Chapter 6

Functions

Part 2

6.10

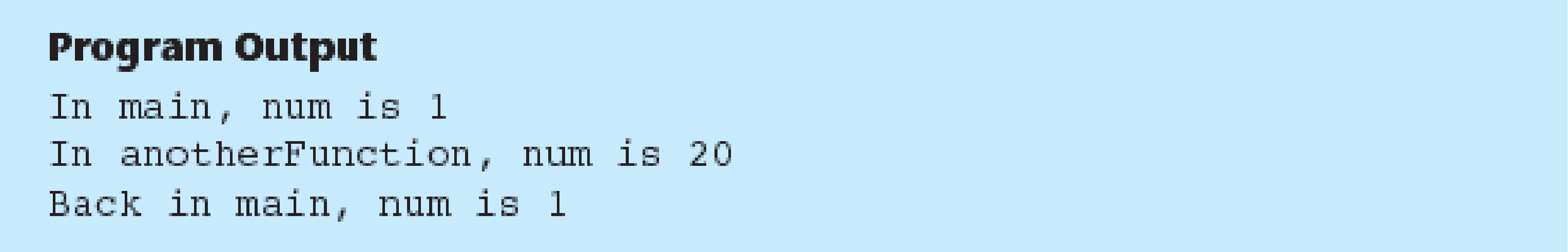
Local and Global Variables

Local and Global Variables

- Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them.
- Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

Program 6-15

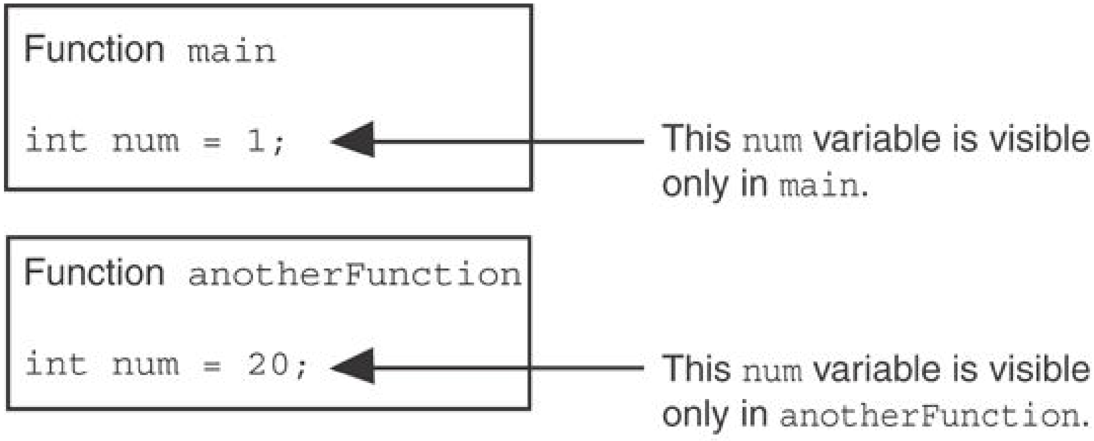
```cpp
// This program shows that variables defined in a function
// are hidden from other functions.
#include <iostream>
using namespace std;

void anotherFunction(); // Function prototype

int main()
{
   int num = 1;    // Local variable

   cout << "In main, num is " << num << endl;
   anotherFunction();
   cout << "Back in main, num is " << num << endl;
   return 0;
}

//*****************************************************
// Definition of anotherFunction                      *
// It has a local variable, num, whose initial value  *
// is displayed.                                      *
//*****************************************************

void anotherFunction()
{
   int num = 20;  // Local variable

   cout << "In anotherFunction, num is " << num << endl;
}
```



Program Output

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 1
```

When the program is executing in `main`, the `num` variable defined in `main` is visible. When `anotherFunction` is called, however, only variables defined inside it are visible, so the `num` variable in `main` is hidden.



Function `main`

`int num = 1;`

This `num` variable is visible only in `main`.

Function `anotherFunction`

`int num = 20;`

This `num` variable is visible only in `anotherFunction`.

Local Variable Lifetime

- A function's local variables exist only while the function is executing. This is known as the *lifetime* of a local variable.

- When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter variables are destroyed.

- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Global Variables and Global Constants

- A global variable is any variable defined outside all the functions in a program.
- The scope of a global variable is the portion of the program from the variable definition to the end.
- This means that a global variable can be accessed by *all* functions that are defined after the global variable is defined.
- Example program 6-16.

Program 6-16

```
// This program shows that a global variable is visible
// to all the functions that appear in a program after
// the variable's declaration.
#include <iostream>
using namespace std;

void anotherFunction(); // Function prototype
int num = 2;         // Global variable

int main()
{
   cout << "In main, num is " << num << endl;
   anotherFunction();
   cout << "Back in main, num is " << num << endl;
   return 0;
}
```

Output

```
In main, num is 2

In anotherFunction, num is 2

But, it is now changed to 52

Back in main, num is 52
```

Program 6-16

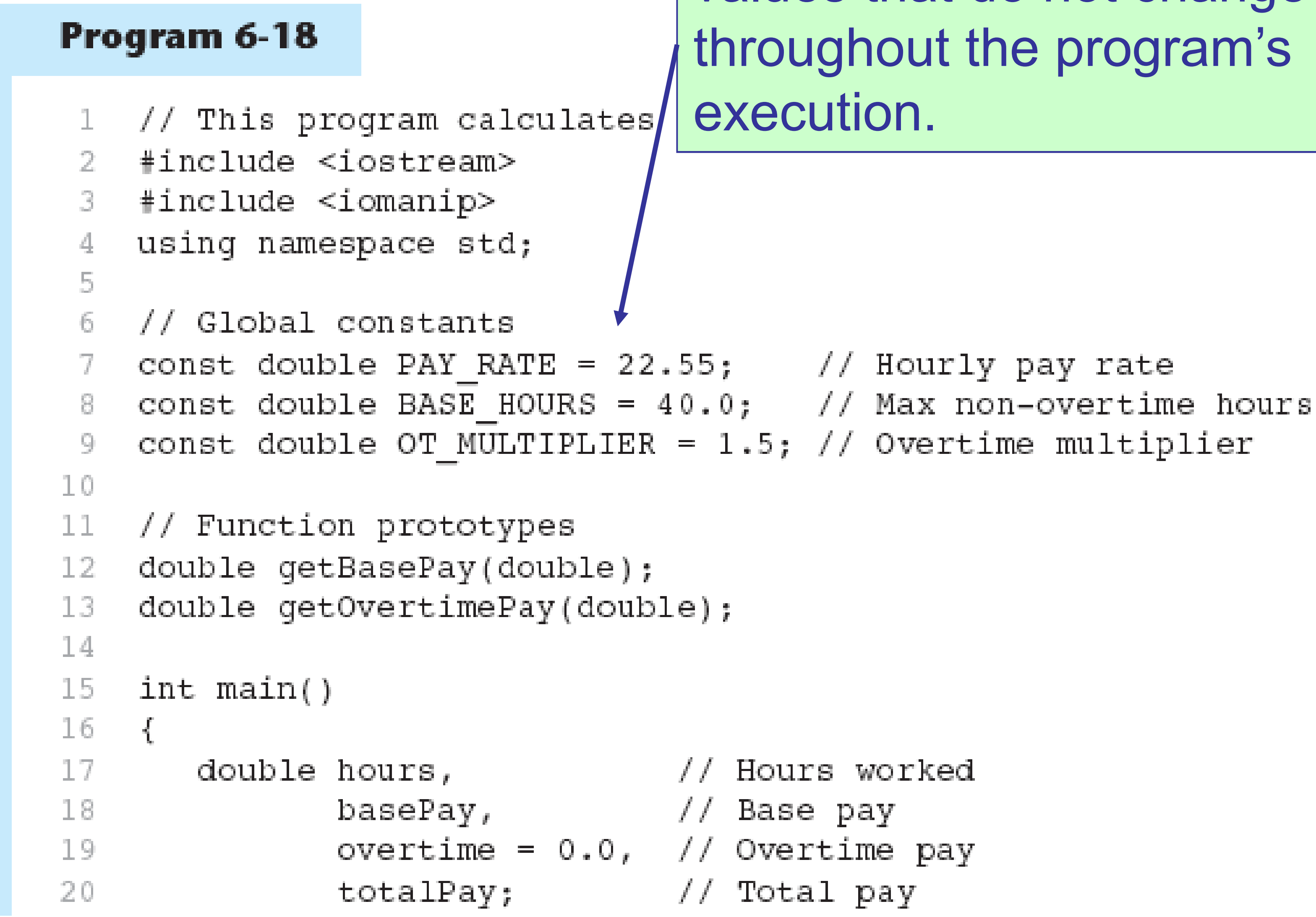
```
//**************************************************
// Definition of anotherFunction                    *
// This function changes the value of the           *
// global variable num.                             *
//**************************************************

void anotherFunction()
{
   cout << "In anotherFunction, num is " << num << endl;
   num = 50;
   cout << "But, it is now changed to " << num << endl;
}
```

Global Variables and Global Constants

- You should avoid using global variables because they make programs difficult to debug.

- Any global that you create should be *global constants*.

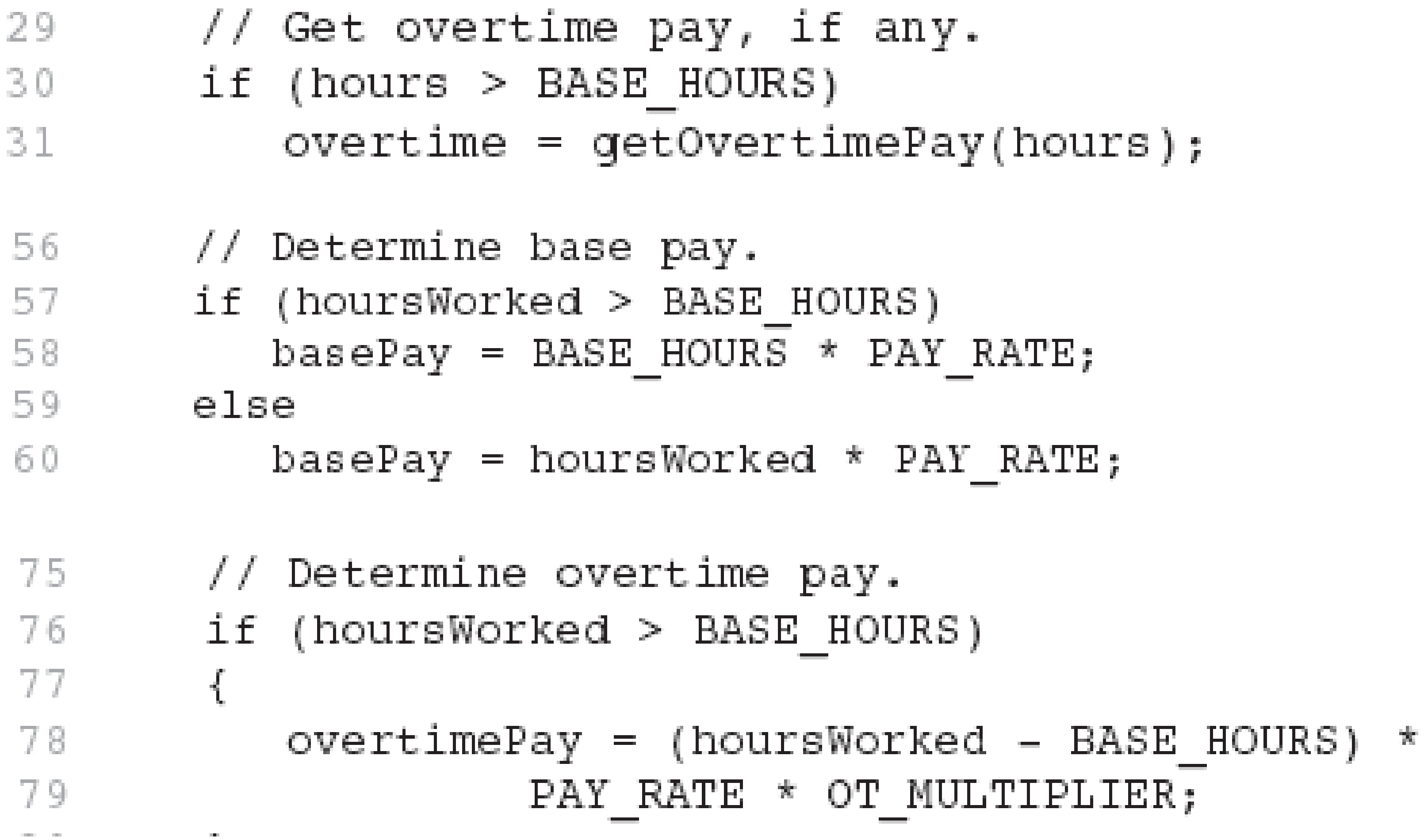
Global constants defined for values that do not change throughout the program’s execution.



Program 6-18

```
// This program calculates
#include <iostream>
#include <iomanip>
using namespace std;

// Global constants
const double PAY_RATE = 22.55;    // Hourly pay rate
const double BASE_HOURS = 40.0;   // Max non-overtime hours
const double OT_MULTIPLIER = 1.5; // Overtime multiplier

// Function prototypes
double getBasePay(double);
double getOvertimePay(double);

int main()
{
   double hours,            // Hours worked
          basePay,          // Base pay
          overtime = 0.0,   // Overtime pay
          totalPay;         // Total pay
```

The constants are then used for those values throughout the program.



```
// Get overtime pay, if any.
if (hours > BASE_HOURS)
   overtime = getOvertimePay(hours);

// Determine base pay.
if (hoursWorked > BASE_HOURS)
   basePay = BASE_HOURS * PAY_RATE;
else
   basePay = hoursWorked * PAY_RATE;

// Determine overtime pay.
if (hoursWorked > BASE_HOURS)
{
   overtimePay = (hoursWorked - BASE_HOURS) *
                 PAY_RATE * OT_MULTIPLIER;
```

Initializing Local and Global Variables

- Local variables are not automatically initialized. They must be initialized by programmer.

- Global variables (not constants) are automatically initialized to `0` (numeric) or `NULL` (character) when the variable is defined.

6.11

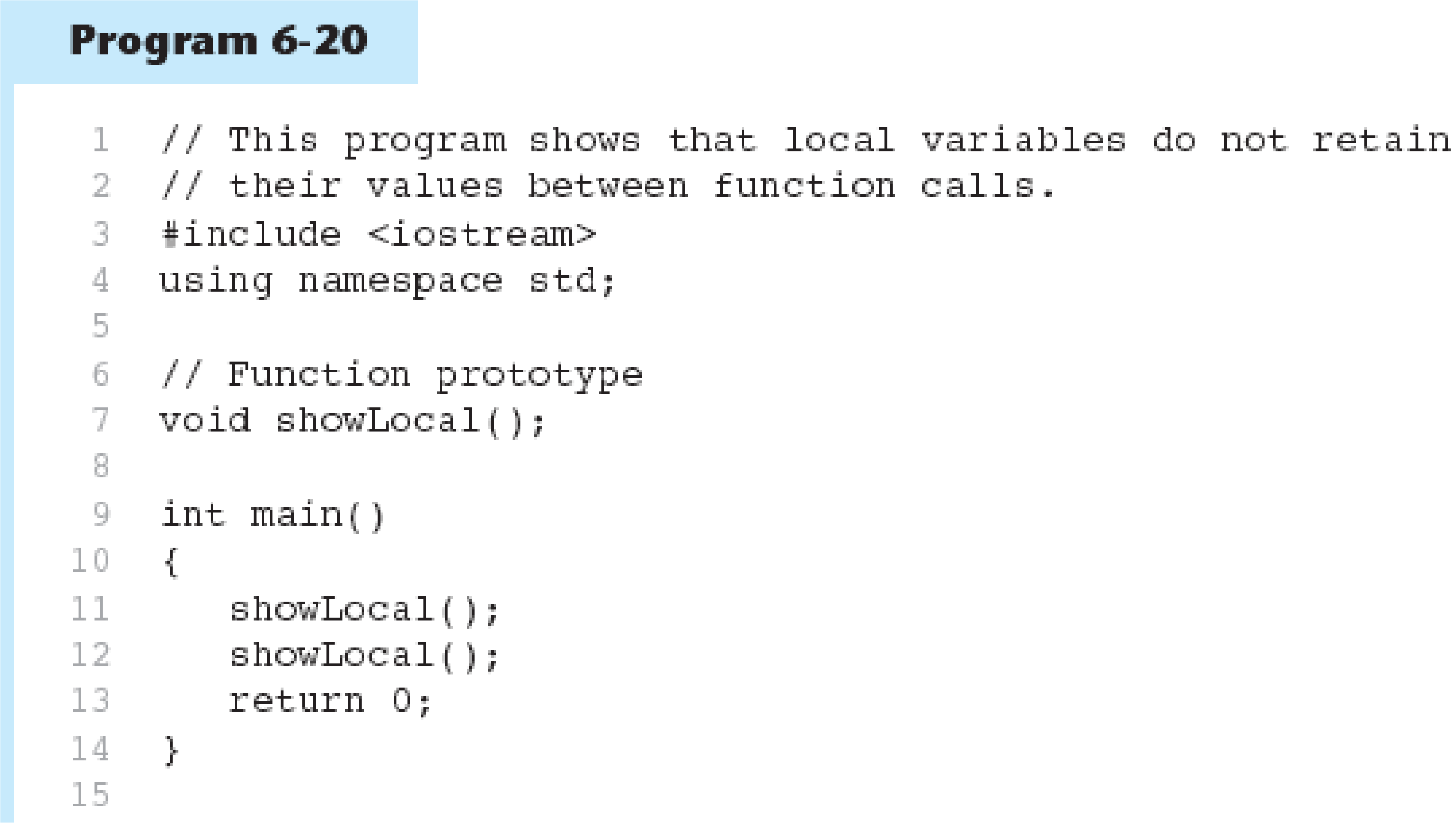
Static Local Variables

Static Local Variables

- Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.

- `static` local variables retain their contents between function calls.

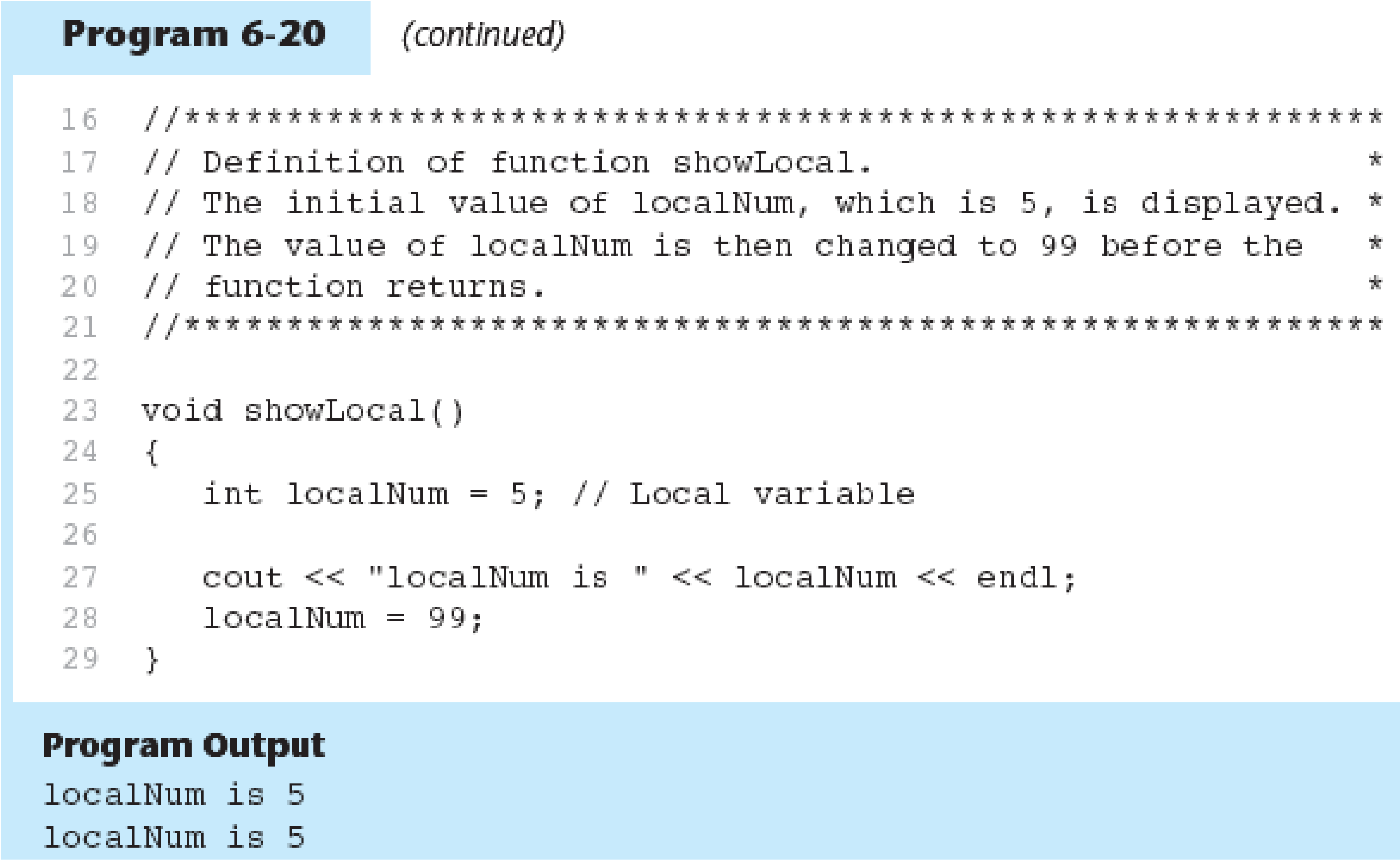
- `static` local variables are defined and initialized only the first time the function is executed. `0` is the default initialization value.



Program 6-20

```
// This program shows that local variables do not retain
// their values between function calls.
#include <iostream>
using namespace std;

// Function prototype
void showLocal();

int main()
{
   showLocal();
   showLocal();
   return 0;
}

```

(Program Continues)



Program 6-20 *(continued)*

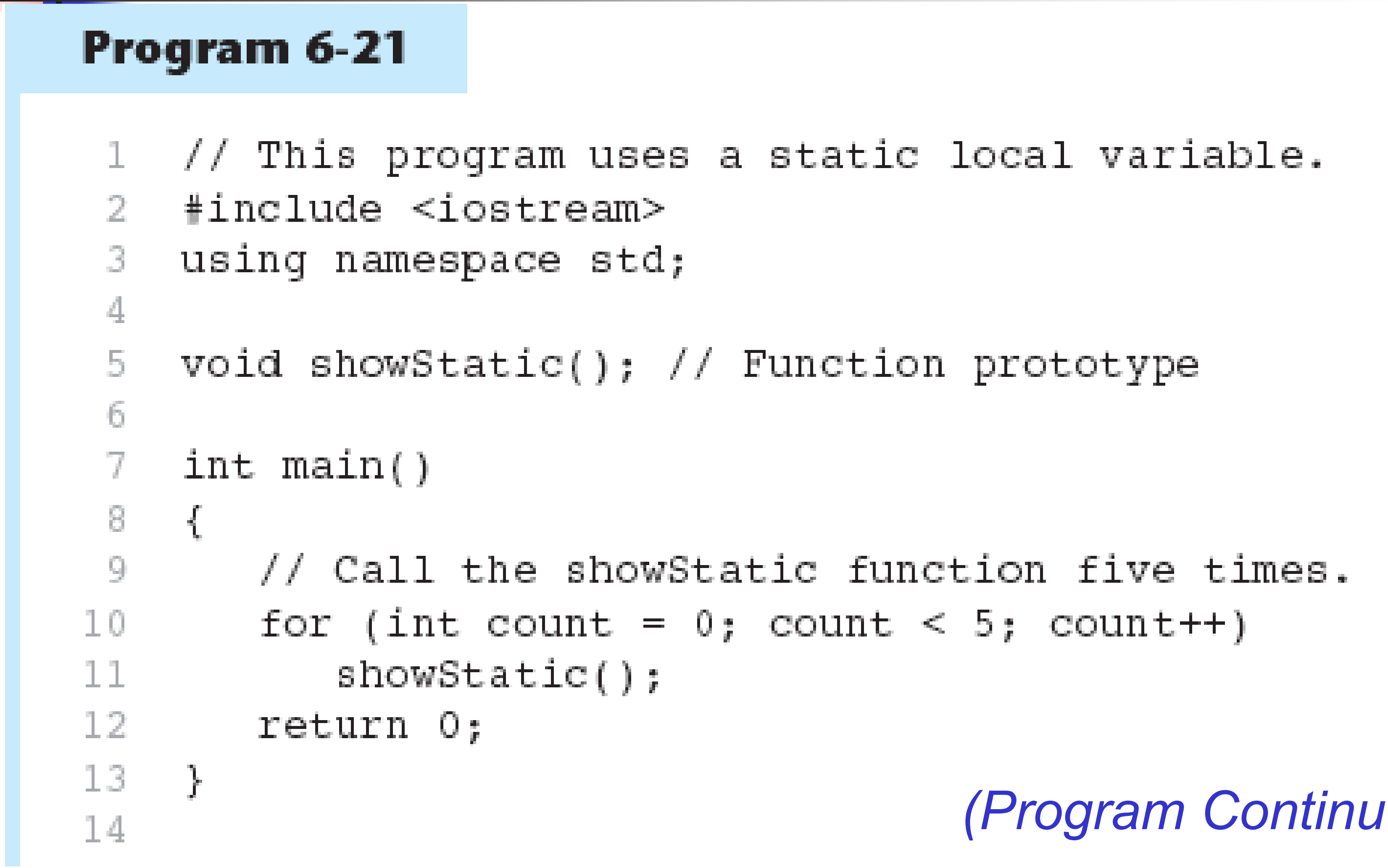
```
//***********************************************************
// Definition of function showLocal.                        *
// The initial value of localNum, which is 5, is displayed. *
// The value of localNum is then changed to 99 before the   *
// function returns.                                        *
//***********************************************************

void showLocal()
{
   int localNum = 5; // Local variable

   cout << "localNum is " << localNum << endl;
   localNum = 99;
}
```

Program Output

```
localNum is 5
localNum is 5
```

In this program, each time **`showLocal`** is called, the **`localNum`** variable is re-created and initialized with the value 5.

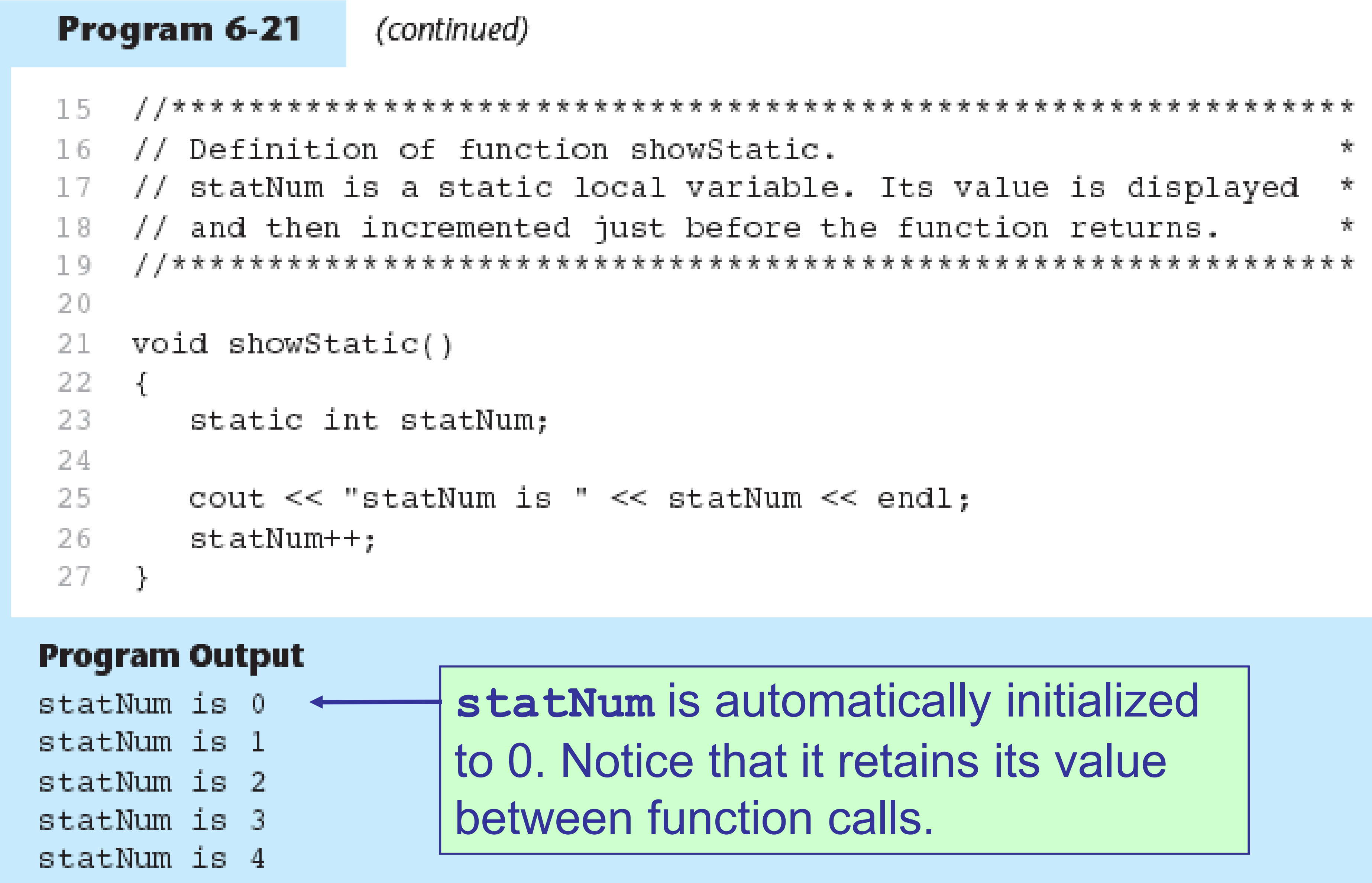
A Different Approach, Using a Static Variable



Program 6-21

```
// This program uses a static local variable.
#include <iostream>
using namespace std;

void showStatic(); // Function prototype

int main()
{
    // Call the showStatic function five times.
    for (int count = 0; count < 5; count++)
        showStatic();
    return 0;
}

```

(Program Continues)



Program 6-21 *(continued)*

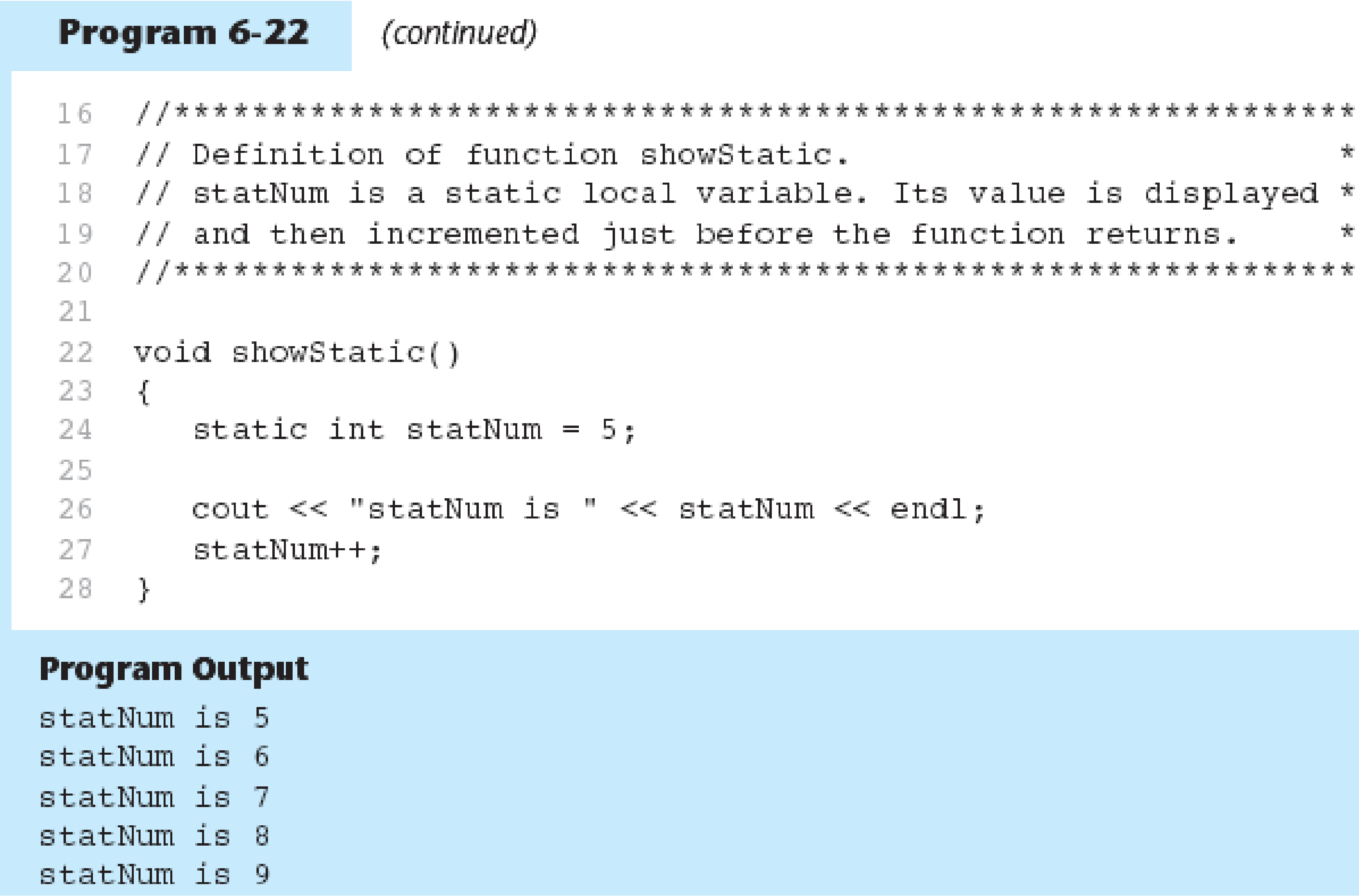
```
//*****************************************************************
// Definition of function showStatic.                             *
// statNum is a static local variable. Its value is displayed     *
// and then incremented just before the function returns.         *
//*****************************************************************

void showStatic()
{
   static int statNum;

   cout << "statNum is " << statNum << endl;
   statNum++;
}
```

Program Output

```
statNum is 0
statNum is 1
statNum is 2
statNum is 3
statNum is 4
```

`statNum` is automatically initialized to 0. Notice that it retains its value between function calls.

If you do initialize a local static variable, the initialization only happens once. See Program 6-22…



Program 6-22 *(continued)*

```
//*************************************************************
// Definition of function showStatic.                         *
// statNum is a static local variable. Its value is displayed *
// and then incremented just before the function returns.     *
//*************************************************************

void showStatic()
{
   static int statNum = 5;

   cout << "statNum is " << statNum << endl;
   statNum++;
}
```

Program Output

```
statNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9
```

6.12

Default Arguments

Default Arguments

A Default argument is an argument that is passed automatically to a parameter if the argument is missing on the function call.

- Must be a constant declared in prototype:

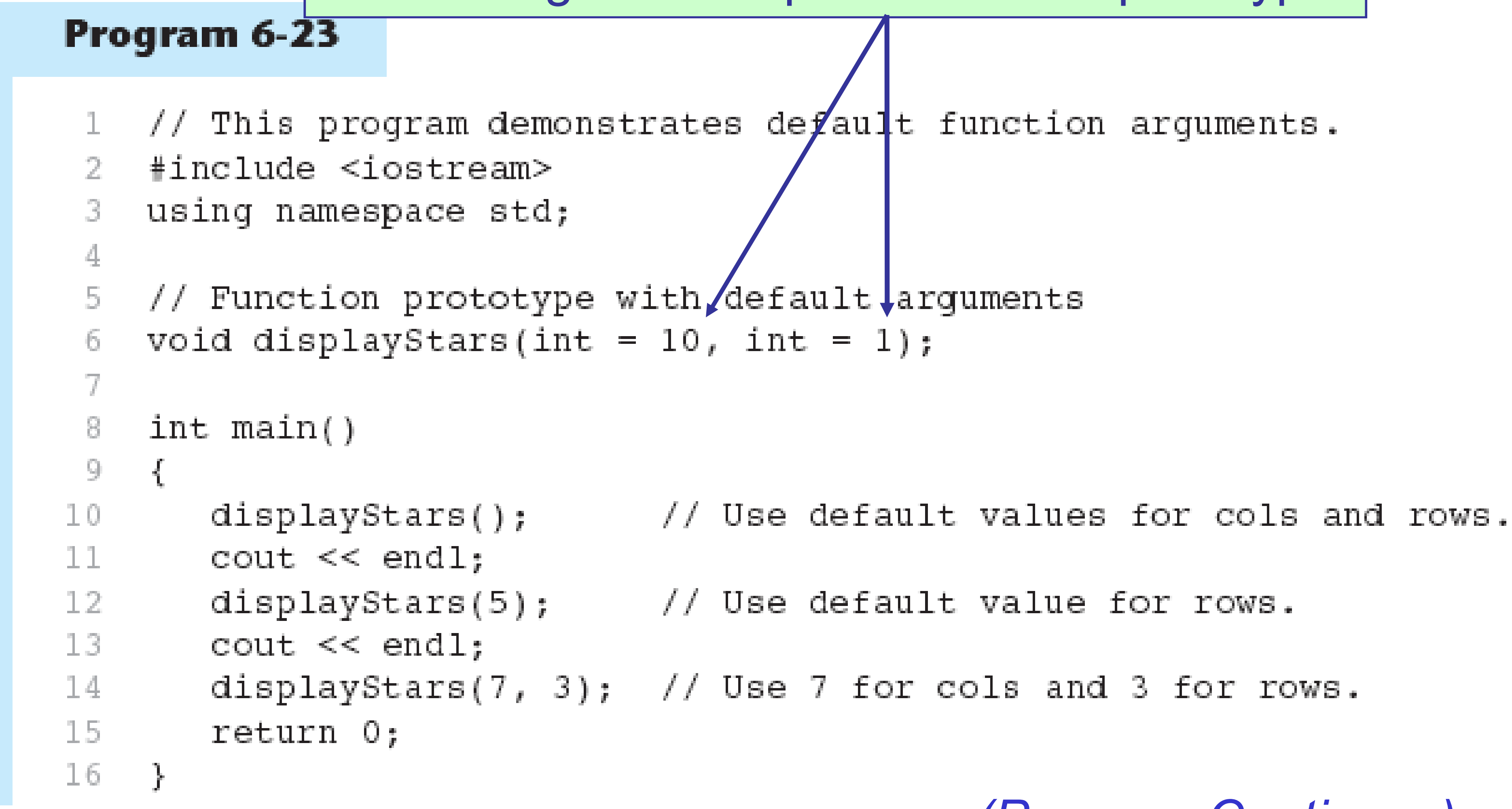
```
void evenOrOdd(int = 0);
```

- Can be declared in header if no prototype

- Multi-parameter functions may have default arguments for some or all of them:

```
int getSum(int, int=0, int=0);
```

Default arguments specified in the prototype

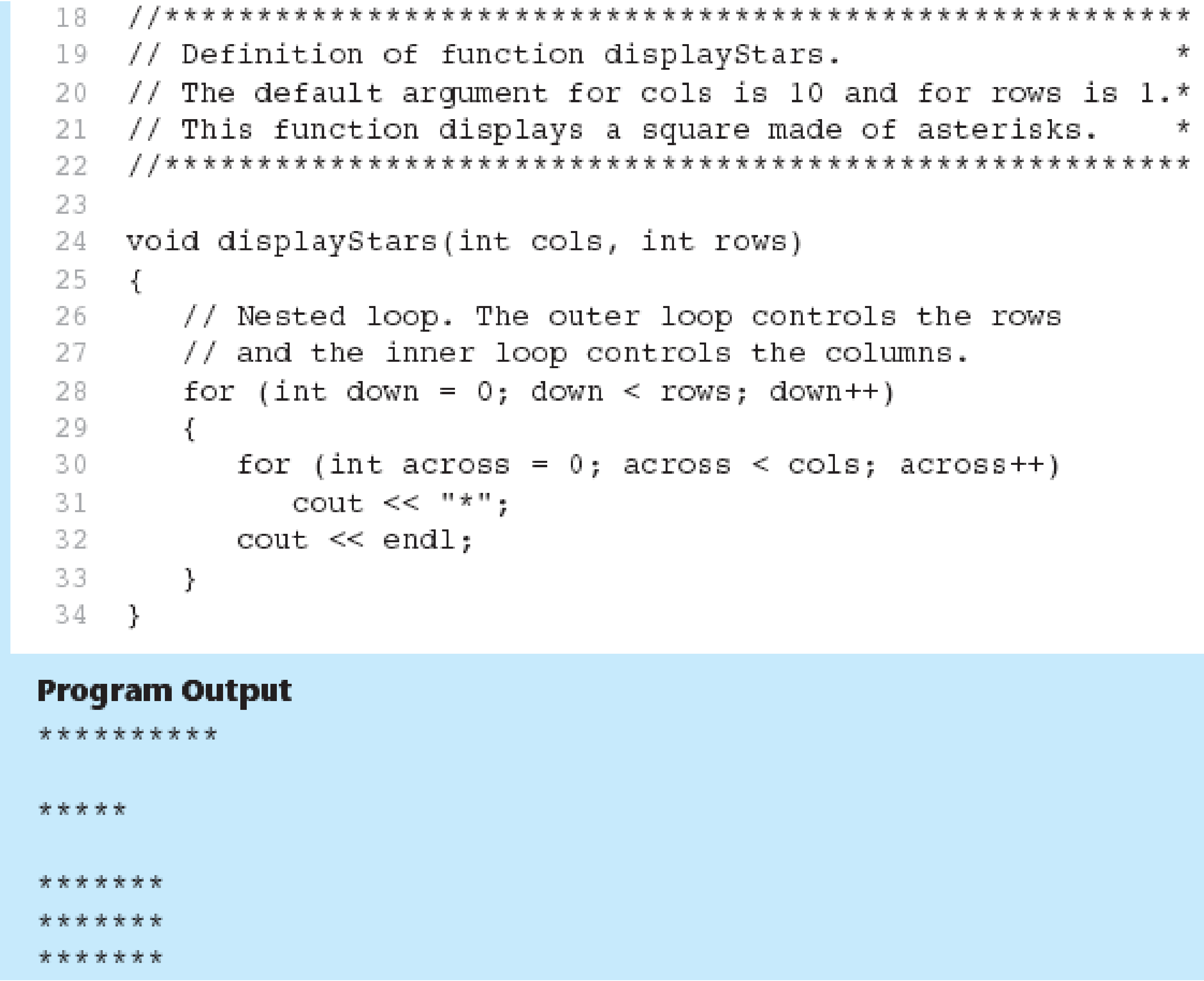


Program 6-23

```
// This program demonstrates default function arguments.
#include <iostream>
using namespace std;

// Function prototype with default arguments
void displayStars(int = 10, int = 1);

int main()
{
   displayStars();        // Use default values for cols and rows.
   cout << endl;
   displayStars(5);       // Use default value for rows.
   cout << endl;
   displayStars(7, 3);    // Use 7 for cols and 3 for rows.
   return 0;
}
```

(Program Continues)

Program 6-23 *(Continued)*



```
//********************************************************
// Definition of function displayStars.                  *
// The default argument for cols is 10 and for rows is 1.*
// This function displays a square made of asterisks.    *
//********************************************************

void displayStars(int cols, int rows)
{
   // Nested loop. The outer loop controls the rows
   // and the inner loop controls the columns.
   for (int down = 0; down < rows; down++)
   {
      for (int across = 0; across < cols; across++)
         cout << "*";
      cout << endl;
   }
}
```

Program Output

```
**********

*****

*******
*******
*******
```

Default Arguments

- If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:

```
int getSum(int, int=0, int=0); // OK
int getSum(int, int=0, int);  // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
  sum = getSum(num1, num2);    // OK
  sum = getSum(num1, , num3);  // NO
```

6.13

Using Reference Variables as Parameters

Using Reference Variables as Parameters

- A mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to ‘return’ more than one value

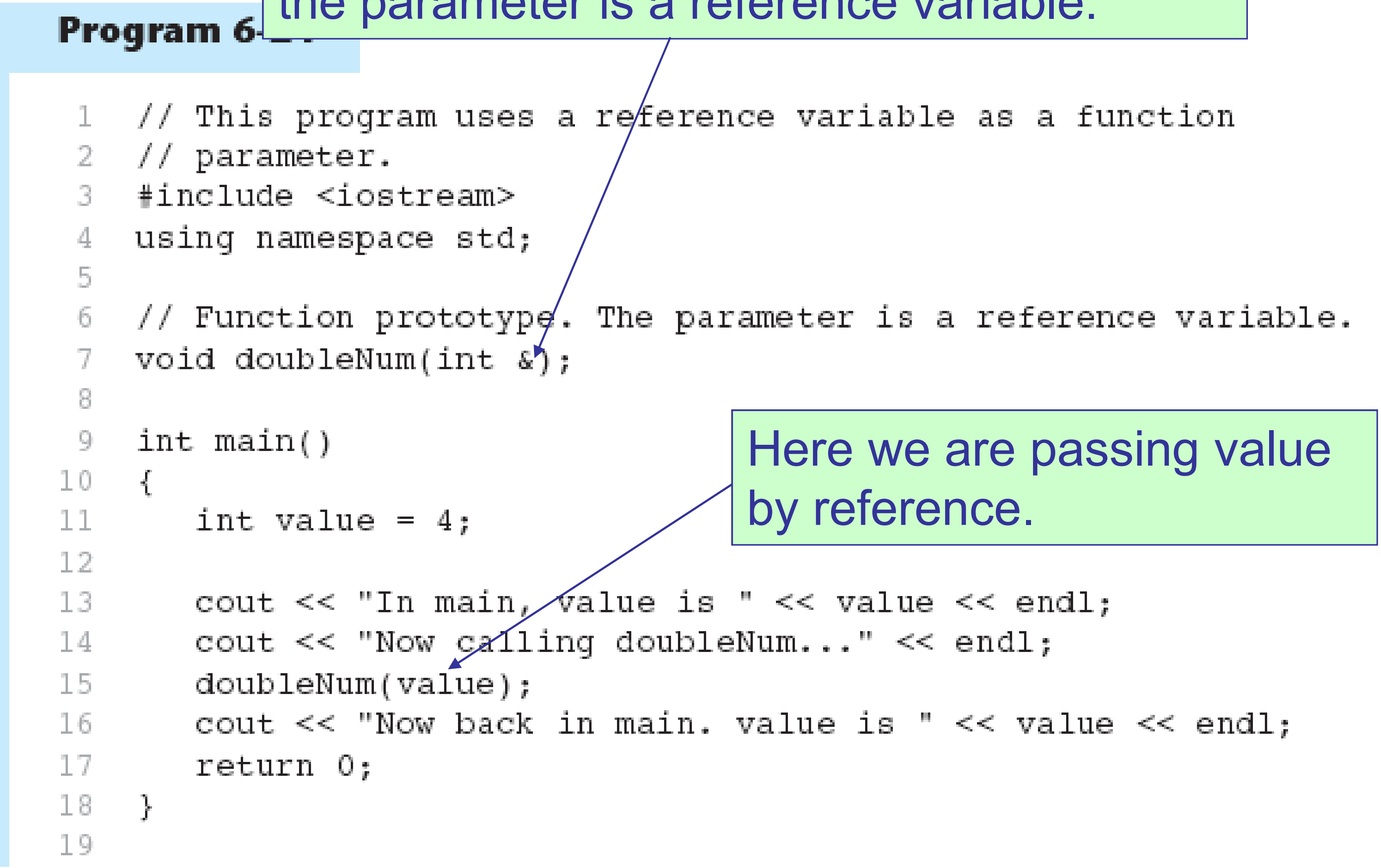
Passing by Reference

- A reference variable is an alias for another variable
- Defined with an ampersand (`&`)

```
void getDimensions(int&, int&);
```

- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by reference*

The & here in the prototype indicates that the parameter is a reference variable.



Program 6-

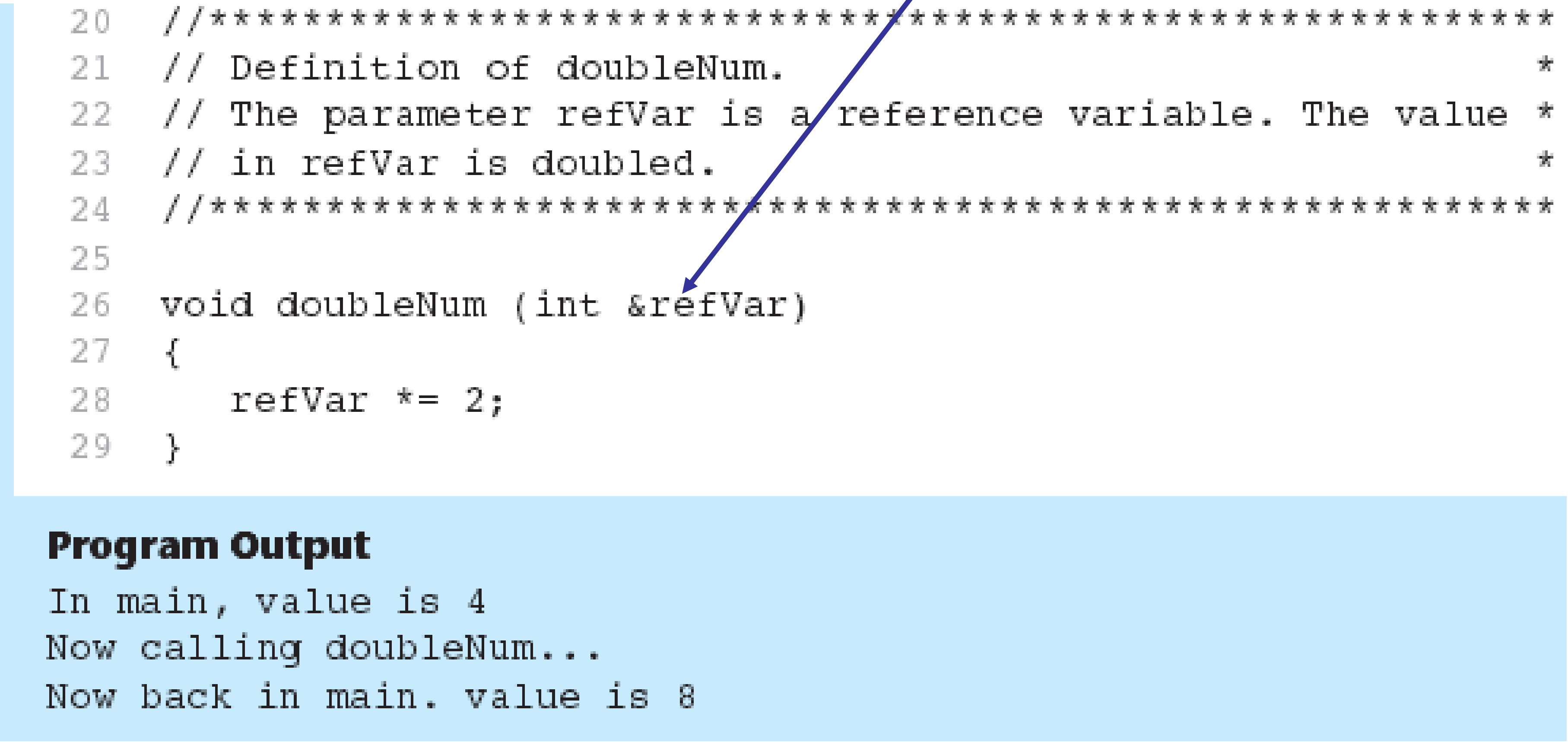
```
// This program uses a reference variable as a function
// parameter.
#include <iostream>
using namespace std;

// Function prototype. The parameter is a reference variable.
void doubleNum(int &);

int main()
{
   int value = 4;

   cout << "In main, value is " << value << endl;
   cout << "Now calling doubleNum..." << endl;
   doubleNum(value);
   cout << "Now back in main. value is " << value << endl;
   return 0;
}

```

Here we are passing value by reference.

(Program Continues)

Program 6-24 *(Continued)*

The & also appears here in the function header.



```
//******************************************************************
// Definition of doubleNum.                                        *
// The parameter refVar is a reference variable. The value *
// in refVar is doubled.                                           *
//******************************************************************

void doubleNum (int &refVar)
{
   refVar *= 2;
}
```

Program Output

```
In main, value is 4
Now calling doubleNum...
Now back in main. value is 8
```

Reference Variable Notes

- Each reference parameter must contain `&`
- Space between type and `&` is unimportant
- Must use `&` in both prototype and header
- Argument passed to reference parameter must be a variable – cannot be an expression or constant
- Use when appropriate – don't use when argument should not be changed by function, or if function needs to return only 1 value

Question: The following program asks the user to enter two numbers. What is the output of the program if the user enters **12** and **14**

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

void func1(int &, int &);
void func2(int &, int &, int &);
void func3 (int, int, int);

int main()
{
        int x = 0, y = 0, z = 0;
        cout << x << " " << y << " "<< z << endl;
        func1(x, y);
        cout << x << " " << y << " "<< z << endl;
        func2(x,  y,  z);
        cout << x << " " << y << " " << z << endl;
        func3 (x, y, z);
        cout << x << " " << y << " " << z << endl;
        return 0;
}
```

```
void  func1(int &a, int &b)
{
   cout <<"Enter two numbers: ";
   cin >> a >> b;
}

void  func2 (int &a, int &b, int &c)
{
        b++;
        c--;
        a = b+c;
}
void  func3 (int a, int b, int c)
{
        a = b-c;
}
```

6.14

Overloading Functions

Overloading Functions

- Overloaded functions have the same name but different parameter lists
- Can be used to create functions that perform the same task but take different parameter types or different number of parameters
- Compiler will determine which version of function to call by argument and parameter lists

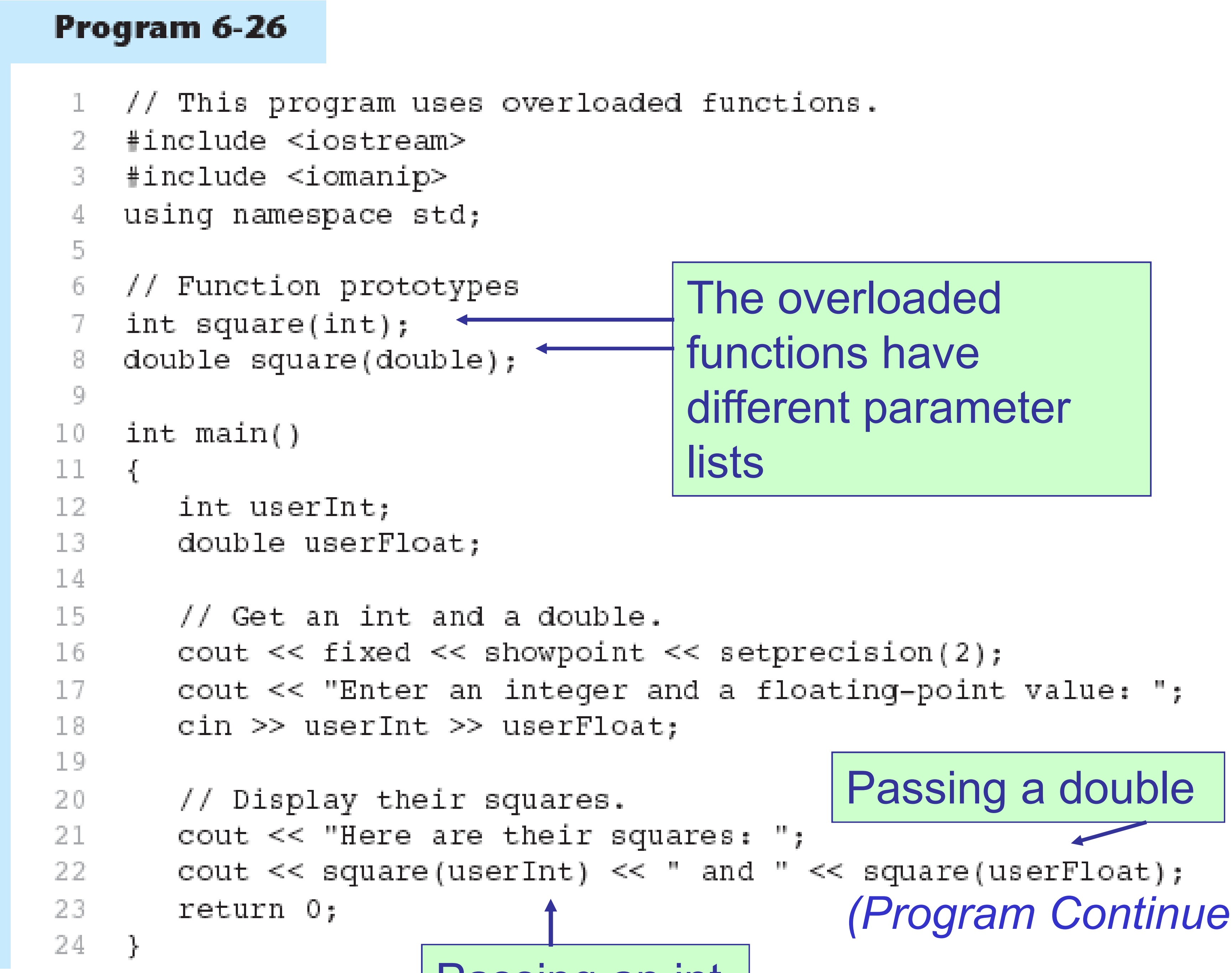
Function Overloading Examples

Using these overloaded functions,

```
void getDimensions(int);              // 1
void getDimensions(int, int);         // 2
void getDimensions(int, double);      // 3
void getDimensions(double, double);// 4
```

the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);              // 1
getDimensions(length, width);       // 2
getDimensions(length, height);      // 3
getDimensions(height, base);        // 4
```



Program 6-26

```
// This program uses overloaded functions.
#include <iostream>
#include <iomanip>
using namespace std;

// Function prototypes
int square(int);
double square(double);

int main()
{
   int userInt;
   double userFloat;

   // Get an int and a double.
   cout << fixed << showpoint << setprecision(2);
   cout << "Enter an integer and a floating-point value: ";
   cin >> userInt >> userFloat;

   // Display their squares.
   cout << "Here are their squares: ";
   cout << square(userInt) << " and " << square(userFloat);
   return 0;
}
```

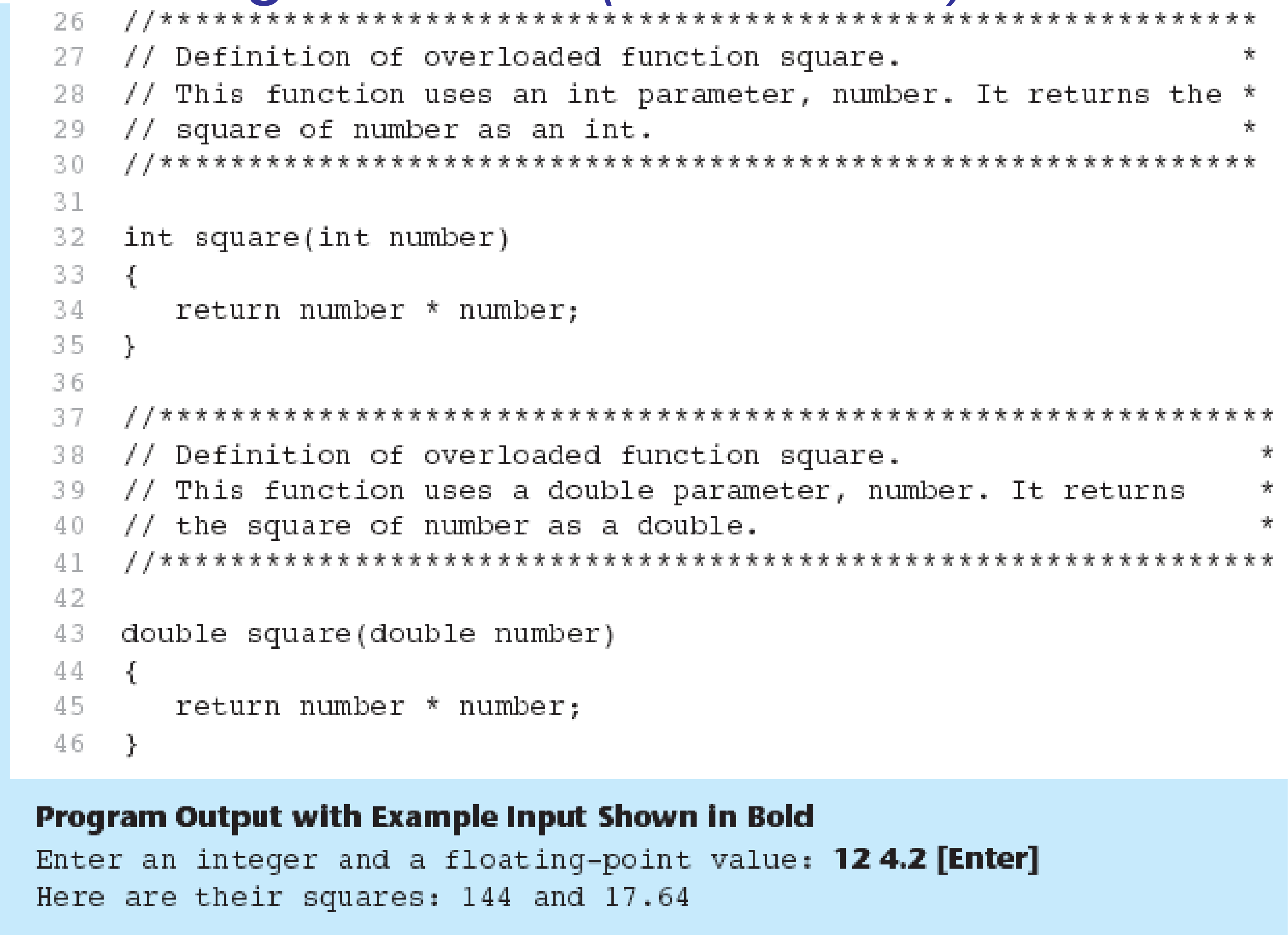
The overloaded functions have different parameter lists

Passing a double

Passing an int

(Program Continues)

Program 6-26 *(Continued)*



```
//**************************************************************
// Definition of overloaded function square.                   *
// This function uses an int parameter, number. It returns the *
// square of number as an int.                                 *
//**************************************************************

int square(int number)
{
   return number * number;
}

//***************************************************************
// Definition of overloaded function square.                    *
// This function uses a double parameter, number. It returns    *
// the square of number as a double.                            *
//***************************************************************

double square(double number)
{
   return number * number;
}
```

Program Output with Example Input Shown in Bold

```
Enter an integer and a floating-point value: 12 4.2 [Enter]
Here are their squares: 144 and 17.64
```

6.15

The `exit()` Function

The `exit()` Function

- Terminates the execution of a program
- Can be called from any function
- Can pass an `int` value to operating system to indicate status of program termination
- Usually used for abnormal termination of program
- Requires `cstdlib` header file

The `exit()` Function

- Example:

  ```
  exit(0);
  ```

- The `cstdlib` header defines two constants that are commonly passed, to indicate success or failure:

  ```
  exit(EXIT_SUCCESS);
  exit(EXIT_FAILURE);
  ```

Program 6-28

```
// This program shows how the exit function causes a program
// to stop executing.
#include <iostream>
#include <cstdlib>   // For exit
using namespace std;

void function();  // Function prototype

int main()
{
   function();
   return 0;
}

//*********************************************************
// This function simply demonstrates that exit can be used  *
// to terminate a program from a function other than main.  *
//*********************************************************

void function()
{
   cout << "This program terminates with the exit function.\n";
   cout << "Bye!\n";
   exit(0);
   cout << "This message will never be displayed\n";
   cout << "because the program has already terminated.\n";
}
```

6.16

Stubs and Drivers

Stubs and Drivers

- Useful for testing and debugging program and function logic and design
- Stub: A dummy function used in place of an actual function
  - Usually displays a message indicating it was called. May also display parameters
- Driver: A function that tests another function by calling it
  - Various arguments are passed and return values are tested

Program 6-28

```
// This program demonstrates stubs and drivers
#include <iostream>

using namespace std;

void function1();  // Function prototype

int main()
{
   function1();
   return 0;
}

// function stub
void function1()
{
   cout << "function1() is called. UNDER CONSTRUCTION.\n";

}
```

Driver of the function1()