning Function

- Syntax:

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

- `functionType` is also called the data type or return type

Syntax: Formal Parameter List

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

Function Call

```
functionName(actual parameter list)
```

Syntax: Actual Parameter List

- The syntax of the actual parameter list is:

```
expression or variable, expression or variable, ...
```

- Formal parameter list can be empty:

```
functionType functionName()
```

- A call to a value-returning function with an empty formal parameter list is:

```
functionName()
```

`return` Statement

- Once a value-returning function computes the value, the function returns this value via the `return` statement
  - It passes this value outside the function via the `return` statement

Syntax: `return` Statement

- The `return` statement has the following syntax:

```
return expr;
```

- In C++, `return` is a reserved word
- When a return statement executes
  - Function immediately terminates
  - Control goes back to the caller
- When a `return` statement executes in the function `main`, the program terminates

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

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

    return max;
}
```

You can also write this function as follows:

```
double larger(double x, double y)
{
    if (x >= y)
        return x;
    else
        return y;
}
```

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

    return y;
}
```

NOTE

1. In the definition of the function `larger`, `x` and `y` are formal parameters.
2. The `return` statement can appear anywhere in the function. Recall that once a `return` statement executes, all subsequent statements are skipped. Thus, it's a good idea to return the value as soon as it is computed.

Function Prototype

- Function prototype: function heading without the body of the function
- Syntax:

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

- It is not necessary to specify the variable name in the parameter list
- The data type of each parameter must be specified

```
//Program: Largest of three numbers

#include <iostream>

using namespace std;

double larger(double x, double y);
double compareThree(double x, double y, double z);

int main()
{
    double one, two;                                    //Line 1

    cout << "Line 2: The larger of 5 and 10 is "
         << larger(5, 10) << endl;                      //Line 2

    cout << "Line 3: Enter two numbers: ";              //Line 3
    cin >> one >> two;                                  //Line 4
    cout << endl;                                       //Line 5

    cout << "Line 6: The larger of " << one
         << " and " << two << " is "
         << larger(one, two) << endl;                   //Line 6

    cout << "Line 7: The largest of 23, 34, and "
         << "12 is " << compareThree(23, 34, 12)
         << endl;                                       //Line 7

    return 0;
}
```

Function Prototype (continued)

```
double larger(double x, double y)
{
    if (x >= y)
        return x;
    else
        return y;
}

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 2: The larger of 5 and 10 is 10
Line 3: Enter two numbers: 25 73

Line 6: The larger of 25 and 73 is 73
Line 7: The largest of 23, 34, and 12 is 34
```

Flow of Execution

- Execution always begins at the first statement in the function `main`
- Other functions are executed only when they are called
- Function prototypes appear before any function definition
  - The compiler translates these first
- The compiler can then correctly translate a function call

Flow of Execution (continued)

- A function call results in transfer of control to the first statement in the body of the called function
- After the last statement of a function is executed, control is passed back to the point immediately following the function call
- A value-returning function returns a value
  - After executing the function the returned value replaces the function call statement

Summary

- Functions (modules) are miniature programs
  - Divide a program into manageable tasks
- C++ provides the standard functions
- Two types of user-defined functions: value-returning functions and void functions
- Variables defined in a function heading are called formal parameters
- Expressions, variables, or constant values in a function call are called actual parameters

Summary (continued)

- In a function call, the number of actual parameters and their types must match with the formal parameters in the order given
- To call a function, use its name together with the actual parameter list
- Function heading and the body of the function are called the definition of the function
- If a function has no parameters, you need empty parentheses in heading and call
- A value-returning function returns its value via the `return` statement

Summary (continued)

- A prototype is the function heading without the body of the function; prototypes end with the semicolon
- Prototypes are placed before every function definition, including `main`
- User-defined functions execute only when they are called
- In a call statement, specify only the actual parameters, not their data types

C++ Programming: From Problem Analysis to Program Design, Fourth Edition

Chapter 7: User-Defined Functions II

Objectives

In this chapter, you will:

- Learn how to construct and use void functions in a program
- Discover the difference between value and reference parameters
- Explore reference parameters and value-returning functions
- Learn about the scope of an identifier

Objectives (continued)

- Examine the difference between local and global identifiers
- Discover static variables
- Learn function overloading
- Explore functions with default parameters

Void Functions

- Void functions and value-returning functions have similar structures
  - Both have a heading part and a statement part
- User-defined void functions can be placed either before or after the function `main`
- If user-defined void functions are placed after the function `main`
  - The function prototype must be placed before the function `main`

Void Functions (continued)

- A void function does not have a return type
  - `return` statement without any value is typically used to exit the function early
- Formal parameters are optional
- A call to a void function is a stand-alone statement

Void Functions without Parameters

- Function definition syntax:

```
void functionName()
{
    statements
}
```

- `void` is a reserved word
- Function call syntax:

```
functionName();
```

Void Functions with Parameters

- Function definition syntax:

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

- Formal parameter list syntax:

```
dataType& variable, dataType& variable, ...
```

- Function call syntax:

```
functionName(actual parameter list);
```

- Actual parameter list syntax:

```
expression or variable,expression or variable, ...
```

Void Functions with Parameters (continued)

EXAMPLE 7-2

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

The function `funexp` has four parameters.

EXAMPLE 7-3

```
void expfun(int one, int& two, char three, double& four)
{
     .
     .
     .
}
```

The function `expfun` has four parameters: (1) `one`, a value parameter of type `int`; (2) `two`, a reference parameter of type `int`; (3) `three`, a value parameter of type `char`, and (4) `four`, a reference parameter of type `double`.

Void Functions with Parameters (continued)

- Value parameter: a formal parameter that receives a copy of the content of corresponding actual parameter
- Reference parameter: a formal parameter that receives the location (memory address) of the corresponding actual parameter

Value Parameters

- If a formal parameter is a value parameter
  - The value of the corresponding actual parameter is copied into it
- The value parameter has its own copy of the data
- During program execution
  - The value parameter manipulates the data stored in its own memory space

Reference Variables as Parameters

- If a formal parameter is a reference parameter
  - It receives the memory address of the corresponding actual parameter
- A reference parameter stores the address of the corresponding actual parameter
- During program execution to manipulate data
  - The address stored in the reference parameter directs it to the memory space of the corresponding actual parameter

Reference Variables as Parameters (continued)

- Reference parameters can:
  - Pass one or more values from a function
  - Change the value of the actual parameter
- Reference parameters are useful in three situations:
  - Returning more than one value
  - Changing the actual parameter
  - When passing the address would save memory space and time

Calculate Grade

```
//This program reads a course score and prints the
//associated course grade.

#include <iostream>
using namespace std;

void getScore(int& score);
void printGrade(int score);

int main()
{
    int courseScore;

    cout << "Line 1: Based on the course score, \n"
         << "    this program computes the "
         << "course grade." << endl;                    //Line 1

    getScore(courseScore);                              //Line 2

    printGrade(courseScore);                            //Line 3

    return 0;
}
```

Calculate Grade (continued)

```
void getScore(int& score)
{
    cout << "Line 4: Enter course score: ";          //Line 4
    cin >> score;                                    //Line 5
    cout << endl << "Line 6: Course score is "
         << score << endl;                           //Line 6
}

void printGrade(int cScore)
{
    cout << "Line 7: Your grade for the course is "; //Line 7

    if (cScore >= 90)                                //Line 8
        cout << "A." << endl;
    else if (cScore >= 80)
        cout << "B." << endl;
    else if(cScore >= 70)
        cout << "C." << endl;
    else if (cScore >= 60)
        cout << "D." << endl;
    else
        cout << "F." << endl;
}
```

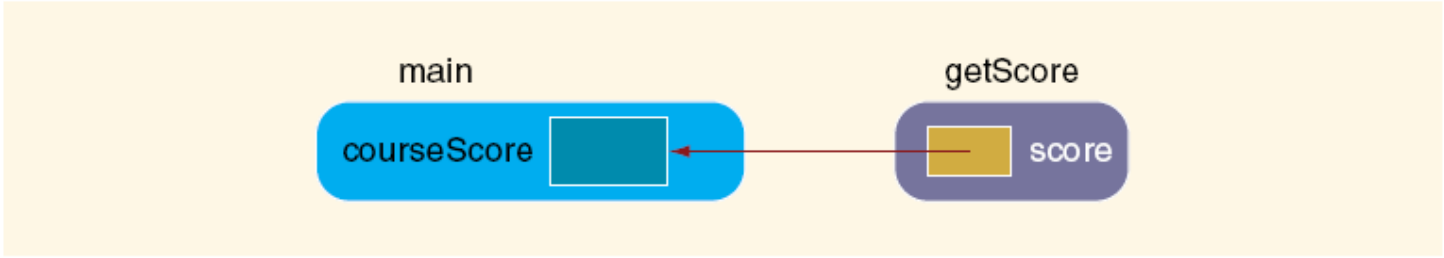


FIGURE 7-1 Variable `courseScore` and the parameter `score`

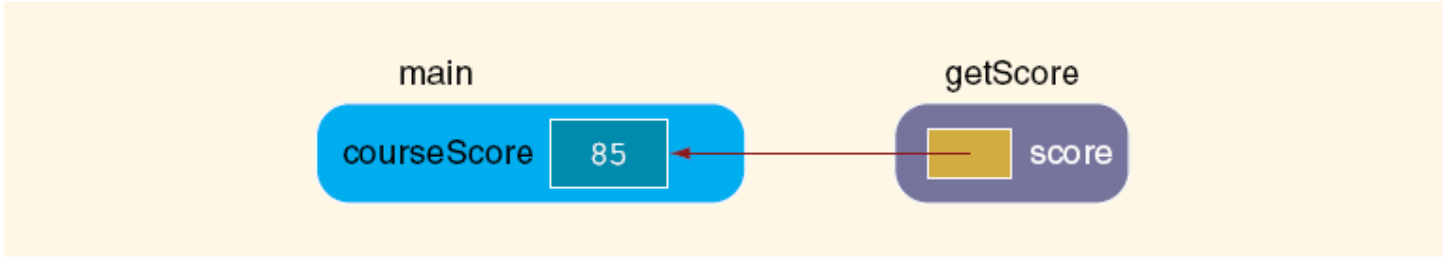


FIGURE 7-2 Variable `courseScore` and the parameter `score` after the statement in Line 5 executes

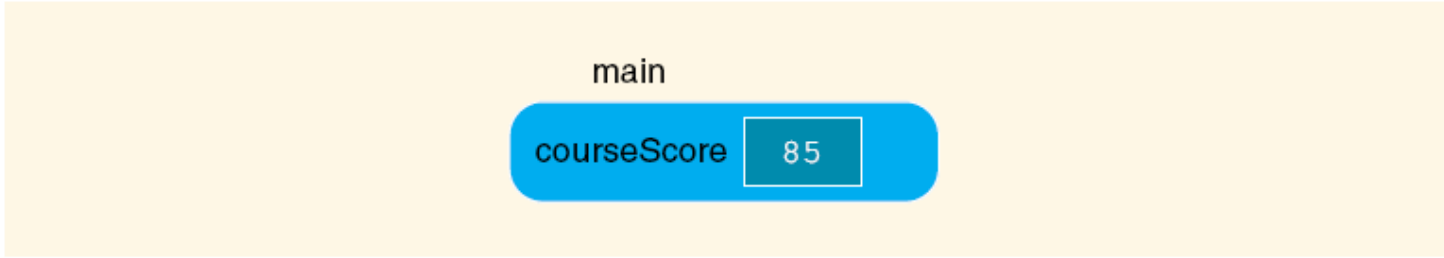


FIGURE 7-3 Variable `courseScore` after the statement in Line 6 is executed and control goes back to `main`

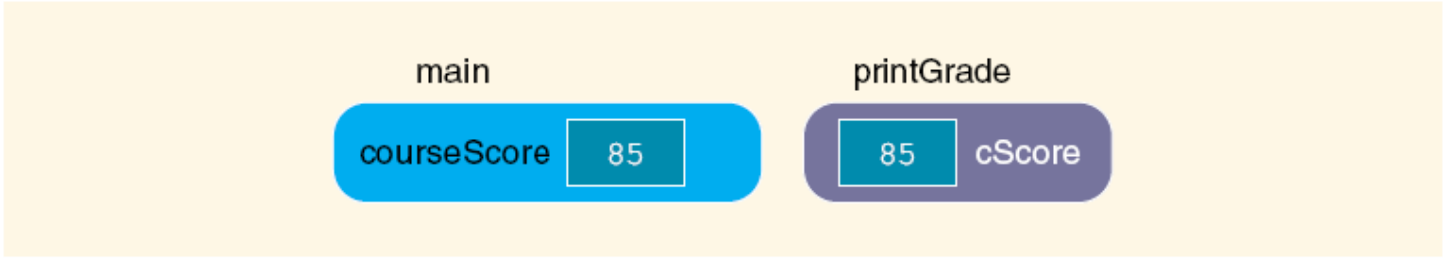


FIGURE 7-4 Variable `courseScore` and the parameter `cScore`

Value and Reference Parameters and Memory Allocation

- When a function is called
  - Memory for its formal parameters and variables declared in the body of the function (called local variables) is allocated in the function data area
- In the case of a value parameter
  - The value of the actual parameter is copied into the memory cell of its corresponding formal parameter

Value and Reference Parameters and Memory Allocation (continued)

- In the case of a reference parameter
  - The address of the actual parameter passes to the formal parameter
- Content of formal parameter is an address
- During execution, changes made by the formal parameter permanently change the value of the actual parameter
- Stream variables (e.g., `ifstream`) should be passed by reference to a function

EXAMPLE 7-7

```
#include <iostream>

using namespace std;

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

int main()
{
    int num1, num2;
    char ch;

    num1 = 10;                                          //Line 1
    num2 = 15;                                          //Line 2
    ch = 'A';                                           //Line 3

    cout << "Line 4: Inside main: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                 //Line 4

    funOne(num1, num2, ch);                             //Line 5

    cout << "Line 6: After funOne: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                 //Line 6

    funTwo(num2, 25, ch);                               //Line 7

    cout << "Line 8: After funTwo: num1 = " << num1
         << ", num2 = " << num2 << ", and ch = "
         << ch << endl;                                 //Line 8

    return 0;
}
```

```
void funOne(int a, int& b, char v)
{
    int one;

    one = a;                                                    //Line 9
    a++;                                                        //Line 10
    b = b * 2;                                                  //Line 11
    v = 'B';                                                    //Line 12

    cout << "Line 13: Inside funOne: a = " << a
         << ", b = " << b << ", v = " << v
         << ", and one = " << one << endl;                      //Line 13
}

void funTwo(int& x, int y, char& w)
{
    x++;                                                        //Line 14
    y = y * 2;                                                  //Line 15
    w = 'G';                                                    //Line 16

    cout << "Line 17: Inside funTwo: x = " << x
         << ", y = " << y << ", and w = " << w
         << endl;                                               //Line 17
}
```

Sample Run:

```
Line 4: Inside main: num1 = 10, num2 = 15, and ch = A
Line 13: Inside funOne: a = 11, b = 30, v = B, and one = 10
Line 6: After funOne: num1 = 10, num2 = 30, and ch = A
Line 17: Inside funTwo: x = 31, y = 50, and w = G
Line 8: After funTwo: num1 = 10, num2 = 31, and ch = G
```

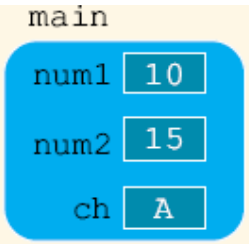


FIGURE 7-5 Values of the variables after the statement in Line 3 executes

```
one = a;                                                    //Line 9
```

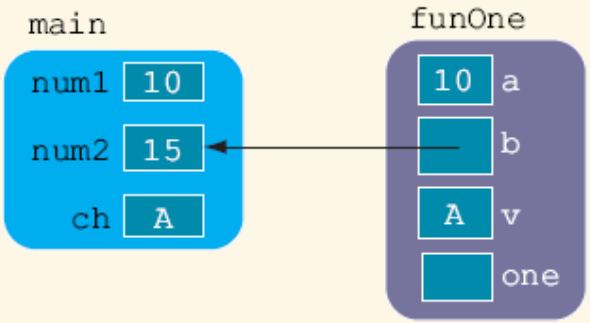


FIGURE 7-6 Values of the variables just before the statement in Line 9 executes

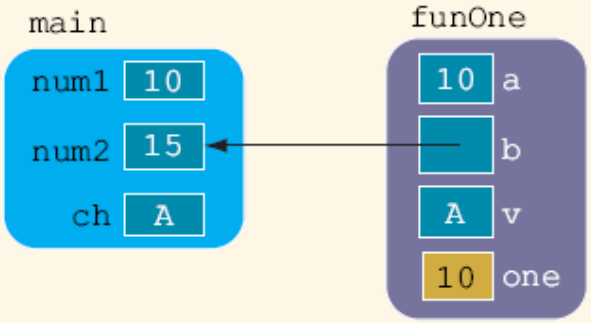


FIGURE 7-7 Values of the variables after the statement in Line 9 executes

```
a++;                                                    //Line 10
b = b * 2;                                              //Line 11
v = 'B';                                                //Line 12
```

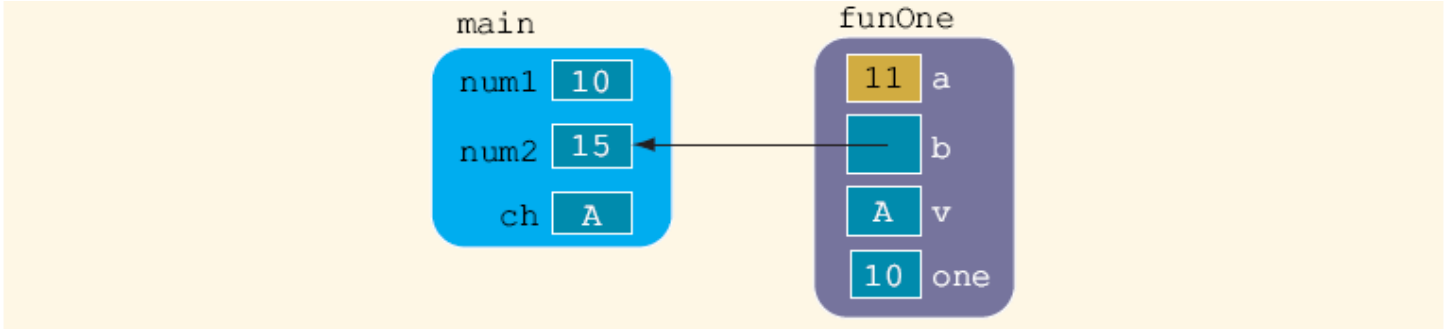


FIGURE 7-8 Values of the variables after the statement in Line 10 executes

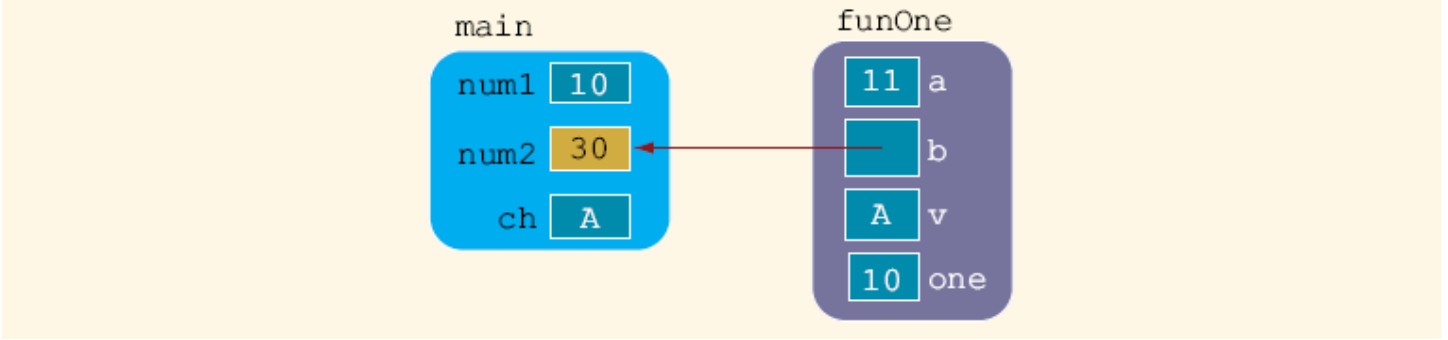


FIGURE 7-9 Values of the variables after the statement in Line 11 executes

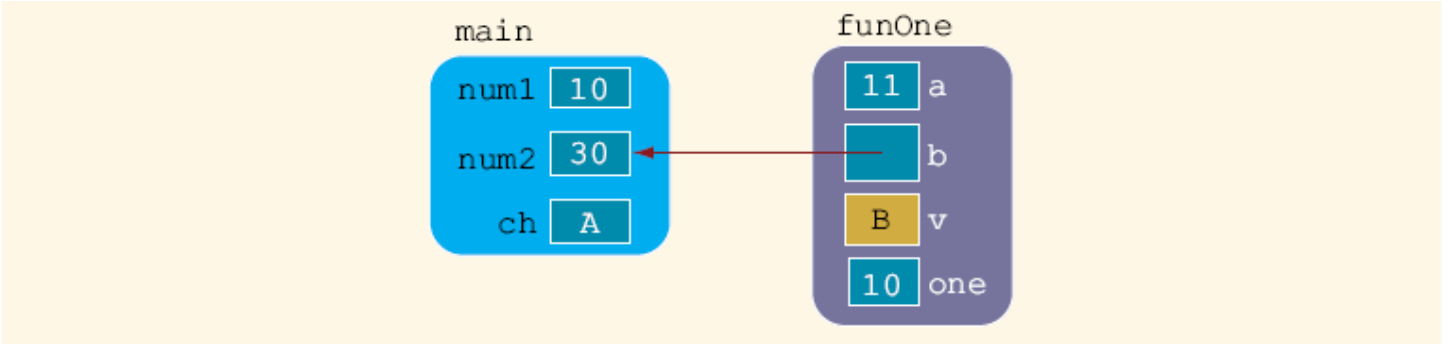


FIGURE 7-10 Values of the variables after the statement in Line 12 executes

```
cout << "Line 13: Inside funOne: a = " << a
     << ", b = " << b << ", v = " << v
     << ", and one = " << one << endl;          //Line 13
```

The statement in Line 13 produces the following output:

```
Line 13: Inside funOne: a = 11, b = 30, v = B, and one = 10
```

```
cout << "Line 6: After funOne: num1 = " << num1
     << ", num2 = " << num2 << ", and ch = "
     << ch << endl;                             //Line 6
```

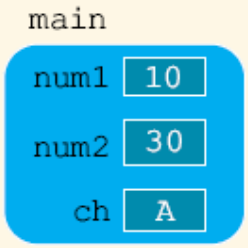


FIGURE 7-11 Values of the variables when control goes back to Line 6

Line 6 produces the following output:

```
Line 6: After funOne: num1 = 10, num2 = 30, and ch = A
```

```
x++;                                                    //Line 14
y = y * 2;                                              //Line 15
```

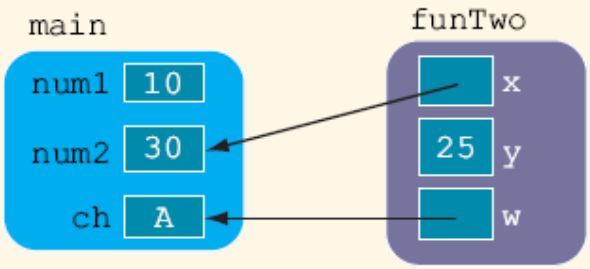


FIGURE 7-12 Values of the variables before the statement in Line 14 executes

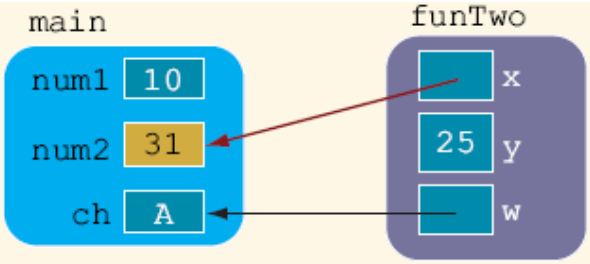


FIGURE 7-13 Values of the variables after the statement in Line 14 executes

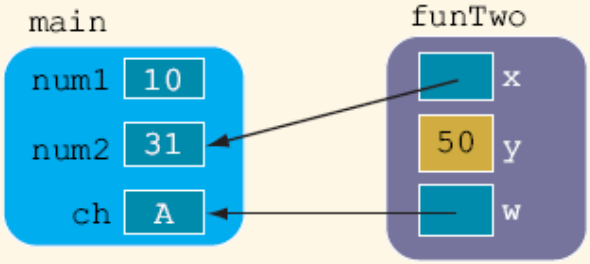


FIGURE 7-14 Values of the variables after the statement in Line 15 executes

```
w = 'G';                                                    //Line 16

cout << "Line 17: Inside funTwo: x = " << x
     << ", y = " << y << ", and w = " << w
     << endl;                                               //Line 17
```

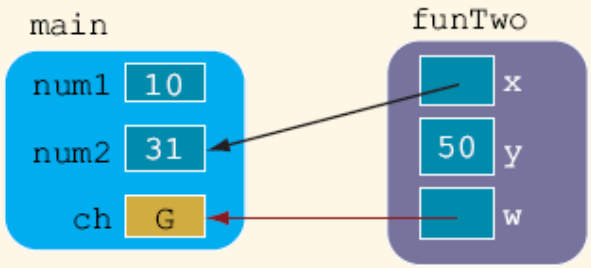


FIGURE 7-15 Values of the variables after the statement in Line 16 executes

Line 17 produces the following output:

```
Line 17: Inside funTwo: x = 31, y = 50, and w = G
```

```
cout << "Line 8: After funTwo: num1 = " << num1
     << ", num2 = " << num2 << ", and ch = "
     << ch << endl;                                         //Line 8
```

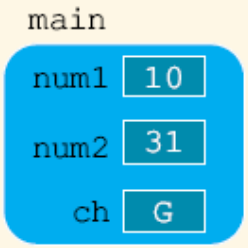


FIGURE 7-16 Values of the variables when control goes to Line 8

```
Line 8: After funTwo: num1 = 10, num2 = 31, and ch = G
```

Reference Parameters and Value-Returning Functions

- You can also use reference parameters in a value-returning function
  - Not recommended
- By definition, a value-returning function returns a single value
  - This value is returned via the return statement
- If a function needs to return more than one value, you should change it to a void function and use the appropriate reference parameters to return the values

Scope of an Identifier

- The scope of an identifier refers to where in the program an identifier is accessible
- Local identifier: identifiers declared within a function (or block)
- Global identifier: identifiers declared outside of every function definition
- C++ does not allow nested functions
  - The definition of one function cannot be included in the body of another function

In general, the following rules apply when an identifier is accessed:

1. Global identifiers (such as variables) are accessible by a function or a block if:
   a. The identifier is declared before the function definition (block),
   b. The function name is different from the identifier,
   c. All parameters of the function have names different than the name of the identifier, and
   d. All local identifiers (such as local variables) have names different than the name of the identifier.
2. **(Nested Block)** An identifier declared within a block is accessible:
   a. Only within the block from the point at which it is declared until the end of the block, and
   b. By those blocks that are nested within that block if the nested block does not have an identifier with the same name as that of the outside block (the block that encloses the nested block).
3. The scope of a function name is similar to the scope of an identifier declared outside any block. That is, the scope of a function name is the same as the scope of a global variable.

Scope of an Identifier (continued)

- Some compilers initialize global variables to default values
- The operator `::` is called the scope resolution operator
- By using the scope resolution operator
  - A global variable declared before the definition of a function (block) can be accessed by the function (or block) even if the function (or block) has an identifier with the same name as the variable

Scope of an Identifier (continued)

- C++ provides a way to access a global variable declared after the definition of a function
  - In this case, the function must not contain any identifier with the same name as the global variable

Global Variables, Named Constants, and Side Effects

- Using global variables has side effects
- A function that uses global variables is not independent
- If more than one function uses the same global variable and something goes wrong
  - It is difficult to find what went wrong and where
  - Problems caused in one area of the program may appear to be from another area
- Global named constants have no side effects

Function Overloading: An Introduction

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

Function Overloading (continued)

- Two functions are said to have different formal parameter lists if both functions 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 them, must differ in at least one position

Function Overloading (continued)

- The following functions all have different formal parameter lists:

```
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)
```

- The following functions have the same formal parameter list:

```
void functionSix(int x, double y, char ch)
void functionSeven(int one, double u, char firstCh)
```

Function Overloading (continued)

- 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

Function Overloading (continued)

- Correct function overloading:

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

- Syntax error:

```
void functionABC(int x, double y)
int functionABC(int x, double y)
```

Summary

- Void function: does not have a data type
  - A `return` statement without any value can be used in a void function to exit it early
  - The heading starts with the word `void`
  - To call the function, you use the function name together with the actual parameters in a stand-alone statement
- Two types of formal parameters:
  - Value parameters
  - Reference parameters

Summary (continued)

- A value parameter receives a copy of its corresponding actual parameter
- A reference parameter receives the memory address of its corresponding actual parameter
  - If a formal parameter needs to change the value of an actual parameter, you must declare this formal parameter as a reference parameter in the function heading

Summary (continued)

- Variables declared within a function (or block) are called local variables
- Variables declared outside of every function definition (and block) are global variables
- Automatic variable: variable for which memory is allocated on function/block entry and deallocated on function/block exit
- Static variable: memory remains allocated throughout the execution of the program
- C++ functions can have default parameters

C++ Programming: From Problem Analysis to Program Design, Fourth Edition

Chapter 9: Arrays

Data Types

- A data type is called simple if variables of that type can store only one value at a time
- A structured data type is one in which each data item is a collection of other data items

Arrays

- Array: a collection of a fixed number of components wherein all of the components have the same data type
- In a one-dimensional array, the components are arranged in a list form
- Syntax for declaring a one-dimensional array:

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

`intExp` evaluates to a positive integer

Arrays (continued)

- Example:

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

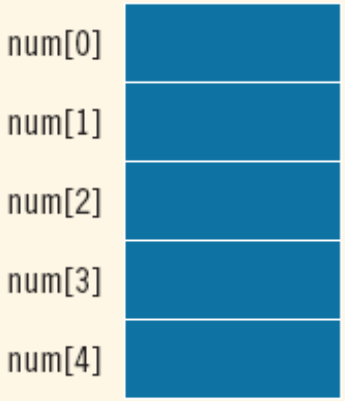


FIGURE 9-1 Array num

Accessing Array Components

- General syntax:

```
arrayName[indexExp]
```

  where `indexExp`, called an **index**, is any expression whose value is a nonnegative integer
- Index value specifies the position of the component in the array
- `[]` is the **array subscripting operator**
- The array index always starts at `0`

Accessing Array Components (continued)

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

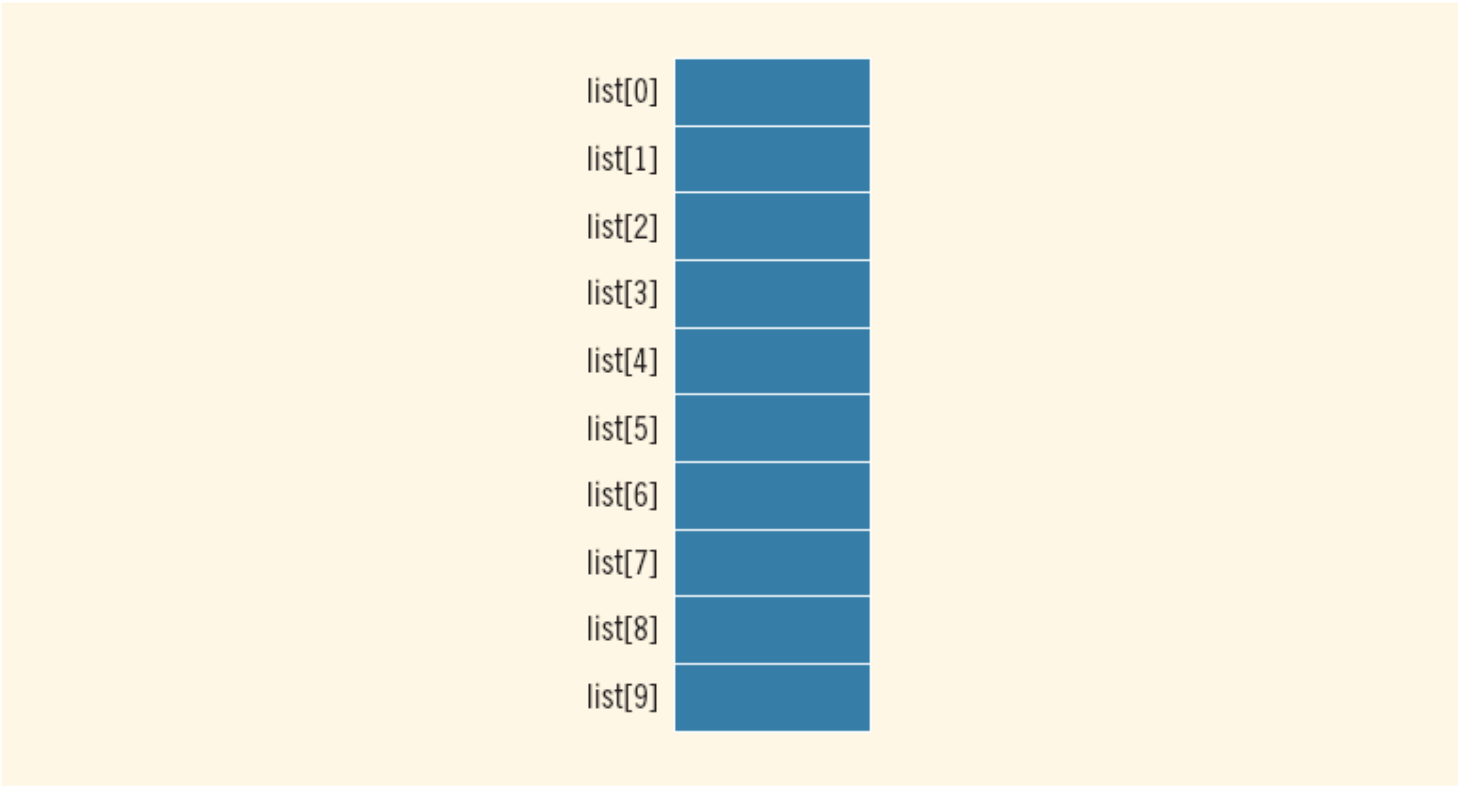


FIGURE 9-2 Array `list`

Accessing Array Components (continued)

```
list[5] = 34;
```

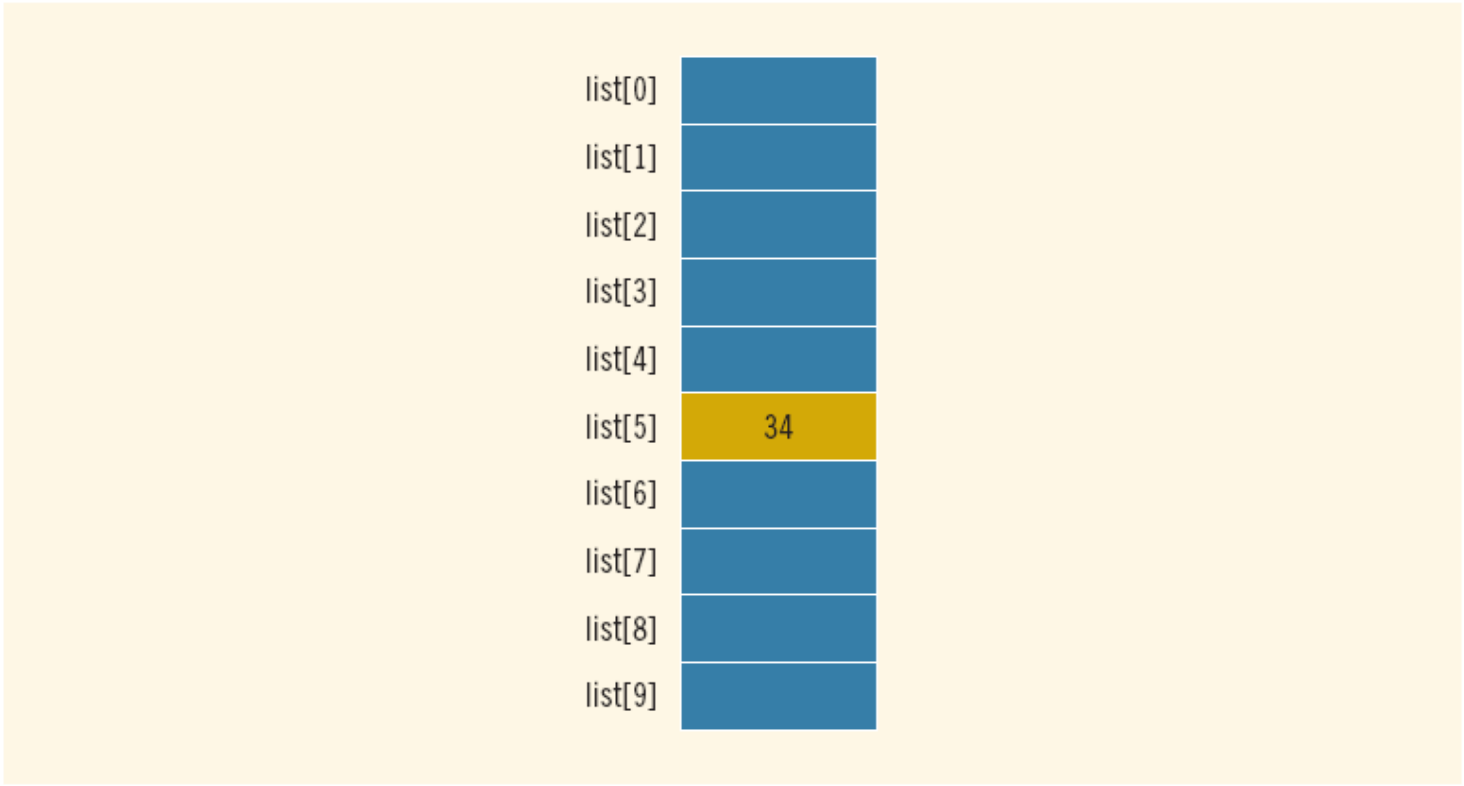


FIGURE 9-3 Array `list` after execution of the statement `list[5]= 34;`

Accessing Array Components (continued)

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

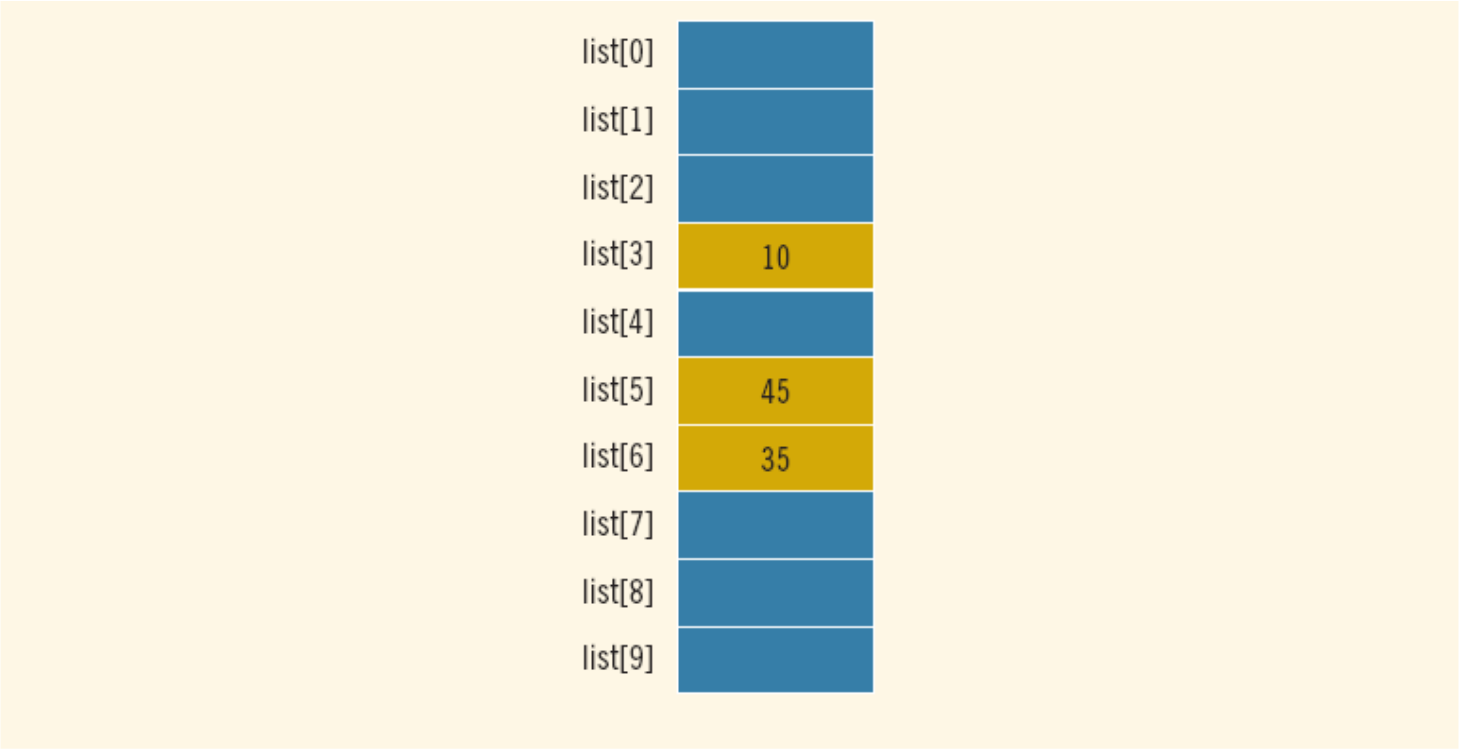


FIGURE 9-4 Array `list` after execution of the statements `list[3]= 10;`, `list[6]= 35;`, and `list[5] = list[3] + list[6];`

Accessing Array Components (continued)

EXAMPLE 9-2

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.

NOTE

When you declare an array, its size must be known. For example, you cannot do the following:

```
int arraySize;                                  //Line 1

cout << "Enter the size of the array: "; //Line 2
cin >> arraySize;                               //Line 3
cout << endl;                                   //Line 4

int list[arraySize];                            //Line 5; not allowed
```

Processing One-Dimensional Arrays

- Some basic operations performed on a one-dimensional array are:
  - Initializing
  - Inputting data
  - Outputting data stored in an array
  - Finding the largest and/or smallest element
- Each operation requires ability to step through the elements of the array
- Easily accomplished by a loop

Processing One-Dimensional Arrays (continued)

- Consider the declaration

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

- Using `for` loops to access array elements:

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

- Example:

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

EXAMPLE 9-3

```
double sales[10];
int index;
double largestSale, sum, average;
```

Initializing an array:

```
for (index = 0; index < 10; index++)
    sales[index] = 0.0;
```

Reading data into an array:

```
for (index = 0; index < 10; index++)
    cin >> sales[index];
```

Printing an array:

```
for (index = 0; index < 10; index++)
    cout << sales[index] << " ";
```

Finding the sum and average of an array:

```
sum = 0;
for (index = 0; index < 10; index++)
    sum = sum + sales[index];

average = sum / 10;
```

Largest element in the array:

```
maxIndex = 0;
for (index = 1; index < 10; index++)
    if (sales[maxIndex] < sales[index])
        maxIndex = index;
largestSale = sales[maxIndex];
```

Array Index Out of Bounds

- If we have the statements:

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

- The component `num[i]` is valid if `i = 0, 1, 2, 3, 4, 5, 6, 7, 8,` or `9`
- The index of an array is in bounds if the `index >=0` and the `index <= ARRAY_SIZE-1`
  - Otherwise, we say the `index` is out of bounds
- In C++, there is no guard against indices that are out of bounds

Array Initialization During Declaration

- Arrays can be initialized during declaration
  - In this case, it is not necessary to specify the size of the array
    - Size determined by the number of initial values in the braces
- Example:

```
double sales[] = {12.25, 32.50, 16.90, 23, 45.68};
```

Partial Initialization of Arrays During Declaration

- The statement:

```
int list[10] = {0};
```

  declares `list` to be an array of `10` components and initializes all of them to zero

- The statement:

```
int list[10] = {8, 5, 12};
```

  declares `list` to be an array of `10` components, initializes `list[0]` to `8`, `list[1]` to `5`, `list[2]` to `12` and all other components are initialized to `0`

Partial Initialization of Arrays During Declaration (continued)

- The statement:

```
int list[] = {5, 6, 3};
```

declares `list` to be an array of `3` components and initializes `list[0]` to `5`, `list[1]` to `6`, and `list[2]` to `3`

- The statement:

```
int list[25]= {4, 7};
```

declares an array of `25` components; initializes `list[0]` to `4` and `list[1]` to `7`; all other components are initialized to `0`

Some Restrictions on Array Processing

- Consider the following statements:

```
int myList[5] = {0, 4, 8, 12, 16};  //Line 1
int yourList[5];  //Line 2
```

- C++ does not allow aggregate operations on an array:

```
yourList = myList;  //illegal
```

- Solution:

```
for (int index = 0; index < 5; index ++)
     yourList[index] = myList[index];
```

Some Restrictions on Array Processing (continued)

- The following is illegal too:

```
cin >> yourList; //illegal
```

- Solution:

```
for (int index = 0; index < 5; index ++)
     cin >> yourList[index];
```

- The following statements are legal, but do not give the desired results:

```
cout << yourList;

if (myList <= yourList)
.
.
```

Two-Dimensional Arrays

- Two-dimensional array: collection of a fixed number of components (of the same type) arranged in two dimensions
  - Sometimes called matrices or tables
- Declaration syntax:

```
dataType  arrayName[intExp1][intExp2];
```

  where `intexp1` and `intexp2` are expressions yielding positive integer values, and specify the number of rows and the number of columns, respectively, in the array

Two-Dimensional Arrays (continued)

```
double sales[10][5];
```

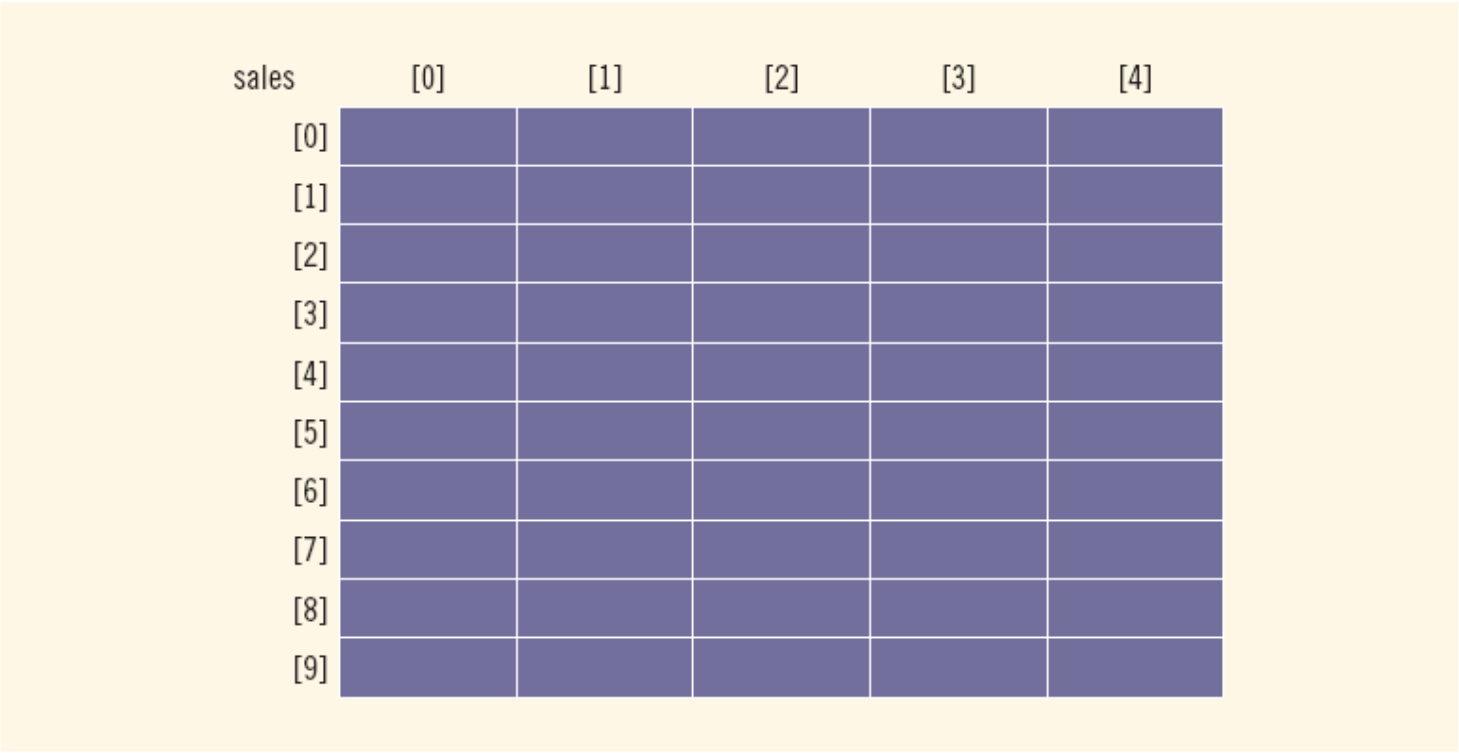


FIGURE 9-8 Two-dimensional array `sales`

Accessing Array Components

- Syntax:

```
arrayName[indexExp1][indexExp2]
```

where `indexexp1` and `indexexp2` are expressions yielding nonnegative integer values, and specify the row and column position

Accessing Array Components (continued)

```
sales[5][3] = 25.75;
```

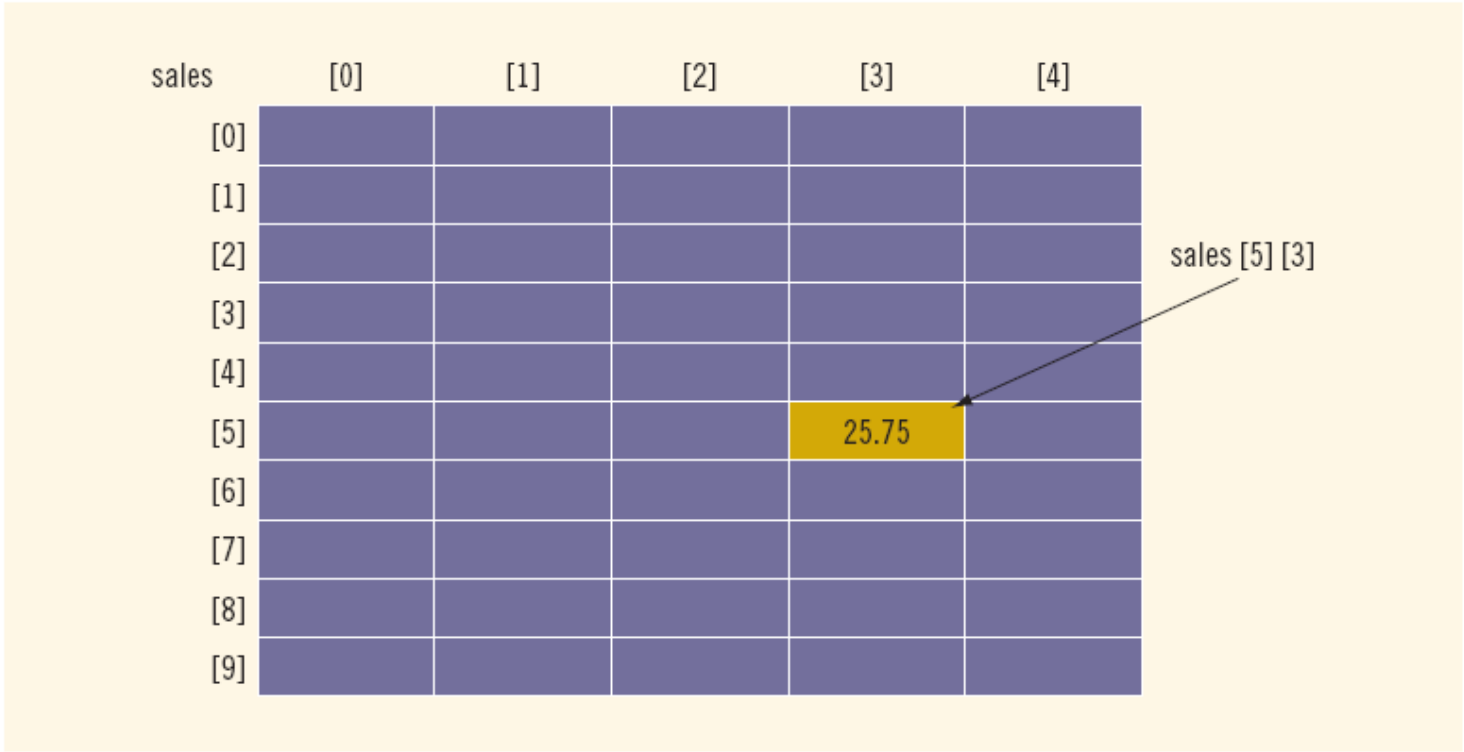


FIGURE 9-9 `sales[5][3]`

Two-Dimensional Array Initialization During Declaration

- Two-dimensional arrays can be initialized when they are declared:

```
int board[4][3] = {{2, 3, 1},
                   {15, 25, 13},
                   {20, 4, 7},
                   {11, 18, 14}};
```

  - Elements of each row are enclosed within braces and separated by commas
  - All rows are enclosed within braces
  - For number arrays, if all components of a row aren't specified, unspecified ones are set to 0

Processing Two-Dimensional Arrays

- Ways to process a two-dimensional array:
  - Process the entire array
  - Process a particular row of the array, called row processing
  - Process a particular column of the array, called column processing
- Each row and each column of a two-dimensional array is a one-dimensional array
  - To process, use algorithms similar to processing one-dimensional arrays

Processing Two-Dimensional Arrays (continued)

```
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;
```

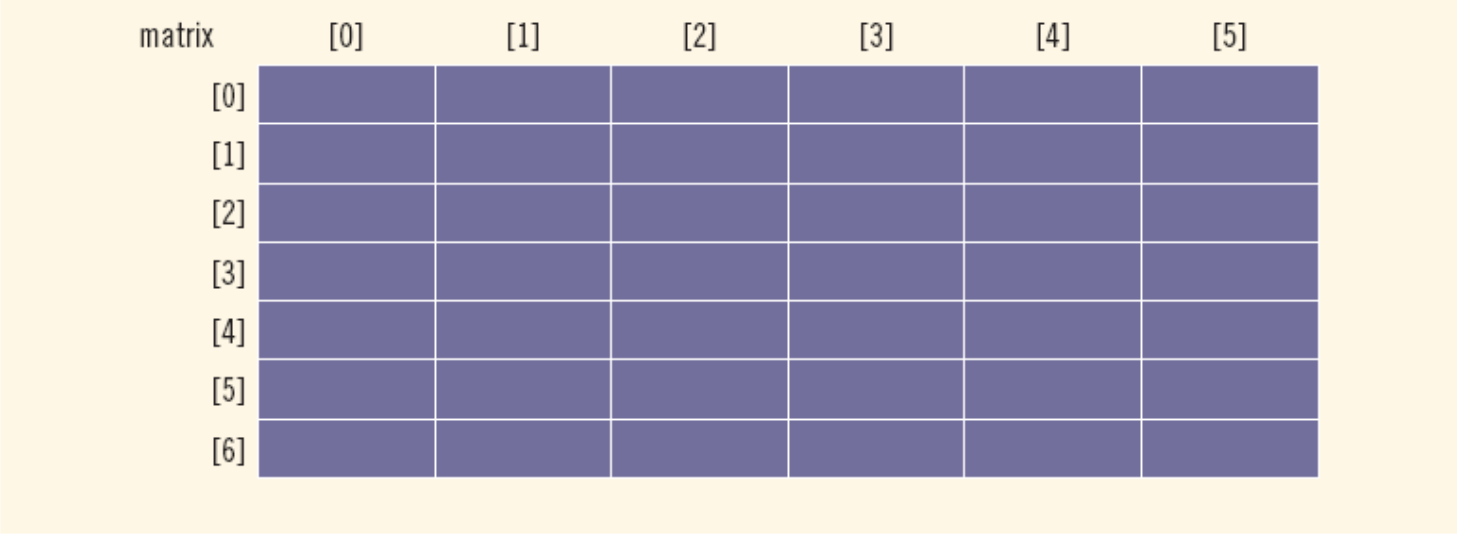


FIGURE 9-13 Two-dimensional array `matrix`

Initialization

- To initialize row number 4 (i.e., fifth row) to 0

```
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    matrix[row][col] = 0;
```

- To initialize the entire matrix to 0:

```
for (row = 0; row < NUMBER_OF_ROWS; row++)
    for (col = 0; col < NUMBER_OF_COLUMNS; col++)
        matrix[row][col] = 0;
```

Print

- To output the components of `matrix`:

```
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

- To input 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

- To find the sum of row number 4 of `matrix`:

```
sum = 0;
row = 4;
for (col = 0; col < NUMBER_OF_COLUMNS; col++)
    sum = sum + matrix[row][col];
```

- 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

- To find 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

```
  //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 (col = 1; col < NUMBER_OF_COLUMNS; col++)
        if (largest < matrix[row][col])
            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 (largest < matrix[row][col])
            largest = matrix[row][col];

    cout << "The largest element in column " << col + 1
         << " = " << largest << endl;
}
```

Summary

- Array: structured data type with a fixed number of components of the same type
  - Components are accessed using their relative positions in the array
- Elements of a one-dimensional array are arranged in the form of a list
- An array index can be any expression that evaluates to a nonnegative integer
  - Must always be less than the size of the array

Summary (continued)

- The base address of an array is the address of the first array component
- When passing an array as an actual parameter, you use only its name
  - Passed by reference only
- A function cannot return a value of the type array

Summary (continued)

- Parallel arrays are used to hold related information
- In a two-dimensional array, the elements are arranged in a table form

Summary

- To access an element of a two-dimensional array, you need a pair of indices:
  - One for the row position
  - One for the column position
- In row processing, a two-dimensional array is processed one row at a time
- In column processing, a two-dimensional array is processed one column at a time