مبانی برنامه نویسی
به زبان سی

۱۴، ۱۶ و ۱۸ دی ۱۳۹۹

جلسه های ۲۶، ۲۷ و ۲۸

ملکی مجد

مباحث این هفته:

- ادامه استراکت
  - مثال بُر زدن کارت
- Enumeration

- آشنایی با زبان برنامه نویسی C++
  - مثال ساده جمع دو عدد
- کتابخانه های استاندارد C++
- تابع های inline
- پارامتر References
- Default Arguments
- عملگر ::
- Function Overloading
- مقدمه ای بر مفهوم شی و UML

10.6 typedef

- The keyword `typedef` provides a mechanism for creating synonyms (or aliases) for previously defined data types.
  - For example, the statement
    - `typedef struct card Card;`
    - defines the new type name `Card` as a synonym for type `struct card`.
  - For example, the following definition
    - ```
      typedef struct {
         char *face;
         char *suit;
      } Card;
      ```

  creates the structure type `Card` without the need for a separate `typedef` statement.
      ```

10.7 Example: High-Performance Card Shuffling and Dealing Simulation

- The program in Fig. 10.3 is based on the card shuffling and dealing simulation discussed in Chapter 7.
- The program represents the deck of cards as an array of structures.

- The declaration
  - `Card deck[ 52 ];`

  declares an array of 52 `Card` structures (i.e., variables of type `struct card`).

```
1  /* Fig. 10.3: fig10_03.c
2     The card shuffling and dealing program using structures */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  /* card structure definition */
8  struct card {
9     const char *face; /* define pointer face */
10    const char *suit; /* define pointer suit */
11 }; /* end structure card */
12
13 typedef struct card Card; /* new type name for struct card */
14
15 /* prototypes */
16 void fillDeck( Card * const wDeck, const char * wFace[],
17    const char * wSuit[] );
18 void shuffle( Card * const wDeck );
19 void deal( const Card * const wDeck );
20
21 int main( void )
22 {
```

initializes the `Card` array in order with Ace through King of each suit

Fig. 10.3 | High-performance card shuffling and dealing simulation. (Part 1 of 4.)

```
23      Card deck[ 52 ]; /* define array of Cards */
24
25      /* initialize array of pointers */
26      const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
27         "Six", "Seven", "Eight", "Nine", "Ten",
28         "Jack", "Queen", "King"};
29
30      /* initialize array of pointers */
31      const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
32
33      srand( time( NULL ) ); /* randomize */
34
35      fillDeck( deck, face, suit ); /* load the deck with Cards */
36      shuffle( deck ); /* put Cards in random order */
37      deal( deck ); /* deal all 52 Cards */
38      return 0; /* indicates successful termination */
39   } /* end main */
40
41   /* place strings into Card structures */
42   void fillDeck( Card * const wDeck, const char * wFace[],
43      const char * wSuit[] )
44   {
45      int i; /* counter */
46
```

Fig. 10.3 | High-performance card shuffling and dealing simulation. (Part 2 of 4.)

```
47      /* loop through wDeck */
48      for ( i = 0; i <= 51; i++ ) {
49         wDeck[ i ].face = wFace[ i % 13 ];
50         wDeck[ i ].suit = wSuit[ i / 13 ];
51      } /* end for */
52   } /* end function fillDeck */
53
54   /* shuffle cards */
55   void shuffle( Card * const wDeck )
56   {
57      int i; /* counter */
58      int j; /* variable to hold random value between 0 - 51 */
59      Card temp; /* define temporary structure for swapping Cards */
60
61      /* loop through wDeck randomly swapping Cards */
62      for ( i = 0; i <= 51; i++ ) {
63         j = rand() % 52;
64         temp = wDeck[ i ];
65         wDeck[ i ] = wDeck[ j ];
66         wDeck[ j ] = temp;
67      } /* end for */
68   } /* end function shuffle */
69
```

This algorithm cannot suffer from indefinite postponement like the shuffling algorithm presented in Chapter 7.

Fig. 10.3 | High-performance card shuffling and dealing simulation. (Part 3 of 4.)

```
70  /* deal cards */
71  void deal( const Card * const wDeck )
72  {
73     int i; /* counter */
74
75     /* loop through wDeck */
76     for ( i = 0; i <= 51; i++ ) {
77        printf( "%5s of %-8s%s", wDeck[ i ].face, wDeck[ i ].suit,
78           ( i + 1 ) % 4 ? "  " : "\n" );
79     } /* end for */
80  } /* end function deal */
```

Fig. 10.3 | High-performance card shuffling and dealing simulation. (Part 4 of 4.)

```
Three of Hearts     Jack of Clubs     Three of Spades      Six of Diamonds
 Five of Hearts    Eight of Spades    Three of Clubs     Deuce of Spades
 Jack of Spades     Four of Hearts    Deuce of Hearts      Six of Clubs
Queen of Clubs     Three of Diamonds  Eight of Diamonds   King of Clubs
 King of Hearts    Eight of Hearts    Queen of Hearts    Seven of Clubs
Seven of Diamonds   Nine of Spades     Five of Clubs     Eight of Clubs
  Six of Hearts    Deuce of Diamonds   Five of Spades     Four of Clubs
Deuce of Clubs      Nine of Hearts    Seven of Hearts     Four of Spades
  Ten of Spades     King of Diamonds    Ten of Hearts     Jack of Diamonds
 Four of Diamonds    Six of Spades     Five of Diamonds    Ace of Diamonds
  Ace of Clubs      Jack of Hearts     Ten of Clubs      Queen of Diamonds
  Ace of Hearts      Ten of Diamonds   Nine of Clubs      King of Spades
  Ace of Spades     Nine of Diamonds  Seven of Spades    Queen of Spades
```

Fig. 10.4 | Output for the high-performance card shuffling and dealing simulation.

10.10 Enumeration Constants

- C provides one final user-defined type called an enumeration.
- An enumeration, introduced by the keyword **`enum`**, is a set of integer enumeration constants represented by identifiers.
  - Values in an `enum` start with `0`, unless specified otherwise, and are incremented by `1`.

- For example, the enumeration
  - ```
    enum months {
        JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
        OCT, NOV, DEC };
    ```

    creates a new type, `enum months`, in which the identifiers are set to the integers `0` to `11`, respectively.
    ```

10.11 Enumeration Constants (Cont.)

- To number the months 1 to 12, use the following enumeration:
  - ```
    enum months {
        JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
        SEP, OCT, NOV, DEC };
    ```
    - Since the first value in the preceding enumeration is explicitly set to 1, the remaining values are incremented from 1, resulting in the values 1 through 12.
- The identifiers in an enumeration must be unique.
- The value of each enumeration constant of an enumeration can be set explicitly in the definition by assigning a value to the identifier.
- Multiple members of an enumeration can have the same constant value.
    ```

Common Programming Error 10.15

Assigning a value to an enumeration constant after it has been defined is a syntax error.

```c
/* Fig. 10.18: fig10_18.c
   Using an enumeration type */
#include <stdio.h>

/* enumeration constants represent months of the year */
enum months {
   JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };

int main( void )
{
   enum months month; /* can contain any of the 12 months */

   /* initialize array of pointers */
   const char *monthName[] = { "", "January", "February", "March",
      "April", "May", "June", "July", "August", "September", "October",
      "November", "December" };

   /* loop through months */
   for ( month = JAN; month <= DEC; month++ ) {
      printf( "%2d%11s\n", month, monthName[ month ] );
   } /* end for */

   return 0; /* indicates successful termination */
} /* end main */
```

Fig. 10.18 | Using an enumeration. (Part 1 of 2.)

```
 1    January
 2   February
 3      March
 4      April
 5        May
 6       June
 7       July
 8     August
 9  September
10    October
11   November
12   December
```

Fig. 10.18 | Using an enumeration. (Part 2 of 2.)

15.1 Introduction

- 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).

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.
  - The name C++ includes C's increment operator (++) to indicate that C++ is an enhanced version of C.

15.3 A Simple Program: Adding Two Integers

- In what follows we illustrate 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).

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     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 } // end function main
```

```
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.)

- 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.
- Line 10 uses the standard input stream object
  - **`std::cin`**—and the stream extraction operator, **`>>`**, to obtain a value from the keyboard.
  - Using the stream extraction operator with `std::cin` takes character input from the standard input stream, which is usually the keyboard.
  - 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.)

- 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`.

  - Soon, we introduce the `using` statement, which will enable us to avoid placing `std::` before each use of a namespace `std` name.

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

- We could have combined the statements in lines 17 and 18 into the statement
  - `std::cout << "Sum is " << number1 + number2 << std::endl;`

You'll notice that we did not have a `return 0;` statement at the end of main in this example.

- 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`.

15.4 C++ Standard Library

- C++ programs consist of pieces called **classes** and **functions**.
- You can take advantage of the rich collections of existing classes and functions in the C++ Standard Library.

1. learning the C++ language itself;
2. learning how to use the classes and functions in the C++ Standard Library.

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.*

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.

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.
  - 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 standard output functions. This header file replaces header file `<iostream.h>`. This header is discussed in detail in Chapter 2, 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 2, 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>`.

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

C++ Standard Library header file	Explanation
`<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.
`<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 2, Exception Handling).

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

C++ Standard Library header file	Explanation
`<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 2, 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.
`<iterator>`	Contains classes for accessing C++ Standard Library container data.
`<algorithm>`	Contains functions for manipulating container data.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This replaces header file `<assert.h>` from pre-standard C++.

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

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.
  - 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.

```
1  // Fig. 15.3: fig15_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
```

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

Complete definition of function `cube` appears before it's used in the program so that the compiler knows how to expand a `cube` function call into its inlined code.

```
16  int main()
17  {
18     double sideValue; // stores value entered by user
19
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  } // end main
```

```
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.)

- Namespaces provide a method for preventing name conflicts in large projects.
  - Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes.
  - Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

- 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.)

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

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.

15.7 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.
  - 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.

15.7 References and Reference Parameters (Cont.)

- 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.

15.7 References and Reference Parameters (Cont.)

- A reference parameter is an alias for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference: follow the parameter's type in the function prototype by an ampersand (&);
  - For example, the following declaration in a function header
    - `int &count`

    read from right to left is pronounced "`count` is a reference to an `int`."
- In the function call, simply mention the variable by name to pass it by reference.
  - 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.

```
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    int x = 2; // value to square using squareByValue
12    int z = 4; // value to square using squareByReference
13
14    // demonstrate squareByValue
15    cout << "x = " << x << " before squareByValue\n";
16    cout << "Value returned by squareByValue: "
17       << squareByValue( x ) << endl;
18    cout << "x = " << x << " after squareByValue\n" << endl;
19
20    // demonstrate squareByReference
21    cout << "z = " << z << " before squareByReference" << endl;
22    squareByReference( z );
23    cout << "z = " << z << " after squareByReference" << endl;
24 } // end main
```

both variables are simply mentioned by name in the function calls

Fig. 15.5 | Passing arguments by value and by reference. (Part 1 of 2.)

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

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

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

Fig. 15.5 | Passing arguments by value and by reference. (Part 2 of 2.)

15.7 References and Reference Parameters (Cont.)

- References can also be used as aliases for other variables within a function.
- For example, 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`.
- Reference variables must be initialized in their declarations
  - 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.
    ```

```cpp
// Fig. 15.6: fig15_06.cpp
// References must be initialized.
#include <iostream>
using namespace std;

int main()
{
   int x = 3;
   int &y = x; // y refers to (is an alias for) x

   cout << "x = " << x << endl << "y = " << y << endl;
   y = 7; // actually modifies x
   cout << "x = " << x << endl << "y = " << y << endl;
} // end main
```

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

Fig. 15.6 | Initializing and using a reference.

```
1  // Fig. 15.7: fig15_07.cpp
2  // References must be initialized.
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 } // end main
```

Microsoft Visual C++ compiler error message:

```
C:\examples\ch15\fig15_07\fig15_07.cpp(10) : error C2530: 'y' :
   references must be initialized
```

GNU C++ compiler error message:

```
fig15_07.cpp:10: error: 'y' declared as a reference but not initialized
```

Fig. 15.7 | Uninitialized reference causes a syntax error.

15.9 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, the programmer 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.
  - Default arguments must be the rightmost (trailing) arguments in a function's parame-ter list.
  - Default arguments should be specified with the first occurrence of the function name
    - typically, in the function prototype.

```
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 | Default arguments to a function. (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   } // end main
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   } // end function boxVolume
```

Common Programming Error 15.6

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

```
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 | Default arguments to a function. (Part 2 of 2.)

15.10 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.

```
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 } // end main
```

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

Fig. 15.9 | Using the unary scope resolution operator.

15.11 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.
- Function overloading is commonly used to create several 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.

```
1  // Fig. 15.10: fig15_10.cpp
2  // Overloaded 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 } // end function square with int argument
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 } // end function square with double argument
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  } // end main
```

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

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

15.11 Function Overloading (Cont.)

- 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 to enable type-safe linkage.
  - This ensures that the proper overloaded function is called and that the argument types conform to the parameter types.

15.13 Introduction to Object Technology and the UML

- 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.

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

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.

15.13 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.

15.13 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.
- 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.

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

- 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.

15.13 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.

15.13 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.

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

- 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.

15.13 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.

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

- 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).
  - 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.

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

- The programmer uses 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.

15.13 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.

15.13 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.

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

- 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?

15.13 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.

15.13 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 these first few chapters do not require an exhaustive OOAD process.

15.13 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.
  - This language, known as the Unified Modeling Language (UML), was developed in the mid-1990s

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

- The Unified Modeling Language is now the most widely used graphical representation scheme for modeling object-oriented systems.
- Those who design systems use the language (in the form of diagrams) to model their systems
  - 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.