6.5 Function Prototypes and Argument Coercion

- Function prototype
  - Also called a function declaration
  - Indicates to the compiler:
    - Name of the function
    - Type of data returned by the function
    - Parameters the function expects to receive
      - Number of parameters
      - Types of those parameters
      - Order of those parameters

Software Engineering Observation 6.6

Function prototypes are required in C++. Use `#include` preprocessor directives to obtain function prototypes for the C++ Standard Library functions from the header files for the appropriate libraries (e.g., the prototype for math function `sqrt` is in header file `<cmath>`; a partial list of C++ Standard Library header files appears in Section 6.6). Also use `#include` to obtain header files containing function prototypes written by you or your group members.

Common Programming Error 6.3

If a function is defined before it is invoked, then the function's definition also serves as the function's prototype, so a separate prototype is unnecessary. If a function is invoked before it is defined, and that function does not have a function prototype, a compilation error occurs.

Software Engineering Observation 6.7

Always provide function prototypes, even though it is possible to omit them when functions are defined before they are used (in which case the function header acts as the function prototype as well). Providing the prototypes avoids tying the code to the order in which functions are defined (which can easily change as a program evolves).

6.5 Function Prototypes and Argument Coercion (Cont.)

- **Function signature (or simply signature)**
  - **The portion of a function prototype that includes the name of the function and the types of its arguments**
    - **Does not specify the function's return type**
  - **Functions in the same scope must have unique signatures**
    - **The scope of a function is the region of a program in which the function is known and accessible**

Common Programming Error 6.4

It is a compilation error if two functions in the same scope have the same signature but different return types.

6.5 Function Prototypes and Argument Coercion (Cont.)

- **Argument Coercion**
  - **Forcing arguments to the appropriate types specified by the corresponding parameters**
    - **For example, calling a function with an integer argument, even though the function prototype specifies a double argument**
      - **The function will still work correctly**

6.5 Function Prototypes and Argument Coercion (Cont.)

- C++ Promotion Rules
  - Indicate how to convert between types without losing data
  - Apply to expressions containing values of two or more data types
    - Such expressions are also referred to as mixed-type expressions
    - Each value in the expression is promoted to the "highest" type in the expression
      - Temporary version of each value is created and used for the expression
        - Original values remain unchanged

6.5 Function Prototypes and Argument Coercion (Cont.)

- C++ Promotion Rules (Cont.)
  - Promotion also occurs when the type of a function argument does not match the specified parameter type
    - Promotion is as if the argument value were being assigned directly to the parameter variable
  - Converting a value to a lower fundamental type
    - Will likely result in the loss of data or incorrect values
    - Can only be performed explicitly
      - By assigning the value to a variable of lower type (some compilers will issue a warning in this case) or
      - By using a cast operator

Data types	
`long double`	
`double`	
`float`	
`unsigned long int`	(synonymous with `unsigned long`)
`long int`	(synonymous with `long`)
`unsigned int`	(synonymous with `unsigned`)
`int`	
`unsigned short int`	(synonymous with `unsigned short`)
`short int`	(synonymous with `short`)
`unsigned char`	
`char`	
`bool`	

Fig. 6.6 | Promotion hierarchy for fundamental data types.

Common Programming Error 6.5

Converting from a higher data type in the promotion hierarchy to a lower type, or between signed and unsigned, can corrupt the data value, causing a loss of information.

Common Programming Error 6.6

It is a compilation error if the arguments in a function call do not match the number and types of the parameters declared in the corresponding function prototype. It is also an error if the number of arguments in the call matches, but the arguments cannot be implicitly converted to the expected types.

6.6 C++ Standard Library 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

C++ Standard Library header file	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and standard output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output. This header file replaces header file `<iostream.h>`.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header file is first used in Section 4.9 and is discussed in more detail in Chapter 15, Stream Input/Output. This header file replaces header file `<iomanip.h>`.
`<cmath>`	Contains function prototypes for math library functions (discussed in Section 6.3). 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. Portions of the header file are covered in Section 6.7; Chapter 11, Operator Overloading; String and Array Objects; Chapter 16, Exception Handling; Chapter 19, Web Programming; Chapter 22, Bits, Characters, C-Strings and `struct`s; and Appendix E, C Legacy Code Topics. This header file replaces header file `<stdlib.h>`.

Fig. 6.7 | C++ Standard Library header files. (Part 1 of 4)

C++ Standard Library header file	Explanation
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header file replaces header file `<time.h>`. This header file is used in Section 6.7.
`<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. The `<vector>` header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these header files in Chapter 23, Standard Template Library (STL).
`<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>`. These topics are discussed in Chapter 8, Pointers and Pointer-Based Strings, and Chapter 22, Bits, Characters, C-Strings and `struct`s.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header file replaces header file `<string.h>`. This header file is used in Chapter 11, Operator Overloading; String and Array Objects.

Fig. 6.7 | C++ Standard Library header files. (Part 2 of 4)

C++ Standard Library header file	Explanation
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header file is discussed in Section 13.8.
`<exception>`, `<stdexcept>`	These header files contain classes that are used for exception handling (discussed in Chapter 16).
`<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 16, Exception Handling.
`<fstream>`	Contains function prototypes for functions that perform input from files on disk and output to files on disk (discussed in Chapter 17, File Processing). This header file replaces header file `<fstream.h>`.
`<string>`	Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 18).
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class `string` and String Stream Processing).
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms. This header file is used in Chapter 23.

Fig. 6.7 | C++ Standard Library header files. (Part 3 of 4)

C++ Standard Library header file	Explanation
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header file is used in Chapter 23, Standard Template Library (STL).
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header file is used in Chapter 23.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This replaces header file `<assert.h>` from pre-standard C++. This header file is used in Appendix F, Preprocessor.
`<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, 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 header files.

Fig. 6.7 | C++ Standard Library header files. (Part 4 of 4)

6.7 Case Study: Random Number Generation

- C++ Standard Library function `rand`
  - Introduces the element of chance into computer applications
  - Example
    - `i = rand();`
      - Generates an unsigned integer between `0` and `RAND_MAX` (a symbolic constant defined in header file `<cstdlib>`)
  - Function prototype for the `rand` function is in `<cstdlib>`

6.7 Case Study: Random Number Generation (Cont.)

- To produce integers in a specific range, use the modulus operator (%) with `rand`
  - Example
    - `rand() % 6;`
      - Produces numbers in the range `0` to `5`
  - This is called scaling, `6` is the scaling factor
  - Shifting can move the range to `1` to `6`
    - `1 + rand() % 6;`

Outline

fig06_08.cpp

(1 of 2)

```
// Fig. 6.8: fig06_08.cpp
// Shifted and scaled random integers.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // contains function prototype for rand
using std::rand;

int main()
{
   // loop 20 times
   for ( int counter = 1; counter <= 20; counter++ )
   {
      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );
```

#include and **using** for function **rand**

Calling function **rand**

Outline

```
      // if counter is divisible by 5, start a new line of output
      if ( counter % 5 == 0 )
         cout << endl;
   } // end for

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

```
      6         6         5         5         6
      5         1         1         5         3
      6         6         2         4         2
      6         2         3         4         1
```

fig06_08.cpp

(2 of 2)

```cpp
// Fig. 6.9: fig06_09.cpp
// Roll a six-sided die 6,000,000 times.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // contains function prototype for rand
using std::rand;

int main()
{
   int frequency1 = 0; // count of 1s rolled
   int frequency2 = 0; // count of 2s rolled
   int frequency3 = 0; // count of 3s rolled
   int frequency4 = 0; // count of 4s rolled
   int frequency5 = 0; // count of 5s rolled
   int frequency6 = 0; // count of 6s rolled

   int face; // stores most recently rolled value

   // summarize results of 6,000,000 rolls of a die
   for ( int roll = 1; roll <= 6000000; roll++ )
   {
      face = 1 + rand() % 6; // random number from 1 to 6
```

Scaling and shifting the value produced by function **`rand`**

Outline

`fig06_09.cpp`

(1 of 3)

Outline

```
      // determine roll value 1-6 and increment appropriate counter
      switch ( face )
      {
         case 1:
            ++frequency1; // increment the 1s counter
            break;
         case 2:
            ++frequency2; // increment the 2s counter
            break;
         case 3:
            ++frequency3; // increment the 3s counter
            break;
         case 4:
            ++frequency4; // increment the 4s counter
            break;
         case 5:
            ++frequency5; // increment the 5s counter
            break;
         case 6:
            ++frequency6; // increment the 6s counter
            break;
         default: // invalid value
            cout << "Program should never get here!";
      } // end switch
   } // end for
```

fig06_09.cpp

Outline

```
   cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
   cout << "   1" << setw( 13 ) << frequency1
      << "\n   2" << setw( 13 ) << frequency2
      << "\n   3" << setw( 13 ) << frequency3
      << "\n   4" << setw( 13 ) << frequency4
      << "\n   5" << setw( 13 ) << frequency5
      << "\n   6" << setw( 13 ) << frequency6 << endl;
   return 0; // indicates successful termination
} // end main
```

```
Face    Frequency
   1       999702
   2      1000823
   3       999378
   4       998898
   5      1000777
   6      1000422
```

Each face value appears approximately 1,000,000 times

Error-Prevention Tip 6.3

Provide a `default` case in a `switch` to catch errors even if you are absolutely, positively certain that you have no bugs!

6.7 Case Study: Random Number Generation (Cont.)

- Function `rand`
  - Generates pseudorandom numbers
  - The same sequence of numbers repeats itself each time the program executes
- Randomizing
  - Conditioning a program to produce a different sequence of random numbers for each execution
- C++ Standard Library function `srand`
  - Takes an unsigned integer argument
  - Seeds the `rand` function to produce a different sequence of random numbers

Outline

```
// Fig. 6.10: fig06_10.cpp
// Randomizing die-rolling program.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include <iomanip>
using std::setw;

#include <cstdlib> // contains prototypes for functions srand and rand
using std::rand;
using std::srand;

int main()
{
   unsigned seed; // stores the seed entered by the user

   cout << "Enter seed: ";
   cin >> seed;
   srand( seed ); // seed random number generator

```

using statement for function **srand**

Data type **unsigned** is short for **unsigned int**

Passing **seed** to **srand** to randomize the program

fig06_10.cpp

(1 of 2)

```
   // loop 10 times
   for ( int counter = 1; counter <= 10; counter++ )
   {
      // pick random number from 1 to 6 and output it
      cout << setw( 10 ) << ( 1 + rand() % 6 );

      // if counter is divisible by 5, start a new line of output
      if ( counter % 5 == 0 )
         cout << endl;
   } // end for

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

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Enter seed: 432
         4         6         3         1         6
         3         1         5         4         2
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

Program outputs show that each unique seed value produces a different sequence of random numbers

6.7 Case Study: Random Number Generation (Cont.)

- **To randomize without having to enter a seed each time**
  - `srand( time( 0 ) );`
    - **This causes the computer to read its clock to obtain the seed value**
  - **Function `time` (with the argument `0`)**
    - **Returns the current time as the number of seconds since January 1, 1970 at midnight Greenwich Mean Time (GMT)**
    - **Function prototype for `time` is in `<ctime>`**

Common Programming Error 6.7

Calling function `srand` more than once in a program restarts the pseudorandom number sequence and can affect the randomness of the numbers produced by `rand`.

Common Programming Error 6.8

Using `srand` in place of `rand` to attempt to generate random numbers is a compilation error—function `srand` does not return a value.

6.7 Case Study: Random Number Generation (Cont.)

- Scaling and shifting random numbers
  - To obtain random numbers in a desired range, use a statement like

    `number =` *shiftingValue* `+ rand() %` *scalingFactor*`;`

    - *shiftingValue* is equal to the first number in the desired range of consecutive integers
    - *scalingFactor* is equal to the width of the desired range of consecutive integers
      - number of consecutive integers in the range

6.8 Case Study: Game of Chance and Introducing `enum`

- **Enumeration**
  - **A set of integer constants represented by identifiers**
    - **The values of enumeration constants start at `0`, unless specified otherwise, and increment by `1`**
    - **The identifiers in an `enum` must be unique, but separate enumeration constants can have the same integer value**
  - **Defining an enumeration**
    - **Keyword `enum`**
    - **A type name**
    - **Comma-separated list of identifier names enclosed in braces**
    - **Example**
      - `enum Months { JAN = 1, FEB, MAR, APR };`

```
// Fig. 6.11: fig06_11.cpp
// Craps simulation.
#include <iostream>
using std::cout;
using std::endl;

#include <cstdlib> // contains prototypes for functions srand and rand
using std::rand;
using std::srand;

#include <ctime> // contains prototype for function time
using std::time;

int rollDice(); // rolls dice, calculates amd displays sum

int main()
{
   // enumeration with constants that represent the game status
   enum Status { CONTINUE, WON, LOST }; // all caps in constants

   int myPoint; // point if no win or loss on first roll
   Status gameStatus; // can contain CONTINUE, WON or LOST

   // randomize random number gener
   srand( time( 0 ) );

   int sumOfDice = rollDice();
```

#include and **using** for function **time**

Enumeration to keep track of the game status

Declaring a variable of the user-defined enumeration type

Seeding the random number generator with the current time

Outline

fig06_11.cpp

```

   // determine game status and point (if needed) based on first roll
   switch ( sumOfDice )
   {
      case 7: // win with 7 on first roll
      case 11: // win with 11 on first roll
         gameStatus = WON;
         break;
      case 2: // lose with 2 on first roll
      case 3: // lose with 3 on first roll
      case 12: // lose with 12 on first roll
         gameStatus = LOST;
         break;
      default: // did not win or lose, so remember point
         gameStatus = CONTINUE; // game is not over
         myPoint = sumOfDice; // remember the point
         cout << "Point is " << myPoint << endl;
         break; // optional at end of switch
   } // end switch

   // while game is not complete
   while ( gameStatus == CONTINUE ) // not WON or LOST
   {
      sumOfDice = rollDice(); // roll dice again

```

Assigning an enumeration constant to **gameStatus**

Comparing a variable of an enumeration type to an enumeration constant

Outline

fig06_11.cpp

(3 of 4)

```
      // determine game status
      if ( sumOfDice == myPoint ) // win by making point
         gameStatus = WON;
      else
         if ( sumOfDice == 7 ) // lose by rolling 7 before point
            gameStatus = LOST;
   } // end while

   // display won or lost message
   if ( gameStatus == WON )
      cout << "Player wins" << endl;
   else
      cout << "Player loses" << endl;

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

// roll dice, calculate sum and display results
int rollDice()
{
   // pick random die values
   int die1 = 1 + rand() % 6; // first die roll
   int die2 = 1 + rand() % 6; // second die roll

   int sum = die1 + die2; // compute sum of die values
```

Function that performs the task of rolling the dice

Outline

```
   // display results of this roll
   cout << "Player rolled " << die1 << " + " << die2
      << " = " << sum << endl;
   return sum; // end function rollDice
} // end function rollDice
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

fig06_11.cpp

(4 of 4)

Good Programming Practice 6.1

Capitalize the first letter of an identifier used as a user-defined type name.

Good Programming Practice 6.2

Use only uppercase letters in the names of enumeration constants. This makes these con-stants stand out in a program and reminds the programmer that enumeration constants are not variables.

Good Programming Practice 6.3

Using enumerations rather than integer constants can make programs clearer and more maintainable. You can set the value of an enumeration constant once in the enumeration declaration.

Common Programming Error 6.9

Assigning the integer equivalent of an enumeration constant to a variable of the enumeration type is a compilation error.

Common Programming Error 6.10

After an enumeration constant has been defined, attempting to assign another value to the enumeration constant is a compilation error.

6.9 Storage Classes

- Each identifier has several attributes
  - Name, type, size and value
  - Also storage class, scope and linkage
- C++ provides five storage-class specifiers:
  - `auto`, `register`, `extern`, `mutable` and `static`
- Identifier's storage class
  - Determines the period during which that identifier exists in memory
- Identifier's scope
  - Determines where the identifier can be referenced in a program

6.9 Storage Classes (Cont.)

- Identifier's linkage
  - Determines whether an identifier is known only in the source file where it is declared or across multiple files that are compiled, then linked together
- An identifier's storage-class specifier helps determine its storage class and linkage

6.9 Storage Classes (Cont.)

- Automatic storage class
  - Declared with keywords `auto` and `register`
  - Automatic variables
    - Created when program execution enters block in which they are defined
    - Exist while the block is active
    - Destroyed when the program exits the block
  - Only local variables and parameters can be of automatic storage class
    - Such variables normally are of automatic storage class

Performance Tip 6.1

Automatic storage is a means of conserving memory, because automatic storage class variables exist in memory only when the block in which they are defined is executing.

Software Engineering Observation 6.8

Automatic storage is an example of the principle of least privilege, which is fundamental to good software engineering. In the context of an application, the principle states that code should be granted only the amount of privilege and access that it needs to accomplish its designated task, but no more. Why should we have variables stored in memory and accessible when they are not needed?

Performance Tip 6.2

The storage-class specifier `register` can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers rather than in memory. If intensely used variables such as counters or totals are maintained in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory is eliminated.

6.9 Storage Classes (Cont.)

- Storage-class specifier `auto`
  - Explicitly declares variables of automatic storage class
  - Local variables are of automatic storage class by default
    - So keyword `auto` rarely is used
- Storage-class specifier `register`
  - Data in the machine-language version of a program is normally loaded into registers for calculations and other processing
    - Compiler tries to store register storage class variables in a register
  - The compiler might ignore `register` declarations
    - May not be sufficient registers for the compiler to use

Common Programming Error 6.11

Using multiple storage-class specifiers for an identifier is a syntax error. Only one storage class specifier can be applied to an identifier. For example, if you include `register`, do not also include `auto`.

Performance Tip 6.3

Often, `register` is unnecessary. Today's optimizing compilers are capable of recognizing frequently used variables and can decide to place them in registers without needing a `register` declaration from the programmer.

6.9 Storage Classes (Cont.)

- Static storage class
  - Declared with keywords `extern` and `static`
  - Static-storage-class variables
    - Exist from the point at which the program begins execution
    - Initialized once when their declarations are encountered
    - Last for the duration of the program
  - Static-storage-class functions
    - The name of the function exists when the program begins execution, just as for all other functions
  - However, even though the variables and the function names exist from the start of program execution, this does not mean that these identifiers can be used throughout the program.

6.9 Storage Classes (Cont.)

- Two types of identifiers with static storage class
  - External identifiers
    - Such as global variables and global function names
  - Local variables declared with the storage class specifier `static`
- Global variables
  - Created by placing variable declarations outside any class or function definition
  - Retain their values throughout the execution of the program
  - Can be referenced by any function that follows their declarations or definitions in the source file

Software Engineering Observation 6.9

Declaring a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. This is another example of the principle of least privilege. In general, except for truly global resources such as `cin` and `cout`, the use of global variables should be avoided except in certain situations with unique performance requirements.

Software Engineering Observation 6.10

Variables used only in a particular function should be declared as local variables in that function rather than as global variables.

6.9 Storage Classes (Cont.)

- **Local variables declared with keyword `static`**
  - **Known only in the function in which they are declared**
  - **Retain their values when the function returns to its caller**
    - **Next time the function is called, the `static` local variables contain the values they had when the function last completed**
  - **If numeric variables of the static storage class are not explicitly initialized by the programmer**
    - **They are initialized to zero**

6.10 Scope Rules

- **Scope**
  - **Portion of the program where an identifier can be used**
  - **Four scopes for an identifier**
    - **Function scope**
    - **File scope**
    - **Block scope**
    - **Function-prototype scope**

6.10 Scope Rules (Cont.)

- **File scope**
  - **For an identifier declared outside any function or class**
    - **Such an identifier is “known” in all functions from the point at which it is declared until the end of the file**
  - **Global variables, function definitions and function prototypes placed outside a function all have file scope**
- **Function scope**
  - **Labels (identifiers followed by a colon such as `start:`) are the only identifiers with function scope**
    - **Can be used anywhere in the function in which they appear**
    - **Cannot be referenced outside the function body**
    - **Labels are implementation details that functions hide from one another**

6.10 Scope Rules (Cont.)

- Block scope
  - Identifiers declared inside a block have block scope
    - Block scope begins at the identifier's declaration
    - Block scope ends at the terminating right brace (`}`) of the block in which the identifier is declared
  - Local variables and function parameters have block scope
    - The function body is their block
  - Any block can contain variable declarations
  - Identifiers in an outer block can be "hidden" when a nested block has a local identifier with the same name
  - Local variables declared `static` still have block scope, even though they exist from the time the program begins execution
    - Storage duration does not affect the scope of an identifier

6.10 Scope Rules (Cont.)

- **Function-prototype scope**
  - Only identifiers used in the parameter list of a function prototype have function-prototype scope
  - Parameter names appearing in a function prototype are ignored by the compiler
    - Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity
    - However, in a single prototype, a particular identifier can be used only once

Common Programming Error 6.12

Accidentally using the same name for an identifier in an inner block that is used for an identifier in an outer block, when in fact the programmer wants the identifier in the outer block to be active for the duration of the inner block, is normally a logic error.

Good Programming Practice 6.4

Avoid variable names that hide names in outer scopes. This can be accomplished by avoiding the use of duplicate identifiers in a program.

Outline

fig06_12.cpp


```
// Fig. 6.12: fig06_12.cpp
// A scoping example.
#include <iostream>
using std::cout;
using std::endl;

void useLocal( void ); // function prototype
void useStaticLocal( void ); // function prototype
void useGlobal( void ); // function prototype

int x = 1; // global variable

int main()
{
   int x = 5; // local variable to main

   cout << "local x in main's outer scope is

   { // start new scope
      int x = 7; // hides x in outer scope

      cout << "local x in main's inner scope is
   } // end new scope

   cout << "local x in main's outer scope is " << x << endl;
```

Declaring a global variable outside any class or function definition

Local variable x that hides global variable x

Local variable x in a block that hides local variable x in outer scope

fig06_12.cpp


```

   useLocal(); // useLocal has local x
   useStaticLocal(); // useStaticLocal has static local x
   useGlobal(); // useGlobal uses global x
   useLocal(); // useLocal reinitializes its local x
   useStaticLocal(); // static local x retains its prior value
   useGlobal(); // global x also retains its value

   cout << "\nlocal x in main is " << x << endl;
   return 0; // indicates successful termination
} // end main

// useLocal reinitializes local variable x during each call
void useLocal( void )
{
   int x = 25; // initialized each time useLocal is called

   cout << "\nlocal x is " << x << " on entering useLocal" << endl;
   x++;
   cout << "local x is " << x << " on exiting useLocal" << endl;
} // end function useLocal
```

Local variable that gets recreated and reinitialized each time **useLocal** is called

Outline

fig06_12.cpp

(3 of 4)

```
// useStaticLocal initializes static local variable x only the
// first time the function is called; value of x is saved
// between calls to this function
void useStaticLocal( void )
{
   static int x = 50; // initialized first time useStaticLocal is called

   cout << "\nlocal static x is " << x << " on entering useStaticLocal"
      << endl;
   x++;
   cout << "local static x is " << x << " on exiting useStaticLocal"
      << endl;
} // end function useStaticLocal

// useGlobal modifies global variable x during each call
void useGlobal( void )
{
   cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
   x *= 10;
   cout << "global x is " << x << " on exiting useGlobal" << endl;
} // end function useGlobal
```

`static` local variable that gets initialized only once

Statement refers to global variable `x` because no local variable named `x` exists

```
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

6.11 Function Call Stack and Activation Records

- **Data structure: collection of related data items**
- **Stack data structure**
  - **Analogous to a pile of dishes**
  - **When a dish is placed on the pile, it is normally placed at the top**
    - **Referred to as pushing the dish onto the stack**
  - **Similarly, when a dish is removed from the pile, it is normally removed from the top**
    - **Referred to as popping the dish off the stack**
  - **A last-in, first-out (LIFO) data structure**
    - **The last item pushed (inserted) on the stack is the first item popped (removed) from the stack**

6.11 Function Call Stack and Activation Records (Cont.)

- Function Call Stack
  - Sometimes called the program execution stack
  - Supports the function call/return mechanism
    - Each time a function calls another function, a stack frame (also known as an activation record) is pushed onto the stack
      - Maintains the return address that the called function needs to return to the calling function
      - Contains automatic variables—parameters and any local variables the function declares

6.11 Function Call Stack and Activation Records (Cont.)

- **Function Call Stack (Cont.)**
  - **When the called function returns**
    - **Stack frame for the function call is popped**
    - **Control transfers to the return address in the popped stack frame**
  - **If a function makes a call to another function**
    - **Stack frame for the new function call is simply pushed onto the call stack**
    - **Return address required by the newly called function to return to its caller is now located at the top of the stack.**
- **Stack overflow**
  - **Error that occurs when more function calls occur than can have their activation records stored on the function call stack (due to memory limitations)**

Outline

fig06_13.cpp

(1 of 1)

```
// Fig. 6.13: fig06_13.cpp
// square function used to demonstrate the function
// call stack and activation records.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int square( int ); // prototype for function square

int main()
{
   int a = 10; // value to square (local automatic variable in main)

   cout << a << " squared: " << square( a ) << endl; // display a squared
   return 0; // indicate successful termination
} // end main

// returns the square of an integer
int square( int x ) // x is a local variable
{
   return x * x; // calculate square and return result
} // end function square
```

Calling function **square**

```
10 squared: 100
```

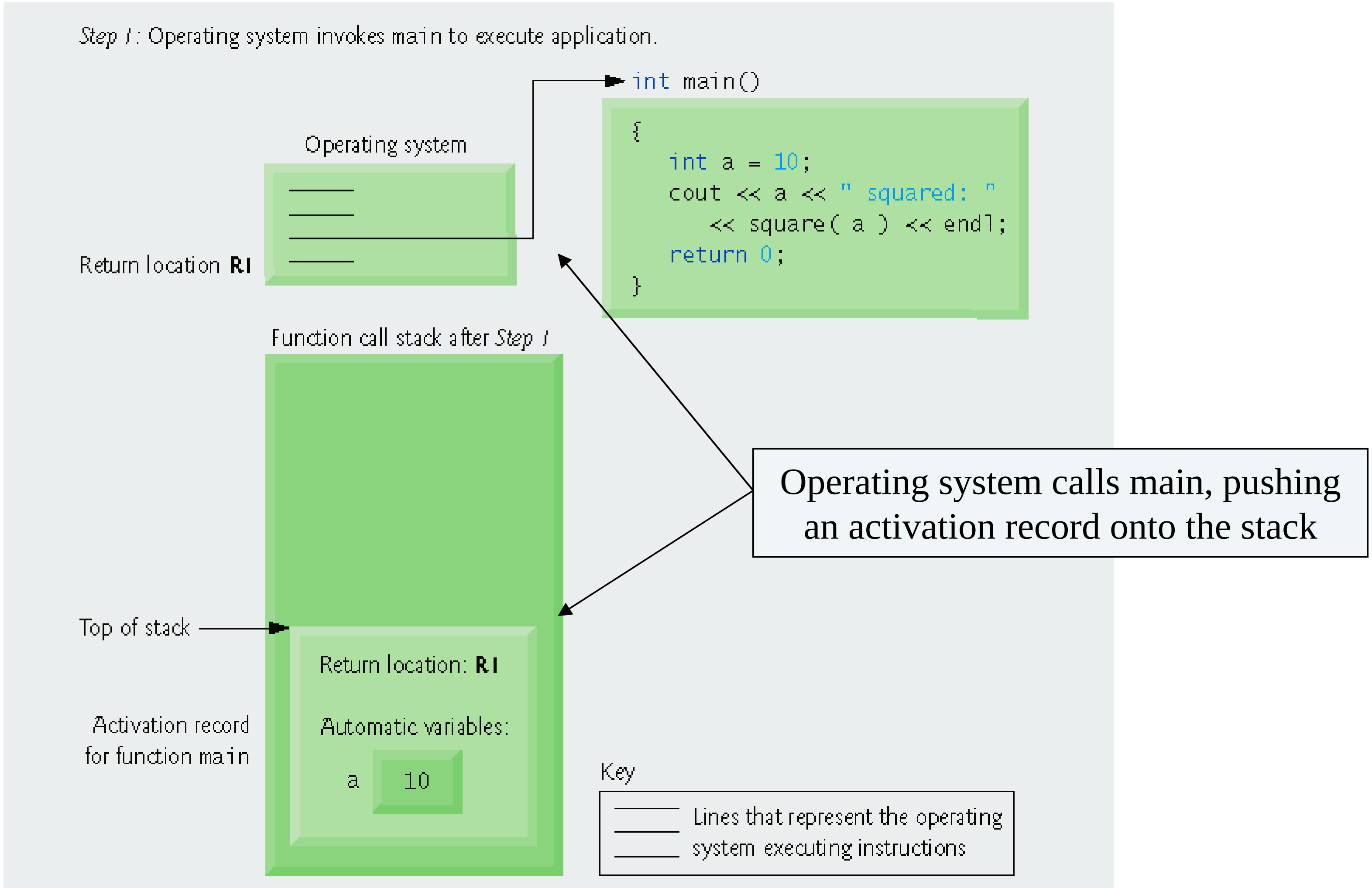

Fig. 6.14 | Function call stack after the operating system invokes `main` to execute the application.

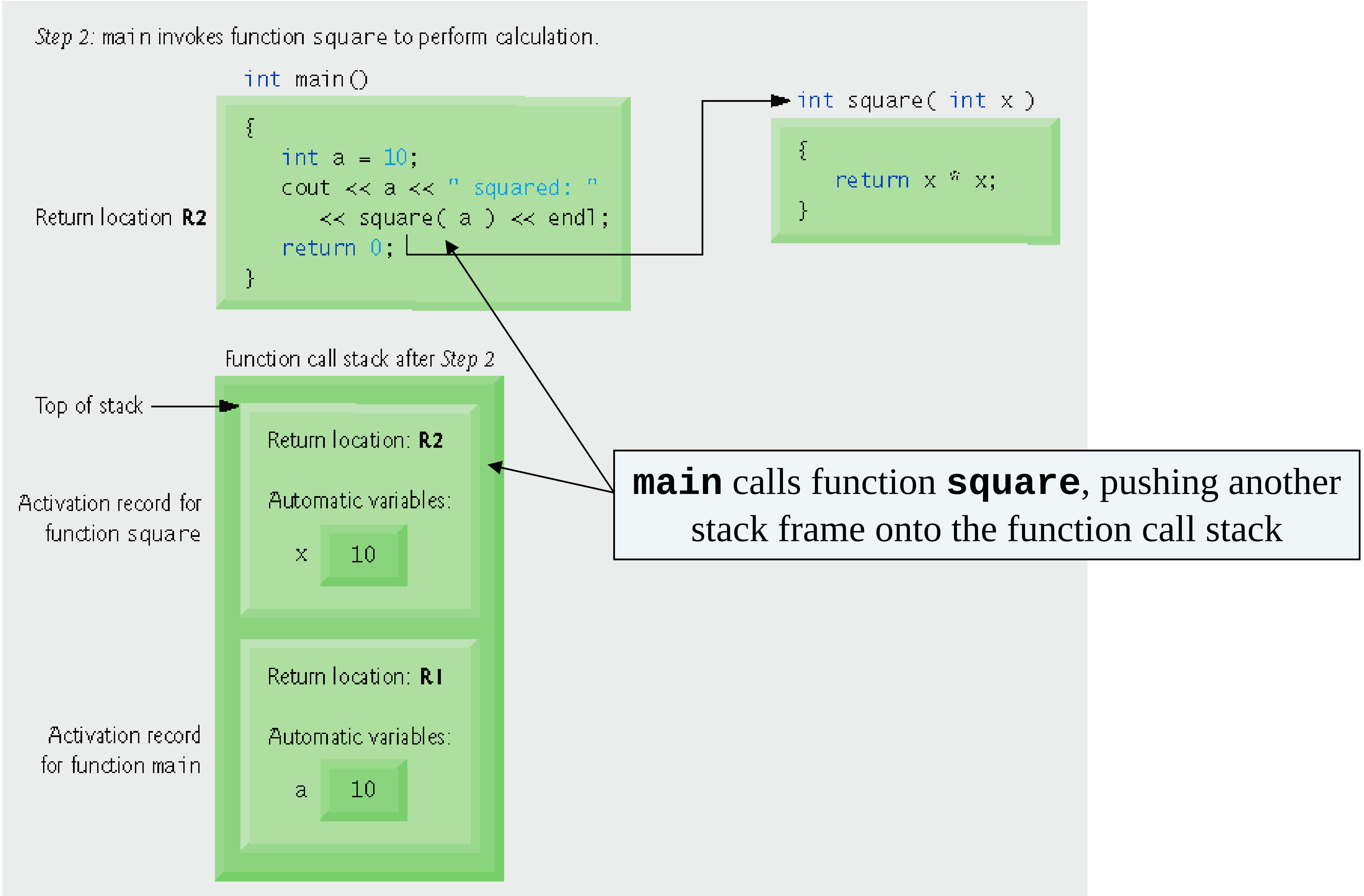

Fig. 6.15 | Function call stack after `main` invokes function `square` to perform the calculation.

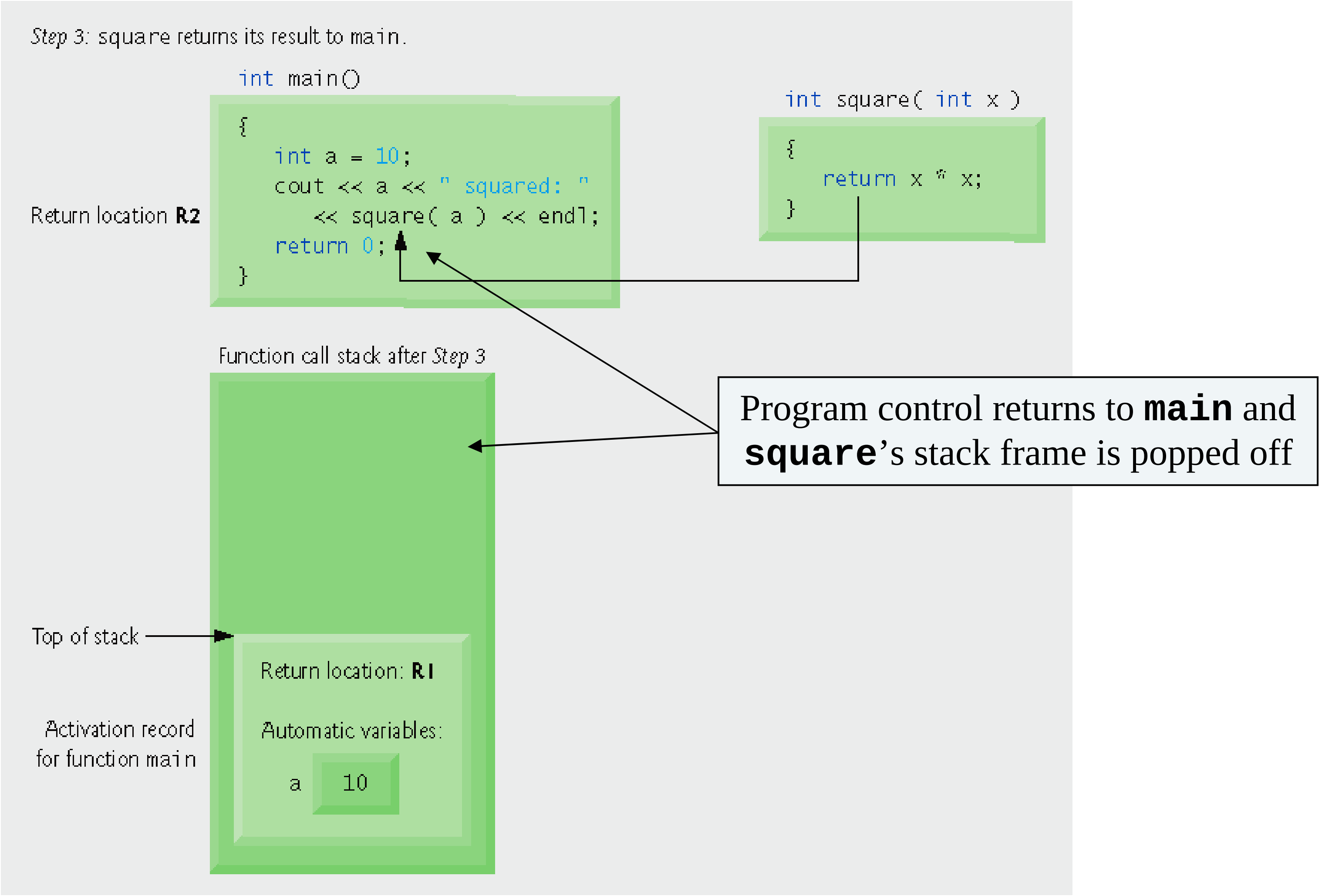

Fig. 6.16 | Function call stack after function `square` returns to `main`.

6.12 Functions with Empty Parameter Lists

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

```
void print();
```

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

Portability Tip 6.2

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++.

```cpp
// Fig. 6.17: fig06_17.cpp
// Functions that take no arguments.
#include <iostream>
using std::cout;
using std::endl;

void function1(); // function that takes no arguments
void function2( void ); // function that takes no arguments

int main()
{
   function1(); // call function1 with no arguments
   function2(); // call function2 with no arguments
   return 0; // indicates successful termination
} // end main

// function1 uses an empty parameter list to specify that
// the function receives no arguments
void function1()
{
   cout << "function1 takes no arguments" << endl;
} // end function1
```

Specify an empty parameter list by putting nothing in the parentheses

Specify an empty parameter list by putting **`void`** in the parentheses

`fig06_17.cpp`

(1 of 2)

```
// function2 uses a void parameter list to specify that
// the function receives no arguments
void function2( void )
{
   cout << "function2 also takes no arguments" << endl;
} // end function2
```

```
function1 takes no arguments
function2 also takes no arguments
```

fig06_17.cpp

Common Programming Error 6.13

C++ programs do not compile unless function prototypes are provided for every function or each function is defined before it is called.

6.13 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 6.11

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.

Good Programming Practice 6.5

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

Performance Tip 6.4

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

Software Engineering Observation 6.12

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.

fig06_18.cpp

(1 of 1)

```cpp
// Fig. 6.18: fig06_18.cpp
// Using an inline function to calculate the volume of a cube.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// Definition of inline function cube. Definition of function appears
// before function is called, so a function prototype is not required.
// First line of function definition acts as the prototype.
inline double cube( const double side )
{
   ret[illegible]de; // calculate cube
} // end function cube

int main()
{
   double sideValue; // stores value entered by user
   cout << "Enter the side length of your cube: ";
   cin >> sideValue; // read value from user

   // calculate cube of sideValue and display result
   cout << "Volume of cube with side "
      << sideValue << " is " << cube( sideValue ) << endl;
   return 0; // indicates successful termination
} // end main
```

`inline` qualifier

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

cube function call that could be inlined

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

6.14 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 6.5

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.

6.14 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 6.6

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 6.13

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

fig06_19.cpp

(1 of 2)

```
// Fig. 6.19: fig06_19.cpp
// Comparing pass-by-value and pass-by-reference with references.
#include <iostream>
using std::cout;
using std::endl;

int squareByValue( int ); // function prototype (value pass)
void squareByReference( int & ); // function prototype (reference pass)

int main()
{
   int x = 2; // value to square using squareByValue
   int z = 4; // value to square using squareByReference

   // demonstrate squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue( x ) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;

   // demonstrate squareByReference
   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;
   return 0; // indicates successful termination
} // end main

```

Function illustrating pass-by-value

Function illustrating pass-by-reference

Variable is simply mentioned by name in both function calls

Outline

```
// squareByValue multiplies number by itself, stores the
// result in number and returns the new value of number
int squareByValue( int number )
{
   return number *= number; // caller's argument not modified
} // end function squareByValue

// squareByReference multiplies numberRef by itself and stores the result
// in the variable to which numberRef refers in function main
void squareByReference( int &numberRef )
{
   numberRef *= numberRef; // caller's argument modified
} // end function squareByReference
```

Receives copy of argument in `main`

Receives reference to argument in `main`

Modifies variable in `main`

fig06_19.cpp


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

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

Common Programming Error 6.14

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 6.7

For passing large objects, 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.

Software Engineering Observation 6.14

Many programmers do not bother to declare parameters passed by value as `const`, even though 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 6.15

For the combined reasons of clarity and performance, many C++ programmers prefer that modifiable arguments be passed to functions by using pointers (which we study in Chapter 8), small nonmodifiable arguments be passed by value and large nonmodifiable arguments be passed to functions by using references to constants.

6.14 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

fig06_20.cpp

(1 of 1)

```
// Fig. 6.20: fig06_20.cpp
// References must be initialized.
#include <iostream>
using std::cout;
using std::endl;

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;
   return 0; // indicates successful termination
} // 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
```

```
1  // Fig. 6.21: fig06_21.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
```

Uninitialized reference

Borland C++ command-line compiler error message:

```
Error E2304 C:\cpphtp5_examples\ch06\Fig06_21\fig06_21.cpp 10:
   Reference variable 'y' must be initialized in function main()
```

Microsoft Visual C++ compiler error message:

```
C:\cpphtp5_examples\ch06\Fig06_21\fig06_21.cpp(10) : error C2530: 'y' :
   references must be initialized
```

GNU C++ compiler error message:

```
fig06_21.cpp:10: error: 'y' declared as a reference but not initialized
```

6.14 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 6.15

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 6.16

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 6.17

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

6.15 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**

Common Programming Error 6.18

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

```cpp
// Fig. 6.22: fig06_22.cpp
// Using default arguments.
#include <iostream>
using std::cout;
using std::endl;

// function prototype that specifies default arguments
int boxVolume( int length = 1, int width = 1, int height = 1 );

int main()
{
   // no arguments--use default values for all dimensions
   cout << "The default box volume is: " << boxVolume();

   // specify length; default width and height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 1 and height 1 is: " << boxVolume( 10 );

   // specify length and width; default height
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 1 is: " << boxVolume( 10, 5 );

   // specify all arguments
   cout << "\n\nThe volume of a box with length 10,\n"
      << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
      << endl;
   return 0; // indicates successful termination
} // 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

fig06_22.cpp

```
// function boxVolume calculates the volume of a box
int boxVolume( int length, int width, int height )
{
   return length * width * height;
} // end function boxVolume
```

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

Good Programming Practice 6.6

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

Software Engineering Observation 6.16

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

Common Programming Error 6.19

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.

6.16 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

Common Programming Error 6.20

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 6.7

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 are intending to access a global variable rather than a nonglobal variable.

```cpp
// Fig. 6.23: fig06_23.cpp
// Using the unary scope resolution operator.
#include <iostream>
using std::cout;
using std::endl;

int number = 7; // global variable named number

int main()
{
   double number = 10.5; // local variable named number

   // display values of local and global variables
   cout << "Local double value of number = " << number
      << "\nGlobal int value of number = " << ::number << endl;
   return 0; // indicates successful termination
} // 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**

Software Engineering Observation 6.17

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 6.4

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 6.5

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.

6.17 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 6.8

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

Outline

fig06_24.cpp

(1 of 2)

```
// Fig. 6.24: fig06_24.cpp
// Overloaded functions.
#include <iostream>
using std::cout;
using std::endl;

// function square for int values
int square( int x )
{
   cout << "square of integer " << x << " is ";
   return x * x;
} // end function square with int argument

// function square for double values
double square( double y )
{
   cout << "square of double " << y << " is ";
   return y * y;
} // end function square with double argument
```

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

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

```cpp

int main()
{
   cout << square( 7 ); // calls int version
   cout << endl;
   cout << square( 7.5 ); // calls double version
   cout << endl;
   return 0; // indicates successful termination
} // end main
```

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

Output confirms that the proper function was called in each case

fig06_24.cpp

6.17 Function Overloading (Cont.)

- **How the compiler differentiates overloaded functions**
  - **Overloaded functions are distinguished by their signatures**
  - **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**

```
// Fig. 6.25: fig06_25.cpp
// Name mangling.

// function square for int values
int square( int x )
{
   return x * x;
} // end function square

// function square for double values
double square( double y )
{
   return y * y;
} // end function square

// function that receives arguments of types
// int, float, char and int &
void nothing1( int a, float b, char c, int &d )
{
   // empty function body
} // end function nothing1
```

fig06_25.cpp

(1 of 2)

Overloaded **square** functions

fig06_25.cpp


```cpp

// function that receives arguments of types
// char, int, float & and double &
int nothing2( char a, int b, float &c, double &d )
{
   return 0;
} // end function nothing2

int main()
{
   return 0; // indicates successful termination
} // 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 6.21

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

Common Programming Error 6.22

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.

6.18 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

6.18 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

```
// Fig. 6.26: maximum.h
// Definition of function template maximum.

template < class T >  // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
   T maximumValue = value1; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if ( value2 > maximumValue )
      maximumValue = value2;

   // determine whether value3 is greater than maximumValue
   if ( value3 > maximumValue )
      maximumValue = value3;

   return maximumValue;
} // end function template maximum
```

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

(1 of 1)

Common Programming Error 6.23

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

```
// Fig. 6.27: fig06_27.cpp
// Function template maximum test program.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "maximum.h" // include definition of function template maximum

int main()
{
   // demonstrate maximum with int values
   int int1, int2, int3;

   cout << "Input three integer values: ";
   cin >> int1 >> int2 >> int3;

   // invoke int version of maximum
   cout << "The maximum integer value is: "
      << maximum( int1, int2, int3 );

   // demonstrate maximum with double values
   double double1, double2, double3;

   cout << "\n\nInput three double values: ";
   cin >> double1 >> double2 >> double3;

   // invoke double version of maximum
   cout << "The maximum double value is: "
      << maximum( double1, double2, double3 );
```

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

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

`fig06_27.cpp`

(1 of 2)

Outline

fig06_27.cpp

(2 of 2)

```
   // demonstrate maximum with char values
   char char1, char2, char3;

   cout << "\n\nInput three characters: ";
   cin >> char1 >> char2 >> char3;

   // invoke char version of maximum
   cout << "The maximum character value is: "
      << maximum( char1, char2, char3 ) << endl;
   return 0; // indicates successful termination
} // end main
```

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

```
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
```

6.19 Recursion

- Recursive function
  - A function that calls itself, either directly, or indirectly (through another function)
- Recursion
  - Base case(s)
    - The simplest case(s), which the function knows how to handle
  - For all other cases, the function typically divides the problem into two conceptual pieces
    - A piece that the function knows how to do
    - A piece that it does not know how to do
      - Slightly simpler or smaller version of the original problem

6.19 Recursion (Cont.)

- Recursion (Cont.)
  - Recursive call (also called the recursion step)
    - The function launches (calls) a fresh copy of itself to work on the smaller problem
    - Can result in many more recursive calls, as the function keeps dividing each new problem into two conceptual pieces
    - This sequence of smaller and smaller problems must eventually converge on the base case
      - Otherwise the recursion will continue forever

6.19 Recursion (Cont.)

- **Factorial**
  - **The factorial of a nonnegative integer *n*, written *n*! (and pronounced "*n* factorial"), is the product**
    - $n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$
  - **Recursive definition of the factorial function**
    - $n! = n \cdot (n - 1)!$
    - **Example**
      - $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$
        $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$
        $5! = 5 \cdot (4!)$

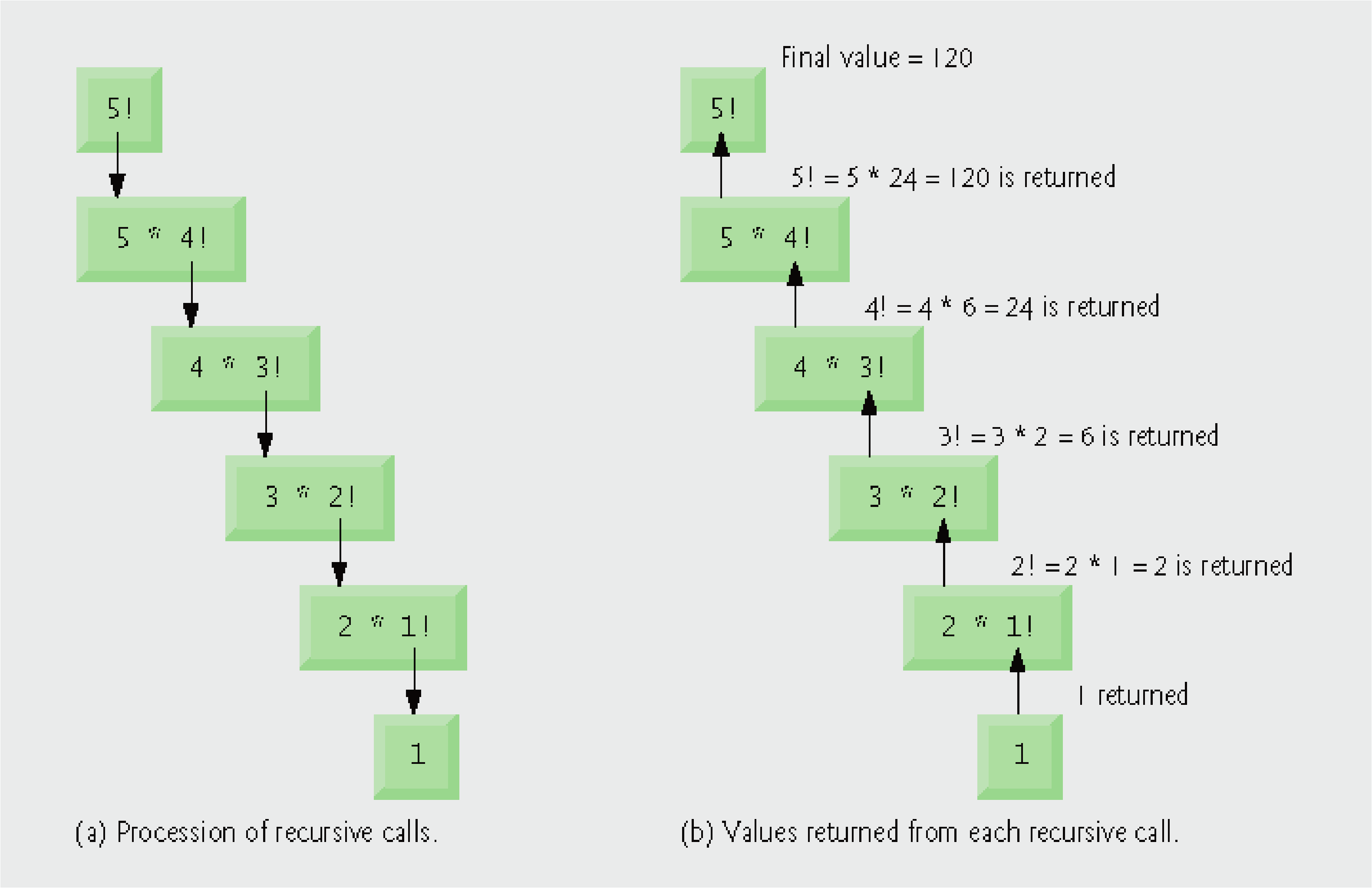

Fig. 6.28 | Recursive evaluation of 5!.

```cpp
// Fig. 6.29: fig06_29.cpp
// Testing the recursive factorial function.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

unsigned long factorial( unsigned long ); // function prototype

int main()
{
   // calculate the factorials of 0 through 10
   for ( int counter = 0; counter <= 10; counter++ )
      cout << setw( 2 ) << counter << "! = " << factorial( counter )
         << endl;

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

First call to **factorial** function

Outline

```
// recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
   if ( number <= 1 ) // test for base case
      return 1; // base cases: 0! = 1 and 1! = 1
   else // recursion step
      return number * factorial( number - 1 );
} // end function factorial
```

Base cases simply **`return 1`**

Recursive call to **`factorial`** function with a slightly smaller problem

`fig06_29.cpp`

(2 of 2)

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Common Programming Error 6.24

Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, causes “infinite” recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

6.20 Example Using Recursion: Fibonacci Series

- **The Fibonacci series**
  - **0, 1, 1, 2, 3, 5, 8, 13, 21, …**
  - **Begins with 0 and 1**
  - **Each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers**
  - **can be defined recursively as follows:**
    - **fibonacci(0) = 0**
    - **fibonacci(1) = 1**
    - **fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)**

```
// Fig. 6.30: fig06_30.cpp
// Testing the recursive fibonacci function.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

unsigned long fibonacci( unsigned long ); // function prototype

int main()
{
   // calculate the fibonacci values of 0 through 10
   for ( int counter = 0; counter <= 10; counter++ )
      cout << "fibonacci( " << counter << " ) = "
         << fibonacci( counter ) << endl;

   // display higher fibonacci values
   cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
   cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
   cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
   return 0; // indicates successful termination
} // end main

```

fig06_30.cpp

Outline

fig06_30.cpp

(2 of 2)

```
// recursive method fibonacci
unsigned long fibonacci( unsigned long number )
{
   if ( ( number == 0 ) || ( number == 1 ) ) // base cases
      return number;
   else // recursion step
      return fibonacci( number - 1 ) + fibonacci( number - 2 );
} // end function fibonacci
```

Base cases

Recursive calls to **fibonacci** function

```
fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 6 ) = 8
fibonacci( 7 ) = 13
fibonacci( 8 ) = 21
fibonacci( 9 ) = 34
fibonacci( 10 ) = 55
fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
fibonacci( 35 ) = 9227465
```

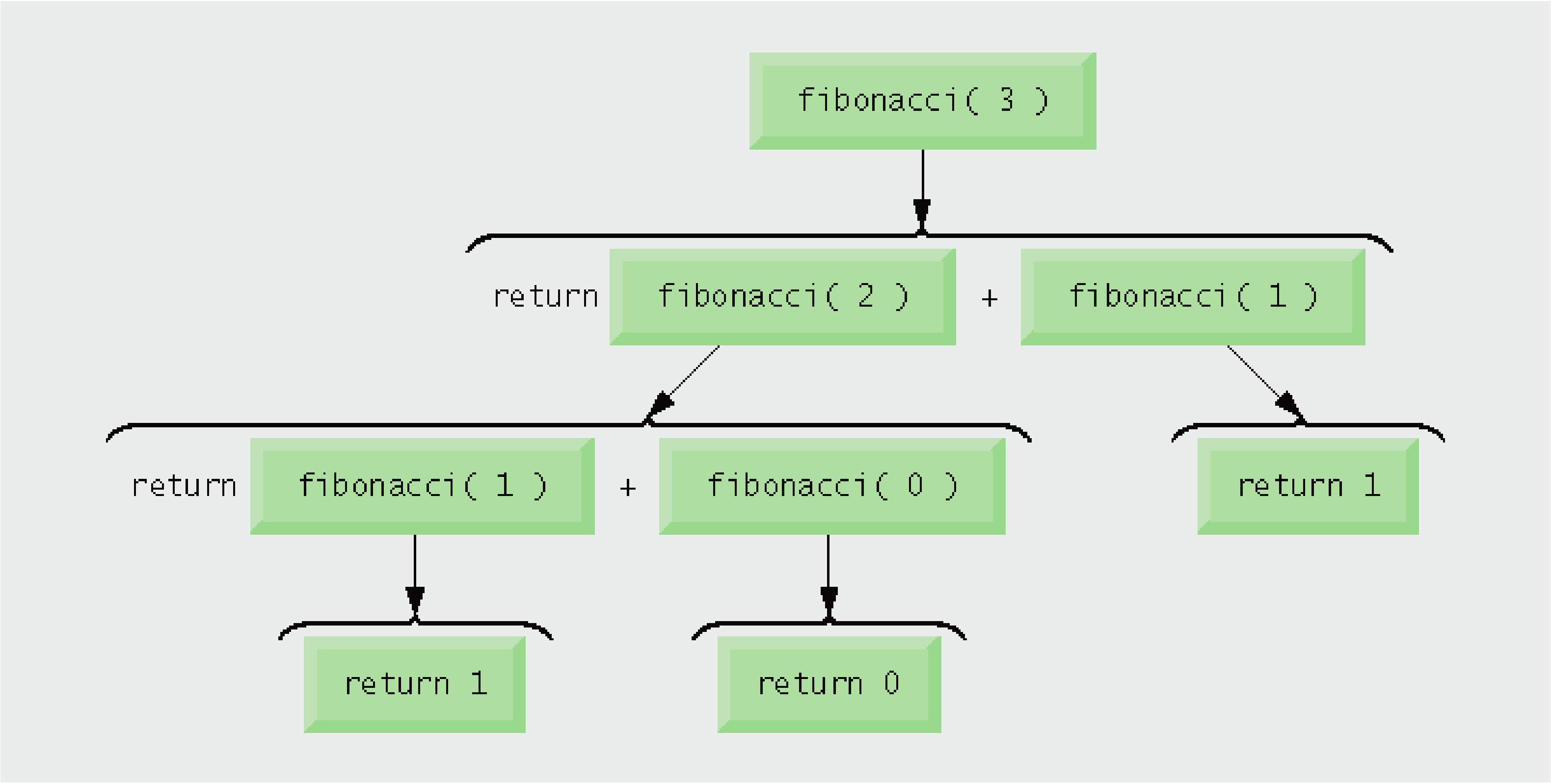

Fig. 6.31 | **Set of recursive calls to function `fibonacci`.**

Common Programming Error 6.25

Writing programs that depend on the order of evaluation of the operands of operators other than &&, `||`, `?:` and the comma (`,`) operator can lead to logic errors.

Portability Tip 6.3

Programs that depend on the order of evaluation of the operands of operators other than &&, ||, ?: and the comma (,) operator can function differently on systems with different compilers.

6.20 Example Using Recursion: Fibonacci Series (Cont.)

- **Caution about recursive programs**
  - **Each level of recursion in function `fibonacci` has a doubling effect on the number of function calls**
    - **i.e., the number of recursive calls that are required to calculate the *n*th Fibonacci number is on the order of 2^n**
    - **20th Fibonacci number would require on the order of 2^{20} or about a million calls**
    - **30th Fibonacci number would require on the order of 2^{30} or about a billion calls.**
  - **Exponential complexity**
    - **Can humble even the world's most powerful computers**

Performance Tip 6.8

Avoid Fibonacci-style recursive programs that result in an exponential "explosion" of calls.

6.21 Recursion vs. Iteration

- **Both are based on a control statement**
  - **Iteration – repetition structure**
  - **Recursion – selection structure**
- **Both involve repetition**
  - **Iteration – explicitly uses repetition structure**
  - **Recursion – repeated function calls**
- **Both involve a termination test**
  - **Iteration – loop-termination test**
  - **Recursion – base case**

6.21 Recursion vs. Iteration (Cont.)

- **Both gradually approach termination**
  - **Iteration modifies counter until loop-termination test fails**
  - **Recursion produces progressively simpler versions of problem**
- **Both can occur infinitely**
  - **Iteration – if loop-continuation condition never fails**
  - **Recursion – if recursion step does not simplify the problem**

```cpp
// Fig. 6.32: fig06_32.cpp
// Testing the iterative factorial function.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

unsigned long factorial( unsigned long ); // function prototype

int main()
{
   // calculate the factorials of 0 through 10
   for ( int counter = 0; counter <= 10; counter++ )
      cout << setw( 2 ) << counter << "! = " << factorial( counter )
         << endl;

   return 0;
} // end main

// iterative function factorial
unsigned long factorial( unsigned long number )
{
   unsigned long result = 1;
```

Outline

fig06_32.cpp

(2 of 2)

```

   // iterative declaration of function factorial
   for ( unsigned long i = number; i >= 1; i-- )
      result *= i;

   return result;
} // end function factorial
```

Iterative approach to finding a factorial

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

6.21 Recursion vs. Iteration (Cont.)

- Negatives of recursion
  - Overhead of repeated function calls
    - Can be expensive in both processor time and memory space
  - Each recursive call causes another copy of the function (actually only the function's variables) to be created
    - Can consume considerable memory
- Iteration
  - Normally occurs within a function
  - Overhead of repeated function calls and extra memory assignment is omitted

Software Engineering Observation 6.18

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.

Performance Tip 6.9

Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

Common Programming Error 6.26

Accidentally having a nonrecursive function call itself, either directly or indirectly (through another function), is a logic error.

Location in Text	Recursion Examples and Exercises
Chapter 6	
Section 6.19, Fig. 6.29	**Factorial function**
Section 6.19, Fig. 6.30	**Fibonacci function**
Exercise 6.7	**Sum of two integers**
Exercise 6.40	**Raising an integer to an integer power**
Exercise 6.42	**Towers of Hanoi**
Exercise 6.44	**Visualizing recursion**
Exercise 6.45	**Greatest common divisor**
Exercise 6.50, Exercise 6.51	**Mystery “What does this program do?” exercise**

Fig. 6.33 | Summary of recursion examples and exercises in the text. (Part 1 of 3)

Location in Text	Recursion Examples and Exercises
Chapter 7	
Exercise 7.18	Mystery "What does this program do?" exercise
Exercise 7.21	Mystery "What does this program do?" exercise
Exercise 7.31	Selection sort
Exercise 7.32	Determine whether a string is a palindrome
Exercise 7.33	Linear search
Exercise 7.34	Binary search
Exercise 7.35	Eight Queens
Exercise 7.36	Print an array
Exercise 7.37	Print a string backward
Exercise 7.38	Minimum value in an array
Chapter 8	
Exercise 8.24	Quicksort
Exercise 8.25	Maze traversal
Exercise 8.26	Generating Mazes Randomly
Exercise 8.27	Mazes of Any Size

Fig. 6.33 | Summary of recursion examples and exercises in the text. (Part 2 of 3)

Location in Text	Recursion Examples and Exercises
Chapter 20	
Section 20.3.3, Figs. 20.5–20.7	Mergesort
Exercise 20.8	Linear search
Exercise 20.9	Binary search
Exercise 20.10	Quicksort
Chapter 21	
Section 21.7, Figs. 21.20–21.22	Binary tree insert
Section 21.7, Figs. 21.20–21.22	Preorder traversal of a binary tree
Section 21.7, Figs. 21.20–21.22	Inorder traversal of a binary tree
Section 21.7, Figs. 21.20–21.22	Postorder traversal of a binary tree
Exercise 21.20	Print a linked list backward
Exercise 21.21	Search a linked list
Exercise 21.22	Binary tree delete
Exercise 21.25	Printing tree

Fig. 6.33 | Summary of recursion examples and exercises in the text. (Part 3 of 3)

6.22 (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System

- **Operation**
  - **A service that objects of a class provide to their clients**
    - **For example, a radio's operations include setting its station and volume**
  - **Implemented as a member function in C++**
  - **Identifying operations**
    - **Examine key verbs and verb phrases in the requirements document**

Class	Verbs and verb phrases
`ATM`	executes financial transactions
`BalanceInquiry`	[none in the requirements document]
`Withdrawal`	[none in the requirements document]
`Deposit`	[none in the requirements document]
`BankDatabase`	authenticates a user, retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Account`	retrieves an account balance, credits a deposit amount to an account, debits a withdrawal amount from an account
`Screen`	displays a message to the user
`Keypad`	receives numeric input from the user
`CashDispenser`	dispenses cash, indicates whether it contains enough cash to satisfy a withdrawal request
`DepositSlot`	receives a deposit envelope

Fig. 6.34 | Verbs and verb phrases for each class in the ATM system.

6.22 (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System (Cont.)

- **Modeling operations in UML**
  - **Each operation is given an operation name, a parameter list and a return type:**
    - ***operationName* (*parameter1* , … , *parameterN*) : *return type***
    - **Each parameter has a parameter name and a parameter type**
      - ***parameterName* : *parameterType***
  - **Some operations may not have return types yet**
    - **Remaining return types will be added as design and implementation proceed**

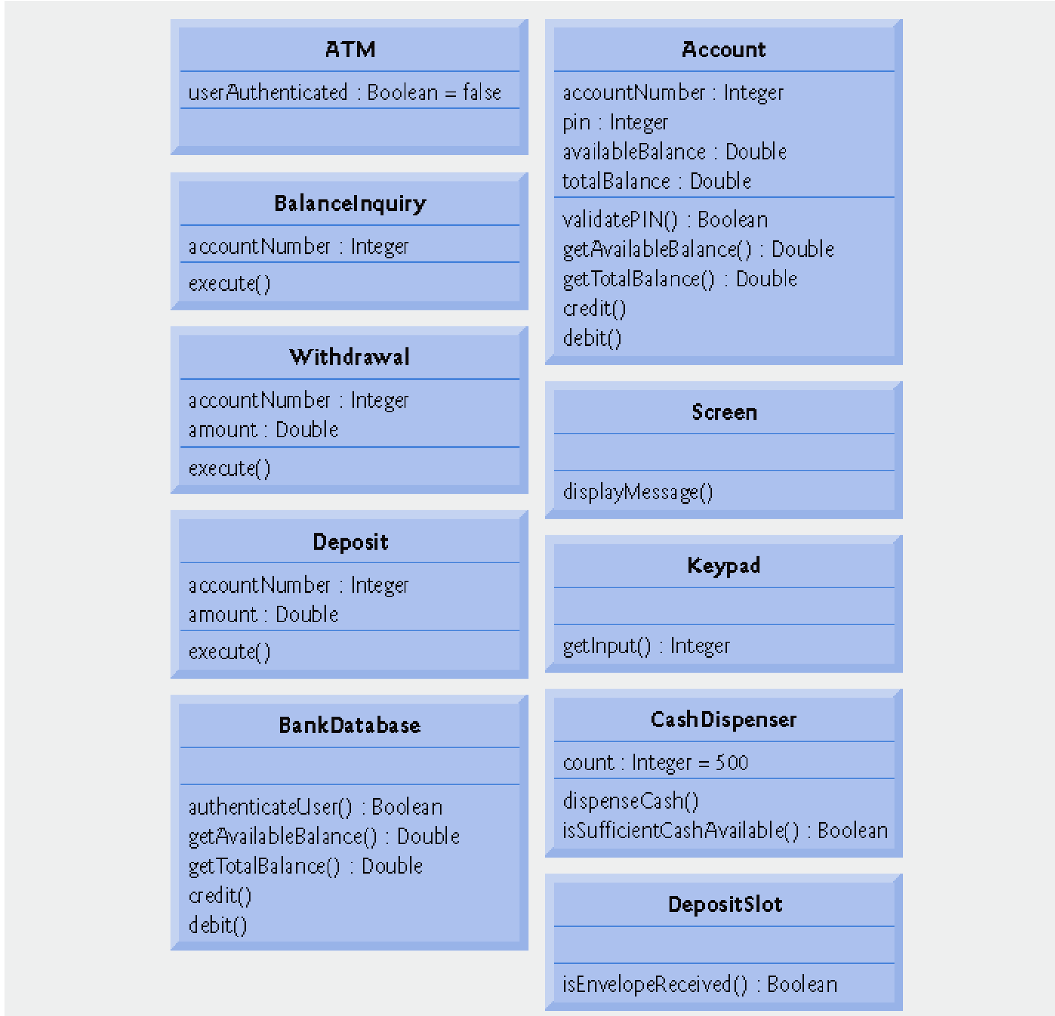

Fig. 6.35 | Classes in the ATM system with attributes and operations.

6.22 (Optional) Software Engineering Case Study: Identifying Class Operations in the ATM System (Cont.)

- **Identifying and modeling operation parameters**
  - **Examine what data the operation requires to perform its assigned task**
  - **Additional parameters may be added later on**

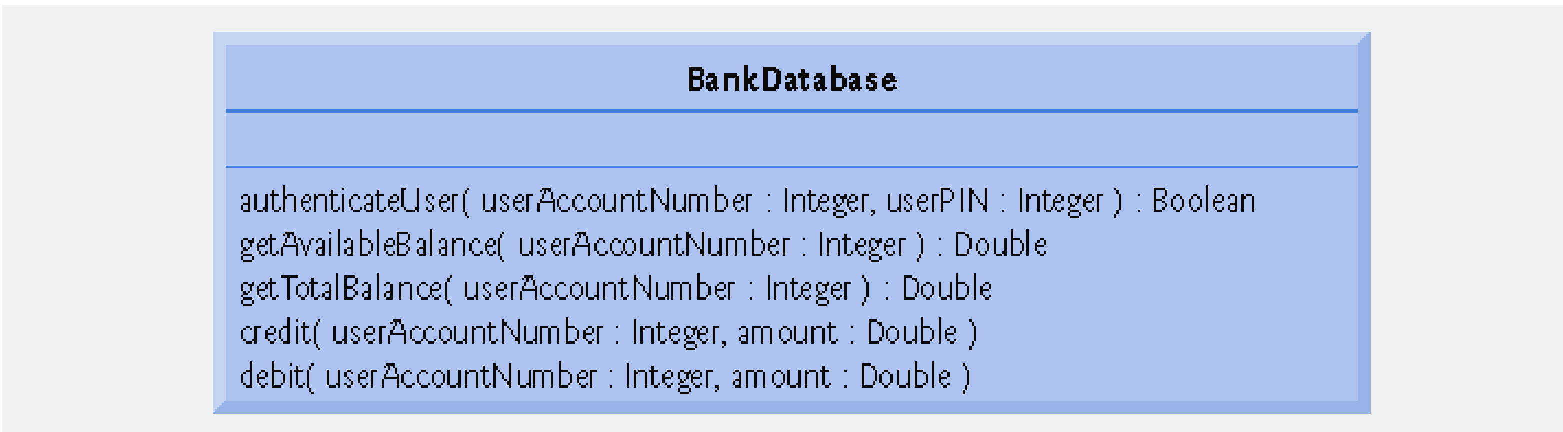

Fig. 6.36 | Class `BankDatabase` with operation parameters.

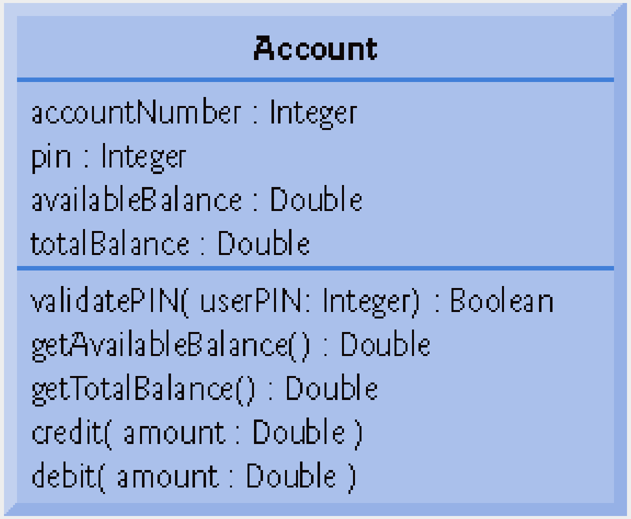

Fig. 6.37 | Class `Account` with operation parameters.

Fig. 6.38 | Class `Screen` with operation parameters.

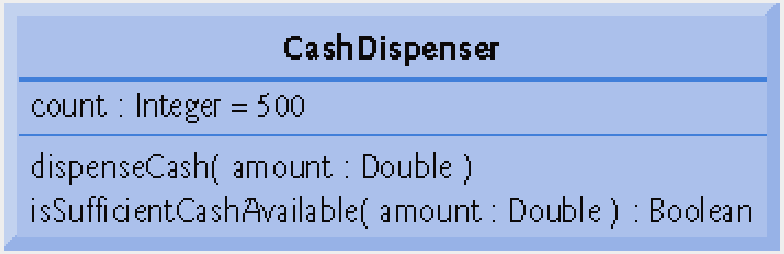

Fig. 6.39 | Class `CashDispenser` with operation parameters.