Chapter 15
C++ as a Better C; Introducing Object Technology

C How to Program, 8/e, GE

Objectives

In this chapter you'll:

- Learn key C++ enhancements to C.
- Learn the C++ Standard Library's header files.
- Use `inline` functions to improve performance.
- Use references to pass arguments to functions by reference.
- Use default arguments that the compiler passes to a function if the corresponding arguments are not provided in a function call.
- Use the unary scope resolution operator to access a global variable.
- Overload functions to create several functions of the same name that perform similar tasks, but on data of different types.
- Create and use function templates that perform identical operations on data of different types.

15.1 Introduction

- The first 14 chapters presented a thorough treatment of procedural programming and top-down program design with C.
- The C++ section introduces two additional programming paradigms—object-oriented programming (with classes, encapsulation, objects, operator overloading, inheritance and polymorphism) and generic programming (with function templates and class templates).
- These chapters emphasize "crafting valuable classes" to create reusable software componentry.

15.2 C++

- C++ improves on many of C's features and provides object-oriented-programming (OOP) capabilities that increase software productivity, quality and reusability.
- This chapter discusses many of C++'s enhancements to C.
- When a programming language becomes as entrenched as C, new requirements demand that the language evolve rather than simply be displaced by a new language.
- C++ was developed by Bjarne Stroustrup at Bell Laboratories and was originally called "C with classes."
- The name C++ includes C's increment operator (++) to indicate that C++ is an enhanced version of C.

15.2 C++ (Cont.)

- The remaining provide an introduction to the version of C++ standardized in the United States through the American National Standards Institute (ANSI) and worldwide through the International Standards Organization (ISO).
- If you need additional technical details on C++, we suggest that you read the C++ standard document "Programming languages—C++" (document number ISO/IEC 14882:2011), which can be purchased from standards organization websites, such as `ansi.org` and `iso.org`.

15.3 A Simple Program: Adding Two Integers

- This section revisits the addition program of Fig. 2.5 and illustrates several important features of the C++ language as well as some differences between C and C++.
- C file names have the `.c` (lowercase) extension.
- C++ file names can have one of several extensions, such as `.cpp`, `.cxx` or `.C` (uppercase).
- We use the extension `.cpp`.
- Figure 15.1 uses C++-style input and output to obtain two integers typed by a user at the keyboard, computes the sum of these values and outputs the result.
- Lines 1 and 2 each begin with `//`, indicating that the remainder of each line is a comment.

15.3 A Simple Program: Adding Two Integers (Cont.)

- C++ allows you to begin a comment with // and use the remainder of the line as comment text.
- A // comment is a maximum of one line long.
- C++ programmers may also use /*...*/ C-style comments, which can be more than one line long.

```
1  // Fig. 15.1: fig15_01.cpp
2  // Addition program that displays the sum of two numbers.
3  #include <iostream> // allows program to perform input and output
4
5  int main()
6  {
7     std::cout << "Enter first integer: "; // prompt user for data
8     int number1;
9     std::cin >> number1; // read first integer from user into number1
10
11    std::cout << "Enter second integer: "; // prompt user for data
12    int number2;
13    std::cin >> number2; // read second integer from user into number2
14    int sum = number1 + number2; // add the numbers; store result in sum
15    std::cout << "Sum is " << sum << std::endl; // display sum; end line
16 }
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 15.1 | Addition program that displays the sum of two numbers.

15.3 A Simple Program: Adding Two Integers (Cont.)

- The C++ preprocessor directive in line 3 exhibits the standard C++ style for including header files from the standard library.
- This line tells the C++ preprocessor to include the contents of the input/output stream header file `<iostream>`.
- This file must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output.
- We discuss `iostream`'s many features in detail in Chapter 2, Stream Input/Output.
- As in C, every C++ program begins execution with function `main` (line 5).

15.3 A Simple Program: Adding Two Integers (Cont.)

- Keyword `int` to the left of main indicates that main returns an integer value.
- C++ requires you to specify the return type, possibly `void`, for all functions.
- In C++, specifying a parameter list with empty parentheses is equivalent to specifying a `void` parameter list in C.
- In C, using empty parentheses in a function definition or prototype is dangerous.
- It disables compile-time argument checking in function calls, which allows the caller to pass any arguments to the function.
- This could lead to runtime errors.

Common Programming Error 15.1

Omitting the return type in a C++ function definition is a syntax error.

15.3 A Simple Program: Adding Two Integers (Cont.)

- Line 7 is a familiar variable declaration.
- Declarations can be placed almost anywhere in a C++ program, but they must appear before their corresponding variables are used in the program.
- For example, in Fig. 15.1, the declaration in line 7 could have been placed immediately before line 10, the declaration in line 12 could have been placed immediately before line 16 and the declaration in line 13 could have been placed immediately before line 17.

15.3 A Simple Program: Adding Two Integers (Cont.)

- Line 9 uses the standard output stream object—`std::cout`—and the stream insertion operator, `<<`, to display the string `"Enter first integer: "`.
- Output and input in C++ are accomplished with streams of characters.
- Thus, when line 9 executes, it sends the stream of characters `"Enter first integer: "` to `std::cout`, which is normally "connected" to the screen.
- We like to pronounce the preceding statement as "`std::cout` gets the character string `"Enter first integer: "`."
- Line 10 uses the standard input stream object—`std::cin`—and the stream extraction operator, `>>`, to obtain a value from the keyboard.

15.3 A Simple Program: Adding Two Integers (Cont.)

- Using the stream extraction operator with `std::cin` takes character input from the standard input stream, which is usually the keyboard.
- We like to pronounce the preceding statement as, "`std::cin` gives a value to `number1`" or simply "`std::cin` gives `number1`."
- When the computer executes the statement in line 10, it waits for the user to enter a value for variable `number1`.
- The user responds by typing an integer (as characters), then pressing the *Enter* key.
- The computer converts the character representation of the number to an integer and assigns this value to the variable `number1`.

15.3 A Simple Program: Adding Two Integers (Cont.)

- Line 15 displays `"Enter second integer: "` on the screen, prompting the user to take action.
- Line 16 obtains a value for variable `number2` from the user.
- The assignment statement in line 17 calculates the sum of the variables `number1` and `number2` and assigns the result to variable `sum`.
- Line 18 displays the character string `Sum is` followed by the numerical value of variable `sum` followed by `std::endl`—a so-called stream manipulator.
- The name `endl` is an abbreviation for "end line."
- The `std::endl` stream manipulator outputs a newline, then "flushes the output buffer."

15.3 A Simple Program: Adding Two Integers (Cont.)

- This simply means that, on some systems where outputs accumulate in the machine until there are enough to "make it worthwhile" to display on the screen, `std::endl` forces any accumulated outputs to be displayed at that moment.
- This can be important when the outputs are prompting the user for an action, such as entering data.
- We place `std::` before `cout`, `cin` and `endl`.
- This is required when we use standard C++ header files.
- The notation `std::cout` specifies that we're using a name, in this case `cout`, that belongs to "namespace" `std`.
- Namespaces are an advanced C++ feature that we do not discuss in these introductory C++ chapters.

15.3 A Simple Program: Adding Two Integers (Cont.)

- For now, you should simply remember to include `std::` before each mention of `cout`, `cin` and `endl` in a program.
- This can be cumbersome—in Fig. 15.3, we introduce the `using` statement, which will enable us to avoid placing `std::` before each use of a namespace `std` name.
- The statement in line 18 outputs values of different types.
- The stream insertion operator "knows" how to output each type of data.
- Using multiple stream insertion operators (`<<`) in a single statement is referred to as concatenating, chaining or cascading stream insertion operations.

15.3 A Simple Program: Adding Two Integers (Cont.)

- Calculations can also be performed in output statements.
- We could have combined the statements in lines 17 and 18 into the statement
  - `std::cout << "Sum is " << number1 + number2 << std::endl;`

  thus eliminating the need for the variable `sum`.
- You'll notice that we did not have a `return 0;` statement at the end of main in this example.

15.3 A Simple Program: Adding Two Integers (Cont.)

- According to the C++ standard, if program execution reaches the end of `main` without encountering a `return` statement, it's assumed that the program terminated successfully—exactly as when the last statement in main is a `return` statement with the value `0`.
- For that reason, we omit the `return` statement at the end of `main` in our C++ programs.

15.3 A Simple Program: Adding Two Integers (Cont.)

- A powerful C++ feature is that users can create their own types called classes (we introduce this capability in Chapter 16 and explore it in depth in Chapters 17–18).
- Users can then "teach" C++ how to input and output values of these new data types using the >> and << operators (this is called operator overloading—a topic we explore in Chapter 19).

15.4 C++ Standard Library

- C++ programs consist of pieces called classes and functions.
- You can program each piece you need to form a C++ program.
- Instead, most C++ programmers take advantage of the rich collections of existing classes and functions in the C++ Standard Library.
- Thus, there are really two parts to learning the C++ "world."
- The first is learning the C++ language itself; the second is learning how to use the classes and functions in the C++ Standard Library.
- Many special-purpose class libraries are supplied by independent software vendors.

Software Engineering Observation 15.1

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

Software Engineering Observation 15.2

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

15.4 C++ Standard Library (Cont)

- The advantage of creating your own functions and classes is that you'll know exactly how they work.
- You'll be able to examine the C++ code.
- The disadvantage is the time-consuming and complex effort that goes into designing, developing and maintaining new functions and classes that are correct and that operate efficiently.

Performance Tip 15.1

Using C++ Standard Library functions and classes instead of writing your own versions can improve program performance, because they are written to perform efficiently. This technique also shortens program development time.

Portability Tip 15.1

Using C++ Standard Library functions and classes instead of writing your own improves program portability, because they are included in every C++ implementation.

15.5 Header Files

- The C++ Standard Library is divided into many portions, each with its own header file.
- The header files contain the function prototypes for the related functions that form each portion of the library.
- The header files also contain definitions of various class types and functions, as well as constants needed by those functions.
- A header file “instructs” the compiler on how to interface with library and user-written components.
- Figure 15.2 lists common C++ Standard Library header files.
- Header file names ending in `.h` are “old-style” headers that have been superceded by C++ Standard Library headers..

C++ Standard Library header file	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and output functions, introduced in Section 15.3, and is covered in more detail in Chapter 21, Stream Input/Output: A Deeper Look.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 15.15 and is discussed in more detail in Chapter 21.
`<cmath>`	Contains function prototypes for the math library functions.
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Chapter 18, Operator Overloading; Class `string` and Chapter 22, Exception Handling: A Deeper Look.
`<ctime>`	Contains function prototypes and types for manipulating the time and date.

Fig. 15.2 | C++ Standard Library header files. (Part 1 of 4.)

C++ Standard Library header file	Explanation
`<array>`, `<vector>`, `<list>`, `<forward_list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<unordered_map>`, `<unordered_set>`, `<set>`, `<bitset>`	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The `<vector>` header is first introduced in Section 15.15.
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header is used in Chapter 18.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 20.8.

Fig. 15.2 | C++ Standard Library header files. (Part 2 of 4.)

C++ Standard Library header file	Explanation
`<exception>`, `<stdexcept>`	These headers contain classes that are used for exception handling (discussed in Chapter 22).
`<memory>`	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 22.
`<fstream>`	Contains function prototypes for functions that perform input from and output to files on disk.
`<string>`	Contains the definition of class `string` from the C++ Standard Library.
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory.
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms.
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers.
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging.

Fig. 15.2 | C++ Standard Library header files. (Part 3 of 4.)

C++ Standard Library header file	Explanation
`<cfloat>`	Contains the floating-point size limits of the system.
`<climits>`	Contains the integral size limits of the system.
`<cstdio>`	Contains function prototypes for the C's standard I/O functions.
`<locale>`	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).
`<limits>`	Contains classes for defining the numerical data type limits on each computer platform.
`<utility>`	Contains classes and functions that are used by many C++ Standard Library headers.

Fig. 15.2 | C++ Standard Library header files. (Part 4 of 4.)

15.5 Header Files (Cont.)

- You can create custom header files.
- Programmer-defined header files should end in `.h`.
- A programmer-defined header file can be included by using the `#include` preprocessor directive.
- For example, the header file `square.h` can be included in a program by placing the directive `#include "square.h"` at the beginning of the program.

15.6 Inline Functions

- Implementing a program as a set of functions is good from a software engineering standpoint, but function calls involve execution-time overhead.
- C++ provides inline functions to help reduce function call overhead—especially for small functions.
- Placing the qualifier `in-line` before a function's return type in the function definition "advises" the compiler to generate a copy of the function's code in place (when appropriate) to avoid a function call.

15.6 Inline Functions (Cont.)

- The trade-off is that multiple copies of the function code are inserted in the program (often making the program larger) rather than there being a single copy of the function to which control is passed each time the function is called.
- The compiler can ignore the `inline`-qualifier and typically does so for all but the smallest functions.

Software Engineering Observation 15.3

Changing to an `inline` *function could require clients of the function to be recompiled. This can be significant in program development and maintenance situations.*

Performance Tip 15.2

Using `inline` *functions can reduce execution time but may increase program size.*

Software Engineering Observation 15.4

The `inline` qualifier should be used only with small, frequently used functions.

15.6 Inline Functions (Cont.)

- Figure 15.3 uses `inline` function `cube` (lines 11–14) to calculate the volume of a cube of side length `side`.
- Keyword `const` in the parameter list of function `cube` tells the compiler that the function does not modify variable `side`.
- This ensures that the value of `side` is not changed by the function when the calculation is performed.
- Notice that the complete definition of function `cube` appears before it's used in the program.
- This is required so that the compiler knows how to expand a `cube` function call into its inlined code.
- For this reason, reusable inline functions are typically placed in header files, so that their definitions can be included in each source file that uses them.

Software Engineering Observation 15.5

The `const` qualifier should be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software can greatly reduce debugging time and improper side effects, and can make a program easier to modify and maintain.

```
1  // Fig. 15.3: fig15_03.cpp
2  // inline function that calculates the volume of a cube.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  // Definition of inline function cube. Definition of function appears
9  // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13    return side * side * side; // calculate the cube of side
14 }
15
16 int main()
17 {
18    double sideValue; // stores value entered by user
19
```

Fig. 15.3 | `inline` function that calculates the volume of a cube. (Part 1 of 2.)

```
20      for ( int i = 1; i <= 3; i++ )
21      {
22         cout << "\nEnter the side length of your cube: ";
23         cin >> sideValue; // read value from user
24
25         // calculate cube of sideValue and display result
26         cout << "Volume of cube with side "
27            << sideValue << " is " << cube( sideValue ) << endl;
28      }
29   }
```

```
Enter the side length of your cube: 1.0
Volume of cube with side 1 is 1

Enter the side length of your cube: 2.3
Volume of cube with side 2.3 is 12.167

Enter the side length of your cube: 5.4
Volume of cube with side 5.4 is 157.464
```

Fig. 15.3 | `inline` function that calculates the volume of a cube. (Part 2 of 2.)

15.6 Inline Functions (Cont.)

- Lines 4–6 are `using` statements that help us eliminate the need to repeat the `std::` prefix.
- Once we include these `using` statements, we can write `cout` instead of `std::cout`, `cin` instead of `std::cin` and `endl` instead of `std::endl`, in the remainder of the program.
- From this point forward, each C++ example contains one or more `using` statements.
- In place of lines 4–6, many programmers prefer to use
  - `using namespace std;`

  which enables a program to use all the names in any standard C++ header file (such as `<iostream>`) that a program might include.

15.6 Inline Functions (Cont.)

- From this point forward in our C++ programs, we'll use the preceding declaration in our programs.
- The `for` statement's condition (line 20) evaluates to either 0 (false) or nonzero (true).
- This is consistent with C.
- C++ also provides type `bool` for representing boolean (true/false) values.
- The two possible values of a `bool` are the keywords `true` and `false`.
- When `true` and `false` are converted to integers, they become the values 1 and 0, respectively.

15.6 Inline Functions (Cont.)

- When non-boolean values are converted to type `bool`, non-zero values become `true`, and zero or null pointer values become `false`.

15.7 C++ Keywords

- Figure 15.4 lists the keywords common to C and C++ and the keywords unique to C++.

C++ Keywords

Keywords common to the C and C++ programming languages

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

C++-only keywords

and	and_eq	asm	bitand	bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	false
friend	inline	mutable	namespace	new
not	not_eq	operator	or	or_eq
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

Fig. 15.4 | C++ keywords. (Part 1 of 2.)

C++ Keywords				
C++11 keywords				
`alignas`	`alignof`	`char16_t`	`char32_t`	`constexpr`
`decltype`	`noexcept`	`nullptr`	`static_assert`	`thread_local`

Fig. 15.4 | C++ keywords. (Part 2 of 2.)

15.8 References and Reference Parameters

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

Performance Tip 15.3

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

15.8 References and Reference Parameters (Cont.)

Reference Parameters

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

Performance Tip 15.4

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.

Software Engineering Observation 15.6

Pass-by-reference can weaken security; the called function can corrupt the caller's data.

15.8 References and Reference Parameters (Cont.)

- Later, we'll show how to achieve the performance advantage of pass-by-reference while simultaneously achieving the software engineering advantage of protecting the caller's data from corruption.
- A reference parameter is an alias for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference, simply follow the pa-rameter's type in the function prototype by an ampersand (&); use the same notation when listing the pa-rameter's type in the function header.

15.8 References and Reference Parameters (Cont.)

- For exam-ple, the following declaration in a function header
  - `int &count`

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

15.8 References and Reference Parameters (Cont.)

Passing Arguments by Value and by Reference

- Figure 15.5 compares pass-by-value and pass-by-reference with reference parameters.
- The "styles" of the arguments in the calls to function `squareByValue` (line 17) and function `squareByReference` (line 22) are identical—both variables are simply mentioned by name in the function calls.
- Without checking the function prototypes or function definitions, it's not possible to tell from the calls alone whether either function can modify its arguments.

15.8 References and Reference Parameters (Cont.)

- Because function prototypes are mandatory, however, the compiler has no trouble resolving the ambiguity.
- Recall that a function proto-type tells the compiler the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order in which they are expected.
- The compiler uses this information to validate function calls.
- In C, function prototypes are not required.
- Making them mandatory in C++ enables type-safe linkage, which ensures that the types of the arguments conform to the types of the pa-rameters.

15.8 References and Reference Parameters (Cont.)

- Otherwise, the compiler reports an error.
- Locating such type errors at compile time helps prevent the runtime errors that can occur in C when arguments of incorrect data types are passed to functions.

```
1  // Fig. 15.5: fig15_05.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using namespace std;
5
6  int squareByValue( int ); // function prototype (value pass)
7  void squareByReference( int & ); // function prototype (reference pass)
8
9  int main()
10 {
11    // demonstrate squareByValue
12    int x = 2;
13    cout << "x = " << x << " before squareByValue\n";
14    cout << "Value returned by squareByValue: "
15       << squareByValue( x ) << endl;
16    cout << "x = " << x << " after squareByValue\n" << endl;
17
18    // demonstrate squareByReference
19    int z = 4;
20    cout << "z = " << z << " before squareByReference" << endl;
21    squareByReference( z );
22    cout << "z = " << z << " after squareByReference" << endl;
23 }
24
```

Fig. 15.5 | Comparing pass-by-value and pass-by-reference with references. (Part 1 of 2.)

```
25  // squareByValue multiplies number by itself, stores the
26  // result in number and returns the new value of number
27  int squareByValue( int number )
28  {
29     return number *= number; // caller's argument not modified
30  }
31
32  // squareByReference multiplies numberRef by itself and stores the result
33  // in the variable to which numberRef refers in the caller
34  void squareByReference( int &numberRef )
35  {
36     numberRef *= numberRef; // caller's argument modified
37  }
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 15.5 | Comparing pass-by-value and pass-by-reference with references. (Part 2 of 2.)

Common Programming Error 15.2

Because reference parameters are mentioned only by name in the body of the called function, you might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original copies of the variables are changed by the function.

Performance Tip 15.5

For passing large objects efficiently, use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object. The called function will not be able to modify the object in the caller.

Software Engineering Observation 15.7

Many programmers do not declare parameters passed by value as `const`*, even when the called function should not modify the passed argument. Keyword* `const` *in this context would protect only a copy of the original argument, not the original argument itself, which when passed by value is safe from modification by the called function.*

15.8 References and Reference Parameters (Cont.)

- To specify a reference to a constant, place the `const` qualifier before the type specifier in the parameter declaration.
- Note in line 35 of Fig. 15.5 the placement of `&` in the parameter list of function `squareByReference`.
- Some C++ programmers prefer to write `int& numberRef` with the ampersand abutting `int`—both forms are equivalent to the compiler.

Software Engineering Observation 15.8

For the combined reasons of clarity and performance, many C++ programmers prefer that modifiable arguments be passed to functions by using pointers, small nonmodifiable arguments be passed by value and large nonmodifiable arguments be passed by using references to constants.

15.8 References and Reference Parameters (Cont.)

References as Aliases within a Function

- References can also be used as aliases for other variables within a function (although they typically are used with functions as shown in Fig. 15.5).
- For ex-ample, the code
  - ```
    int count = 1; // declare integer variable count
    int &cRef = count; // create cRef as an alias for count
    cRef++; // increment count (using its alias cRef)
    ```

  increments variable `count` by using its alias `cRef`.
    ```

15.8 References and Reference Parameters (Cont.)

- Reference variables must be initialized in their declarations, as we show in line 9 of both Fig. 15.6 and Fig. 15.7, and cannot be reassigned as aliases to other variables.
- Once a reference is declared as an alias for a variable, all operations "performed" on the alias (i.e., the reference) are actually performed on the original variable.
- The alias is simply another name for the original variable.

15.8 References and Reference Parameters (Cont.)

- Taking the address of a reference and comparing references do not cause syntax errors-; rather, each operation occurs on the variable for which the reference is an alias.
- Unless it's a reference to a constant, a reference argument must be an *lvalue* (e.g., a variable name), not a constant or expression that returns an *rvalue* (e.g., the result of a calculation).

```
1  // Fig. 15.6: fig15_06.cpp
2  // Initializing and using a reference.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 3;
9     int &y = x; // y refers to (is an alias for) x
10
11    cout << "x = " << x << endl << "y = " << y << endl;
12    y = 7; // actually modifies x
13    cout << "x = " << x << endl << "y = " << y << endl;
14 }
```

```
x = 3
y = 3
x = 7
y = 7
```

Fig. 15.6 | Initializing and using a reference.

```
1  // Fig. 15.7: fig15_07.cpp
2  // Uninitialized reference is a syntax error.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 3;
9     int &y; // Error: y must be initialized
10
11    cout << "x = " << x << endl << "y = " << y << endl;
12    y = 7;
13    cout << "x = " << x << endl << "y = " << y << endl;
14 }
```

Fig. 15.7 | Uninitialized reference is a syntax error. (Part 1 of 2.)

Microsoft Visual C++ compiler error message:

```
fig15_07.cpp(9) : error C2530: 'y' :
   references must be initialized
```

GNU C++ compiler error message:

```
fig15_07.cpp:9: error: 'y' declared as a reference but not initialized
```

Xcode LLVM compiler error message:

```
Declaration of reference variable 'y' requires an initializer
```

Fig. 15.7 | Uninitialized reference is a syntax error. (Part 2 of 2.)

15.8 References and Reference Parameters (Cont.)

Returning a Reference from a Function

- Returning references from functions can be dangerous.
- When returning a reference to a variable declared in the called function, the variable should be declared `static` within that function.
- Otherwise, the reference refers to an automatic variable that is discarded when the function terminates; such a variable is "undefined," and the program's behavior is unpredictable.
- References to undefined variables are called dangling references.

Common Programming Error 15.3

Not initializing a reference variable when it's declared is a compilation error, unless the declaration is part of a function's parameter list. Reference parameters are initialized when the function in which they're declared is called.

Common Programming Error 15.4

Attempting to reassign a previously declared reference to be an alias to another variable is a logic error. The value of the other variable is simply assigned to the variable for which the reference is already an alias.

Common Programming Error 15.5

Returning a reference to an automatic variable in a called function is a logic error. Some compilers issue a warning when this occurs.

15.8 References and Reference Parameters (Cont.)

Error Messages for Uninitialized References

- The C++ standard does not specify the error messages that compilers use to indicate particular errors.
- For this reason, we show in Fig. 15.7 the error messages produced by several compilers when a reference is not initialized.

15.9 Empty Parameter Lists

- C++, like C, allows you to define functions with no parameters.
- In C++, an empty parameter list is specified by writing either `void` or nothing at all in parentheses.
- The prototypes
  - ```
    void print();
    void print( void );
    ```

  each specify that function `print` does not take arguments and does not return a value.
- These prototypes are equivalent.
    ```

Portability Tip 15.2

The meaning of an empty function parameter list in C++ is dramatically different than in C. In C, it means all argument checking is disabled (i.e., the function call can pass any arguments it wants). In C++, it means that the function takes no arguments. Thus, C programs using this feature might cause compilation errors when compiled in C++.

Common Programming Error 15.6

It's a compilation error to specify default arguments in both a function's prototype and header.

15.10 Default Arguments

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

15.10 Default Arguments (Cont.)

- Default arguments must be the rightmost (trailing) arguments in a function's parame-ter list.
- When calling a function with two or more default arguments, if an omitted argu-ment is not the rightmost argument in the argument list, then all arguments to the right of that argument also must be omitted.
- Default arguments should be specified with the first oc-currence of the function name—typically, in the function prototype.
- If the function prototype is omitted because the function definition also serves as the prototype, then the default arguments should be specified in the function header.

15.10 Default Arguments (Cont.)

- Default values can be any expression, including con-stants, global variables or function calls.
- Default argu-ments also can be used with `in-line` functions.
- Figure 15.8 demonstrates using default arguments in calculating the volume of a box.
- The function prototype for `boxVolume` (line 7) specifies that all three parameters have been given default values of `1`.
- We provided variable names in the function prototype for readability, but these are not required.

```
1  // Fig. 15.8: fig15_08.cpp
2  // Using default arguments.
3  #include <iostream>
4  using namespace std;
5
6  // function prototype that specifies default arguments
7  int boxVolume( int length = 1, int width = 1, int height = 1 );
8
9  int main()
10 {
11    // no arguments--use default values for all dimensions
12    cout << "The default box volume is: " << boxVolume();
13
14    // specify length; default width and height
15    cout << "\n\nThe volume of a box with length 10,\n"
16       << "width 1 and height 1 is: " << boxVolume( 10 );
17
18    // specify length and width; default height
19    cout << "\n\nThe volume of a box with length 10,\n"
20       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
21
```

Fig. 15.8 | Using default arguments. (Part 1 of 2.)

```
22      // specify all arguments
23      cout << "\n\nThe volume of a box with length 10,\n"
24         << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
25         << endl;
26   }
27
28   // function boxVolume calculates the volume of a box
29   int boxVolume( int length, int width, int height )
30   {
31      return length * width * height;
32   }
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Fig. 15.8 | Using default arguments. (Part 2 of 2.)

15.10 Default Arguments (Cont.)

- The first call to `boxVolume` (line 12) specifies no arguments, thus using all three default values of 1.
- The second call (line 16) passes a `length` argument, thus using default values of 1 for the `width` and `height` arguments.
- The third call (line 20) passes arguments for `length` and `width`, thus using a default value of 1 for the `height` argument.
- The last call (line 24) passes arguments for `length`, `width` and `height`, thus using no default values.

15.10 Default Arguments (Cont.)

- Any arguments passed to the function explicitly are assigned to the function's parameters from left to right.
- Therefore, when `boxVolume` receives one argument, the function assigns the value of that argument to its `length` parameter (i.e., the leftmost parameter in the parameter list).
- When `boxVolume` receives two arguments, the function assigns the values of those arguments to its `length` and `width` parameters in that order.
- Finally, when `boxVolume` receives all three arguments, the function assigns the values of those arguments to its `length`, `width` and `height` parameters, respectively.

Good Programming Practice 15.1

Using default arguments can simplify writing function calls. However, some programmers feel that explicitly specifying all arguments is clearer.

Software Engineering Observation 15.9

If the default values for a function change, all client code must be recompiled.

Common Programming Error 15.7

In a function definition, specifying and attempting to use a default argument that is not a rightmost (trailing) argument (while not simultaneously defaulting all the rightmost arguments) is a syntax error.

15.11 Unary Scope Resolution Operator

- It's possible to declare local and global variables of the same name.
- This causes the global variable to be "hidden" by the local variable in the local scope.
- C++ provides the unary scope resolution operator (`::`) to access a global variable when a local variable of the same name is in scope.
- The unary scope resolution operator cannot be used to access a local variable of the same name in an outer block.

15.11 Unary Scope Resolution Operator (Cont.)

- A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.
- Figure 15.9 demonstrates the unary scope resolution operator with global and local variables of the same name (lines 6 and 10, respectively).
- To emphasize that the local and global versions of variable `number` are distinct, the program declares one variable of type `int` and the other `double`.

```
1  // Fig. 15.9: fig15_09.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4  using namespace std;
5
6  int number = 7; // global variable named number
7
8  int main()
9  {
10    double number = 10.5; // local variable named number
11
12    // display values of local and global variables
13    cout << "Local double value of number = " << number
14       << "\nGlobal int value of number = " << ::number << endl;
15 }
```

```
Local double value of number = 10.5
Global int value of number = 7
```

Fig. 15.9 | Using the unary scope resolution operator.

15.11 Unary Scope Resolution Operator (Cont.)

- Using the unary scope resolution operator (::) with a given variable name is optional when the only variable with that name is a global variable.

Common Programming Error 15.8

It's an error to attempt to use the unary scope resolution operator (::) to access a nonglobal variable in an outer block. If no global variable with that name exists, a compilation error occurs. If a global variable with that name exists, this is a logic error, because the program will refer to the global variable when you intended to access the nonglobal variable in the outer block.

Good Programming Practice 15.2

Always using the unary scope resolution operator (::) to refer to global variables makes it clear that you intend to access a global variable rather than a nonglobal variable.

Software Engineering Observation 15.10

Always using the unary scope resolution operator (::) to refer to global variables makes programs easier to modify by reducing the risk of name collisions with nonglobal variables.

Error-Prevention Tip 15.1

Always using the unary scope resolution operator (::) to refer to a global variable eliminates logic errors that might occur if a nonglobal variable hides the global variable.

Error-Prevention Tip 15.2

Avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors.

15.12 Function Overloading

- C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least as far as the parameter types or the number of parameters or the order of the parameter types are concerned).
- This capability is called function overloading. When an overloaded function is called, the C++ compiler selects the proper function by examining the number, types and order of the arguments in the call.

15.12 Function Overloading (Cont.)

- Function overloading is commonly used to create sev-eral functions of the same name that perform similar tasks, but on data of different types.
- For example, many functions in the math library are overloaded for different numeric data types.

Good Programming Practice 15.3

Overloading functions that perform closely related tasks can make programs more readable and understandable.

15.12 Function Overloading (Cont.)

Overloaded `square` Functions

- Figure 15.10 uses overloaded `square` functions to calculate the square of an `int` (lines 7–11) and the square of a `double` (lines 14–18).
- Line 22 invokes the `int` version of function `square` by passing the literal value `7`.
- C++ treats whole-number literal values as type `int` by default.

15.12 Function Overloading (Cont.)

- Similarly, line 24 invokes the `double` version of function `square` by passing the literal value `7.5`, which C++ treats as a `double` value by default.
- In each case the compiler chooses the proper function to call, based on the type of the argument.
- The outputs confirm that the proper function was called in each case.

```
1  // Fig. 15.10: fig15_10.cpp
2  // Overloaded square functions.
3  #include <iostream>
4  using namespace std;
5
6  // function square for int values
7  int square( int x )
8  {
9     cout << "square of integer " << x << " is ";
10    return x * x;
11 }
12
13 // function square for double values
14 double square( double y )
15 {
16    cout << "square of double " << y << " is ";
17    return y * y;
18 }
19
```

Fig. 15.10 | Overloaded `square` functions. (Part 1 of 2.)

```
20  int main()
21  {
22     cout << square( 7 ); // calls int version
23     cout << endl;
24     cout << square( 7.5 ); // calls double version
25     cout << endl;
26  }
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Fig. 15.10 | Overloaded `square` functions. (Part 2 of 2.)

Common Programming Error 15.9

Creating overloaded functions with identical parameter lists and different return types is a compilation error.

15.12 Function Overloading (Cont.)

How the Compiler Differentiates Overloaded Functions

- Overloaded functions are distinguished by their signatures—a combination of a function's name and its parameter types (in order).
- The compiler encodes each function identifier with the number and types of its parameters (sometimes referred to as name mangling or name decoration) to enable type-safe linkage.
- This ensures that the proper overloaded function is called and that the argument types conform to the pa-rameter types.
- Figure 15.11 was compiled with GNU C++.

15.12 Function Overloading (Cont.)

- Rather than showing the execution out-put of the program (as we normally would), we show the mangled function names produced in assembly language by GNU C++.
- Each mangled name (other than `main`) begins with two underscores (`__`) followed by the letter `Z`, a number and the function name.
- The number that follows `Z` specifies how many characters are in the function's name.
- For example, function square has 6 characters in its name, so its mangled name is prefixed with `__Z6`.
- The function name is then followed by an encoding of its parameter list.

15.12 Function Overloading (Cont.)

- In the parameter list for function `nothing2` (line 25; see the fourth output line), `c` represents a `char`, `i` represents an `int`, `Rf` represents a `float &` (i.e., a reference to a `float`) and `Rd` represents a `double &` (i.e., a reference to a `double`).
- In the parameter list for function `noth-ing1`, `i` represents an `int`, `f` represents a `float`, `c` represents a `char` and `Ri` rep-resents an `int &`.
- The two `square` functions are distinguished by their parameter lists; one specifies `d` for `double` and the other specifies `i` for `int`.

15.12 Function Overloading (Cont.)

- The return types of the functions are not specified in the mangled names.
- Overloaded functions can have different return types, but if they do, they must also have different parameter lists.
- Again, you cannot have two functions with the same signature and different return types.
- Function-name mangling is compiler specific.
- Also, function `main` is not mangled, because it cannot be overloaded.

```
1  // Fig. 15.11: fig15_11.cpp
2  // Name mangling to enable type-safe linkage.
3
4  // function square for int values
5  int square( int x )
6  {
7     return x * x;
8  }
9
10 // function square for double values
11 double square( double y )
12 {
13    return y * y;
14 }
15
16 // function that receives arguments of types
17 // int, float, char and int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20    // empty function body
21 }
22
```

Fig. 15.11 | Name mangling to enable type-safe linkage. (Part 1 of 2.)

```
23   // function that receives arguments of types
24   // char, int, float & and double &
25   int nothing2( char a, int b, float &c, double &d )
26   {
27      return 0;
28   }
29
30   int main()
31   {
32      return 0; // indicates successful termination
33   }
```

```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
_main
```

Fig. 15.11 | Name mangling to enable type-safe linkage. (Part 2 of 2.)

15.12 Function Overloading (Cont.)

- The compiler uses only the parameter lists to distinguish between functions of the same name.
- Overloaded functions need not have the same number of parameters.
- You should use caution when overloading functions with default parameters, because this may cause ambiguity.

Common Programming Error 15.10

A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having in a program both a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in a compilation error when an attempt is made to call that function with no arguments. The compiler does not know which version of the function to choose.

15.12 Function Overloading (Cont.)

Overloaded Operators

- In Chapter 19, we discuss how to overload operators to define how they operate on objects of user-defined data types.
- (In fact, we've been using overloaded operators, including the stream insertion operator << and the stream extraction operator >>, each of which is overloaded to be able to display data of all the fundamental types.
- We say more about overloading << and >> to be able to handle objects of user-defined types in Chapter 19.)
- Section 15.12 introduces function templates for automatically generating overloaded functions that perform identical tasks on data of different types.

15.13 Function Templates

- Overloaded functions are used to perform similar operations that may involve different program logic on different data types.
- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.
- You write a single function template definition.
- Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately.
- Thus, defining a single function template essentially defines a whole family of overloaded functions.

15.13 Function Templates (Cont.)

- Figure 15.12 contains the definition of a function template (lines 4–18) for a `maximum` function that determines the largest of three values.
- All function template definitions begin with the `template` keyword (line 4) followed by a template parameter list to the function template enclosed in angle brackets (< and >).
- Every parameter in the template parameter list (each is referred to as a formal type parameter) is preceded by keyword `typename` or keyword `class` (which are synonyms).
- The formal type parameters are placeholders for fundamental types or user-defined types.

15.13 Function Templates (Cont.)

- These placeholders are used to specify the types of the function's parameters (line 5), to specify the function's return type (line 5) and to declare variables within the body of the function definition (line 7).
- A function template is defined like any other function, but uses the formal type parameters as placeholders for actual data types.
- The function template in Fig. 15.12 declares a single formal type parameter `T` (line 4) as a placeholder for the type of the data to be tested by function `maximum`.

15.13 Function Templates (Cont.)

- The name of a type parameter must be unique in the template parameter list for a particular template definition.
- When the compiler detects a `maximum` invoca-tion in the program source code, the type of the data passed to `maximum` is substituted for `T` throughout the template definition, and C++ creates a complete source-code function for determining the maximum of three values of the specified data type.
- Then the newly created function is com-piled.
- Thus, templates are a means of code generation.

```
1  // Fig. 15.12: maximum.h
2  // Function template maximum header file.
3
4  template < class T >  // or template< typename T >
5  T maximum( T value1, T value2, T value3 )
6  {
7     T maximumValue = value1; // assume value1 is maximum
8
9     // determine whether value2 is greater than maximumValue
10    if ( value2 > maximumValue )
11       maximumValue = value2;
12
13    // determine whether value3 is greater than maximumValue
14    if ( value3 > maximumValue )
15       maximumValue = value3;
16
17    return maximumValue;
18 }
```

Fig. 15.12 | Function template `maximum` header file.

Common Programming Error 15.11

Not placing keyword `class` *or keyword* `typename` *before every formal type parameter of a function template (e.g., writing* `<class S, T>` *instead of* `<class S, class T>`*) is a syntax error.*

15.13 Function Templates (Cont.)

- Figure 15.13 uses the `maximum` function template (lines 18, 28 and 38) to determine the largest of three `int` values, three `double` values and three `char` values.

```
1  // Fig. 15.13: fig15_13.cpp
2  // Demonstrating function template maximum.
3  #include <iostream>
4  using namespace std;
5
6  #include "maximum.h" // include definition of function template maximum
7
8  int main()
9  {
10    // demonstrate maximum with int values
11    int int1, int2, int3;
12
13    cout << "Input three integer values: ";
14    cin >> int1 >> int2 >> int3;
15
16    // invoke int version of maximum
17    cout << "The maximum integer value is: "
18       << maximum( int1, int2, int3 );
19
```

Fig. 15.13 | Demonstrating function template `maximum`. (Part 1 of 3.)

```
20      // demonstrate maximum with double values
21      double double1, double2, double3;
22
23      cout << "\n\nInput three double values: ";
24      cin >> double1 >> double2 >> double3;
25
26      // invoke double version of maximum
27      cout << "The maximum double value is: "
28         << maximum( double1, double2, double3 );
29
30      // demonstrate maximum with char values
31      char char1, char2, char3;
32
33      cout << "\n\nInput three characters: ";
34      cin >> char1 >> char2 >> char3;
35
36      // invoke char version of maximum
37      cout << "The maximum character value is: "
38         << maximum( char1, char2, char3 ) << endl;
39   }
```

Fig. 15.13 | Demonstrating function template `maximum`. (Part 2 of 3.)

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Fig. 15.13 | Demonstrating function template `maximum`. (Part 3 of 3.)

Software Engineering Observation 15.11

Reuse of existing classes when building new classes and programs saves time, money and effort. Reuse also helps you build more reliable and effective systems, because existing classes often have gone through extensive testing, debugging and performance tuning.

15.13 Function Templates (Cont.)

- In Fig. 15.13, three functions are created as a result of the calls in lines 18, 28 and 38—expecting three `int` values, three `double` values and three `char` values, respectively.
- For example, the function template specialization created for type `int` replaces each occurrence of `T` with `int`.

15.14 Introduction to C++ Standard Library Class Template `vector`

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

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- As you learned in Chapter 7, pointer variables (known more commonly as pointers) contain memory addresses as their values.
- Array names are simply *pointers* to where the arrays begin in memory, and, of course, two arrays will always be at different memory locations.
- When an array is passed to a general-purpose function designed to handle arrays of any size, the size of the array must be passed as an additional argument.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- Furthermore, one array cannot be assigned to another with the assignment operator(s)—array names are `const` pointers, so they cannot be used on the left side of an assignment operator.
- These and other capabilities certainly seem like "naturals" for dealing with arrays, but C++ does not provide such capabilities.
- However, the C++ Standard Library provides class template vector to allow you to create a more powerful and less error-prone alternative to arrays.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- The `vector` class template is available to anyone building applications with C++.
- The notations that the vector example uses might be unfamiliar to you, because vectors use template notation.
- In the previous section, we discussed function templates.
- In Chapter 22, we discuss class templates.
- For now, you should feel comfortable using class template vector by mimicking the syntax in the example we show in this section.
- Chapter 22 presents class template `vector` (and several other standard C++ container classes) in detail.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- The program of Fig. 15.14 demonstrates capabilities provided by C++ Standard Library class template `vector` that are not available for C-style pointer-based arrays.
- Standard class template `vector` is defined in header `<vector>` (line 5) and belongs to namespace `std`.
- Chapter 22 discusses the full functionality of vector.
- At the end of this section, we'll demonstrate class `vector`'s bounds checking capabilities and introduce C++'s exception-handling mechanism, which can be used to detect and handle an out-of-bounds `vector` index.

```
1  // Fig. 15.14: fig15_14.cpp
2  // Demonstrating C++ Standard Library class template vector.
3  #include <iostream>
4  #include <iomanip>
5  #include <vector>
6  using namespace std;
7
8  void outputVector( const vector< int > & ); // display the vector
9  void inputVector( vector< int > & ); // input values into the vector
10
11 int main()
12 {
13    vector< int > integers1( 7 ); // 7-element vector< int >
14    vector< int > integers2( 10 ); // 10-element vector< int >
15
16    // print integers1 size and contents
17    cout << "Size of vector integers1 is " << integers1.size()
18       << "\nvector after initialization:" << endl;
19    outputVector( integers1 );
20
21    // print integers2 size and contents
22    cout << "\nSize of vector integers2 is " << integers2.size()
23       << "\nvector after initialization:" << endl;
24    outputVector( integers2 );
```

Fig. 15.14 | Demonstrating C++ Standard Library class template `vector`. (Part 1 of 7.)

```
25
26      // input and print integers1 and integers2
27      cout << "\nEnter 17 integers:" << endl;
28      inputVector( integers1 );
29      inputVector( integers2 );
30
31      cout << "\nAfter input, the vectors contain:\n"
32         << "integers1:" << endl;
33      outputVector( integers1 );
34      cout << "integers2:" << endl;
35      outputVector( integers2 );
36
37      // use inequality (!=) operator with vector objects
38      cout << "\nEvaluating: integers1 != integers2" << endl;
39
40      if ( integers1 != integers2 )
41         cout << "integers1 and integers2 are not equal" << endl;
42
```

Fig. 15.14 | Demonstrating C++ Standard Library class template `vector`. (Part 2 of 7.)

```
43      // create vector integers3 using integers1 as an
44      // initializer; print size and contents
45      vector< int > integers3( integers1 ); // copy constructor
46
47      cout << "\nSize of vector integers3 is " << integers3.size()
48         << "\nvector after initialization:" << endl;
49      outputVector( integers3 );
50
51      // use overloaded assignment (=) operator
52      cout << "\nAssigning integers2 to integers1:" << endl;
53      integers1 = integers2; // assign integers2 to integers1
54
55      cout << "integers1:" << endl;
56      outputVector( integers1 );
57      cout << "integers2:" << endl;
58      outputVector( integers2 );
59
60      // use equality (==) operator with vector objects
61      cout << "\nEvaluating: integers1 == integers2" << endl;
62
63      if ( integers1 == integers2 )
64         cout << "integers1 and integers2 are equal" << endl;
65
```

Fig. 15.14 | Demonstrating C++ Standard Library class template `vector`. (Part 3 of 7.)

```
66      // use square brackets to create rvalue
67      cout << "\nintegers1[5] is " << integers1[ 5 ];
68
69      // use square brackets to create lvalue
70      cout << "\n\nAssigning 1000 to integers1[5]" << endl;
71      integers1[ 5 ] = 1000;
72      cout << "integers1:" << endl;
73      outputVector( integers1 );
74
75      // attempt to use out-of-range index
76      try
77      {
78         cout << "\nAttempt to display integers1.at( 15 )" << endl;
79         cout << integers1.at( 15 ) << endl; // ERROR: out of range
80      }
81      catch ( out_of_range &ex )
82      {
83         cout << "An exception occurred: " << ex.what() << endl;
84      }
85   }
86
```

Fig. 15.14 | Demonstrating C++ Standard Library class template `vector`. (Part 4 of 7.)

```
87  // output vector contents
88  void outputVector( const vector< int > &array )
89  {
90     size_t i; // declare control variable
91
92     for ( i = 0; i < array.size(); ++i )
93     {
94        cout << setw( 12 ) << array[ i ];
95
96        if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
97           cout << endl;
98     }
99
100    if ( i % 4 != 0 )
101       cout << endl;
102 }
103
104 // input vector contents
105 void inputVector( vector< int > &array )
106 {
107    for ( size_t i = 0; i < array.size(); ++i )
108       cin >> array[ i ];
109 }
```

Fig. 15.14 | Demonstrating C++ Standard Library class template `vector`. (Part 5 of 7.)

```
Size of vector integers1 is 7
vector after initialization:
           0           0           0           0
           0           0           0

Size of vector integers2 is 10
vector after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

Fig. 15.14 | Demonstrating C++ Standard Library class template `vector`. (Part 6 of 7.)

```
Size of vector integers3 is 7
vector after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to display integers1.at( 15 )
An exception occurred: invalid vector<T> subscript
```

Fig. 15.14 | Demonstrating C++ Standard Library class template `vector`. (Part 7 of 7.)

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

Creating vector Objects

- Lines 13–14 create two vector objects that store values of type `int`—`integers1` contains seven elements, and `integers2` contains 10 elements.
- By default, all the elements of each `vector` object are set to `0`.
- Note that `vector`s can be defined to store *any* data type, by replacing `int` in `vector<int>` with the appropriate data type.
- This notation, which specifies the type stored in the `vector`, is similar to the template notation that Section 15.12 introduced with function templates.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

`vector` Member Function `size`; Function `outputVector`

- Line 17 uses `vector` member function `size` to obtain the size (i.e., the number of elements) of `integers1`.
- Line 19 passes `integers1` to function `outputVector` (lines 88–102), which uses square brackets, `[]` (line 94), to obtain the value in each element of the `vector` for output.
- Note the resemblance of this notation to that used to access the value of an array element.
- Lines 22 and 24 perform the same tasks for `integers2`.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- Member function `size` of class template `vector` returns the number of elements in a `vector` as a value of type `size_t` (which represents the type unsigned `int` on many systems).
- As a result, line 90 declares the control variable `i` to be of type `size_t`, too.
- On some compilers, declaring `i` as an `int` causes the compiler to issue a warning message, since the loop-continuation condition (line 92) would compare a signed value (i.e., `int i`) and an unsigned value (i.e., a value of type `size_t` returned by function `size`).

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

Function `inputVector`

- Lines 28–29 pass `integers1` and `integers2` to function `inputVector` (lines 105–109) to read values for each vector's elements from the user.
- The function uses square brackets (`[]`) to form `lvalues` that are used to store the input values in each `vector` element.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

Comparing `vector` Objects for Inequality

- Line 40 demonstrates that `vector` objects can be compared with one another using the `!=` operator.
- If the contents of two `vector`s are not equal, the operator returns `true`; otherwise, it returns `false`.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

Initializing One `vector` with the Contents of Another

- The C++ Standard Library class template `vector` allows you to create a new `vector` object that is initialized with the contents of an existing `vector`.
- Line 45 creates a `vector` object `integers3` and initializes it with a copy of `integers1`.
- This invokes `vector`'s so-called copy constructor to perform the copy operation.
- Lines 47–49 output the size and contents of `integers3` to demonstrate that it was initialized correctly.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

Assigning `vector`s and Comparing `vector`s for Equality

- Line 53 assigns `integers2` to `integers1`, demonstrating that the assignment (`=`) operator can be used with `vector` objects.
- Lines 55–58 output the contents of both objects to show that they now contain identical values.
- Line 63 then compares `integers1` to `integers2` with the equality (`==`) operator to determine whether the contents of the two objects are equal after the assignment in line 53 (which they are).

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

Using the `[]` Operator to Access and Modify `vector` Elements

- Lines 67 and 71 use square brackets (`[]`) to obtain a `vector` element as an *rvalue* and as an *lvalue* , respectively.
- Recall that an *rvalue* cannot be modified, but an `lvalue` can.
- As is the case with C-style pointer-based arrays, C++ does not perform any bounds checking when *`vector` elements are accessed with square brackets*.
- Therefore, you must ensure that operations using [ ] do not accidentally attempt to manipulate elements outside the bounds of the `vector`.
- Standard class template `vector` does, however, provide bounds checking in its member function `at`, which we use at line 79 and discuss shortly.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

Exception Handling: Processing an Out-of-Range Subscript

- An exception indicates a problem that occurs while a program executes.
- The name "exception" suggests that the problem occurs infrequently—if the "rule" is that a statement normally executes correctly, then the problem represents the "exception to the rule."
- Exception handling enables you to create fault-tolerant programs that can resolve (or handle) exceptions.
- In many cases, this allows a program to continue executing as if no problems were encountered.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- For example, Fig. 15.14 still runs to completion, even though an attempt was made to access an out-of-range subscript.
- More severe problems might prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem, then terminate.
- When a function detects a problem, such as an invalid array subscript or an invalid argument, it throws an exception—that is, an exception occurs.
- Here we introduce exception handling briefly. We'll discuss it in detail in Chapter 24.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

The `try` Statement

- To handle an exception, place any code that might throw an exception in a `try` statement (lines 76–84).
- The `try` block (lines 76–80) contains the code that might throw an exception, and the `catch` block (lines 81–84) contains the code that handles the exception if one occurs.
- You can have many catch blocks to handle different types of exceptions that might be thrown in the corresponding `try` block.
- If the code in the `try` block executes successfully, lines 81–84 are ignored.
- The braces that delimit `try` and `catch` blocks' bodies are required.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- The vector member function `at` provides bounds checking and throws an exception if its argument is an invalid subscript.
- By default, this causes a C++ program to terminate.
- If the subscript is valid, function at returns the element at the specified location as a modifiable *lvalue* or an unmodifiable *lvalue*, depending on the context in which the call appears.
- An unmodifiable *lvalue* is an expression that identifies an object in memory (such as an element in a `vector`), but cannot be used to modify that object.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

Executing the `catch` Block

- When the program calls `vector` member function `at` with the argument 15 (line 79), the function attempts to access the element at location 15, which is outside the `vector`'s bounds—`integers1` has only 10 elements at this point.
- Because bounds checking is performed at execution time, `vector` member function at generates an exception—specifically line 79 throws an `out_of_range` exception (from header `<stdexcept>`) to notify the program of this problem.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- At this point, the `try` block terminates immediately and the `catch` block begins executing—if you declared any variables in the `try` block, they're now out of scope and are not accessible in the `catch` block.
- [Note: To avoid compilation errors with GNU C++, you may need to include header `<stdexcept>` to use class `out_of_range`.]

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- The `catch` block declares a type (`out_of_range`) and an exception parameter (`ex`) that it receives as a reference.
- The `catch` block can handle exceptions of the specified type.
- Inside the block, you can use the parameter's identifier to interact with a caught exception object.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

`what` *Member Function of the Exception Parameter*

- When lines 81–84 `catch` the exception, the program displays a message indicating the problem that occurred.
- Line 83 calls the exception object's `what` member function to get the error message that is stored in the exception object and display it.
- Once the message is displayed in this example, the exception is considered handled and the program continues with the next statement after the `catch` block's closing brace.
- In this example, the end of the program is reached, so the program terminates.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

Summary of This Example

- In this section, we demonstrated the C++ Standard Library class template `vector`, a robust, reusable class that can replace C-style pointer-based arrays.
- In Chapter 19, you'll see that vector achieves many of its capabilities by "overloading" C++'s built-in operators, and you'll learn how to customize operators for use with your own classes in similar ways.

15.14 Introduction to C++ Standard Library Class Template `vector` (Cont.)

- For example, we create an `Array` class that, like class template `vector`, improves upon basic array capabilities.
- Our `Array` class also provides additional features, such as the ability to input and output entire arrays with operators >> and <<, respectively.

15.15 Introduction to Object Technology and the UML

- Now we introduce object orientation, a natural way of thinking about the world and writing computer programs.
- Our goal here is to help you develop an object-oriented way of thinking and to introduce you to the Unified Modeling Language™ (UML™)—a graphical language that allows people who design object-oriented software systems to use an industry-standard notation to represent them.
- In this section, we introduce basic object-oriented concepts and terminology.

15.15 Introduction to Object Technology and the UML (Cont.)

Basic Object Technology Concepts

- We begin our introduction to object orientation with some key terminology.
- Everywhere you look in the real world you see objects—people, animals, plants, cars, planes, buildings, computers and so on.
- Humans think in terms of objects.
- Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects we see around us every day.

15.15 Introduction to Object Technology and the UML (Cont.)

- Objects have some things in common.
- They all have attributes (e.g., size, shape, color and weight), and they all exhibit behaviors (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water).
- We'll study the kinds of attributes and behaviors that software objects have.
- Humans learn about existing objects by studying their attributes and observing their behaviors.

15.15 Introduction to Object Technology and the UML (Cont.)

- Different objects can have similar attributes and can exhibit similar behaviors.
- Comparisons can be made, for example, between babies and adults and between humans and chimpanzees.
- Object-oriented design (OOD) models software in terms similar to those that people use to describe real-world objects.
- It takes advantage of class relationships, where objects of a certain class, such as a class of vehicles, have the same characteristics—cars, trucks, little red wagons and roller skates have much in common.

15.15 Introduction to Object Technology and the UML (Cont.)

- OOD takes advantage of inheritance relationships, where new classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own.
- An object of class "convertible" certainly has the characteristics of the more general class "automobile," but more specifically, the roof goes up and down.
- Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes, behaviors and interrelationships just as we describe real-world objects.

15.15 Introduction to Object Technology and the UML (Cont.)

- OOD also models communication between objects.
- Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages.
- A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.
- OOD encapsulates (i.e., wraps) attributes and operations (behaviors) into objects—an object's attributes and operations are intimately tied together.

15.15 Introduction to Object Technology and the UML (Cont.)

- Objects have the property of information hiding.
- This means that objects may know how to communicate with one another across well-defined interfaces, but normally they're not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves.
- We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally—as long as we know how to use the accelerator pedal, the brake pedal, the steering wheel and so on.

15.15 Introduction to Object Technology and the UML (Cont.)

- Information hiding, as we'll see, is crucial to good software engineering.
- Languages like C++ are object oriented.
- Programming in such a language is called object-oriented programming (OOP), and it allows you to implement an object-oriented design as a working software system.
- Languages like C, on the other hand, are procedural, *so programming tends to be action oriented.*
- In C, the unit of programming is the function.
- In C++, the unit of programming is the "class" from which objects are eventually instantiated (an OOP term for "created").

15.15 Introduction to Object Technology and the UML (Cont.)

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

15.15 Introduction to Object Technology and the UML (Cont.)

Classes, Data Members and Member Functions

- C++ programmers concentrate on creating their own user-defined types called classes.
- Each class contains data as well as the set of functions that manipulate that data and provide services to clients (i.e., other classes or functions that use the class).
- The data components of a class are called data members.
- For example, a bank account class might include an account number and a balance.
- The function components of a class are called member functions (typically called methods in other object-oriented programming languages such as Java).

15.15 Introduction to Object Technology and the UML (Cont.)

- For example, a bank account class might include member functions to make a deposit (increasing the balance), make a withdrawal (decreasing the balance) and inquire what the current balance is.
- You use built-in types (and other user-defined types) as the "building blocks" for constructing new user-defined types (classes).
- The nouns in a system specification help the C++ programmer determine the set of classes from which objects are created that work together to implement the system.
- Classes are to objects as blueprints are to houses—a class is a "plan" for building an object of the class.

15.15 Introduction to Object Technology and the UML (Cont.)

- Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class.
- You cannot cook meals in the kitchen of a blueprint; you can cook meals in the kitchen of a house.
- You cannot sleep in the bedroom of a blueprint; you can sleep in the bedroom of a house.
- Classes can have relationships with other classes.
- In an object-oriented design of a bank, the "bank teller" class relates to other classes, such as the "customer" class, the "cash drawer" class, the "safe" class, and so on.
- These relationships are called associations.
- Packaging software as classes makes it possible for future software systems to reuse the classes.

15.15 Introduction to Object Technology and the UML (Cont.)

- Indeed, with object technology, you can build much of the new software you'll need by combining existing classes, just as automobile manufacturers combine interchangeable parts.
- Each new class you create can become a valuable software asset that you and others can reuse to speed and enhance the quality of future software development efforts.

15.15 Introduction to Object Technology and the UML (Cont.)

Introduction to Object-Oriented Analysis and Design (OOAD)

- Soon you'll be writing programs in C++.
- How will you create the code for your programs?
- Perhaps, like many beginning programmers, you'll simply turn on your computer and start typing.
- This approach may work for small programs, but what if you were asked to create a software system to control thousands of automated teller machines for a major bank?
- Or what if you were asked to work on a team of 1000 software developers building the next generation of the U.S. air traffic control system?

15.15 Introduction to Object Technology and the UML (Cont.)

- For projects so large and complex, you could not simply sit down and start writing programs.
- To create the best solutions, you should follow a detailed process for analyzing your project's requirements (i.e., determining what the system is supposed to do) and developing a design that satisfies them (i.e., deciding how the system should do it).
- Ideally, you would go through this process and carefully review the design (or have your design reviewed by other software professionals) before writing any code.

15.15 Introduction to Object Technology and the UML (Cont.)

- If this process involves analyzing and designing your system from an object-oriented point of view, it's called object-oriented analysis and design (OOAD).
- Experienced programmers know that analysis and design can save many hours by helping avoid an ill-planned system development approach that has to be abandoned partway through its implementation, possibly wasting considerable time, money and effort.
- OOAD is the generic term for the process of analyzing a problem and developing an approach for solving it.
- Small problems like the ones discussed in the next few chapters do not require an exhaustive OOAD process.

15.15 Introduction to Object Technology and the UML (Cont.)

- As problems and the groups of people solving them increase in size, the methods of OOAD quickly become more appropriate than pseudocode.
- Ideally, a group should agree on a strictly defined process for solving its problem and a uniform way of communicating the results of that process to one another.
- Although many different OOAD processes exist, a single graphical language for communicating the results of any OOAD process has come into wide use.

15.15 Introduction to Object Technology and the UML (Cont.)

- This language, known as the Unified Modeling Language (UML), was developed in the mid-1990s under the initial direction of three software methodologists: Grady Booch, James Rumbaugh and Ivar Jacobson.
- In the 1980s, increasing numbers of organizations began using OOP to build their applications, and a need developed for a standard OOAD process.
- Many methodologists—including Booch, Rumbaugh and Jacobson—individually produced and promoted separate processes to satisfy this need.

15.15 Introduction to Object Technology and the UML (Cont.)

History of the UML

- Each process had its own notation, or "language" (in the form of graphical diagrams), to convey the results of analysis and design.
- In 1994, James Rumbaugh joined Grady Booch at Rational Software Corporation (now a division of IBM), and the two began working to unify their popular processes.
- They soon were joined by Ivar Jacobson.
- In 1996, the group released early versions of the UML to the software engineering community and requested feedback.
- Around the same time, an organization known as the Object Management Group™ (OMG™) invited submissions for a common modeling language.

15.15 Introduction to Object Technology and the UML (Cont.)

- The OMG (`www.omg.org`) is a nonprofit organization that promotes the standardization of object-oriented technologies by issuing guidelines and specifications, such as the UML.
- Several corporations—among them HP, IBM, Microsoft, Oracle and Rational Software—had already recognized the need for a common modeling language.
- In response to the OMG's request for proposals, these companies formed UML Partners—the consortium that developed the UML version 1.1 and submitted it to the OMG.

15.15 Introduction to Object Technology and the UML (Cont.)

- The OMG accepted the proposal and, in 1997, assumed responsibility for the continuing maintenance and revision of the UML.
- We present the terminology and notation of the current version of the UML—UML version 2—throughout the C++ section of this book.

What Is the UML?

- The Unified Modeling Language is now the most widely used graphical representation scheme for modeling object-oriented systems.

15.15 Introduction to Object Technology and the UML (Cont.)

- Those who design systems use the language (in the form of diagrams) to model their systems, as we do throughout the C++ section of this book.
- An attractive feature of the UML is its flexibility.
- The UML is extensible (i.e., capable of being enhanced with new features) and is independent of any particular OOAD process.
- UML modelers are free to use various processes in designing systems, but all developers can now express their designs with one standard set of graphical notations.
- For more information, visit our UML Resource Center at `www.deitel.com/UML/`.