Topic 3
Lecture 3b
Functions and an Introduction to Recursion

CSCI 140: C++ Language & Objects

Prof. Dominick Atanasio

Book: C++: How to Program 10 ed.

Agenda

- ~~Construct programs modularly from functions.~~
- ~~Use common math library functions.~~
- ~~Use function prototypes to declare functions.~~
- ~~Use random-number generation to implement game playing applications.~~
- ~~Use C++14 digit separators to make numeric literals more readable.~~
- Visibility of identifiers (Scope).
- Function call/return mechanism
- Passing data to functions and returning results.
- Inline function, references, and default arguments.
- Overloaded Functions.
- Function templates.
- Recursive functions.

6.10 Scope Rules

- The portion of the program where an identifier can be used is known as its scope.
- This section discusses
  - block scope
  - global namespace scope

6.10 Scope Rules (cont.)

- Identifiers declared inside a block have block scope, which begins at the identifier's declaration and ends at the terminating right brace (}) of the enclosing block.
  - Local variables have block scope, as do function parameters.
- Any block can contain variable declarations.
- In nested blocks, if an identifier in an outer block has the same name as an identifier in an inner block, the one in the outer block is "hidden" until the inner block terminates.
- The inner block "sees" its own local variable's value and not that of the enclosing block's identically named variable.

Error-Prevention Tip 6.9

Avoid variable names in inner scopes that hide names in outer scopes. Most compilers will warn you about this issue.

6.10 Scope Rules (cont.)

- An identifier declared outside any function or class has global namespace scope.
  - "known" in all functions from the point at which it's declared until the end of the file.
- Function definitions, function prototypes placed outside a function, class definitions and global variables all have global namespace scope.
- Global variables are created by placing variable declarations outside any class or function definition. Such variables retain their values throughout a program's execution.

Software Engineering Observation 6.5

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—except for truly global resources such as* `cin` *and* `cout`*, global variables should be avoided. In general, variables should be declared in the narrowest scope in which they need to be accessed.*

6.10 Scope Rules (cont.)

- Scope Demonstration
- The program of Fig. 6.11 demonstrates scoping issues with global variables, automatic local variables and static local variables.

```
1  // Fig. 6.11: fig06_11.cpp
2  // Scoping example.
3  #include <iostream>
4  using namespace std;
5
6  void useLocal(); // function prototype
7  void useStaticLocal(); // function prototype
8  void useGlobal(); // function prototype
9
10 int x{1}; // global variable
11
12 int main() {
13    cout << "global x in main is " << x << endl;
14
15    int x{5}; // local variable to main
16
17    cout << "local x in main's outer scope is " << x << endl;
18
19    { // block starts a new scope
20       int x{7}; // hides both x in outer scope and global x
21
22       cout << "local x in main's inner scope is " << x << endl;
23    }
24
```

Fig. 6.11 | Scoping example. (Part 1 of 4.)

```
25      cout << "local x in main's outer scope is " << x << endl;
26
27      useLocal(); // useLocal has local x
28      useStaticLocal(); // useStaticLocal has static local x
29      useGlobal(); // useGlobal uses global x
30      useLocal(); // useLocal reinitializes its local x
31      useStaticLocal(); // static local x retains its prior value
32      useGlobal(); // global x also retains its prior value
33
34      cout << "\nlocal x in main is " << x << endl;
35  }
36
37  // useLocal reinitializes local variable x during each call
38  void useLocal() {
39      int x{25}; // initialized each time useLocal is called
40
41      cout << "\nlocal x is " << x << " on entering useLocal" << endl;
42      ++x;
43      cout << "local x is " << x << " on exiting useLocal" << endl;
44  }
45
```

Fig. 6.11 | Scoping example. (Part 2 of 4.)

```
1  // useStaticLocal initializes static local variable x only the
2  // first time the function is called; value of x is saved
3  // between calls to this function
4  void useStaticLocal() {
5     static int x{50}; // initialized first time useStaticLocal is called
6
7     cout << "\nlocal static x is " << x << " on entering useStaticLocal"
8        << endl;
9     ++x;
10    cout << "local static x is " << x << " on exiting useStaticLocal"
11       << endl;
12 }
13
14 // useGlobal modifies global variable x during each call
15 void useGlobal() {
16    cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
17    x *= 10;
18    cout << "global x is " << x << " on exiting useGlobal" << endl;
19 }
```

Fig. 6.11 | Scoping example. (Part 3 of 4.)

6.11 Function Call Stack and Activation Records

- To understand how C++ performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a stack.
- Analogous to a pile of dishes.
  - When a dish is placed on the pile, it's normally placed at the top—referred to as pushing.
  - Similarly, when a dish is removed from the pile, it's normally removed from the top—referred to as popping.
- Last-in, first-out (LIFO) data structures—the last item pushed (inserted) is the first item popped (removed).

6.11 Function Call Stack and Activation Records (cont.)

- Function-Call Stack
- One of the most important mechanisms for computer science students to understand is the function call stack (or program execution stack).
  - supports the function call/return mechanism.
  - Also supports the creation, maintenance and destruction of each called function's automatic variables.

6.11 Function Call Stack and Activation Records (cont.)

- Stack Frames
- Each function eventually must return control to the function that called it.
- Each time a function calls another function, an entry is pushed onto the function call stack.
  - This entry, called a stack frame or an activation record, contains the return address that the called function needs in order to return to the calling function.

6.11 Function Call Stack and Activation Records (cont.)

- Automatic Local Variables and Stack Frames
- When a function call returns, the stack frame for the function call is popped, and control transfers to the return address in the popped stack frame.
- Stack Overflow
  - The amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the function call stack.
  - If more function calls occur than can have their activation records stored on the function call stack, an a fatal error known as stack overflow occurs.

6.11 Function Call Stack and Activation Records (cont.)

- Function Call Stack in Action
- Now let's consider how the call stack supports the operation of a square function called by main (Fig. 6.12).
- First the operating system calls main—this pushes an activation record onto the stack (shown in Fig. 6.13).
- The activation record tells main how to return to the operating system (i.e., transfer to return address R1) and contains the space for main's automatic variable (i.e., a, which is initialized to 10).

```
1  // Fig. 6.12: fig06_12.cpp
2  // square function used to demonstrate the function
3  // call stack and activation records.
4  #include <iostream>
5  using namespace std;
6
7  int square(int); // prototype for function square
8
9  int main() {
10    int a{10}; // value to square (local variable in main)
11
12    cout << a << " squared: " << square(a) << endl; // display a squared
13 }
14
15 // returns the square of an integer
16 int square(int x) { // x is a local variable
17    return x * x; // calculate square and return result
18 }
```

```
10 squared: 100
```

Fig. 6.12 | `square` function used to demonstrate the function-call stack and activation records.

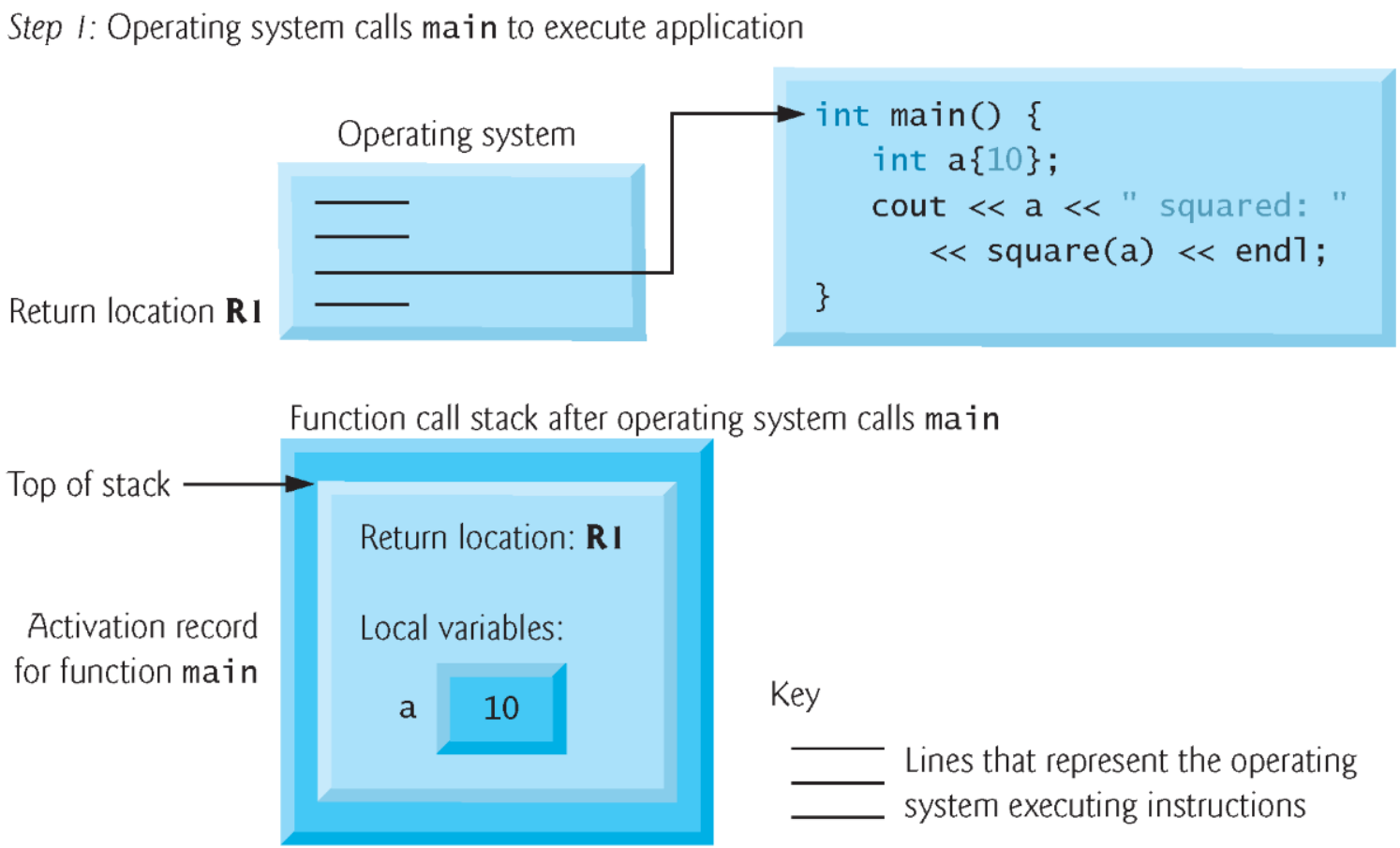

Fig. 6.13 | Function-call stack after the operating system calls `main` to execute the program.

6.12 Function Call Stack and Activation Records (cont.)

- Function main—before returning to the operating system—now calls function square in line 12 of Fig. 6.12.
- This causes a stack frame for square (lines 16–18) to be pushed onto the function call stack (Fig. 6.14).
- This stack frame contains the return address that square needs to return to main (i.e., R2) and the memory for square's automatic variable (i.e., x).

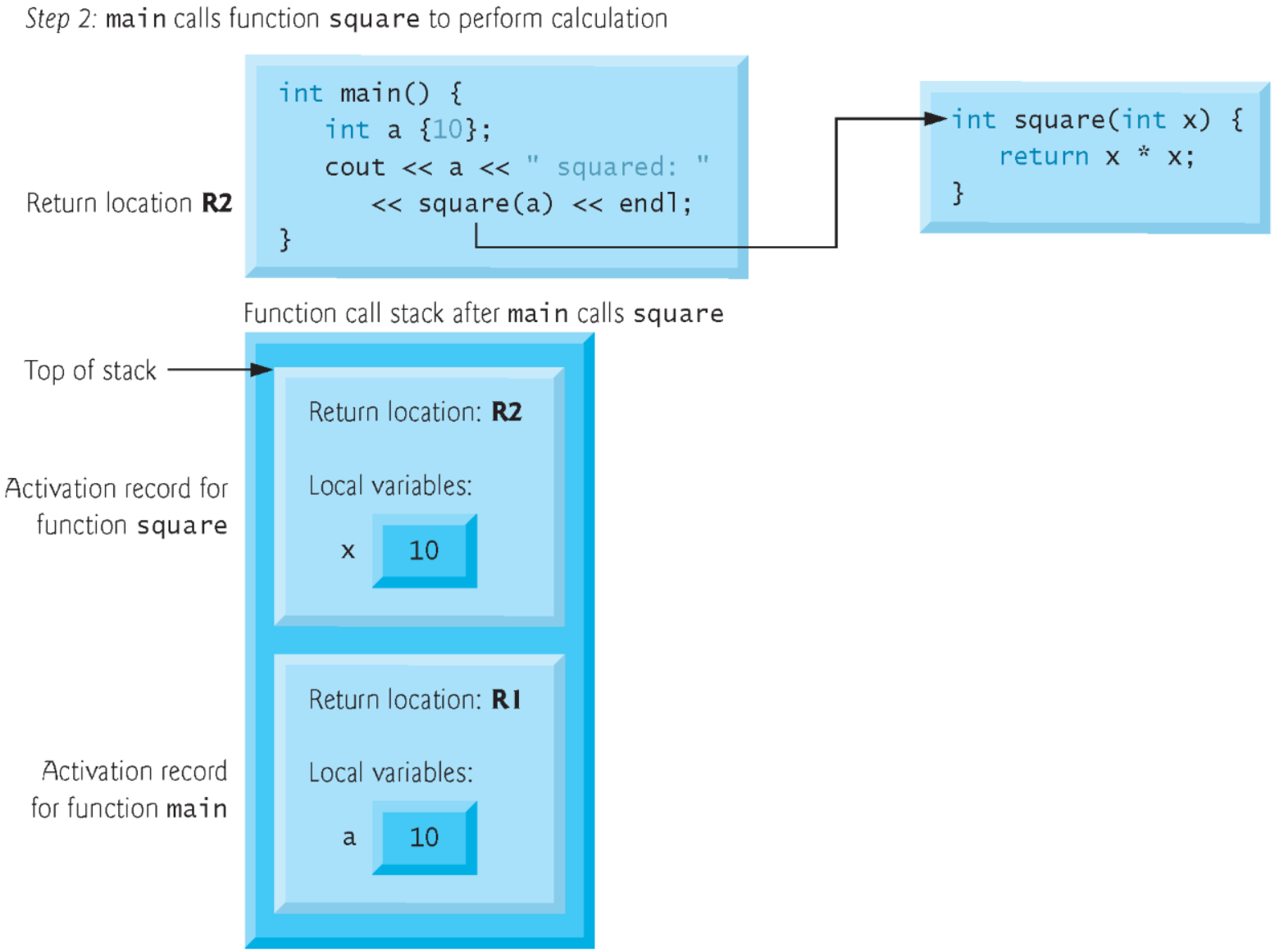

Fig. 6.14 | Function-call stack after `main` calls `square` to perform the calculation.

6.12 Function Call Stack and Activation Records (cont.)

- After square calculates the square of its argument, it needs to return to main—and no longer needs the memory for its automatic variable x. So square's stack frame is popped from the stack—giving square the return location in main (i.e., R2) and losing square's automatic variable.
- Figure 6.15 shows the function call stack after square's activation record has been popped.

Step 3: `square` returns its result to `main`

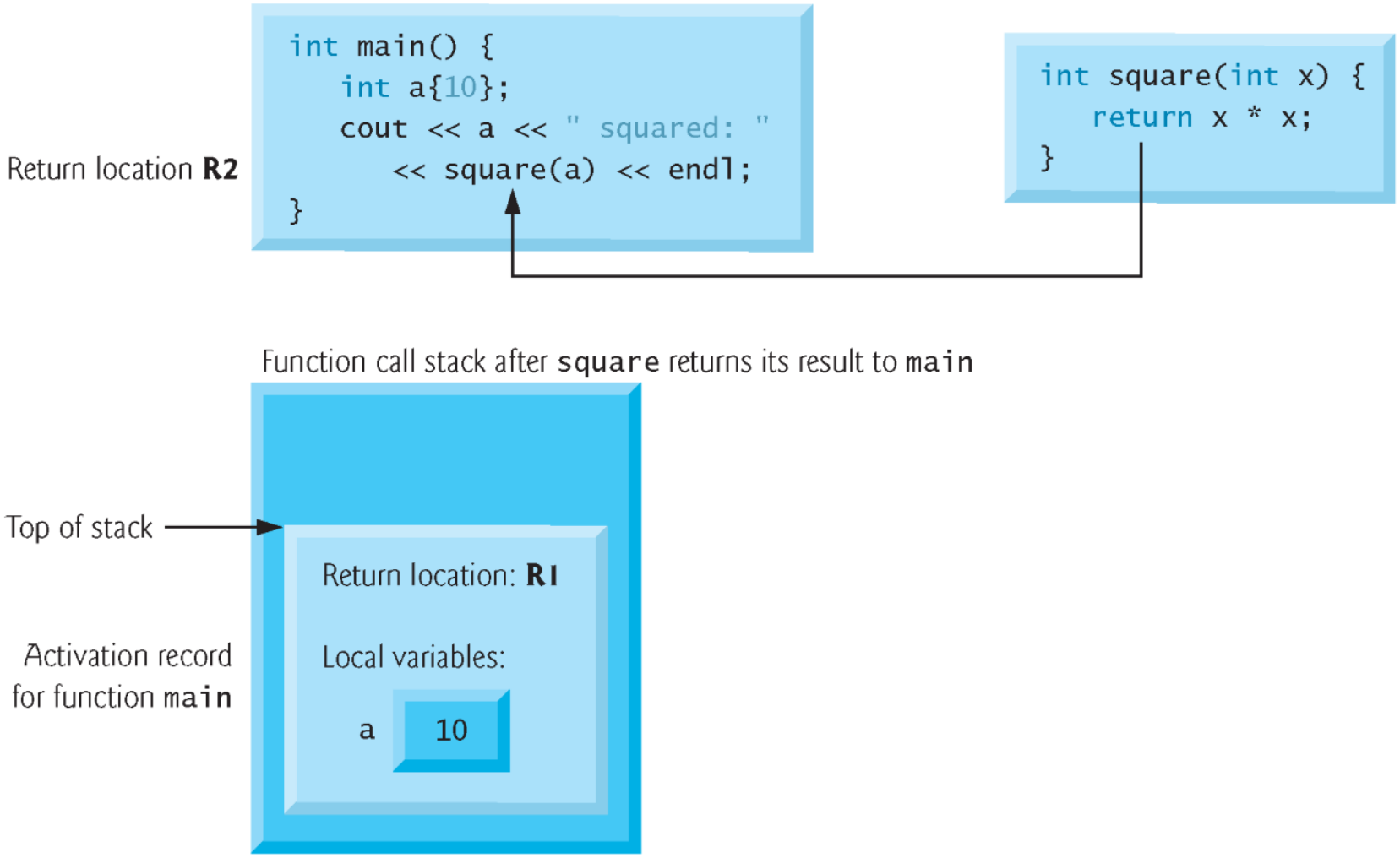

Fig. 6.15 | Function-call stack after function `square` returns to `main`.

6.12 Inline Functions

- C++ provides inline functions to help reduce function call overhead.
- Placing the 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 every place where the function is called (when appropriate) to avoid a function call.
- This often makes the program larger.
- Reusable inline functions are typically placed in headers, so that their definitions can be included in each source file that uses them.

Performance Tip 6.1

Compilers can inline code for which you have not explicitly used the `inline` *keyword. Today's optimizing compilers are so sophisticated that it's best to leave inlining decisions to the compiler.*

6.14 Inline Functions

- Figure 6.16 uses inline function cube (lines 9–11) to calculate the volume of a cube.
- Keyword *const* in function cube's parameter list (line 9) tells the compiler that the function does not modify variable side.
- This ensures that side's value is not changed by the function during the calculation.

Software Engineering Observation 6.8

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

```
1  // Fig. 6.16: fig06_16.cpp
2  // inline function that calculates the volume of a cube.
3  #include <iostream>
4  using namespace std;
5
6  // Definition of inline function cube. Definition of function appears
7  // before function is called, so a function prototype is not required.
8  // First line of function definition acts as the prototype.
9  inline double cube(const double side) {
10    return side * side * side; // calculate cube
11 }
12
13 int main() {
14    double sideValue; // stores value entered by user
15    cout << "Enter the side length of your cube: ";
16    cin >> sideValue; // read value from user
17
18    // calculate cube of sideValue and display result
19    cout << "Volume of cube with side "
20       << sideValue << " is " << cube(sideValue) << endl;
21 }
```

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

Fig. 6.16 | `inline` function that calculates the volume of a cube.

6.13 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.**
- To specify a reference to a constant, place the const qualifier before the type specifier in the parameter declaration.

Performance Tip 6.2

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.

6.15 References and Reference Parameters (cont.)

- With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data.
- A reference parameter is an alias for its corresponding argument in a function call.
- To indicate that a function parameter is passed by reference, simply follow the parameter's type in the function prototype by an ampersand (&); use the same convention when listing the parameter's type in the function header.
- Figure 6.17 compares pass-by-value and pass-by-reference with reference parameters.

Performance Tip 6.3

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 6.9

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

```
1  // Fig. 6.17: fig06_17cpp
2  // Passing arguments by value and by reference.
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    int x{2}; // value to square using squareByValue
11    int z{4}; // value to square using squareByReference
12
13    // demonstrate squareByValue
14    cout << "x = " << x << " before squareByValue\n";
15    cout << "Value returned by squareByValue: "
16       << squareByValue(x) << endl;
17    cout << "x = " << x << " after squareByValue\n" << endl;
18
19    // demonstrate squareByReference
20    cout << "z = " << z << " before squareByReference" << endl;
21    squareByReference(z);
22    cout << "z = " << z << " after squareByReference" << endl;
23 }
24
```

Fig. 6.17 | Passing arguments by value and by reference. (Part 1 of 2.)

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

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

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

Fig. 6.17 | Passing arguments by value and by reference. (Part 2 of 2.)

6.15 References and Reference Parameters (cont.)

- References can also be used as aliases for other variables within a function.
- Reference variables must be initialized in their declarations and cannot be reassigned as aliases to other variables.
- Once a reference is declared as an alias for another variable, all operations supposedly performed on the alias are actually performed on the original variable.

6.15 References and Reference Parameters (cont.)

- To specify that a reference parameter should not be allowed to modify the corresponding argument, place the *const* qualifier before the type name in the parameter's declaration.
- string objects can be large, so they (and objects in general) should be passed to functions by reference.

6.15 References and Reference Parameters (cont.)

- Functions can return references, but this can be dangerous.
- When returning a reference to a variable declared in the called function, the variable should be declared static in that function.

6.14 Default Arguments

- It common for a program to invoke a function repeatedly with the same argument value for a particular parameter.
- 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.
- Figure 6.18 demonstrates using default arguments to calculate a box's volume.

```
1  // Fig. 6.18: fig06_18.cpp
2  // Using default arguments.
3  #include <iostream>
4  using namespace std;
5
6  // function prototype that specifies default arguments
7  unsigned int boxVolume(unsigned int length = 1, unsigned int width = 1,
8     unsigned int height = 1);
9
10 int main() {
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. 6.18 | Using default arguments. (Part 1 of 2.)

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

```
The default box volume is: 1

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

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

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

Fig. 6.18 | Using default arguments. (Part 2 of 2.)

6.14 Default Arguments

- Default arguments must be the rightmost (trailing) arguments in a function's parameter list.
- When calling a function with two or more default arguments, if an omitted argument is not the rightmost argument, then all arguments to the right of that argument also must be omitted.
- Default arguments must be specified with the first occurrence of the function name—typically, in the function prototype.
- Default values can be any expression, including constants, global variables or function calls. Default arguments also can be used with inline functions.

Good Programming Practice 6.4

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

6.15 Unary Scope Resolution Operator

- C++ provides the unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope.
- Using the unary scope resolution operator (::) with a given variable name is optional when the only variable with that name is a global variable.
- Figure 6.19 shows the unary scope resolution operator with local and global variables of the same name.

```
1  // Fig. 6.19: fig06_19.cpp
2  // 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     double number{10.5}; // local variable named number
10
11    // display values of local and global variables
12    cout << "Local double value of number = " << number
13       << "\nGlobal int value of number = " << ::number << endl;
14 }
```

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

Fig. 6.19 | Unary scope resolution operator.

Good Programming Practice 6.5

Always using the unary scope resolution operator (`::`) to refer to global variables (even if there is no collision with a local-variable name) makes it clear that you're intending to access a global variable rather than a local variable.

6.16 Function Overloading

- C++ enables several functions of the same name to be defined, as long as they have different signatures.
- This is called function overloading.
- The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.
- Function overloading is used to create several functions of the same name that perform similar tasks, but on different data types.

6.16 Function Overloading (cont.)

- How the Compiler Differentiates Among Overloaded Functions
- Overloaded functions are distinguished by their **signatures**.
- A signature is a combination of a function's name and its parameter types (in order).
- The compiler encodes each function identifier with the types of its parameters (sometimes referred to as name mangling or name decoration) to enable type-safe linkage.
  - Ensures that the proper overloaded function is called and that the types of the arguments conform to the types of the parameters.
- Figure 6.21 was compiled with GNU C++.
- Rather than showing the execution output of the program, we show the mangled function names produced in assembly language by GNU C++.

```
1  // Fig. 6.21: fig06_21.cpp
2  // Name mangling to enable type-safe linkage.
3
4  // function square for int values
5  int square(int x) {
6     return x * x;
7  }
8
9  // function square for double values
10 double square(double y) {
11    return y * y;
12 }
13
14 // function that receives arguments of types
15 // int, float, char and int&
16 void nothing1(int a, float b, char c, int& d) { }
17
18 // function that receives arguments of types
19 // char, int, float& and double&
20 int nothing2(char a, int b, float& c, double& d) {
21    return 0;
22 }
23
24 int main() { }
```

Fig. 6.21 | Name mangling to enable type-safe linkage. (Part 1 of 2.)

```
__Z6squarei
__Z6squared
__Z8nothing1ifcRi
__Z8nothing2ciRfRd
main
```

Fig. 6.21 | Name mangling to enable type-safe linkage. (Part 2 of 2.)

6.16 Function Overloading (cont.)

- For GNU C++, each mangled name (other than main) begins with two underscores (__) followed by the letter Z, a number and the function name.
  - The number specifies how many characters are in the function's name.
- The compiler distinguishes the two square functions by their parameter lists—one specifies i for int and the other d for double.
- The return types of the functions are not specified in the mangled names.
- Overloaded functions can have different return types, but if they do, they must also have different parameter lists.
- Function-name mangling is compiler specific.

6.17 Function Templates

- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.
- You write a single function template definition.
- Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately.

6.17 Function Templates (cont.)

- Figure 6.22 defines a maximum function that determines the largest of three values.
- All function template definitions begin with the *template* keyword followed by a template parameter list enclosed in angle brackets (< and >).
- Every parameter in the template parameter list is preceded by keyword *typename* or keyword *class*.
- The type parameters are placeholders for fundamental types or user-defined types.
  - Used to specify the types of the function's parameters, to specify the function's return type and to declare variables within the body of the function definition.

```
1  // Fig. 6.22: maximum.h
2  // Function template maximum header.
3  template <typename T>  // or template<class T>
4  T maximum(T value1, T value2, T value3) {
5     T maximumValue{value1}; // assume value1 is maximum
6
7     // determine whether value2 is greater than maximumValue
8     if (value2 > maximumValue) {
9        maximumValue = value2;
10    }
11
12    // determine whether value3 is greater than maximumValue
13    if (value3 > maximumValue) {
14       maximumValue = value3;
15    }
16
17    return maximumValue;
18 }
```

Fig. 6.22 | Function template `maximum` header.

6.19 Function Templates (cont.)

- Figure 6.23 uses the maximum function template to determine the largest of three int values, three double values and three char values, respectively.
- Separate functions are created as a result of the calls—expecting three int values, three double values and three char values, respectively.

```
1  // Fig. 6.23: fig06_23.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  #include "maximum.h" // include definition of function template maximum
5  using namespace std;
6
7  int main() {
8     // demonstrate maximum with int values
9     cout << "Input three integer values: ";
10    int int1, int2, int3;
11    cin >> int1 >> int2 >> int3;
12
13    // invoke int version of maximum
14    cout << "The maximum integer value is: "
15       << maximum(int1, int2, int3);
16
17    // demonstrate maximum with double values
18    cout << "\n\nInput three double values: ";
19    double double1, double2, double3;
20    cin >> double1 >> double2 >> double3;
21
```

Fig. 6.23 | Function template `maximum` test program. (Part 1 of 2.)

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

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

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

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

Fig. 6.23 | Function template `maximum` test program. (Part 2 of 2.)

6.18 Recursion

- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- Recursive problem-solving approaches have a number of elements in common.
  - A recursive function is called to solve a problem.
  - The function actually knows how to solve only the simplest case(s), or so-called base case(s).
  - If the function is called with a base case, the function simply returns a result.
  - If the function is called with a more complex problem, it typically divides the problem into two conceptual pieces—a piece that the function knows how to do and a piece that it does not know how to do.
  - This new problem looks like the original, so the function calls a copy of itself to work on the smaller problem—this is referred to as a recursive call and is also called the recursion step.

Common Programming Error 6.11

Omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case causes an infinite recursion error, typically causing a stack overflow. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution.

6.18 Recursion (cont.)

- Factorial
- The factorial of a nonnegative integer n, written n! (and pronounced “n factorial”), is the product
  - $n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$
- with 1! equal to 1, and 0! defined to be 1.

6.18 Recursion (cont.)

- Iterative Factorial
- The factorial of an integer, number, greater than or equal to 0, can be calculated iteratively (nonrecursively) by using a loop.
- Recursive Factorial
- A recursive definition of the factorial function is arrived at by observing the following algebraic relationship:
  - $n! = n \cdot (n - 1)!$

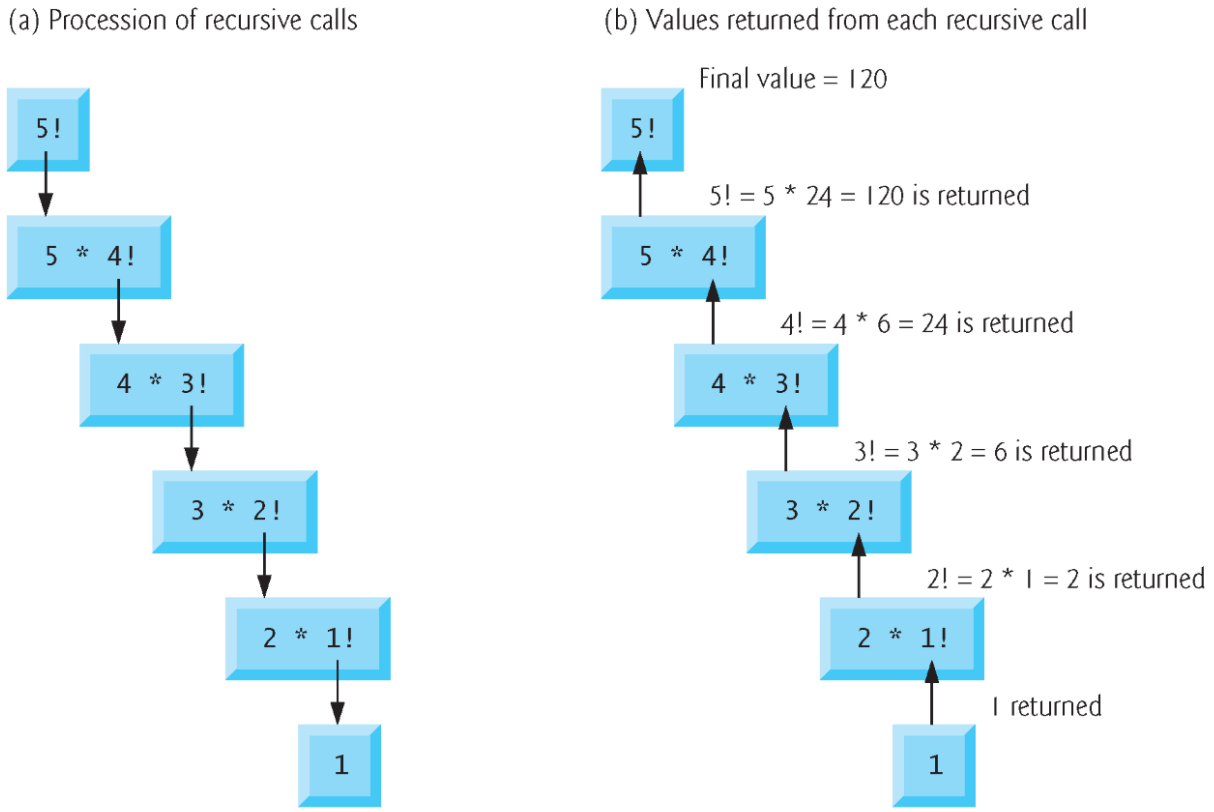

Fig. 6.24 | Recursive evaluation of 5!.

6.19 Example Using Recursion: Fibonacci Series

- The Fibonacci series
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.
- The series occurs in nature and, in particular, describes a form of spiral.
- The ratio of successive Fibonacci numbers converges on a constant value of 1.618….
- This frequently occurs in nature and has been called the golden ratio or the golden mean.
- Humans tend to find the golden mean aesthetically pleasing.

Software Engineering Observation 6.11

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

Performance Tip 6.6

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