Computer Programming
Lecture 6

Hung-Yu Wei
Department of Electrical Engineering
National Taiwan University

function

- What is a function?

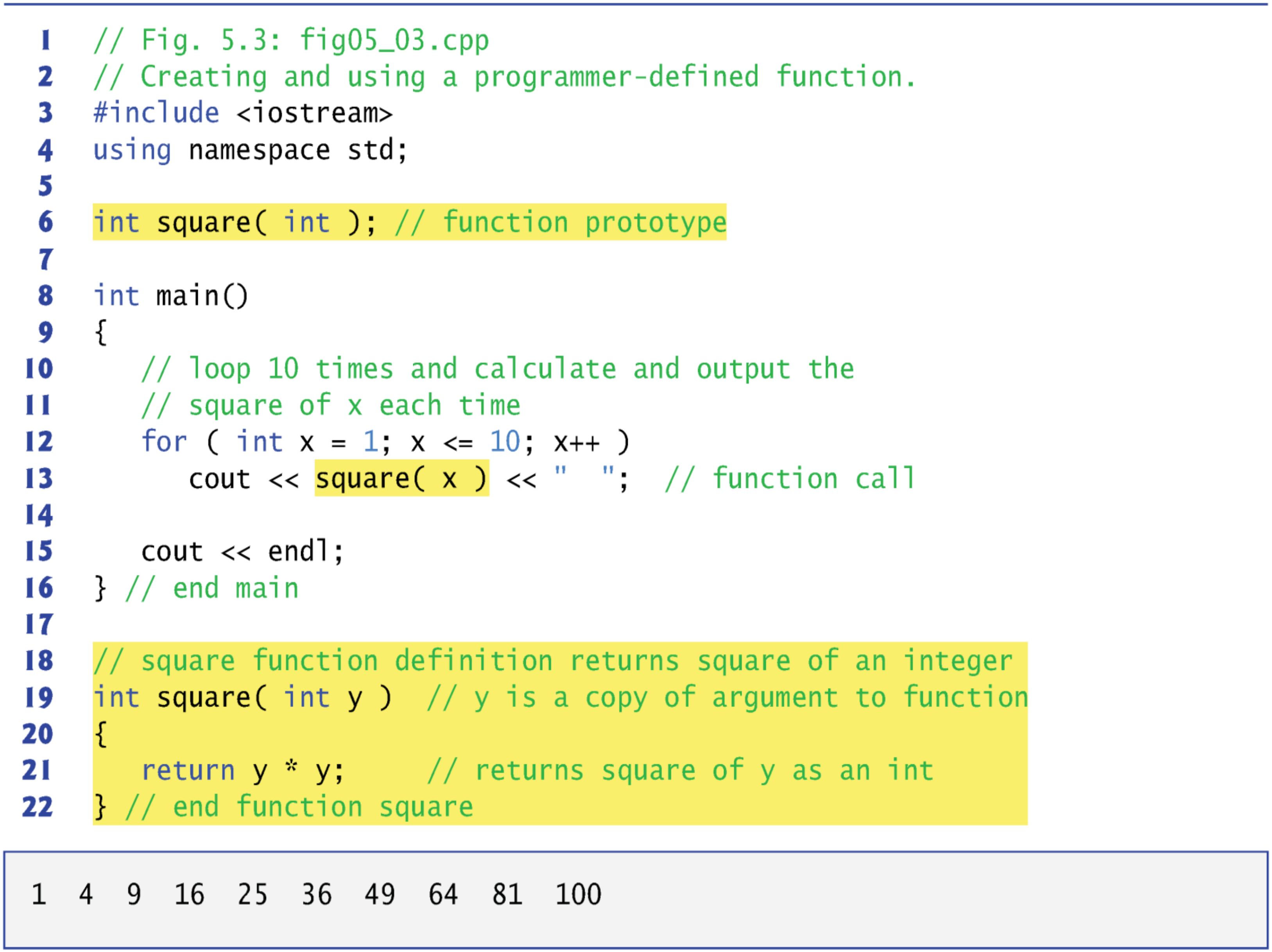
  $y=f(x)$

  $z=g(x1,x2,x3)$

- Function name: f, g
- Input: x, x1, x2,x3
- Output: y, z

- Syntax

```
Return-value-type FunctionName (parameter list)
{

   statements
}
```



```cpp
// Fig. 5.3: fig05_03.cpp
// Creating and using a programmer-defined function.
#include <iostream>
using namespace std;

int square( int ); // function prototype

int main()
{
   // loop 10 times and calculate and output the
   // square of x each time
   for ( int x = 1; x <= 10; x++ )
      cout << square( x ) << "  ";  // function call

   cout << endl;
} // end main

// square function definition returns square of an integer
int square( int y )  // y is a copy of argument to function
{
   return y * y;     // returns square of y as an int
} // end function square
```

```
1   4   9   16   25   36   49   64   81   100
```

Fig. 5.3 | Programmer-defined function `square`.

Function	Description	Example

`ceil( x )`	rounds *x* to the smallest integer not less than *x*	`ceil( 9.2 )` is `10.0`
		`ceil( -9.8 )` is `-9.0`

`cos( x )`	trigonometric cosine of *x* (*x* in radians)	`cos( 0.0 )` is `1.0`

`exp( x )`	exponential function *ex*	`Exp( 1.0 )` is `2.71828`
		`exp( 2.0 )` is `7.38906`

`fabs( x )`	absolute value of *x*	`fabs( 5.1 )` is `5.1`
		`fabs( 0.0 )` is `0.0`
		`fabs( -8.76 )` is `8.76`

`floor( x )`	rounds *x* to the largest integer not greater than *x*	`floor( 9.2 )` is `9.0`
		`floor( -9.8 )` is `-10.0`

`fmod( x, y )`	remainder of *x*/*y* as a floating-point number	`fmod( 2.6, 1.2 )` is `0.2`

`log( x )`	natural logarithm of *x* (base *e*)	`log( 2.718282 )` is `1.0`
		`log( 7.389056 )` is `2.0`

`log10( x )`	logarithm of *x* (base 10)	`log10( 10.0 )` is `1.0`
		`log10( 100.0 )` is `2.0`

`pow( x, y )`	*x* raised to power *y* (*xy*)	`pow( 2, 7 )` is `128`
		`pow( 9, .5 )` is `3`

`sin( x )`	trigonometric sine of *x* (*x* in radians)	`sin( 0.0 )` is `0`
`sqrt( x )`	square root of *x* (where *x* is a nonnegative value)	`sqrt( 9.0 )` is `3.0`
`tan( x )`	trigonometric tangent of *x* (*x* in radians)	`tan( 0.0 )` is `0`

5.5 Function with multiple parameters

- Similar to one-parameter function
- Example
  - Function prototype

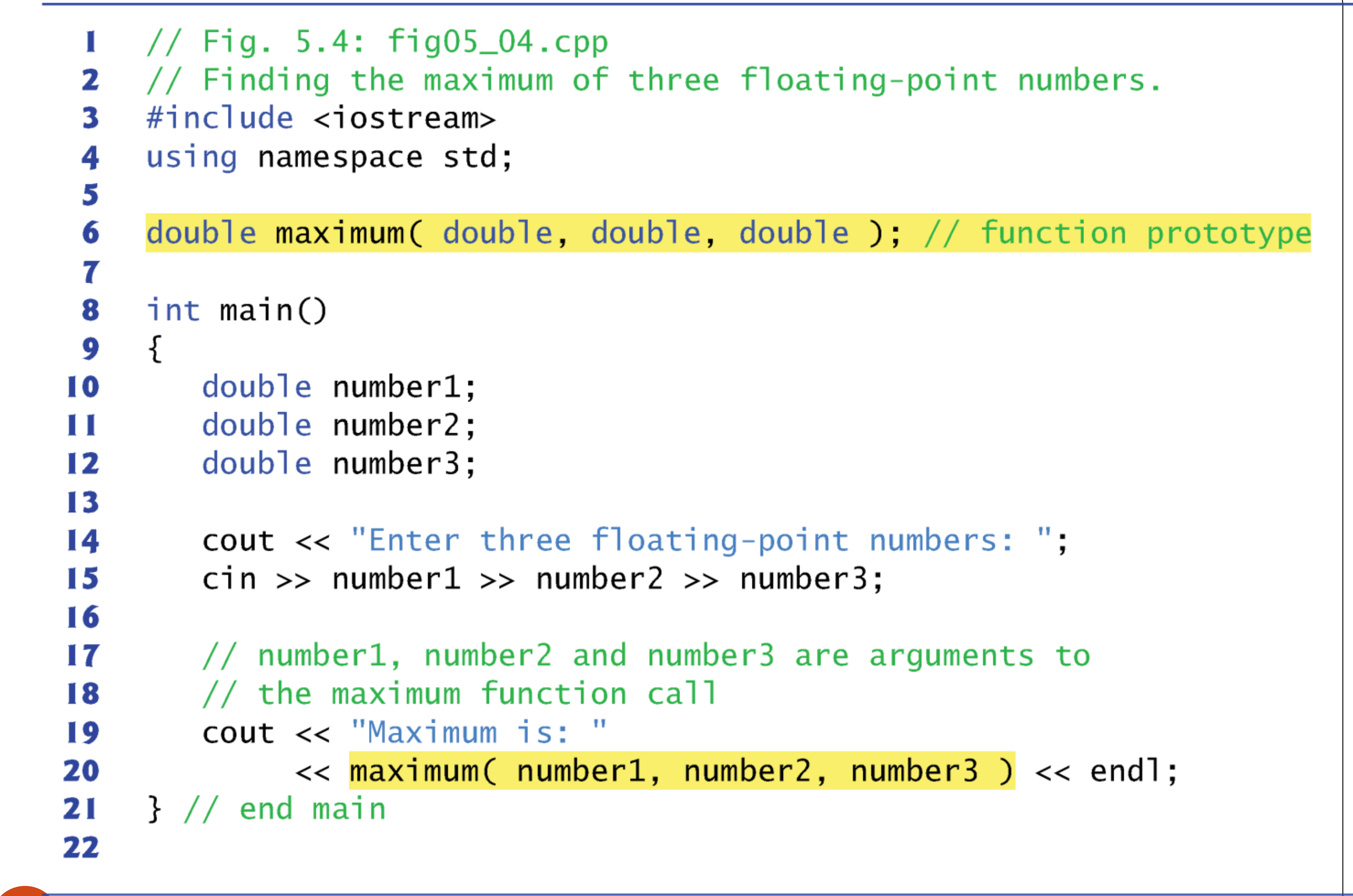
    ```
    int funcName(int, int, int);
    ```
  - Implementation

    ```
    int funcName (int x, int y, int z)
    { statements;
        }
    ```
  - Call (invoke) the function

    ```
    funcName(5, 10, 15);
    ```

Example Program

- What does this program do?
  - Input 3 grade values
  - Determine the maximum grade out of these 3 inputs
- New function
  - int maximum(int, int, int);
  - Determine the maximum value of 3 inputs



```cpp
// Fig. 5.4: fig05_04.cpp
// Finding the maximum of three floating-point numbers.
#include <iostream>
using namespace std;

double maximum( double, double, double ); // function prototype

int main()
{
   double number1;
   double number2;
   double number3;

   cout << "Enter three floating-point numbers: ";
   cin >> number1 >> number2 >> number3;

   // number1, number2 and number3 are arguments to
   // the maximum function call
   cout << "Maximum is: "
        << maximum( number1, number2, number3 ) << endl;
} // end main

```

Fig. 5.4 | Programmer-defined **maximum** function. (Part 1 of 3.)

```cpp
// function maximum definition;
// x, y and z are parameters
double maximum( double x, double y, double z )
{
   double max = x; // assume x is largest

   if ( y > max ) // if y is larger,
      max = y; // assign y to max

   if ( z > max ) // if z is larger,
      max = z; // assign z to max

   return max; // max is largest value
} // end function maximum
```

```
Enter three floating-point numbers: 99.32 37.3 27.1928
Maximum is: 99.32
```

```
Enter three floating-point numbers: 1.1 3.333 2.22
Maximum is: 3.333
```

```
Enter three floating-point numbers: 27.9 14.31 88.99
Maximum is: 88.99
```

Fig. 5.4 | Programmer-defined maximum function. (Part 2 of 2.)

void

- Without Input

```
Return_type functionName()
int MyFunction1()
```

- Without output

```
void functionName (input)
void MyFunction2 (int x)
```

- Without input and without output

```
void MyFuction3 ()
```

Common error: Multiple Input

- Wrong

  ```
  void functionName(double x,y,z)
  ```

- Correct

  ```
  void functionName (double x, double y, double z)
  ```

Leave a function

- Where to do after leaving a function
  - Return to the control point where the function is called (e.g. in main)
- Condition to leave a function
  - Function does not return a value (void)
    - Until everything is executed (when reaching })
    - return;
  - Function with return value
    - return *expression*;

- For example
  - return 0;

5.6 argument coercion

- Function signature
  - Name of the function
  - Types of its arguments
- Argument coercion
  - Forcing argument to the correct data types specified by parameter declaration
  - Example: I can input integers to a function that declares "double" input

Data Types: Promotion hierarchy

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	

Promotion rules

- It is okay to convert "lower data type" to "higher data type"
- Convert "higher data type" to "lower data type" may get incorrect results

C++ standard library

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.

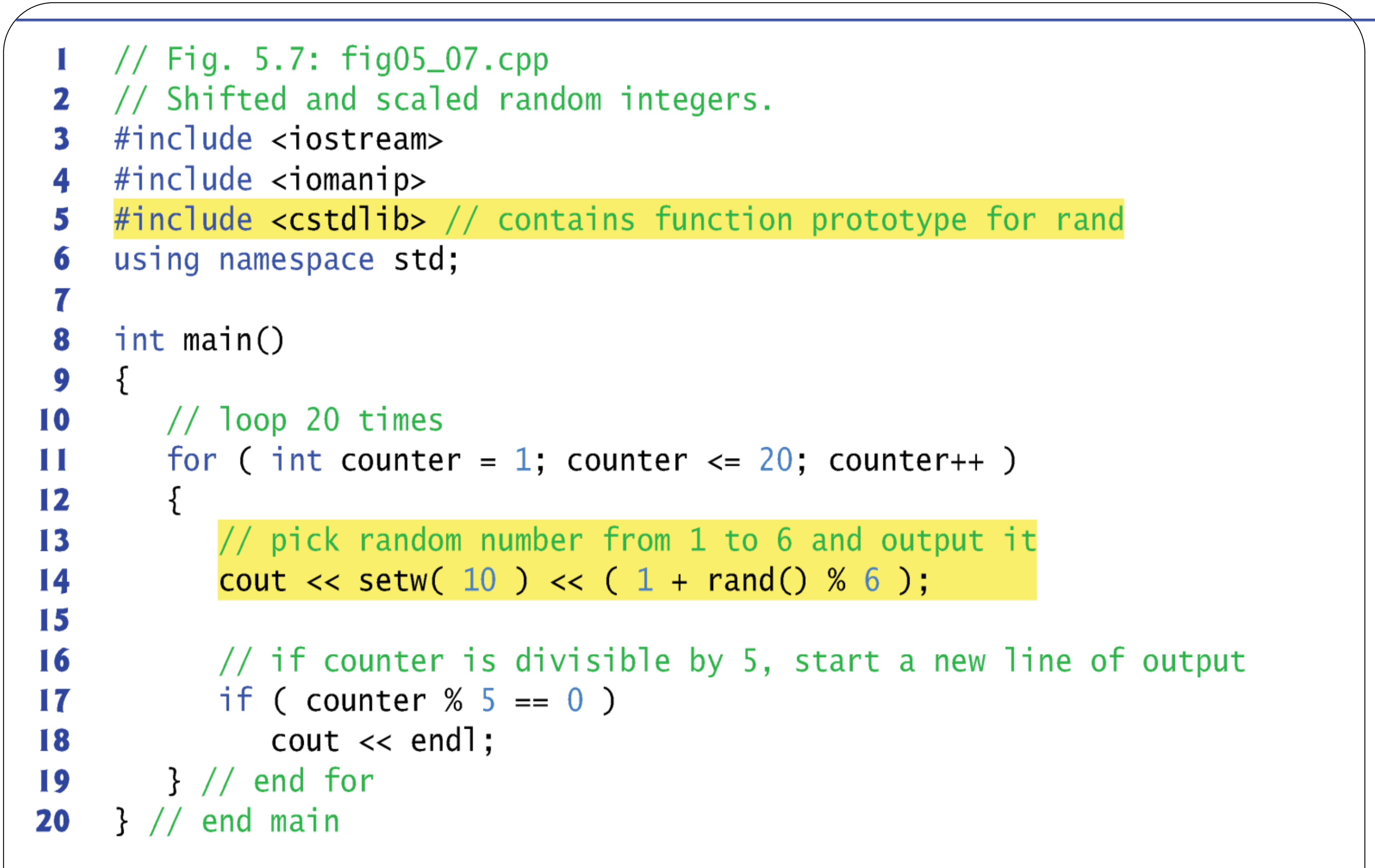
C++ Standard Library header file	Explanation
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header file is discussed in Section 13.8.
`<exception>`, `<stdexcept>`	These header files contain classes that are used for exception handling (discussed 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.

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.

5.8 Random Number Generation

- rand();
  - Generate unsigned integer
  - 0~RAND_MAX (e.g. 32767)
- include <cstdlib>
- Example: randomly generate one of {0,1,2,3,4,5}

  rand() % 6



```cpp
// Fig. 5.7: fig05_07.cpp
// Shifted and scaled random integers.
#include <iostream>
#include <iomanip>
#include <cstdlib> // contains function prototype for rand
using namespace std;

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 );

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

Fig. 5.7 | Shifted, scaled integers produced by `1 + rand() % 6`. (Part 1 of 2.)

Shifting and scaling a random variable

- Example
  - Face = 1+rand() % 6;
- Generalized method

  rv= shiftValue + rand() % scaleFactor
  - shiftValue: the minimal integer
  - scaleFactor: range of desired integers

- Ex: [20 21 22 23]
- 20+rand()%4;
- Ex: [2 4 6 8 10]
  - 2*(1+rand()%5)

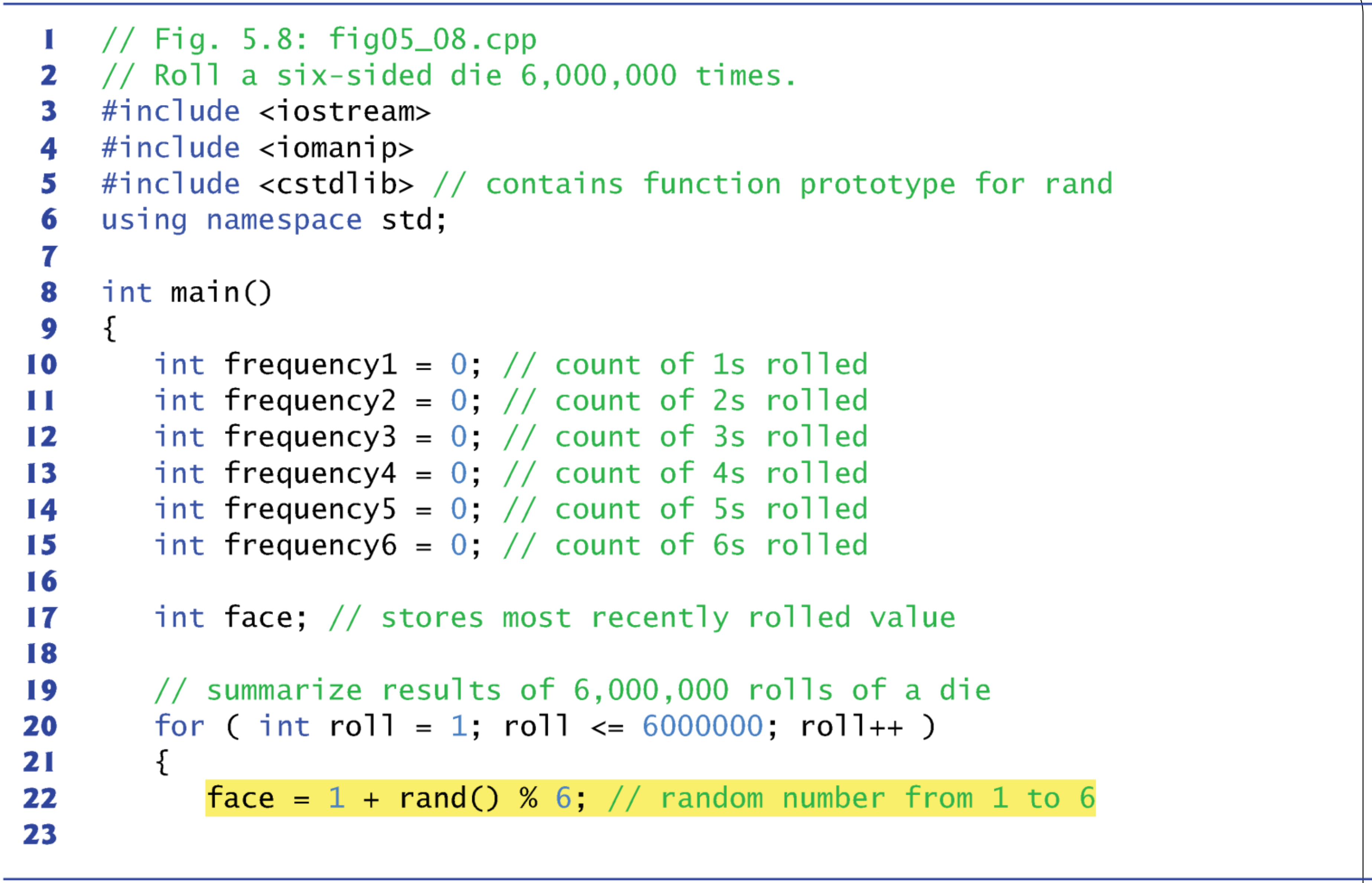
Randomizing random number generator

- rand()
  - Use pseudorandom numbers
- Use "seed" for random number generation
  - srand
    - An integer as the seed
  - Example 1: manually input for seeding

```
cin >> seed;
srand(seed);
```

  - Example 2: read the clock for seeding

```
srand (time(0));
```

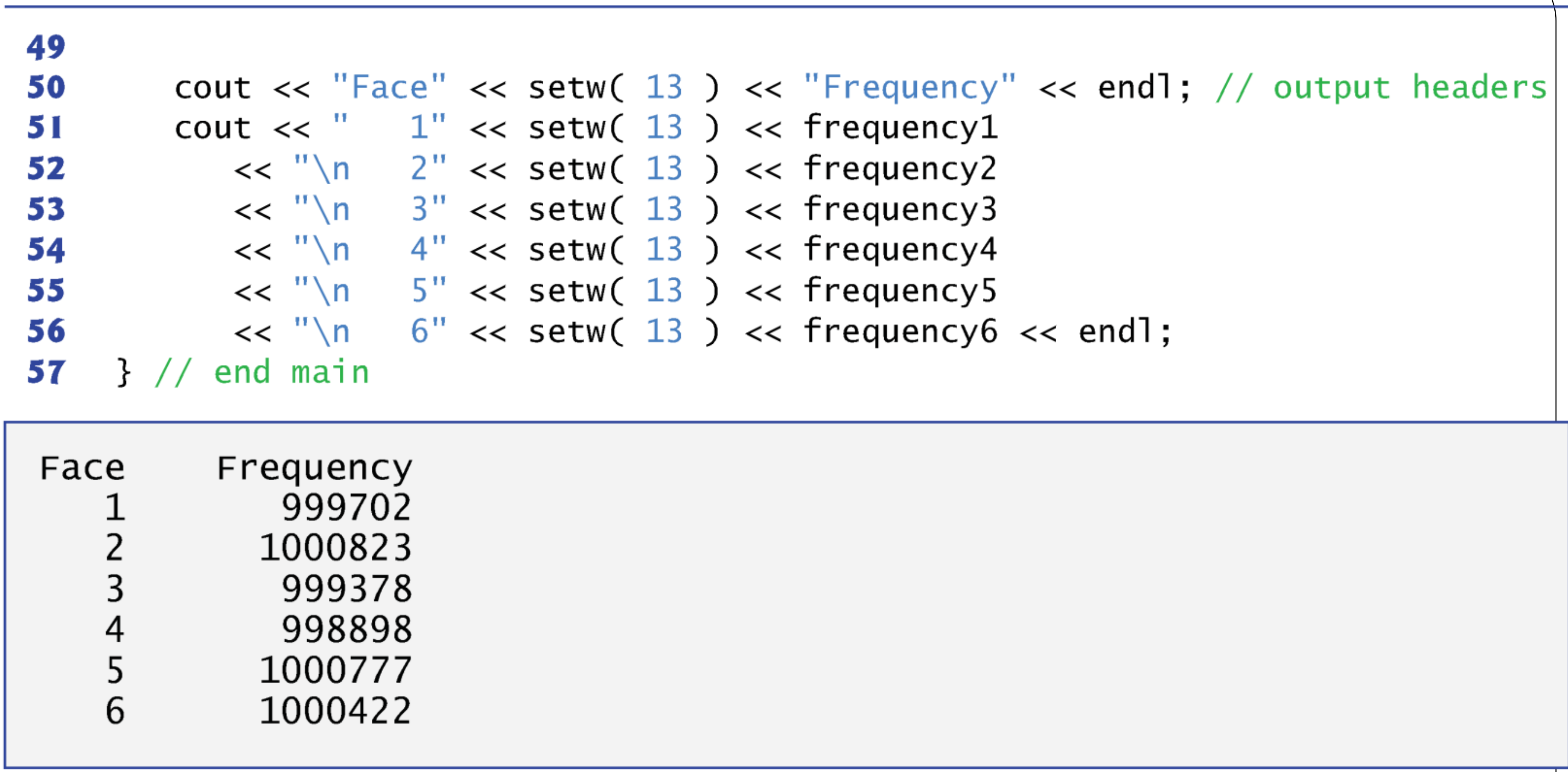


```cpp
// Fig. 5.8: fig05_08.cpp
// Roll a six-sided die 6,000,000 times.
#include <iostream>
#include <iomanip>
#include <cstdlib> // contains function prototype for rand
using namespace std;

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

```

Fig. 5.8 | Rolling a six-sided die 6,000,000 times. (Part 1 of 3.)

```
      // 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
```

Fig. 5.8 | Rolling a six-sided die 6,000,000 times. (Part 2 of 3.)



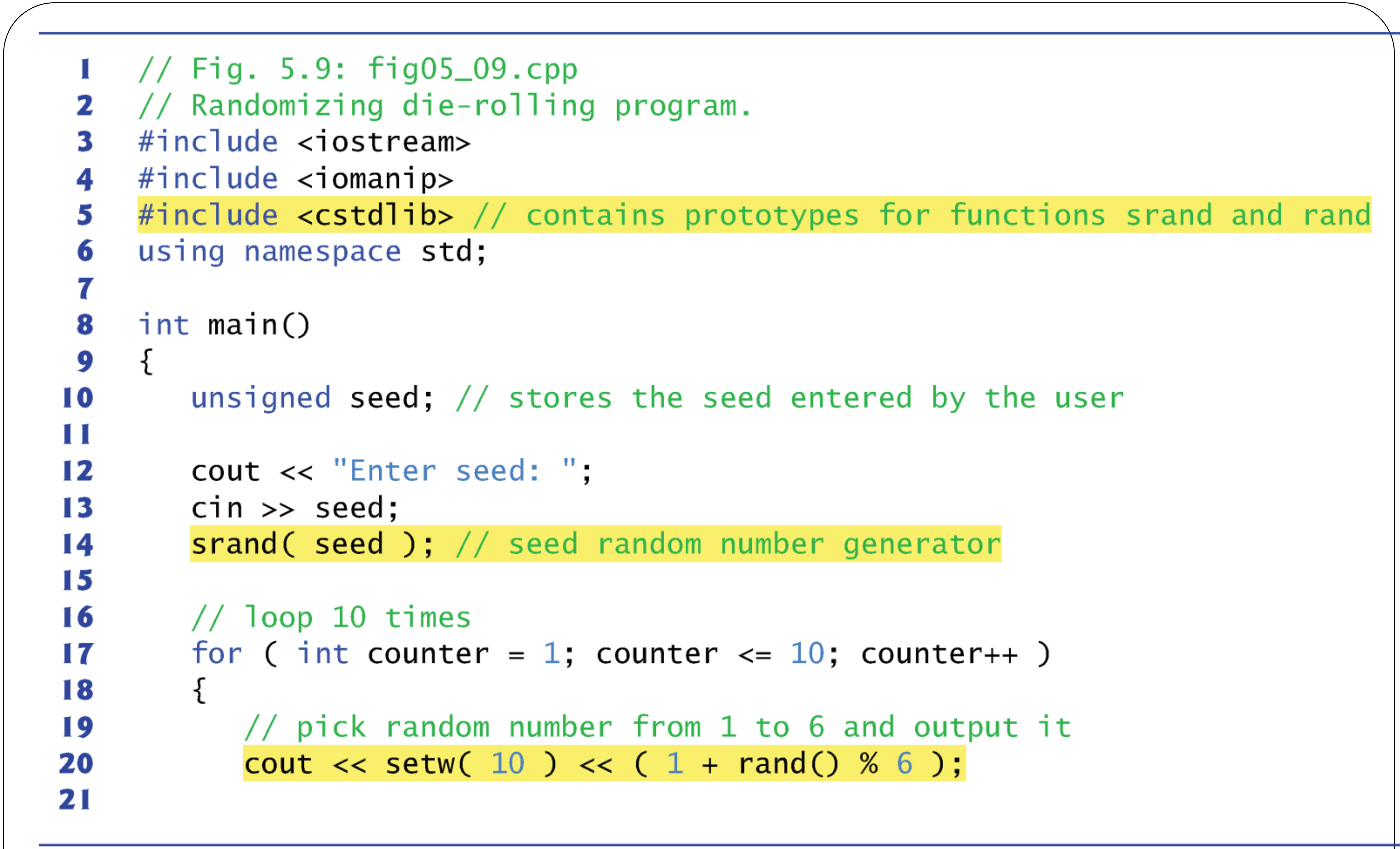
```
   cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
   cout << "   1" << setw( 13 ) << frequency1
      << "\n   2" << setw( 13 ) << frequency2
      << "\n   3" << setw( 13 ) << frequency3
      << "\n   4" << setw( 13 ) << frequency4
      << "\n   5" << setw( 13 ) << frequency5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

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

Fig. 5.8 | Rolling a six-sided die 6,000,000 times. (Part 3 of 3.)

Generating Random Number

- Pseudo-random number
- Seed
- Functions
  - Rand
  - Srand



```
// Fig. 5.9: fig05_09.cpp
// Randomizing die-rolling program.
#include <iostream>
#include <iomanip>
#include <cstdlib> // contains prototypes for functions srand and rand
using namespace std;

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

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

   // 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 );

```

Fig. 5.9 | Randomizing the die-rolling program. (Part 1 of 2.)

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

Fig. 5.9 | Randomizing the die-rolling program. (Part 2 of 2.)

5.15 Function: *pass-by-reference*

- Pass-by-value
  - We have used this for a while
  - Send a copy of the parameter "value" to a function call
    - Send input "value" to a function
    - Return output "value" back
- Pass-by-reference
  - Send the "reference" of parameters to a function call
  - A function can directly access the data and modify it if needed

Pass-by-reference

- &
  - Use for reference
  - Example

    int &x
- Syntax

```
return_type FunctionName (int &x)
 {
     statements;
  }
```

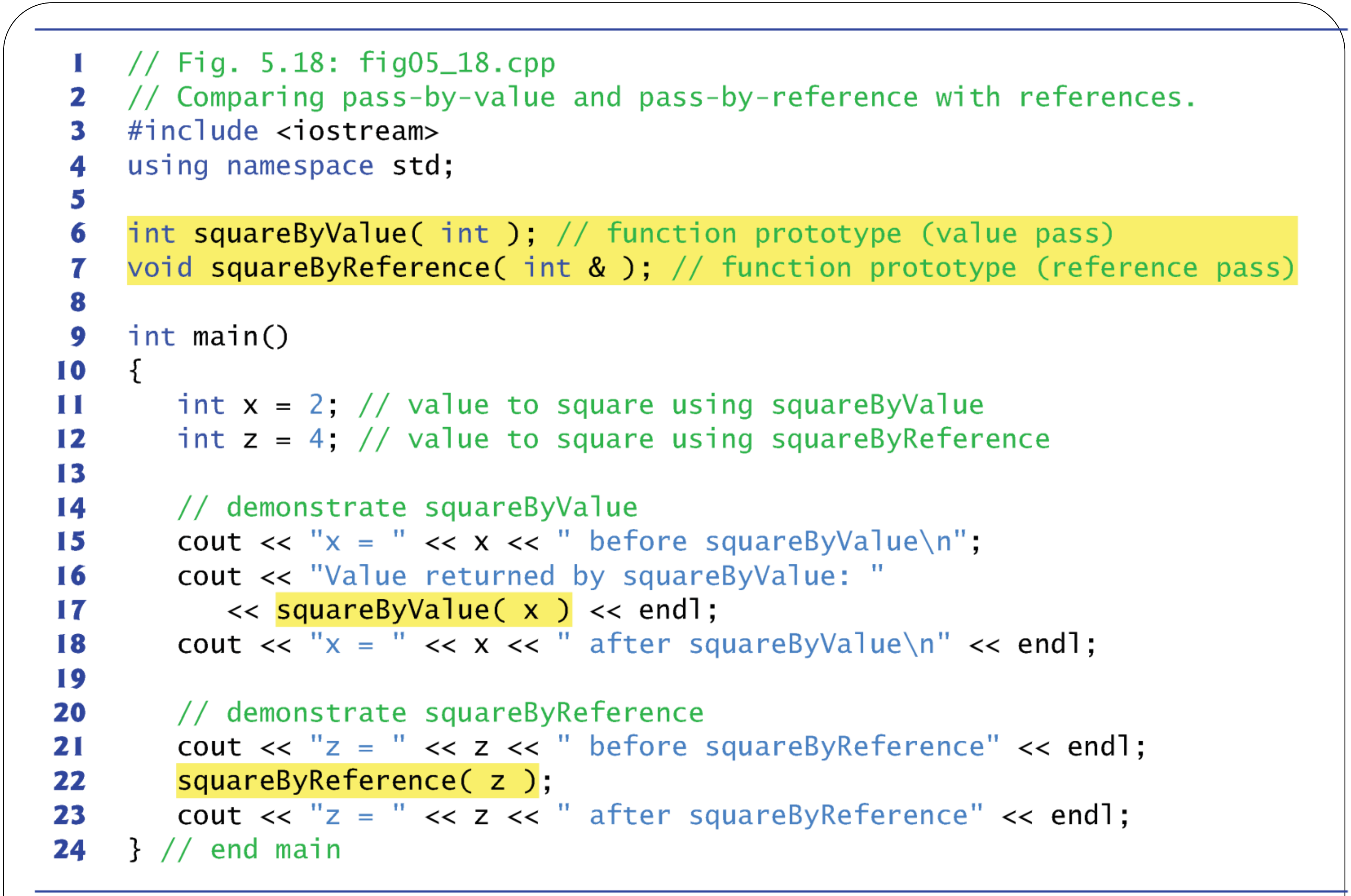
Examples

- Pass-by-value

```
int square (int x)
 {
    return x*=x;
 }
```

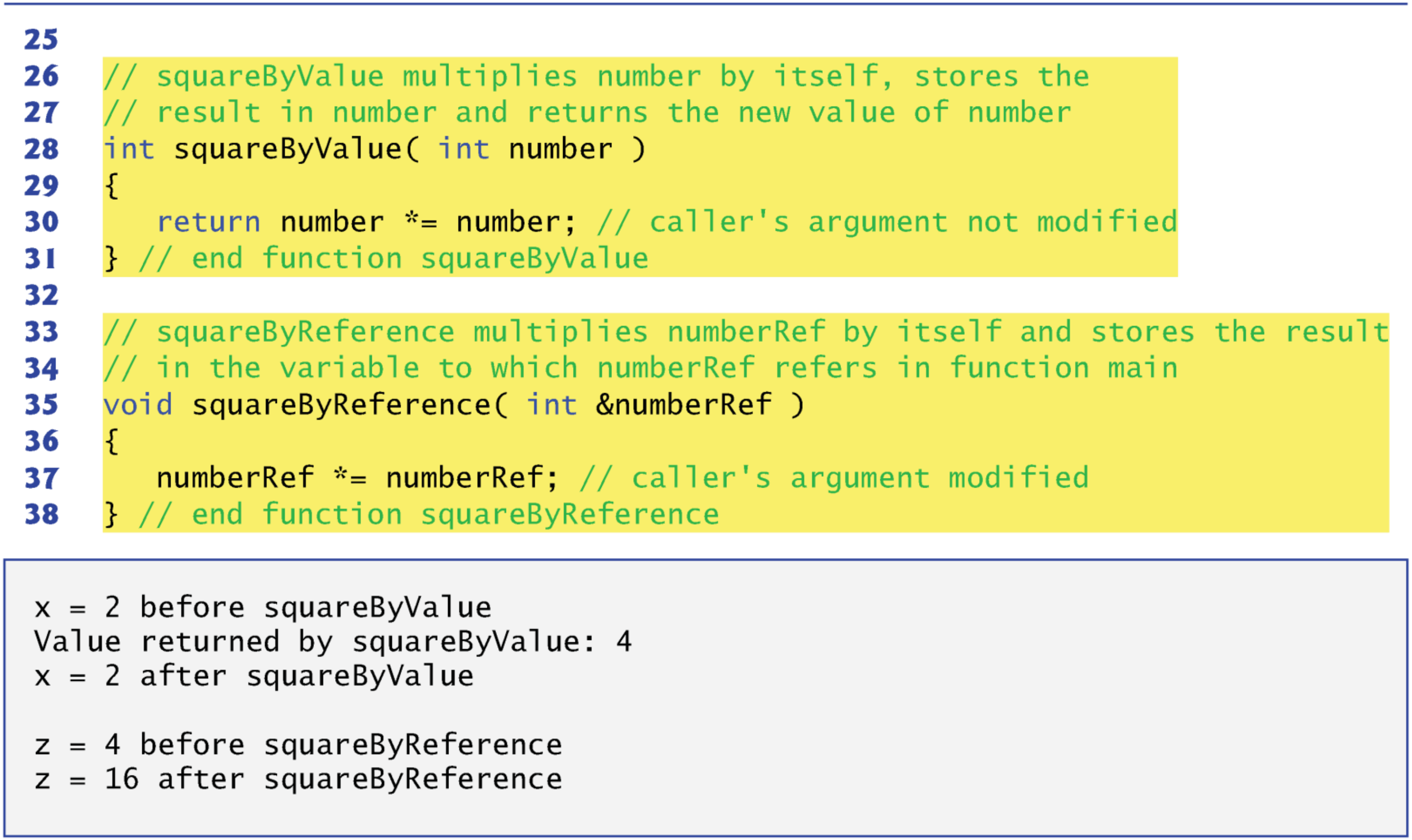
- Pass-by-reference

```
 void square (int &x)
 {
     x*=x;
 }
```



```cpp
// Fig. 5.18: fig05_18.cpp
// Comparing pass-by-value and pass-by-reference with references.
#include <iostream>
using namespace std;

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;
} // end main
```

Fig. 5.18 | Passing arguments by value and by reference. (Part 1 of 2.)



```
// 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
```

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

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

Fig. 5.18 | Passing arguments by value and by reference. (Part 2 of 2.)

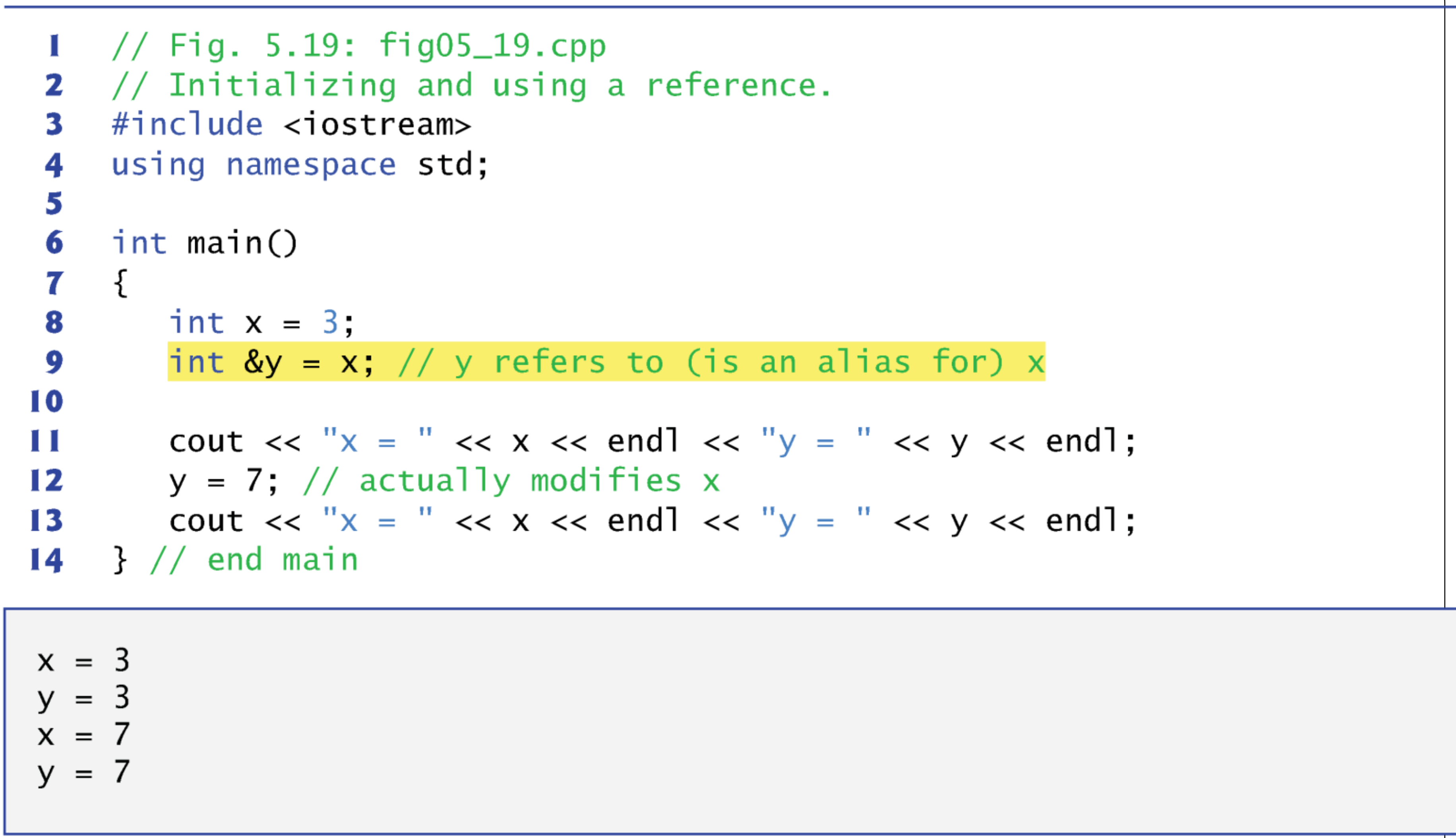
Use pass-by-reference

- Useful when you pass large objects
- When you use a function, know it is pass-by-reference or pass-by-value

References as aliases

- Another name of the original variable
- Reference must be initialized
- Example

```
int x=3;
int &y=x;
```



```cpp
// Fig. 5.19: fig05_19.cpp
// Initializing and using a reference.
#include <iostream>
using namespace std;

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

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

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

Fig. 5.19 | Initializing and using a reference.

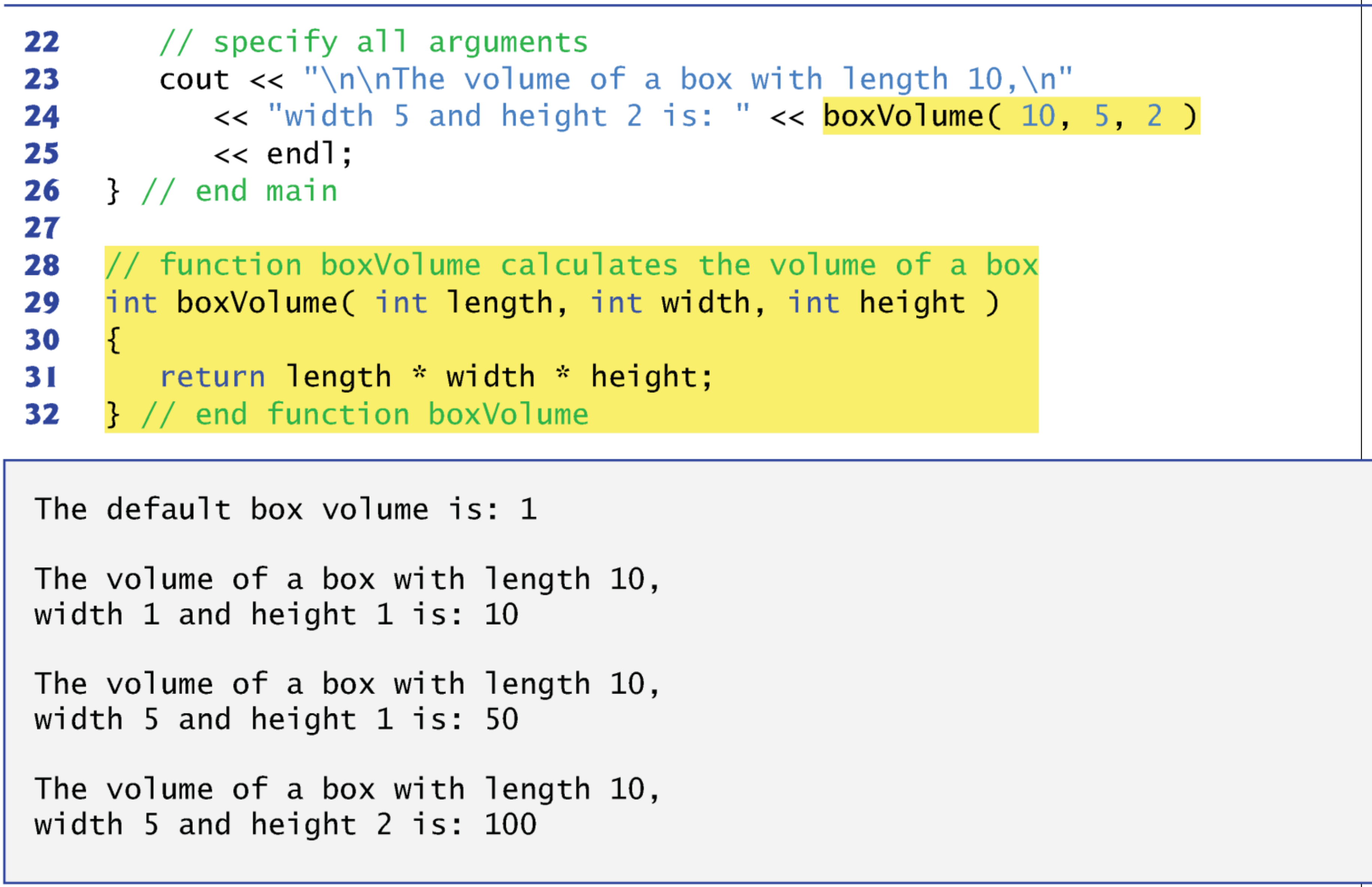
5.16 Default arguments

- A default value to be passed to a parameter
  - Default arguments
  - Unless otherwise specify, the parameter will use the default argument value
- Example: function prototype that specifies default arguments

  ```
  int boxVolume (int L=1, int W=1, int H=1)
  ```

```cpp
// Fig. 5.21: fig05_21.cpp
// Using default arguments.
#include <iostream>
using namespace std;

// 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 );

```

Fig. 5.21 | Default arguments to a function. (Part 1 of 2.)



```cpp
   // 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;
} // end main

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

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

5.18 function overloading

- Several functions have the same name
- Functions have different sets of parameters
  - Type difference
  - Order difference
  - Number difference
- Usually, we use overloading functions to perform similar tasks

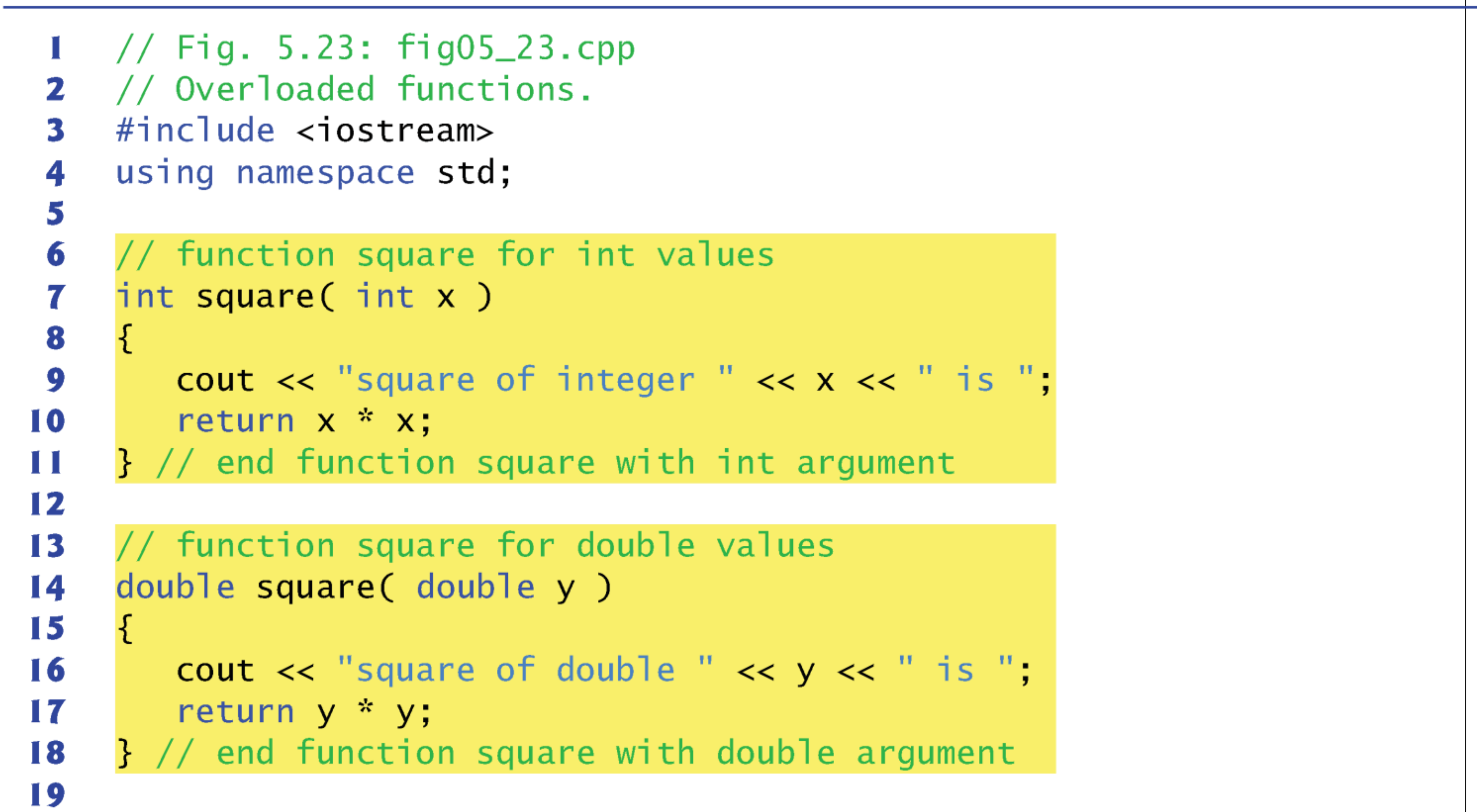
Example: overload functions

- Function with int values

```
int square (int x)
{ return x*x; }
```

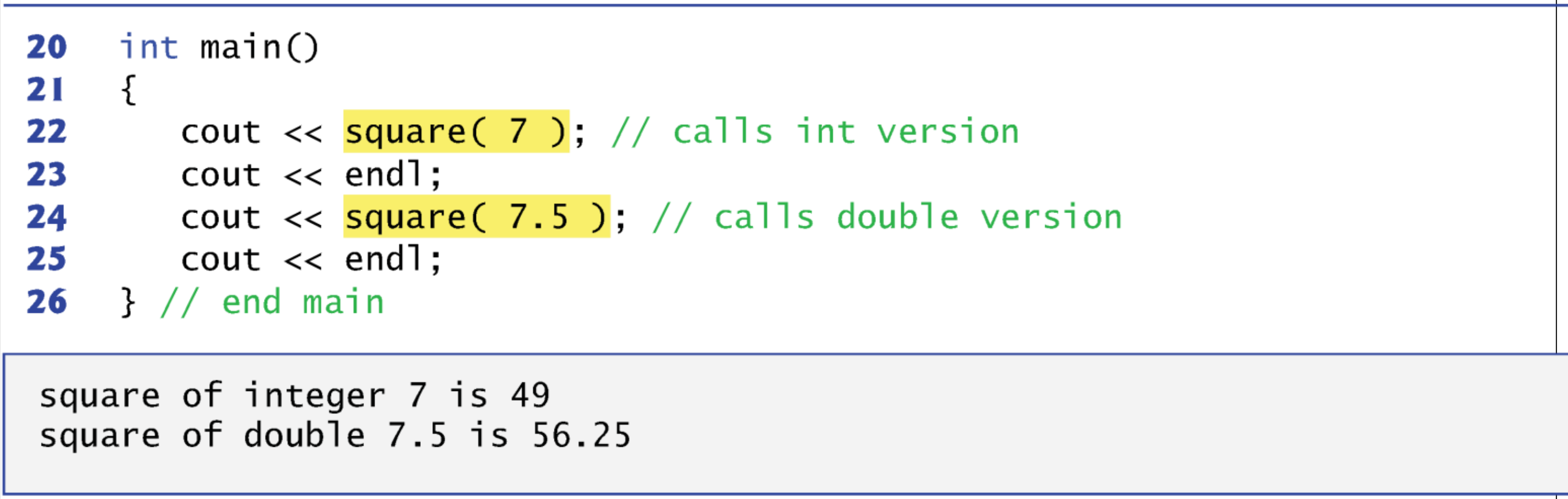
- Function with double values

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



```cpp
// Fig. 5.23: fig05_23.cpp
// Overloaded functions.
#include <iostream>
using namespace std;

// 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

```

Fig. 5.23 | Overloaded `square` functions. (Part 1 of 2.)



```cpp
int main()
{
   cout << square( 7 ); // calls int version
   cout << endl;
   cout << square( 7.5 ); // calls double version
   cout << endl;
} // end main
```

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

Fig. 5.23 | Overloaded `square` functions. (Part 2 of 2.)

Compiling overloaded funcitons

- Compiler encodes each function with the parameters (number, type, order)
  - Name mangling (Name decoration)
- Type-safe linkage
  - Make sure the proper overloaded function is called
- Examples of overloaded function
  - <<
  - >>
  - Input/output data with different data types