18

C++ as a Better C; Introducing Object Technology

… no science of behavior can change the essential nature of man …

—Burrhus Frederic Skinner

Nothing can have value without being an object of utility.

—Karl Marx

The new electronic interdependence recreates the world in the image of a global village.

—Marshall Herbert McLuhan

Knowledge is the conformity of the object and the intellect.

—Averroës

Many things, having full reference
To one consent, may work contrariously …

—William Shakespeare

A wise skepticism is the first attribute of a good critic.

—James Russell Lowell

OBJECTIVES

In this chapter you will learn:

- Several C++ enhancements to C.
- The header files of the C++ Standard Library.
- To use `inline` functions.
- To create and manipulate references.
- To use default function arguments.
- To use the unary scope resolution operator to access a global variable in a scope that contains a local variable of the same name.
- To overload function definitions.
- To create and use function templates that perform identical operations on many different types.

Outline

Outline

18.1 Introduction

- **The C++ section introduces two additional programming paradigms**
  - **Object-oriented programming**
    - **With classes, encapsulation, objects, operator overloading, inheritance and polymorphism**
  - **Generic programming**
    - **With function templates and class templates**
- **Emphasize “crafting valuable classes” to create reusable software componentry**

18.2 C++

- C++ improves on many of C's features and provides object-oriented-programming (OOP) capabilities
  - Increase software productivity, quality and reusability
- 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
  - Originally called "C with classes"
  - The name C++ includes C's increment operator (++)
    - Indicate that C++ is an enhanced version of C
- C++ standardized in the United States through the American National Standards Institute (ANSI) and worldwide through the International Standards Organization (ISO)

Outline

```
1  // Fig. 18.1: fig18_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     int number1; // first integer to add
8
9     std::cout << "Enter first integer: "; // prompt user for data
10    std::cin >> number1; // read first integer from user into number1
11
12    int number2; // second integer to add
13    int sum; // sum of number1 and number2
14
15    std::cout << "Enter second integer: "; // prompt user for data
16    std::cin >> number2; // read second integer from user into number2
17    sum = number1 + number2; // add the numbers; store result in sum
18    std::cout << "Sum is " << sum << std::endl; // display sum; end line
19
20    return 0; // indicate that program ended successfully
21 } // end function main
```

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

fig18_01.cpp

Include the contents of the `iostream`

Declare integer variables

Use stream extraction operator with standard input stream to obtain user input

Stream manipulator **`std::endl`** outputs a newline, then "flushes output buffer"

Concatenating, chaining or cascading stream insertion operations

18.3 A Simple Program: Adding Two Integers

- C++ file names can have one of several extensions
  - Such as: .cpp, .cxx or .C (uppercase)
- Commenting
  - A // comment is a maximum of one line long
  - A /*…*/ C-style comments can be more than one line long
- `iostream`
  - Must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output
- C++ requires you to specify the return type, possibly `void`, for all functions
  - Specifying a parameter list with empty parentheses is equivalent to specifying a `void` parameter list in C

Common Programming Error 18.1

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

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

- **Declarations can be placed almost anywhere in a C++ program**
  - **They must appear before their corresponding variables are used in the program**
- **Input stream object**
  - **`std::cin` from `<iostream>`**
    - **Usually connected to keyboard**
    - **Stream extraction operator `>>`**
      - **Waits for user to input value, press *Enter* (Return) key**
      - **Stores value in variable to right of operator**
        - **Converts value to variable data type**
    - **Example**
      - `std::cin >> number1;`
        - **Reads an integer typed at the keyboard**
        - **Stores the integer in variable `number1`**

Good Programming Practice 18.1

Always place a blank line between a declaration and adjacent executable statements. This makes the declarations stand out in the program, enhancing program clarity.

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

- **Stream manipulator `std::endl`**
  - **Outputs a newline**
  - **Flushes the output buffer**
- **The notation `std::cout` specifies that we are using a name (`cout`) that belongs to a "namespace" (`std`)**

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

- **Concatenating stream insertion operations**
  - **Use multiple stream insertion operators in a single statement**
    - **Stream insertion operation knows how to output each type of data**
  - **Also called chaining or cascading**
  - **Example**
    - `std::cout << "Sum is " << number1 + number2 << std::endl;`
      - **Outputs `"Sum is “`**
      - **Then, outputs sum of `number1` and `number2`**
      - **Then, outputs newline and flushes output buffer**

18.4 C++ Standard Library

- **C++ programs**
  - **Built from pieces called classes and functions**
- **C++ Standard Library**
  - **Rich collections of existing classes and functions**
    - **Reusable in new applications**

Software Engineering Observation 18.1

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 18.2

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.

Performance Tip 18.1

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 18.1

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

18.5 Header Files

- **C++ Standard Library header files**
  - **Each contains a portion of the Standard Library**
    - **Function prototypes for the related functions**
    - **Definitions of various class types and functions**
    - **Constants needed by those functions**
  - **"Instruct" the compiler on how to interface with library and user-written components**
  - **Header file names ending in `.h`**
    - **Are "old-style" header files**
    - **Superseded by the C++ Standard Library header files**
  - **Use `#include` directive to include class in a program**

C++ Standard Library header files	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and standard output functions. This header file replaces header file `<iostream.h>`. This header is discussed in detail in Chapter 26, Stream Input/Output.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header file replaces header file `<iomanip.h>`. This header is used in Chapter 26, Stream Input/Output.
`<cmath>`	Contains function prototypes for math library functions. This header file replaces header file `<math.h>`.
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. This header file replaces header file `<stdlib>`.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header file replaces header file `<time.h>`.
`<vector>`, `<list>` `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution.

Fig. 18.2 | C++ Standard Library header files. (Part 1 of 3.)

C++ Standard Library header files	Explanation
`<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. This header file replaces header file `<ctype.h>`.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header file replaces header file `<string.h>`.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time).
`<exception>`, `<stdexcept>`	These header files contain classes that are used for exception handling (discussed in Chapter 27, Exception Handling).
`<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 27, Exception Handling.
`<fstream>`	Contains function prototypes for functions that perform input from files on disk and output to files on disk. This header file replaces header file `<fstream.h>`.
`<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.

Fig. 18.2 | C++ Standard Library header files. (Part 2 of 3.)

C++ Standard Library header files	Explanation
`<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. This replaces header file `<assert.h>` from pre-standard C++.
`<cfloat>`	Contains the floating-point size limits of the system. This header file replaces header file `<float.h>`.
`<climits>`	Contains the integral size limits of the system. This header file replaces header file `<limits.h>`.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions and information used by them. This header file replaces header file `<stdio.h>`.
`<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, and so on).
`<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 header files.

Fig. 18.2 | C++ Standard Library header files. (Part 3 of 3.)

18.6 Inline Functions

- **Inline functions**
  - **Reduce function call overhead—especially for small functions**
  - **Qualifier `inline` 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**
  - **Trade-off of inline functions**
    - **Multiple copies of the function code are inserted in the program (often making the program larger)**
  - **The compiler can ignore the `inline` qualifier and typically does so for all but the smallest functions**

Software Engineering Observation 18.3

Any change to an `inline` function could require all clients of the function to be recompiled. This can be significant in some program development and maintenance situations.

Performance Tip 18.2

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

Software Engineering Observation 18.4

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

Outline

fig18_03.cpp

(1 of 2)

```
1  // Fig. 18.3: fig18_03.cpp
2  // Using an inline function to calculate 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 } // end function cube
15
16 int main()
17 {
18    double sideValue; // stores value entered by user
19
```

inline qualifier

Complete function definition so the compiler knows how to expand a **cube** function call into its inlined code.

Outline

fig18_03.cpp

(2 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
30    return 0; // indicates successful termination
31 } // end main
```

cube function call that could be inlined

```
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
```

Software Engineering Observation 18.5

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.

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`			

Fig. 18.4 | C++ keywords. (Part 1 of 2.)

C++ keywords

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. 18.4 | C++ keywords. (Part 2 of 2.)

18.6 Inline Functions (Cont.)

- `using` statements help eliminate the need to repeat the namespace prefix
  - Ex: `std::`
- `for` statement's condition evaluates to either 0 (false) or nonzero (true)
  - Type `bool` represents 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
      - When non-boolean values are converted to type `bool`, non-zero values become `true`, and zero or `null` pointer values become `false`

18.7 References and Reference Parameters

- **Two ways to pass arguments to functions**
  - **Pass-by-value**
    - **A *copy* of the argument's value is passed to the called function**
    - **Changes to the copy do not affect the original variable's value in the caller**
      - **Prevents accidental side effects of functions**
  - **Pass-by-reference**
    - **Gives called function the ability to access and modify the caller's argument data directly**

Performance Tip 18.3

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.

18.7 References and Reference Parameters (Cont.)

- **Reference Parameter**
  - **An alias for its corresponding argument in a function call**
  - **& placed after the parameter type in the function prototype and function header**
  - **Example**
    - **`int &count` in a function header**
      - **Pronounced as "`count` is a reference to an `int`"**
  - **Parameter name in the body of the called function actually refers to the original variable in the calling function**

Performance Tip 18.4

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 18.6

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

Outline

```
1  // Fig. 18.5: fig18_05.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int squareByValue( int ); // function prototype (value pass)
8  void squareByReference( int & ); // function prototype (reference pass)
9
10 int main()
11 {
12    int x = 2; // value to square using squareByValue
13    int z = 4; // value to square using squareByReference
14
15    // demonstrate squareByValue
16    cout << "x = " << x << " before squareByValue\n";
17    cout << "Value returned by squareByValue: "
18       << squareByValue( x ) << endl;
19    cout << "x = " << x << " after squareByValue\n" <<
20
21    // demonstrate squareByReference
22    cout << "z = " << z << " before squareByReference" << endl;
23    squareByReference( z );
24    cout << "z = " << z << " after squareByReference" << endl;
25    return 0; // indicates successful termination
26 } // end main
```

Function illustrating pass-by-value

Function illustrating pass-by-reference

Variable is simply mentioned by name in both function calls

fig18_05.cpp

(1 of 2)

Outline

```
27
28 // squareByValue multiplies number by itself, stores the
29 // result in number and returns the new value of number
30 int squareByValue( int number )
31 {
32    return number *= number; // caller's argument not modified
33 } // end function squareByValue
34
35 // squareByReference multiplies numberRef by itself and stores the result
36 // in the variable to which numberRef refers in the caller
37 void squareByReference( int &numberRef )
38 {
39    numberRef *= numberRef; // caller's argument modified
40 } // end function squareByReference
```

Receives copy of argument in `main`

Receives reference to argument in `main`

Modifies variable in `main`

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

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

fig18_05.cpp

(2 of 2)

Common Programming Error 18.2

Because reference parameters are mentioned only by name in the body of the called function, the programmer 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 18.5

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 18.7

Many programmers do not bother to declare parameters passed by value as `const`, even when the called function should not be modifying 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.

Software Engineering Observation 18.8

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.

18.7 References and Reference Parameters (Cont.)

- **References**
  - **Can also be used as aliases for other variables within a function**
    - **All operations supposedly performed on the alias (i.e., the reference) are actually performed on the original variable**
    - **An alias is simply another name for the original variable**
    - **Must be initialized in their declarations**
      - **Cannot be reassigned afterward**
  - **Example**
    - ```
      int count = 1;
      int &cRef = count;
      cRef++;
      ```
      - **Increments `count` through alias `cRef`**


      ```

Outline

fig18_06.cpp

```
1  // Fig. 18.6: fig18_06.cpp
2  // References must be initialized.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y = x; // y refers to (is an alias for) x
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7; // actually modifies x
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

Creating a reference as an alias to another variable in the function

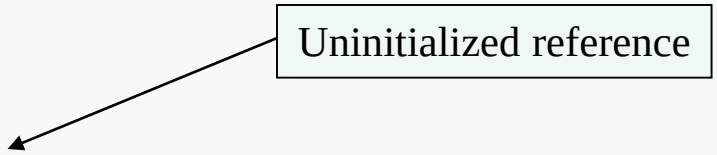
Assign **7** to **x** through alias **y**

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

Outline

fig18_07.cpp

```
1  // Fig. 18.7: fig18_07.cpp
2  // References must be initialized.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y; // Error: y must be initialized
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7;
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```



Borland C++ command-line compiler error message:

```
Error E2304 C:\examples\ch18\Fig18_07\fig18_07.cpp 10:
   Reference variable 'y' must be initialized in function main()
```

Microsoft Visual C++ compiler error message:

```
C:\examples\ch18\Fig18_07\fig18_07.cpp(10) : error C2530: 'y' :
   references must be initialized
```

GNU C++ compiler error message:

```
fig18_07.cpp:10: error: 'y' declared as a reference but not initialized
```

18.7 References and Reference Parameters (Cont.)

- **Returning a reference from a function**
  - **Functions can return references to variables**
    - **Should only be used when the variable is `static`**
  - **Dangling reference**
    - **Returning a reference to an automatic variable**
      - **That variable no longer exists after the function ends**

Common Programming Error 18.3

Not initializing a reference variable when it is 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 are declared is called.

Common Programming Error 18.4

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 18.5

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

18.8 Empty Parameter Lists

- **Empty parameter list**
  - **Specified by writing either `void` or nothing at all in parentheses**
  - **For example:**

```
void print();
void print(void);
```

Specifies that function `print` does not take arguments and does not return a value

Portability Tip 18.2

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 explicitly takes no arguments. Thus, C programs using this feature might cause compilation errors when compiled in C++.

18.9 Default Arguments

- **Default argument**
  - **A default value to be passed to a parameter**
    - **Used when the function call does not specify an argument for that parameter**
  - **Must be the rightmost argument(s) in a function's parameter list**
  - **Should be specified with the first occurrence of the function name**
    - **Typically the function prototype**

Outline

fig18_08.cpp

(1 of 2)

```
1  // Fig. 18.8: fig18_08.cpp
2  // Using default arguments.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function prototype that specifies default arguments
8  int boxVolume( int length = 1, int width = 1, int height = 1 );
9
10 int main()
11 {
12    // no arguments--use default values for all dimensions
13    cout << "The default box volume is: " << boxVolume();
14
15    // specify length; default width and height
16    cout << "\n\nThe volume of a box with length 10,\n"
17       << "width 1 and height 1 is: " << boxVolume( 10 );
18
19    // specify length and width; default height
20    cout << "\n\nThe volume of a box with length 10,\n"
21       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
22
23    // specify all arguments
24    cout << "\n\nThe volume of a box with length 10,\n"
25       << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
26       << endl;
27    return 0; // indicates successful termination
28 } // end main
```

Default arguments

Calling function with no arguments

Calling function with one argument

Calling function with two arguments

Calling function with three arguments

Outline

```
29
30 // function boxVolume calculates the volume of a box
31 int boxVolume( int length, int width, int height )
32 {
33    return length * width * height;
34 } // end function boxVolume
```

```
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
```

fig18_08.cpp

Note that default arguments were specified in the function prototype, so they are not specified in the function header

Common Programming Error 18.6

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

Good Programming Practice 18.2

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

Software Engineering Observation 18.9

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

Common Programming Error 18.7

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.

18.10 Unary Scope Resolution Operator

- **Unary scope resolution operator (::)**
  - **Used to access a global variable when a local variable of the same name is in scope**
  - **Cannot be used to access a local variable of the same name in an outer block**

Outline

fig18_09.cpp

```
1  // Fig. 18.9: fig18_09.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int number = 7; // global variable named number
8
9  int main()
10 {
11    double number = 10.5; // local variable named number
12
13    // display values of local and global variables
14    cout << "Local double value of number = " << number
15       << "\nGlobal int value of number = " << ::number << endl;
16    return 0; // indicates successful termination
17 } // end main
```

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

Unary scope resolution operator used to access global variable **number**

Common Programming Error 18.8

It is 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 18.3

Always using the unary scope resolution operator (`::`) to refer to global variables makes programs easier to read and understand, because it makes it clear that you intend to access a global variable rather than a nonglobal variable.

Software Engineering Observation 18.10

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 18.1

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

Error-Prevention Tip 18.2

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.

18.11 Function Overloading

- **Overloaded functions**
  - **Overloaded functions have**
    - **Same name**
    - **Different sets of parameters**
  - **Compiler selects proper function to execute based on number, types and order of arguments in the function call**
  - **Commonly used to create several functions of the same name that perform similar tasks, but on different data types**

Good Programming Practice 18.4

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

Outline

fig18_10.cpp

```
1  // Fig. 18.10: fig18_10.cpp
2  // Overloaded functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function square for int values
8  int square( int x )
9  {
10    cout << "square of integer " << x << " is ";
11    return x * x;
12 } // end function square with int argument
13
14 // function square for double values
15 double square( double y )
16 {
17    cout << "square of double " << y << " is ";
18    return y * y;
19 } // end function square with double argument
20
21 int main()
22 {
23    cout << square( 7 ); // calls int version
24    cout << endl;
25    cout << square( 7.5 ); // calls double version
26    cout << endl;
27    return 0; // indicates successful termination
28 } // end main
```

Defining a **square** function for **int**s

Defining a **square** function for **double**s

Output confirms that the proper function was called in each case

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

18.11 Function Overloading (Cont.)

- **How the compiler differentiates overloaded functions**
  - **Overloaded functions are distinguished by their signatures**
    - **Signature is a combination of a function's name and its parameter types (in order)**
  - **Name mangling or name decoration**
    - **Compiler encodes each function identifier with the number and types of its parameters to enable type-safe linkage**
  - **Type-safe linkage ensures that**
    - **Proper overloaded function is called**
    - **Types of the arguments conform to types of the parameters**

Outline

fig18_11.cpp

(1 of 2)

```
1  // Fig. 18.11: fig18_11.cpp
2  // Name mangling.
3
4  // function square for int values
5  int square( int x )
6  {
7     return x * x;
8  } // end function square
9
10 // function square for double values
11 double square( double y )
12 {
13    return y * y;
14 } // end function square
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 } // end function nothing1
```

Overloaded **square** functions

Outline

fig18_11.cpp

(2 of 2)

```
22
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 } // end function nothing2
29
30 int main()
31 {
32    return 0; // indicates successful termination
33 } // end main
```

```
@square$qi
@square$qd
@nothing1$qifcri
@nothing2$qcirfrd
_main
```

Mangled names of overloaded functions

main is not mangled because it cannot be overloaded

Common Programming Error 18.9

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

Common Programming Error 18.10

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 use that function name in a call passing no arguments. The compiler does not know which version of the function to choose.

18.12 Function Templates

- **Function templates**
  - **More compact and convenient form of overloading**
    - **Identical program logic and operations for each data type**
  - **Function template definition**
    - **Written by programmer once**
    - **Essentially defines a whole family of overloaded functions**
    - **Begins with the `template` keyword**
    - **Contains template parameter list of formal type parameters for the function template enclosed in angle brackets (<>)**
    - **Formal type parameters**
      - **Preceded by keyword `typename` or keyword `class`**
      - **Placeholders for fundamental types or user-defined types**

18.12 Function Templates(Cont.)

- **Function-template specializations**
  - **Generated automatically by the compiler to handle each type of call to the function template**
  - **Example for function template `max` with type parameter `T` called with `int` arguments**
    - **Compiler detects a `max` invocation in the program code**
    - **`int` is substituted for `T` throughout the template definition**
    - **This produces function-template specialization `max< int >`**

Outline

```
1  // Fig. 18.12: maximum.h
2  // Definition of function template maximum.
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 } // end function template maximum
```

Using formal type parameter **T** in place of data type

Common Programming Error 18.11

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.

Outline

fig18_13.cpp

(1 of 2)

```
1  // Fig. 18.13: fig18_13.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include "maximum.h" // include definition of function template maximum
9
10 int main()
11 {
12    // demonstrate maximum with int values
13    int int1, int2, int3;
14
15    cout << "Input three integer values: ";
16    cin >> int1 >> int2 >> int3;
17
18    // invoke int version of maximum
19    cout << "The maximum integer value is: "
20       << maximum( int1, int2, int3 );
21
22    // demonstrate maximum with double values
23    double double1, double2, double3;
24
25    cout << "\n\nInput three double values: ";
26    cin >> double1 >> double2 >> double3;
27
```

Invoking **maximum** with **int** arguments

```
28    // invoke double version of maximum
29    cout << "The maximum double value is: "
30       << maximum( double1, double2, double3 );
31
32    // demonstrate maximum with char values
33    char char1, char2, char3;
34
35    cout << "\n\nInput three characters: ";
36    cin >> char1 >> char2 >> char3;
37
38    // invoke char version of maximum
39    cout << "The maximum character value is: "
40       << maximum( char1, char2, char3 ) << endl;
41    return 0; // indicates successful termination
42 } // end main
```

Invoking **maximum** with **double** arguments

Invoking **maximum** with **char** arguments

fig18_13.cpp

(2 of 2)

```
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
```

18.13 Introduction to Object Technology and the UML

- **Object orientation**
  - **A natural way of thinking about the world and computer programs**
- **Unified Modeling Language (UML)**
  - **Graphical language that uses common notation**
  - **Allows developers to represent object-oriented designs**

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

- **Objects**
  - **Reusable software components that model real-world items**
  - **Examples are all around you**
    - **People, animals, cars, telephones, microwave ovens, etc.**
  - **Have attributes**
    - **Size, shape, color, weight, etc.**
  - **Exhibit behaviors**
    - **Babies cry, crawl, sleep, etc.; cars accelerate, brake, turn, etc.**

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

- **Object-oriented design (OOD)**
  - **Models real-world objects in software**
  - **Models communication among objects**
  - **Encapsulates attributes and operations (behaviors)**
    - **Information hiding**
    - **Communication through well-defined interfaces**
- **Object-oriented language**
  - **Programming in object oriented languages is called object-oriented programming (OOP)**
  - **C++ is an object-oriented language**
    - **Programmers can create user-defined types called classes**
      - **Contain data members (attributes) and member functions (behaviors)**

Software Engineering Observation 18.11

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

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

- **Object-Oriented Analysis and Design (OOAD)**
  - **Analyze program requirements, then develop solution**
  - **Essential for large programs**
  - **Plan in pseudocode or UML**

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

- **History of the UML**
  - **Used to approach OOAD**
  - **Object Management Group (OMG) supervised**
    - **Brainchild of Booch, Rumbaugh and Jacobson**
  - **Version 2 is current version**
- **UML**
  - **Graphical representation scheme**
  - **Enables developers to model object-oriented systems**
  - **Flexible and extendible**