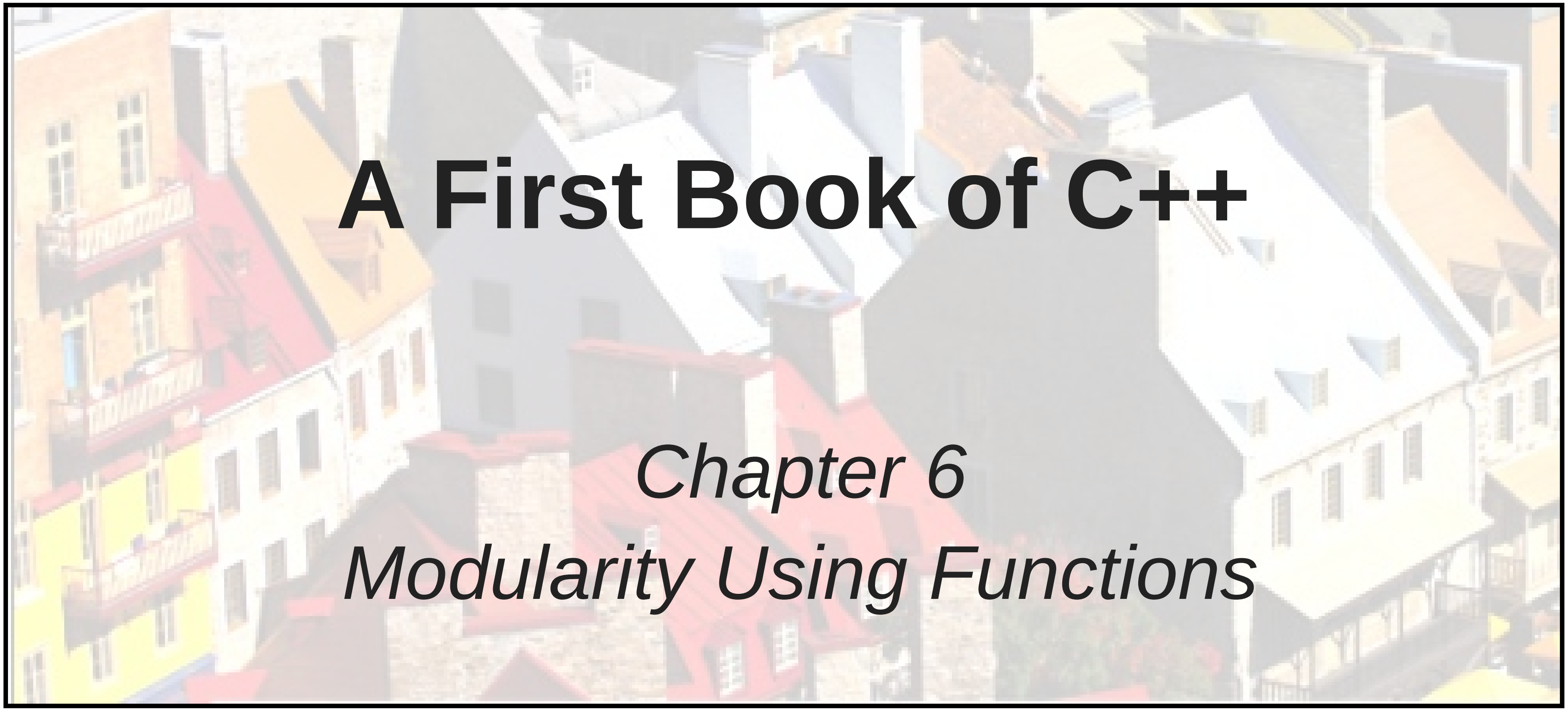


A First Book of C++

Chapter 6

Modularity Using Functions

Objectives

- In this chapter, you will learn about:
  - Function and Parameter Declarations
  - Returning a Single Value
  - Returning Multiple Values
  - Variable Scope
  - Variable Storage Class
  - Common Programming Errors
  - Generating Random Numbers

Function and Parameter Declarations

- All C++ programs must contain a `main()` function
  - May also contain unlimited additional functions
- Major programming concerns when creating functions:
  - How does a function interact with other functions (including `main`)?
  - Correctly passing data to function
  - Correctly returning values from a function

Function and Parameter Declarations (cont'd.)

- Function call process:
  - Give function name
  - Pass data to function as arguments in parentheses following function name
- Only after called function receives data successfully can the data be manipulated within the function

Function and Parameter Declarations (cont'd.)

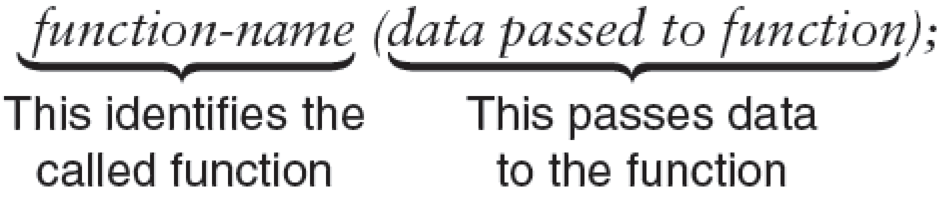
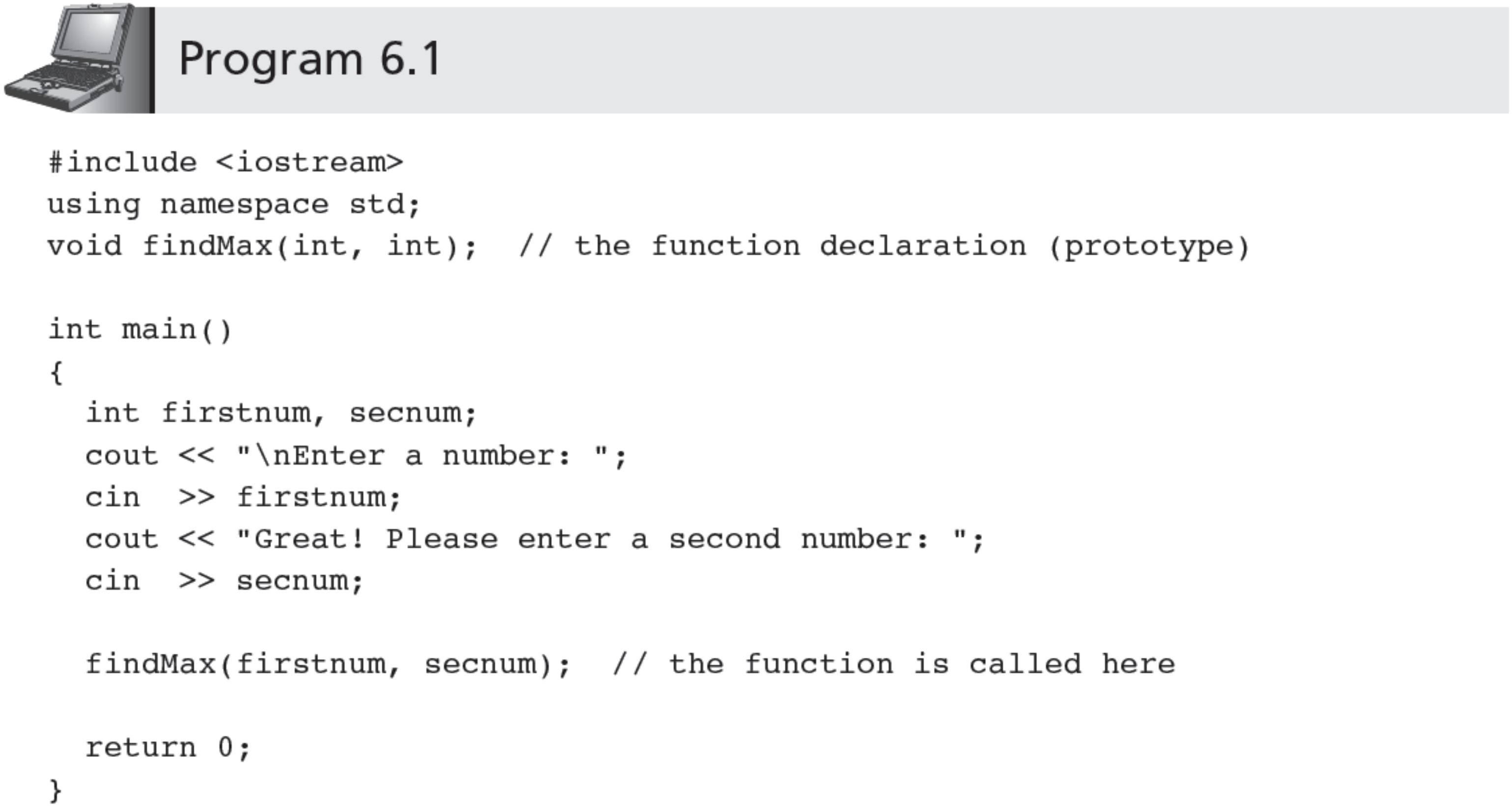

function-name (data passed to function);
This identifies the called function
This passes data to the function

Figure 6.1 Calling and passing data to a function

Function and Parameter Declarations (cont'd.)



Program 6.1

```
#include <iostream>
using namespace std;
void findMax(int, int);  // the function declaration (prototype)

int main()
{
  int firstnum, secnum;
  cout << "\nEnter a number: ";
  cin  >> firstnum;
  cout << "Great! Please enter a second number: ";
  cin  >> secnum;

  findMax(firstnum, secnum);  // the function is called here

  return 0;
}
```

Function and Parameter Declarations (cont'd.)

- Program 6.1 not complete
  - `findMax()` must be written and added
    - Done in slide 15
- Complete program components:
  - `main()`: referred to as **calling function**
  - `findMax()`: referred to as **called function**
- Complete program can be compiled and executed

Function Prototypes

- **Function prototype**: declaration statement for a function
  - Before a function can be called, it must be declared to the calling function
  - Tells the calling function:
    - The type of value to be returned, if any
    - The data type and order of values the calling function should transmit to the called function

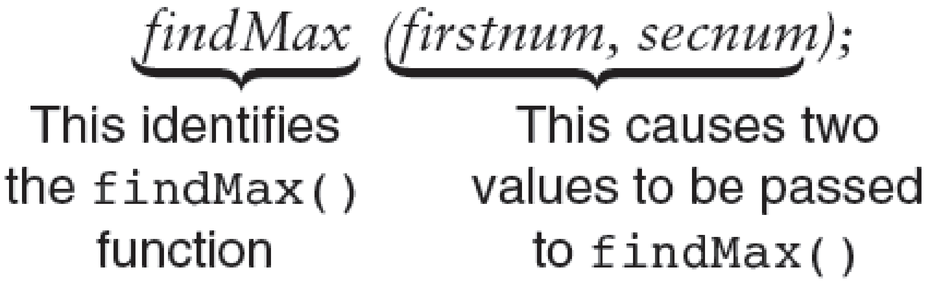
Function Prototypes (cont'd.)

- Example: the function prototype in Program 6.1

```
void findMax(int, int);
```

  - Declares that `findMax()` expects two integer values sent to it
  - `findMax()` returns no value (`void`)
- Prototype statement placement options:
  - Together with variable declaration statements just above calling function name (as in Program 6.1)
  - In a separate header file to be included using a **`#include`** preprocessor statement

Calling a Function



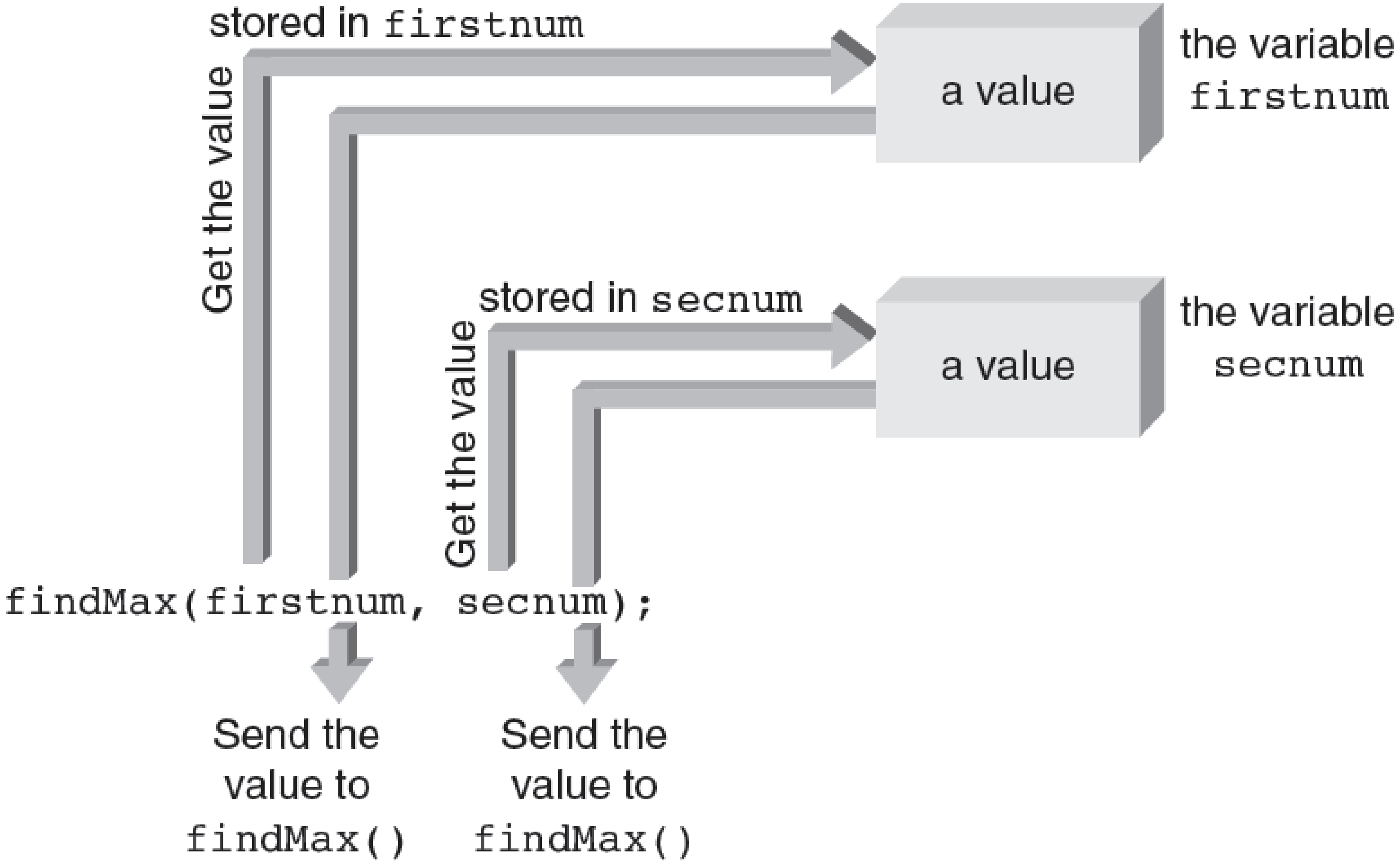
findMax (firstnum, secnum);

This identifies the `findMax()` function

This causes two values to be passed to `findMax()`

Figure 6.2 Calling and passing two values to `findMax()`

Calling a Function (cont'd.)



```
findMax(firstnum, secnum);
```

Figure 6.3 The `findMax()` function receives actual values

Defining a Function

- A function is defined when it is written
  - Can then be used by any other function that suitably declares it
- Format: two parts
  - **Function header** identifies:
    - Data type returned by the function
    - Function name
    - Number, order, and type of arguments expected by the function
  - **Function body**: statements that operate on data
    - Returns one value back to the calling function

Defining a Function (cont'd.)

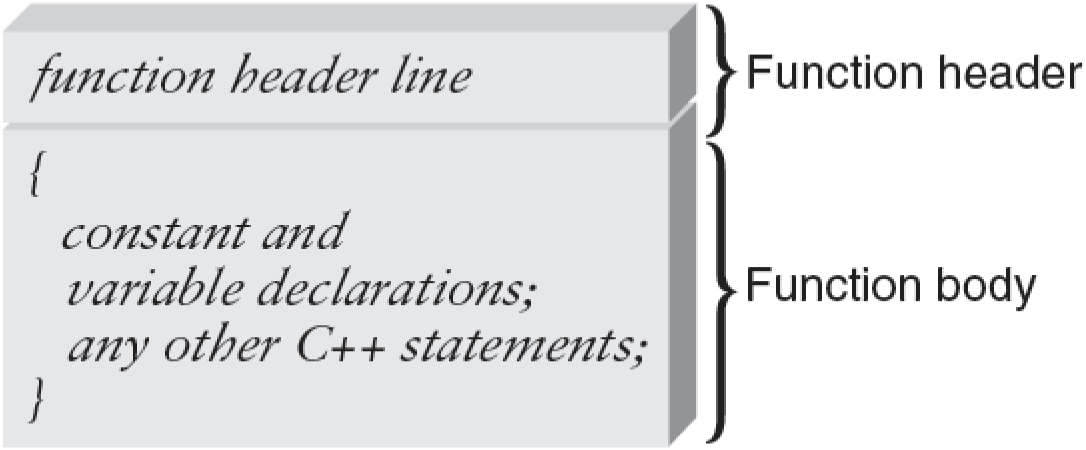


Figure 6.4 The general format of a function

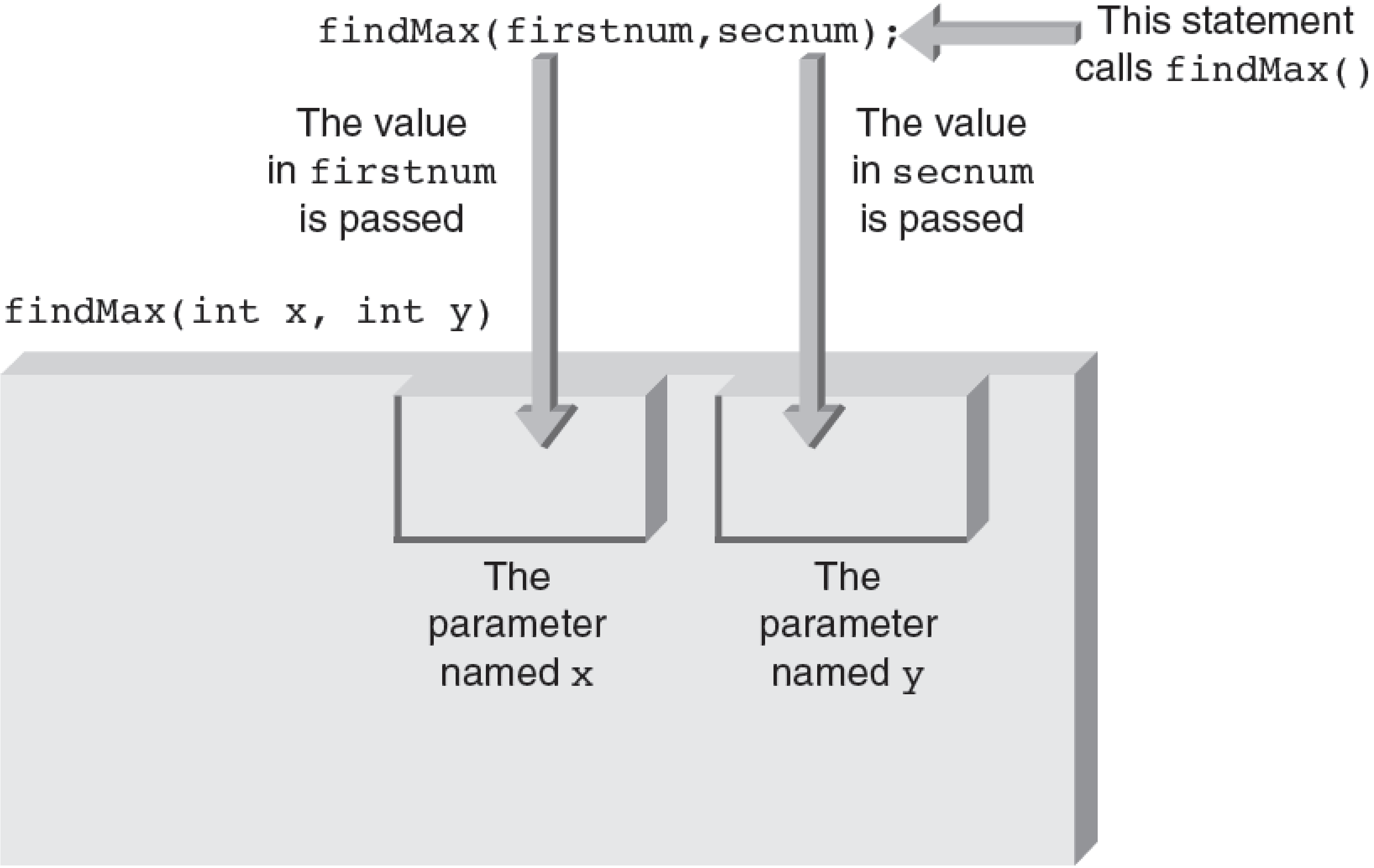

findMax(firstnum,secnum);
This statement calls findMax()
The value in firstnum is passed
The value in secnum is passed
findMax(int x, int y)
The parameter named x
The parameter named y

Figure 6.5 Storing values in parameters

Defining a Function (cont'd.)

`findMax( )` function definition (from Program 6.1)

```
void findMax (int x, int y)
{                       // start of function body
      int maxnum;       // variable declaration
      if (x >= y)       // find the maximum number
        maxnum = x;
      else
        maxnum = y;
      cout << "\nThe maximum of the two numbers is "
           <<maxnum<< endl;
      return;
}     // end of function body and end of function
```

Defining a Function (cont'd.)

- Order of functions in a program:
  - Any order is allowed
  - `main()` usually first
    - `main()` is the driver function
    - Gives reader overall program concept before details of each function encountered
- Each function defined outside any other function
  - Each function separate and independent
  - Nesting functions is *never* permitted

Placement of Statements

- Requirement: items that must be either declared or defined before they are used:
    - Preprocessor directives
    - Named constants
    - Variables
    - Functions
- Otherwise, C++ is flexible in requirements for ordering of statements

Placement of Statements (cont'd.)

- Recommended ordering of statements
  - Good programming practice

```
preprocessor directives
function prototypes
int main()
{
   // symbolic constants
   // variable declarations
   // other executable statements
   // return value
}
// function definitions
```

Function Stubs

- Possible programming approach:
  - Write `main()` first and add functions as developed
  - Program cannot be run until all functions are included
- Stub: beginning of a final function
  - Can be used as a placeholder for a function until the function is completed
  - A “fake” function that accepts parameters and returns values in proper form
  - Allows `main` to be compiled and tested before all functions are completed

Functions with Empty Parameter Lists

- Extremely limited use
- Prototype format:

```
int display ();
int display (void);
```

- Information provided in above prototypes:
  - `display` takes no parameters
  - `display` returns an integer

Default Arguments

- Values listed in function prototype
  - Automatically transmitted to the called function when the arguments are omitted from function call
- Example:

```
void example (int, int = 5, double = 6.78);
```

  - Provides default values for last two arguments
  - Following function calls are valid:

```
example(7, 2, 9.3) // no defaults used
example(7, 2) // same as example(7, 2, 6.78)
example(7)  // same as example(7, 5, 6.78)
```

Reusing Function Names (Overloading)

- **Function overloading**: using same function name for more than one function
  - Compiler must be able to determine which function to use based on data types of parameters (not data type of return value)
- Each function must be written separately
  - Each exists as a separate entity
- Use of same name does not require code to be similar
  - Good programming practice: functions with the same name perform similar operations

Reusing Function Names (Overloading) (cont'd.)

Example: two functions named `cdabs( )`

```
void cdabs(int x) // compute and display the absolute
                    //value of an integer
{
   if ( x < 0 )
      x = -x;
   cout << "The absolute value of the integer is " << x <<
  endl;
}
void cdabs(float x) // compute and display the
                      //absolute value of a float
{
   if ( x < 0 )
      x = -x;
   cout << "The absolute value of the float is " << x <<
  endl;
}
```

Reusing Function Names (Overloading) (cont'd.)

- Function call: `cdabs(10);`
  - Causes compiler to use the function named `cdabs()` that expects and integer argument
- Function call: `cdabs(6.28f);`
  - Causes compiler to use the function named `cdabs()` that expects a double-precision argument
- Major use of overloaded functions
  - Constructor functions

Function Templates

- Most high-level languages require each function to be coded separately
  - Can lead to a profusion of names
- Example: functions to find the absolute value
  - Three separate functions and prototypes required

```
void abs (int);
void fabs (float);
void dabs (double);
```

- Each function performs the same operation
  - Only difference is data type handled

Function Templates (cont'd.)

- Example of function template:

```
template <class T>
void showabs(T number)
{
   if (number < 0)
      number = -number;
   cout << "The absolute value of the number "
             << " is " << number << endl;
   return;
}
```

- Template allows for one function instead of three
  - T represents a general data type
  - T replaced by an actual data type when compiler encounters a function call

Function Templates (cont'd.)

- Example (cont'd.):

```
int main()
{
   int num1 = -4;
   float num2 = -4.23F;
   double num3 = -4.23456;
   showabs(num1);
   showabs(num2);
   showabs(num3);
   return 0;
}
```

- Output from above program:

```
The absolute value of the number is 4
The absolute value of the number is 4.23
The absolute value of the number is 4.23456
```

Returning a Single Value

- Passing data to a function:
  - Called function receives only a copy of data sent to it
  - Protects against unintended change
  - Passed arguments called **pass by value** arguments
  - A function can receive many values (arguments) from the calling function

Returning a Single Value (cont'd.)

- Returning data from a function
  - Only one value directly returned from function
  - Called function header indicates type of data returned
- Examples:

```
void findMax(int x, int y)
```

  - `findMax` accepts two integer parameters and returns no value

```
float findMax (float x, float y)
```

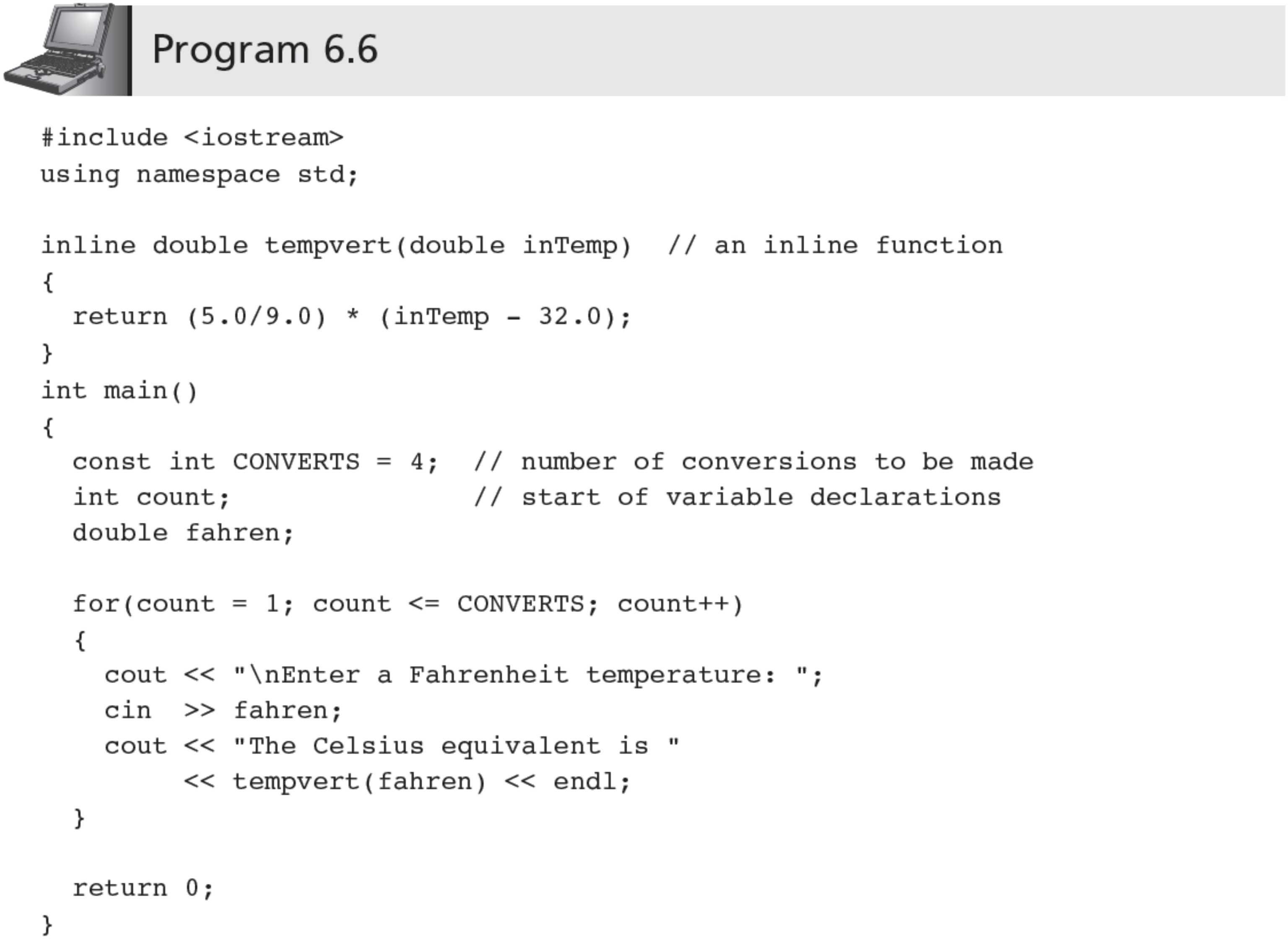
  - `findMax` accepts two floating-point values and returns a floating-point value
- To return a value, a function must use a `return` statement

Inline Functions

- Calling functions associated overhead
  - Placing arguments in reserved memory (stack)
  - Passing control to the function
  - Providing stack space for any returned value
  - Returning to correct point in calling program
- Overhead justified when function is called many times
  - Better than repeating code

Inline Functions (cont'd.)

- Overhead not justified for small functions that are not called frequently
  - Still convenient to group repeating lines of code into a common function name
- **Inline function**: avoids overhead problems
  - C++ compiler instructed to place a copy of inline function code into the program wherever the function is called

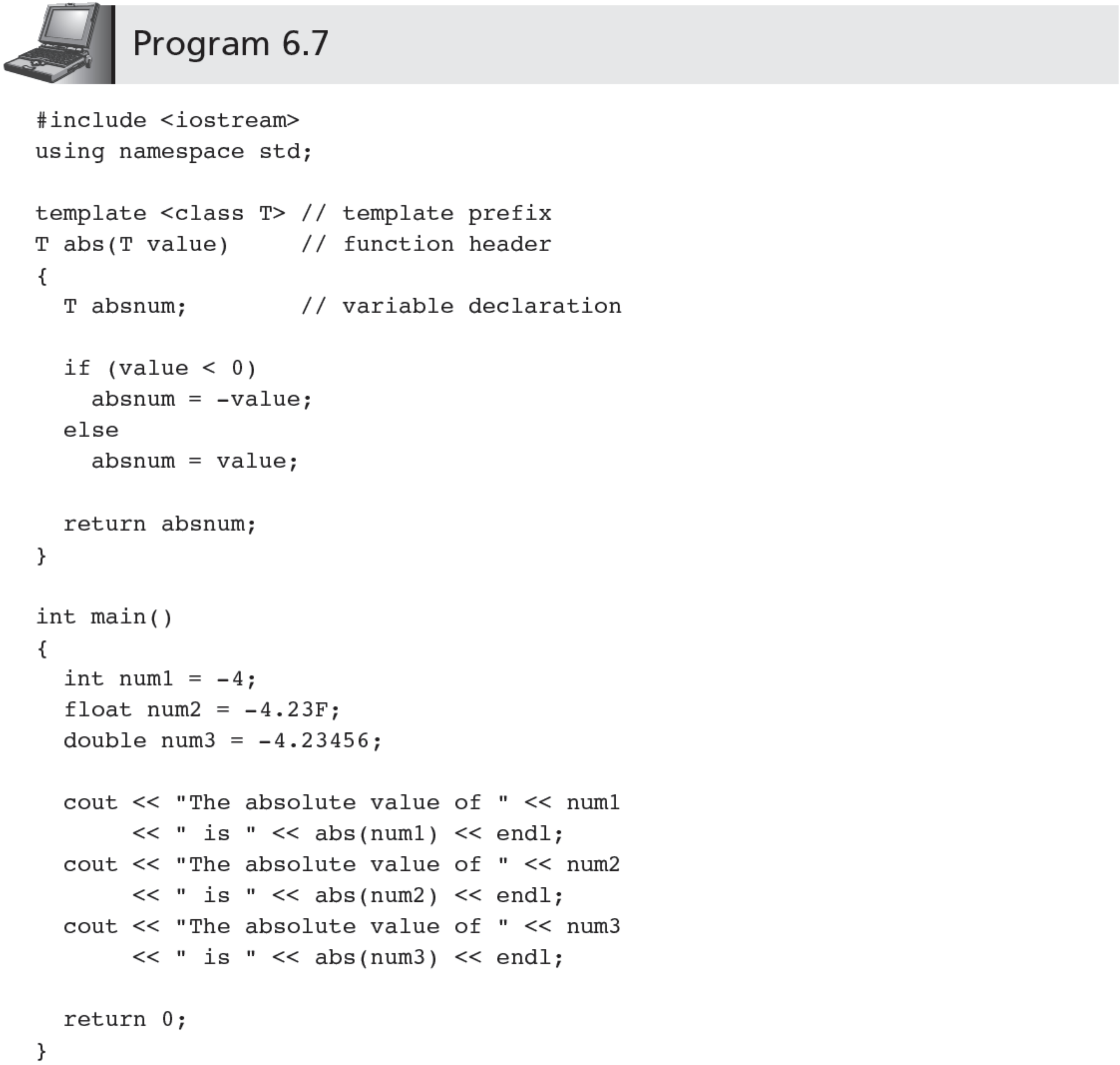


Program 6.6

```
#include <iostream>
using namespace std;

inline double tempvert(double inTemp)  // an inline function
{
  return (5.0/9.0) * (inTemp - 32.0);
}
int main()
{
  const int CONVERTS = 4;  // number of conversions to be made
  int count;               // start of variable declarations
  double fahren;

  for(count = 1; count <= CONVERTS; count++)
  {
    cout << "\nEnter a Fahrenheit temperature: ";
    cin  >> fahren;
    cout << "The Celsius equivalent is "
         << tempvert(fahren) << endl;
  }

  return 0;
}
```

Templates with a Return Value

- Returning a value from a function template is identical to returning a value from a function
- Data type T is also used to declare the return type of the function



Program 6.7

```
#include <iostream>
using namespace std;

template <class T> // template prefix
T abs(T value)     // function header
{
  T absnum;        // variable declaration

  if (value < 0)
    absnum = -value;
  else
    absnum = value;

  return absnum;
}

int main()
{
  int num1 = -4;
  float num2 = -4.23F;
  double num3 = -4.23456;

  cout << "The absolute value of " << num1
       << " is " << abs(num1) << endl;
  cout << "The absolute value of " << num2
       << " is " << abs(num2) << endl;
  cout << "The absolute value of " << num3
       << " is " << abs(num3) << endl;

  return 0;
}
```

Returning Multiple Values

- Called function usually receives values as pass by value
  - A distinct advantage of C++
- Sometimes desirable to allow function to have direct access to variables
  - Address of variable must be passed to function
  - Function can directly access and change the value stored there
- **Pass by reference**: passing addresses of variables received from calling function

Passing and Using Reference Parameters

- Reference parameter: receives the address of an argument passed to called function
- Example: accept two addresses in function `newval()`
- Function header:

```
void newval (double& num1, double& num2)
```

  - Ampersand, &, means “the address of”
- Function prototype:

```
void newval (double&, double&);
```

Variable Scope

- **Scope**: section of program where identifier is valid (known or visible)
- **Local variables** (local scope): variables created inside a function
  - Meaningful only when used in expressions inside the function in which it was declared
- **Global variables** (global scope): variables created outside any function
  - Can be used by all functions placed after the global variable declaration

Scope Resolution Operator

- Local variable with the same name as a global variable
  - All references to variable name within scope of local variable refer to the local variable
  - Local variable name takes precedence over global variable name
- Scope resolution operator (`::`)
  - When used before a variable name, the compiler is instructed to use the global variable

```
::number   // scope resolution operator
        // causes global variable to be used
```

Misuse of Globals

- Avoid overuse of globals
  - Too many globals eliminates safeguards provided by C++ to make functions independent
  - Misuse does not apply to function prototypes
    - Prototypes are typically global
- Difficult to track down errors in a large program using globals
  - Global variable can be accessed and changed by any function following the global declaration

Variable Storage Category

- Scope has a space and a time dimension
- Time dimension (lifetime): length of time that storage locations are reserved for a variable
  - All variable storage locations released back to operating system when program finishes its run
  - During program execution, interim storage locations are reserved
    - **Storage category**: determines length of time that variable's storage locations are reserved
    - Four classes: `auto`, `static`, `extern`, `register`

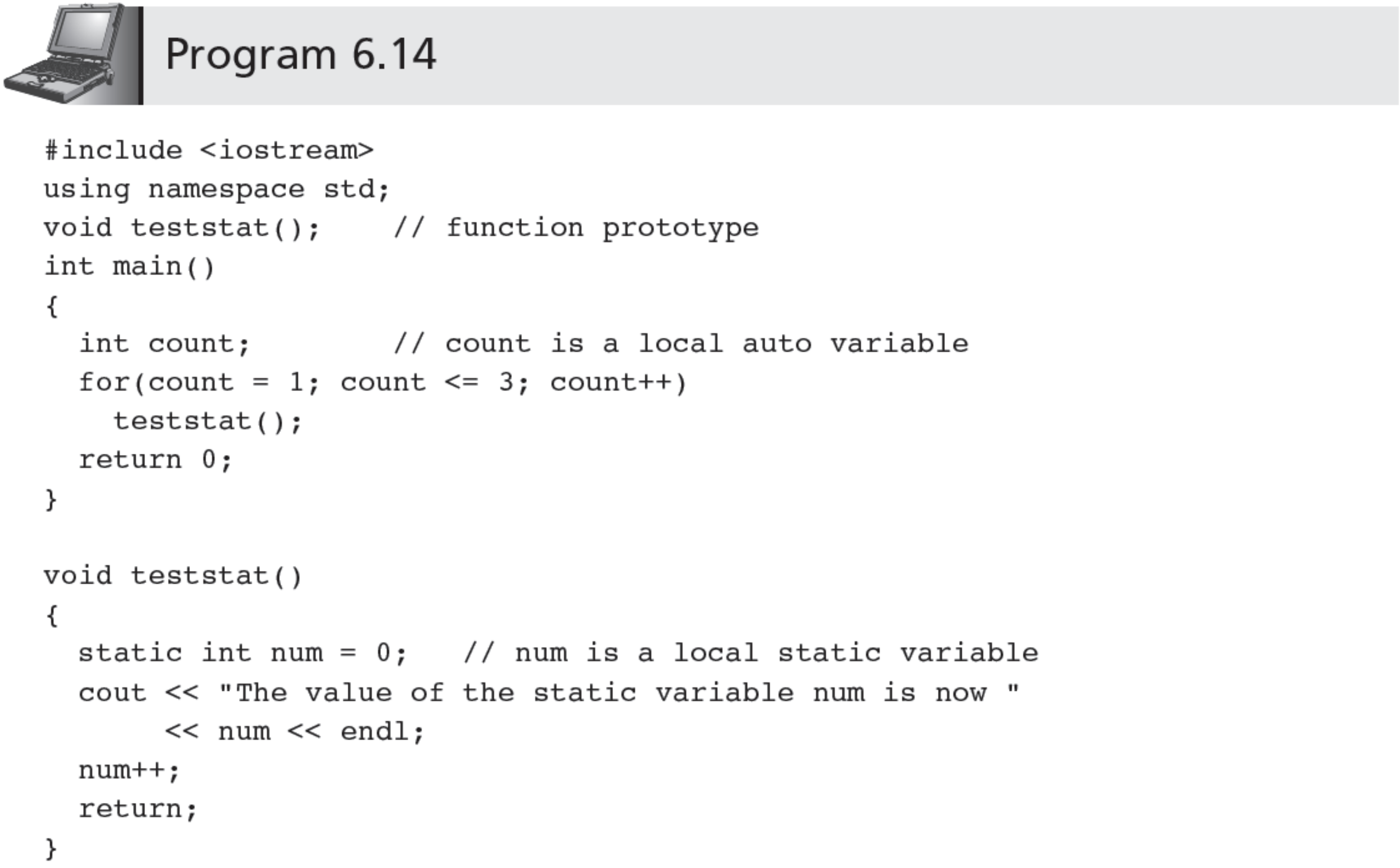
Local Variable Storage Categories

- Local variable can only be members of `auto`, `static`, or `register` class
- `auto` class: default, if no class description included in variable's declaration statement
- Storage for `auto` local variables automatically reserved (created)
  - Each time a function declaring `auto` variables is called
  - Local `auto` variables are "alive" until function returns control to calling function

Local Variable Storage Categories (cont'd.)

- `static` storage class: allows a function to remember local variable values between calls
  - `static` local variable lifetime = lifetime of program
  - Value stored in variable when function is finished is available to function next time it is called
- Initialization of `static` variables (local and global)
  - Done one time only, when program first compiled
  - Only constants or constant expressions allowed

Local Variable Storage Categories (cont'd.)



Program 6.14

```
#include <iostream>
using namespace std;
void teststat();    // function prototype
int main()
{
  int count;        // count is a local auto variable
  for(count = 1; count <= 3; count++)
    teststat();
  return 0;
}

void teststat()
{
  static int num = 0;   // num is a local static variable
  cout << "The value of the static variable num is now "
       << num << endl;
  num++;
  return;
}
```

Local Variable Storage Categories (cont'd.)

- `register` storage class: same as `auto` class except for location of storage for class variables
  - – Uses high-speed registers
  - – Can be accessed faster than normal memory areas
    - Improves program execution time
- Some computers do not support `register` class
  - – Variables automatically switched to `auto` class

Global Variable Storage Classes

- Global variables: created by definition statements external to a function
  - Do not come and go with the calling of a function
  - Once created, a global variable is alive until the program in which it is declared finishes executing
  - May be declared as members of `static` or `extern` classes
- Purpose: to extend the scope of a global variable beyond its normal boundaries

Common Programming Errors

- Passing incorrect data types between functions
  - Values passed must correspond to data types declared for function parameters
- Declaring same variable name in calling and called functions
  - A change to one local variable does not change value in the other
- Assigning same name to a local and a global variable
  - Use of a variable’s name only affects local variable’s contents unless the `::` operator is used

Common Programming Errors (cont'd.)

- Omitting a called function’s prototype
  - The calling function must be alerted to the type of value that will be returned
- Terminating a function’s header line with a semicolon
- Forgetting to include the data type of a function’s parameters within the function header line

Summary

- A function is called by giving its name and passing data to it
  - If a variable is an argument in a call, the called function receives a copy of the variable's value
- Common form of a user-written function:

```
returnDataType functionName(parameter list)
{
   declarations and other C++ statements;
     return expression;
 }
```

Summary (cont'd.)

- A function’s return type is the data type of the value returned by the function
  - If no type is declared, the function is assumed to return an integer value
  - If the function does not return a value, it should be declared as a `void` type
- Functions can directly return at most a single data type value to their calling functions
  - This value is the value of the expression in the `return` statement

Summary (cont'd.)

- Reference parameter: passes the address of a variable to a function
- Function prototype: function declaration
- Scope: determines where in a program the variable can be used
- Variable storage category: determines how long the value in a variable will be retained

Chapter Supplement: Generating Random Numbers

- **Random numbers**
  - Series of numbers whose order can’t be predicted
  - In practice, finding truly random numbers is hard
- **Pseudorandom numbers**
  - Random enough for the type of applications being programmed
- All C++ compilers provide two general-purpose functions for generating random numbers
  - `rand()` and `srand()`

Scaling

- **Scaling**
  - Procedure for adjusting the random numbers produced by a random-number generator to fall in a specified range
- Scaling random numbers to lie in the range 0.0 to 1.0

```
double(rand())/RAND_MAX
```

- Scaling a random number as an integer value between 0 and *N*

```
rand() % (N+1)
int(double(rand())/RAND_MAX * N)
```