高级语言程序设计
High-level Language Programming

Lecture 8 Functions

Yitian Shao (shaoyitian@hit.edu.cn)
School of Computer Science and Technology

Functions

Course Overview

- Function
- Function arguments
- Parsing arguments
- Mathematical functions
- Function overloading
- Recursion
- The scope of a variable

8.1 Function

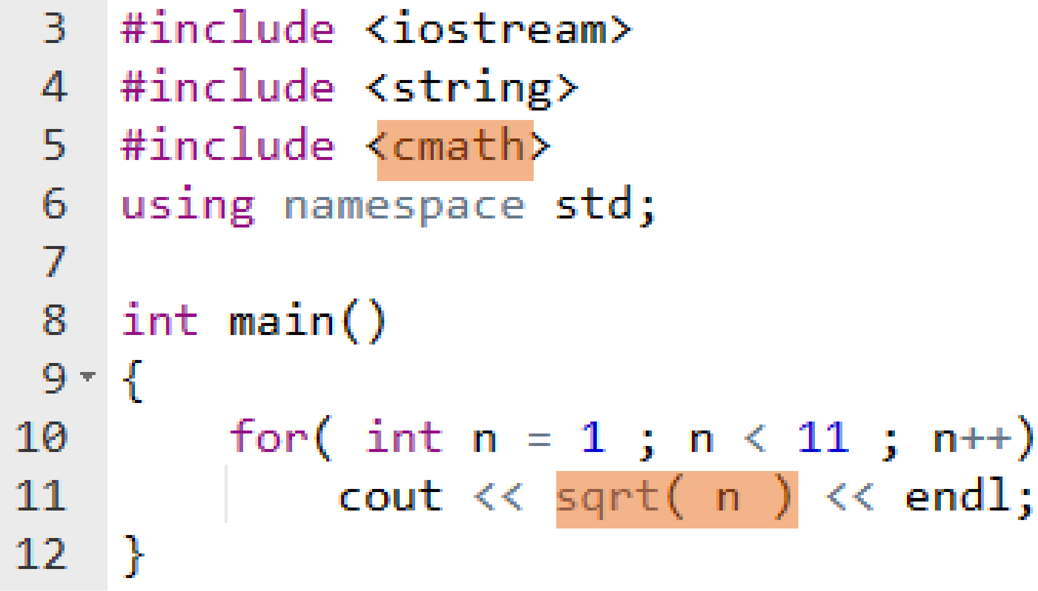
A **function** is **a block of statements** called by name to carry out a specific task

- In order to reduce the complexity of programs, they have to be broken into smaller, less complex parts.

- **Functions** and **classes** are the building blocks of a C++ program.

8.1 Function

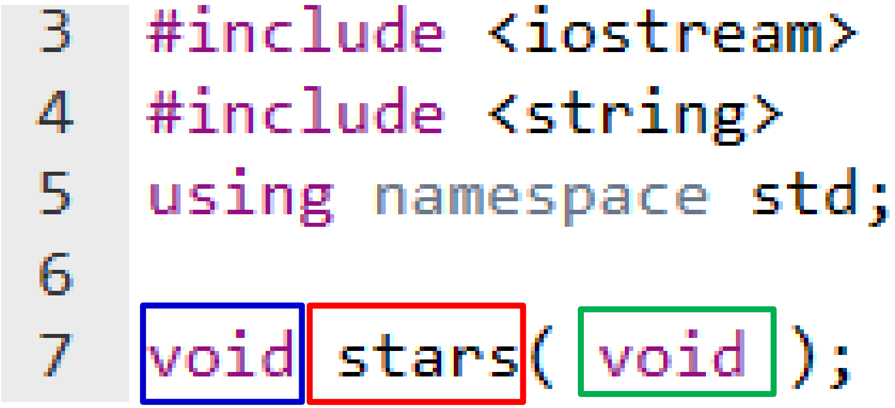
- Functions in the standard library: built-in, pre-written in C++



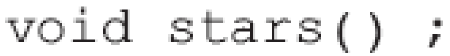
```
 3  #include <iostream>
 4  #include <string>
 5  #include <cmath>
 6  using namespace std;
 7
 8  int main()
 9  {
10      for( int n = 1 ; n < 11 ; n++)
11          cout << sqrt( n ) << endl;
12  }
```

Line 11 calls the function *sqrt()* to calculate the square root of the value in the variable n

8.1 Function



```
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  void stars( void );
```



```
void stars() ;
```

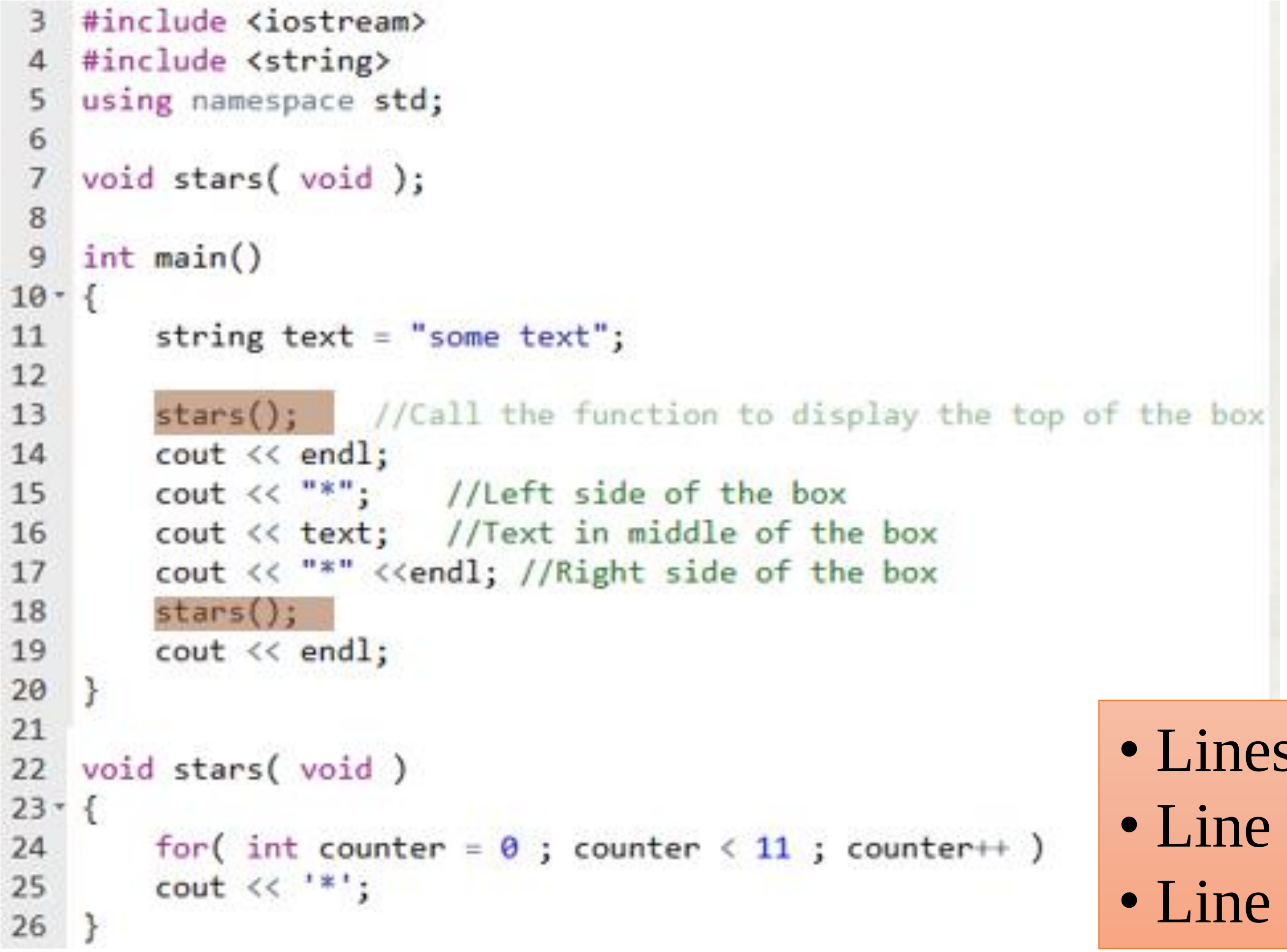
Return nothing

Receive nothing

- Like variables, functions must be declared before they are used.
- Line 7 declares stars (identifier) to be a function
- The first void on line 7 declares the type of the function stars()
- The second void informs the compiler that 'stars' will **not receive any data** from the calling program. <u>The second void is optional.</u>

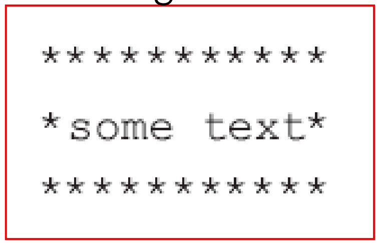
8.1 Function

- Program Example



```
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   void stars( void );
8
9   int main()
10  {
11      string text = "some text";
12
13      stars();    //Call the function to display the top of the box
14      cout << endl;
15      cout << "*";     //Left side of the box
16      cout << text;    //Text in middle of the box
17      cout << "*" <<endl; //Right side of the box
18      stars();
19      cout << endl;
20  }
21
22  void stars( void )
23  {
24      for( int counter = 0 ; counter < 11 ; counter++ )
25      cout << '*';
26  }
```

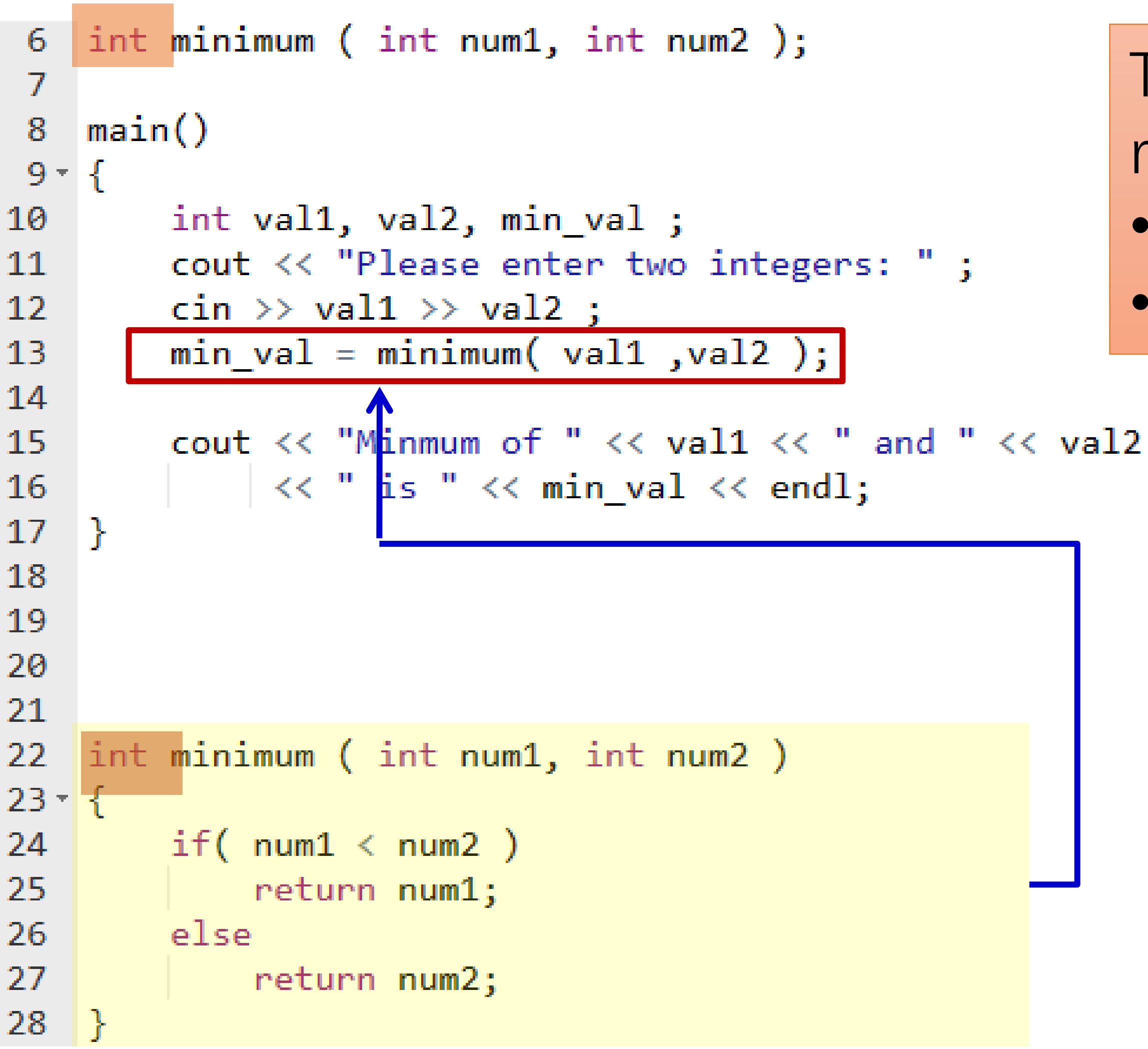
Running Results



```
***********
*some text*
***********
```

- Lines 22 to 26 define the function
- Line 22: function header
- Line 24- 25: function body

8.1 Function



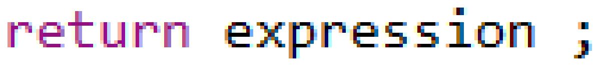
```
int minimum ( int num1, int num2 );

main()
{
    int val1, val2, min_val ;
    cout << "Please enter two integers: " ;
    cin >> val1 >> val2 ;
    min_val = minimum( val1 ,val2 );

    cout << "Minmum of " << val1 << " and " << val2
         << " is " << min_val << endl;
}




int minimum ( int num1, int num2 )
{
    if( num1 < num2 )
        return num1;
    else
        return num2;
}
```

To define and call a function with return value, you should notice:

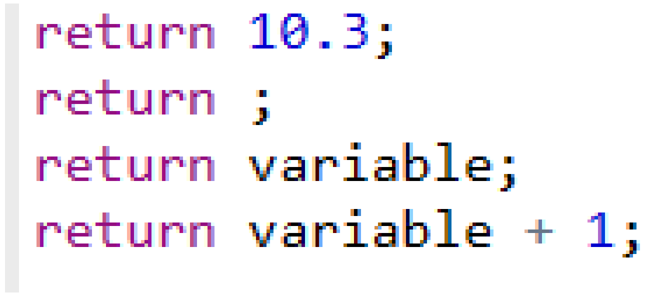
- function prototype
- function header

8.1 Function

- The general format of the return statement:
  - Examples:

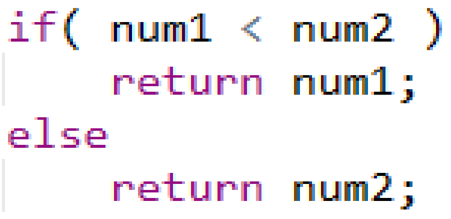


```
return expression ;
```

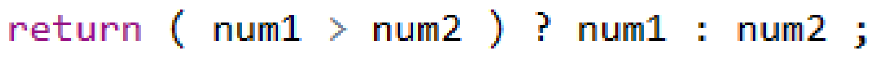


```
return 10.3;
return ;
return variable;
return variable + 1;
```

  - These two blocks are equivalent:



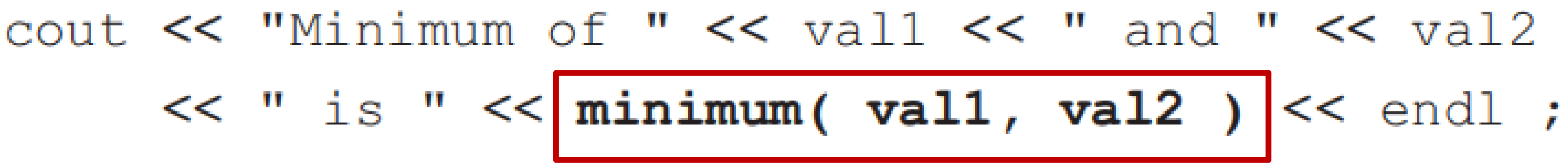
```
if( num1 < num2 )
    return num1;
else
    return num2;
```



```
return ( num1 > num2 ) ? num1 : num2 ;
```

8.1 Function

- A function call can be used anywhere in a program where a variable can be used

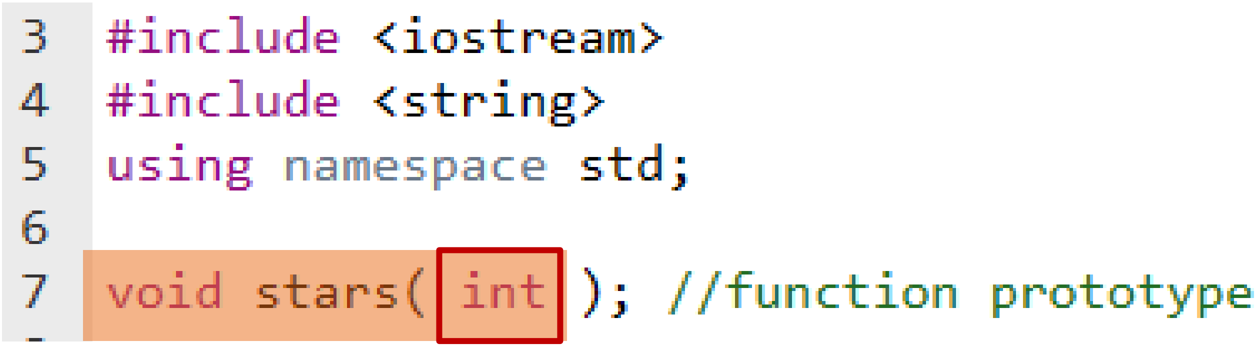


```
cout << "Minimum of " << val1 << " and " << val2
     << " is " << minimum( val1, val2 ) << endl ;
```

8.2 Function arguments

- Function star() can display 11 asterisks, then how to display a variable number of asterisks?
- New function stars():

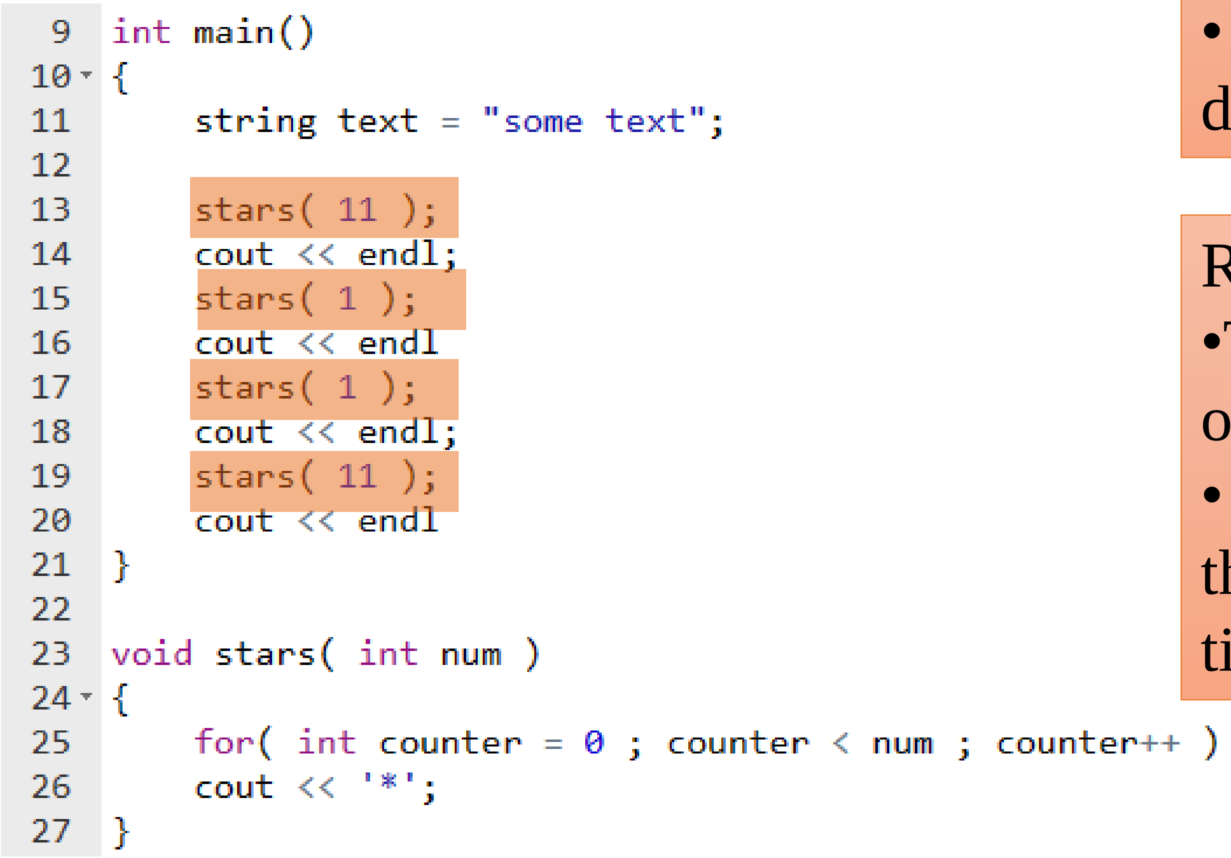
  take a value passed to it and display the number of asterisks specified in that value.



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

void stars( int ); //function prototype
```

8.2 Function arguments

- Program Example



```
 9  int main()
10  {
11      string text = "some text";
12  
13      stars( 11 );
14      cout << endl;
15      stars( 1 );
16      cout << endl
17      stars( 1 );
18      cout << endl;
19      stars( 11 );
20      cout << endl
21  }
22  
23  void stars( int num )
24  {
25      for( int counter = 0 ; counter < num ; counter++ )
26      cout << '*';
27  }
```

Remark

- This number is called an ***argument***
- Received by the parameter *num* declared as an integer in line 23

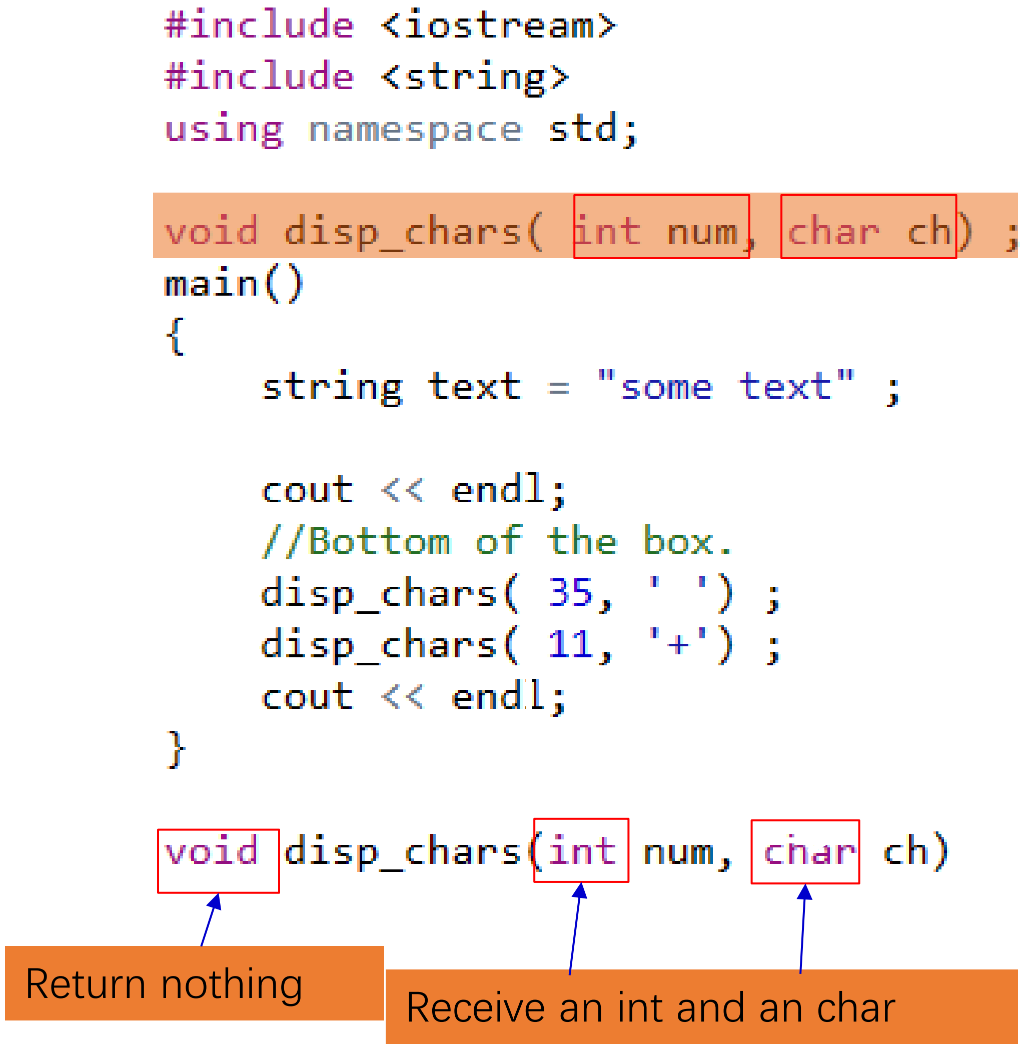
Remark

- The parameters of a function are known only within the function.
- Line 25 of the function stars() now uses the variable num to decide how many times * is displayed.

8.2 Function arguments

- A new program specifies:
    - the number
    - character to display

8.2 Function arguments



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

void disp_chars( int num, char ch) ;
main()
{
    string text = "some text" ;

    cout << endl;
    //Bottom of the box.
    disp_chars( 35, ' ') ;
    disp_chars( 11, '+') ;
    cout << endl;
}

void disp_chars(int num, char ch)
```

Return nothing

Receive an int and an char

disp_chars() uses two parameters:

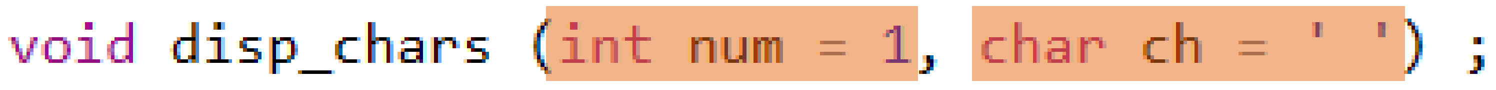
- *num* (the number of times to display a character)
- *ch* (the character to display)

8.2 Function arguments

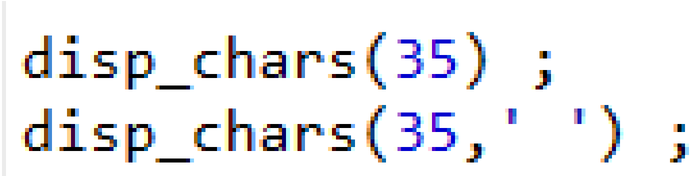
- The variable names used in the prototype are often the same as those used for the parameters in the function header.
- It is good practice to **leave comments** after the function prototype to **describe** the function and its parameters.
- The prototype and the accompanying comments are known as the *function interface.*

8.2 Function arguments

- A function parameter that is not passed a value can be assigned a *default value.*



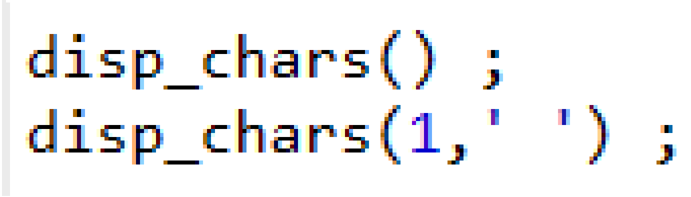
```
void disp_chars (int num = 1, char ch = ' ') ;
```



```
disp_chars(35) ;
disp_chars(35,' ') ;
```

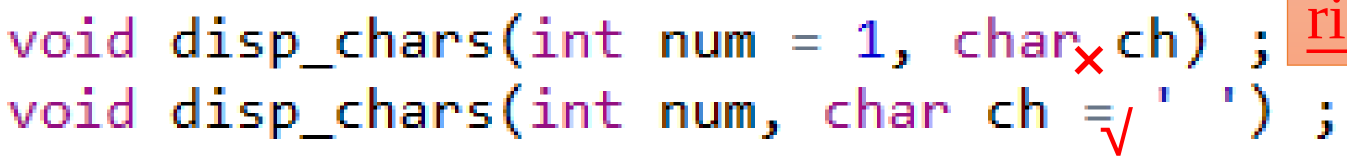
•The second argument is omitted.

•These two are equivalent.



```
disp_chars() ;
disp_chars(1,' ') ;
```

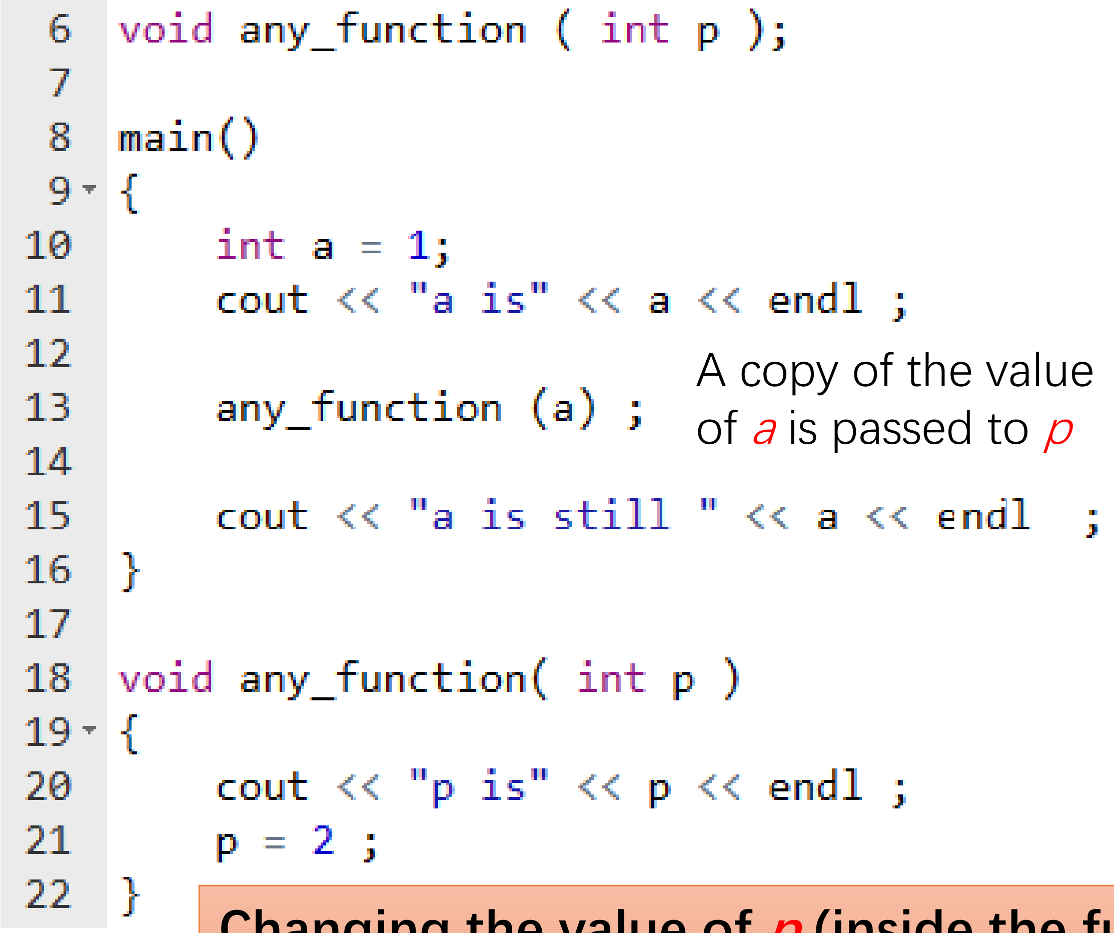
•If a parameter is provided with a default value, all the parameters to its right must also have a default value.



```
void disp_chars(int num = 1, char ✕ ch) ;
void disp_chars(int num, char ch = √ ' ') ;
```

8.3 Passing arguments

- *Passing by value*
  - A copy of the argument values is passed to the function parameters
  - The value of the argument cannot be changed within the function

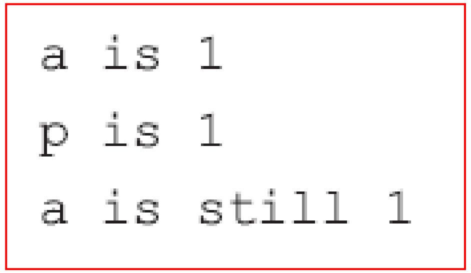


```
void any_function ( int p );

main()
{
    int a = 1;
    cout << "a is" << a << endl ;

    any_function (a) ;

    cout << "a is still " << a << endl  ;
}

void any_function( int p )
{
    cout << "p is" << p << endl ;
    p = 2 ;
}
```

A copy of the value of *a* is passed to *p*

Changing the value of *p* (inside the function) has no effect on *a*

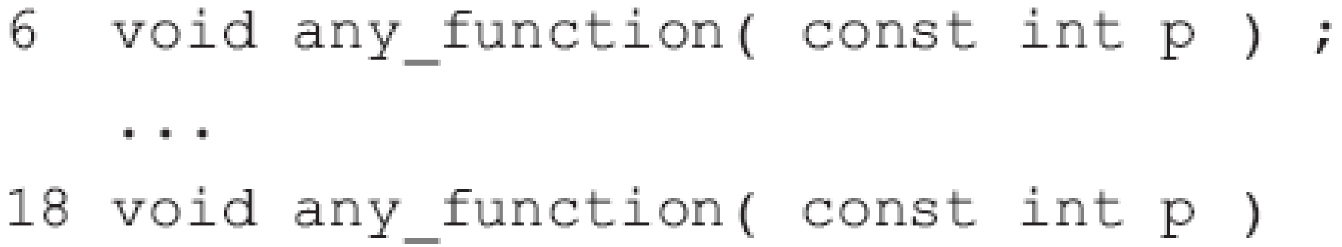
Running Results



```
a is 1
p is 1
a is still 1
```

8.3 Passing arguments

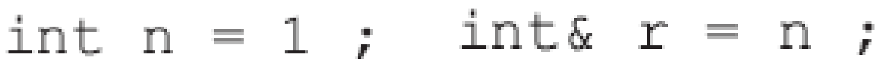
- The value of the parameter can be prevented from any changes within a function by making it a constant.
  - To do this, place the keyword ***const*** before the parameter in the function prototype and function header .



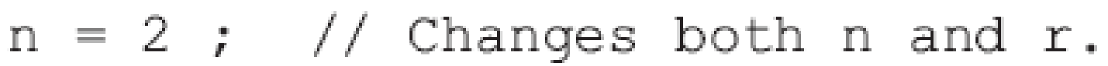
```
void any_function( const int p ) ;
 ...
void any_function( const int p )
```

8.3 Passing arguments

- ***Passing by reference****:* A ***reference*** is a synonym or an alias for an existing variable.
  - A reference to a variable is defined by **adding & after the variable’s data type**.



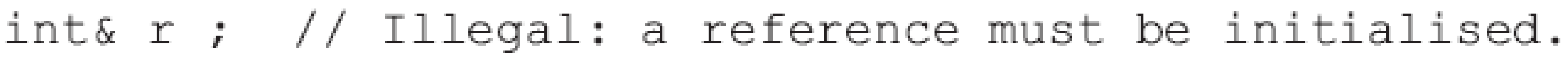
```
int n = 1 ;   int& r = n ;
```



```
n = 2 ;   // Changes both n and r.
```

- *r* is a reference to *n*
- ***n* and *r* both refer to the same value**
- *r* is not a copy of *n*, but is merely another name for *n*
- A change to *n* will also result in a change to *r*

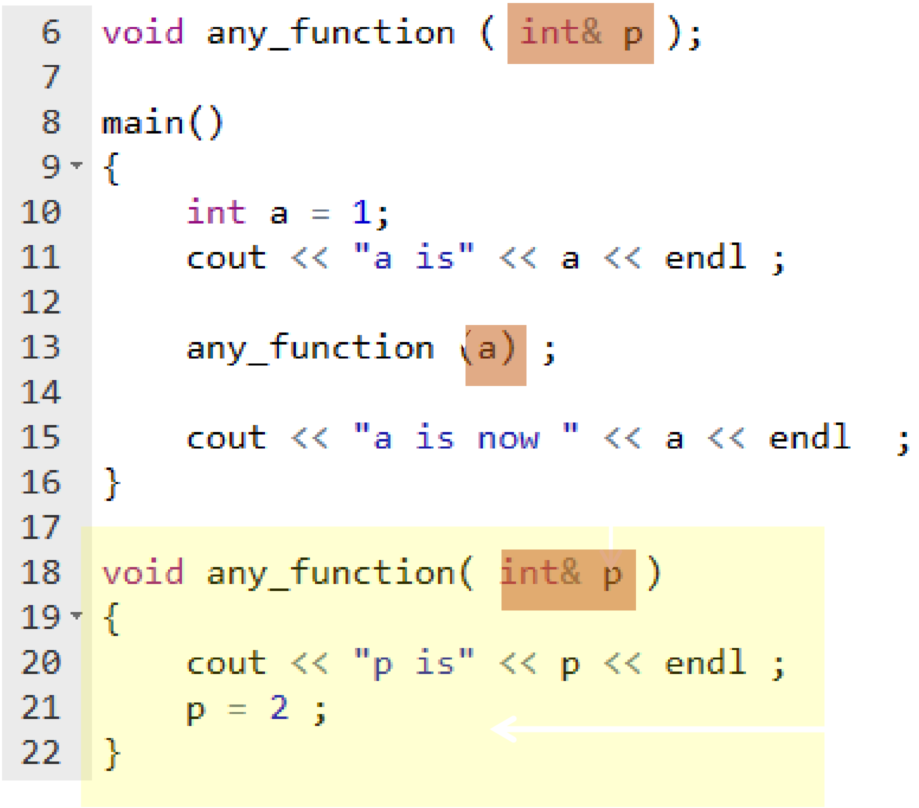
  - A reference must always be initialized when it is defined



```
int& r ;   // Illegal: a reference must be initialised.
```

8.3 Passing arguments

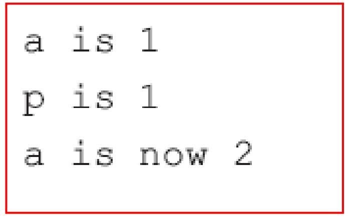
- Program Example



```
void any_function ( int& p );

main()
{
    int a = 1;
    cout << "a is" << a << endl ;

    any_function (a) ;

    cout << "a is now " << a << endl   ;
}

void any_function( int& p )
{
    cout << "p is" << p << endl ;
    p = 2 ;
}
```

a and *p* refer to the same storage location

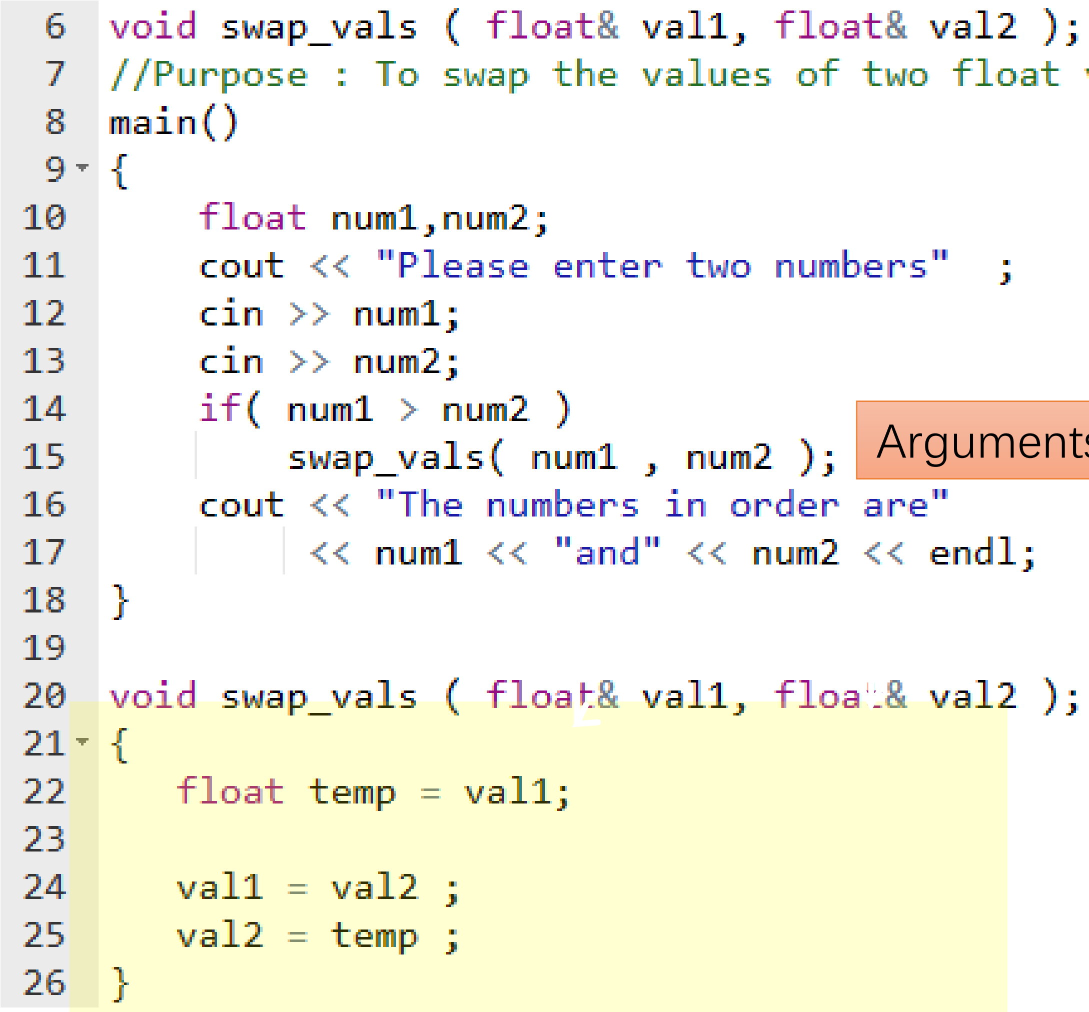
Running Results



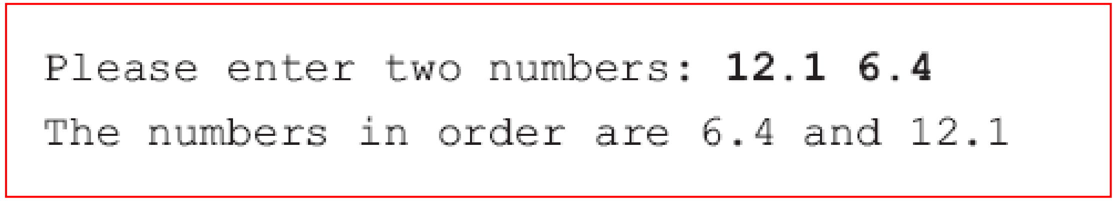
```
a is 1
p is 1
a is now 2
```

Changing the value of p also changes the value of a

8.3 Passing arguments



```
void swap_vals ( float& val1, float& val2 );
//Purpose : To swap the values of two float variables
main()
{
    float num1,num2;
    cout << "Please enter two numbers"  ;
    cin >> num1;
    cin >> num2;
    if( num1 > num2 )
        swap_vals( num1 , num2 );
    cout << "The numbers in order are"
         << num1 << "and" << num2 << endl;
}

void swap_vals ( float& val1, float& val2 );
{
   float temp = val1;

   val1 = val2 ;
   val2 = temp ;
}
```



```
Please enter two numbers: 12.1 6.4
The numbers in order are 6.4 and 12.1
```

Arguments passed by reference

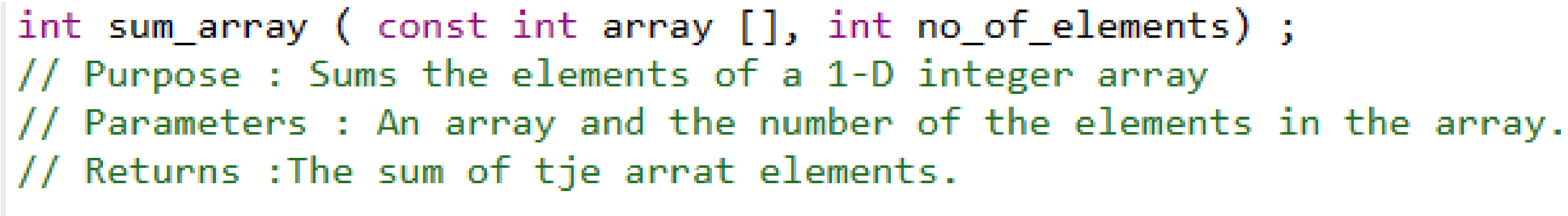
local variable

•The variable temp is a *local variable to the function* swap_vals().

•Local variables are known only within the function where they are defined.

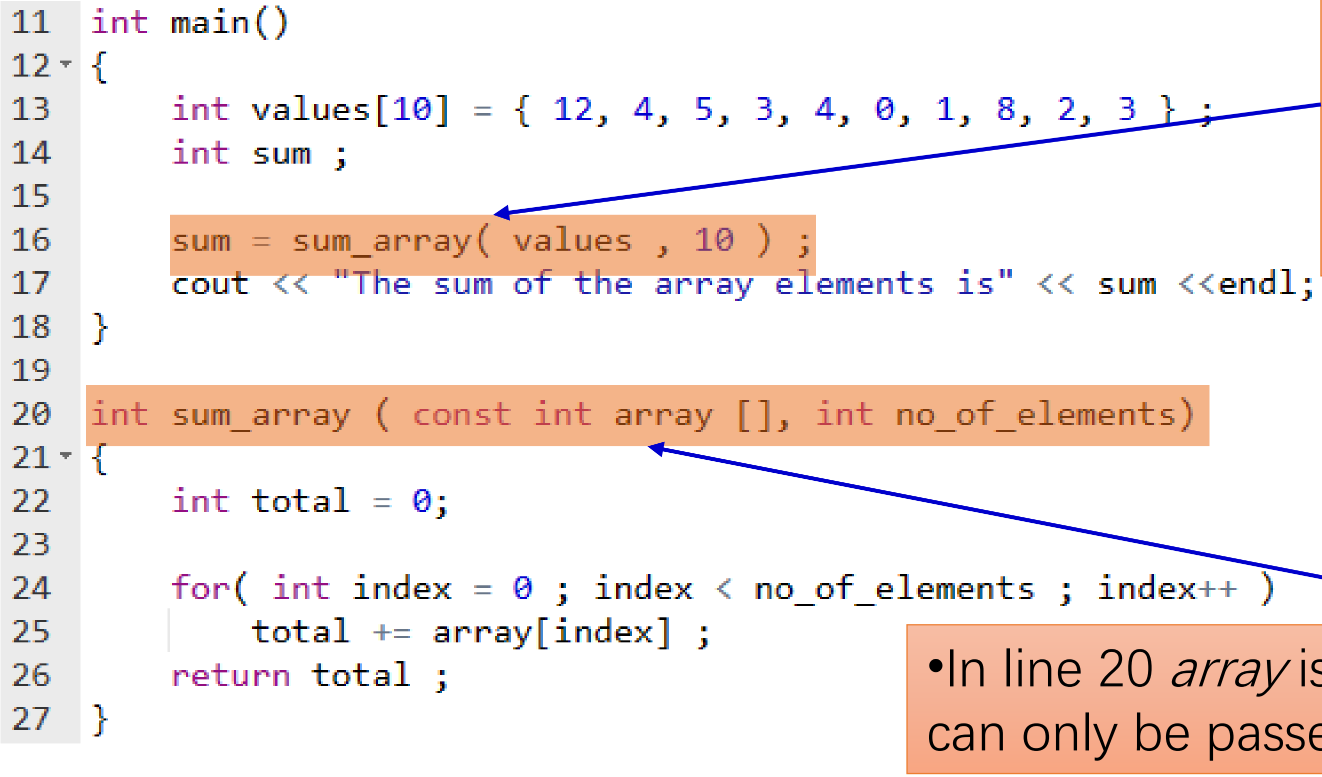
8.3 Passing arguments

- **Arrays** can **only** be **passed by reference** to a function.
  - To avoid the overhead of copying all the elements of an array
  - Program Example: contains a function sum_array() that sums the elements of an integer array passed to it from main().



```
int sum_array ( const int array [], int no_of_elements) ;
// Purpose : Sums the elements of a 1-D integer array
// Parameters : An array and the number of the elements in the array.
// Returns :The sum of tje arrat elements.
```

8.3 Passing arguments



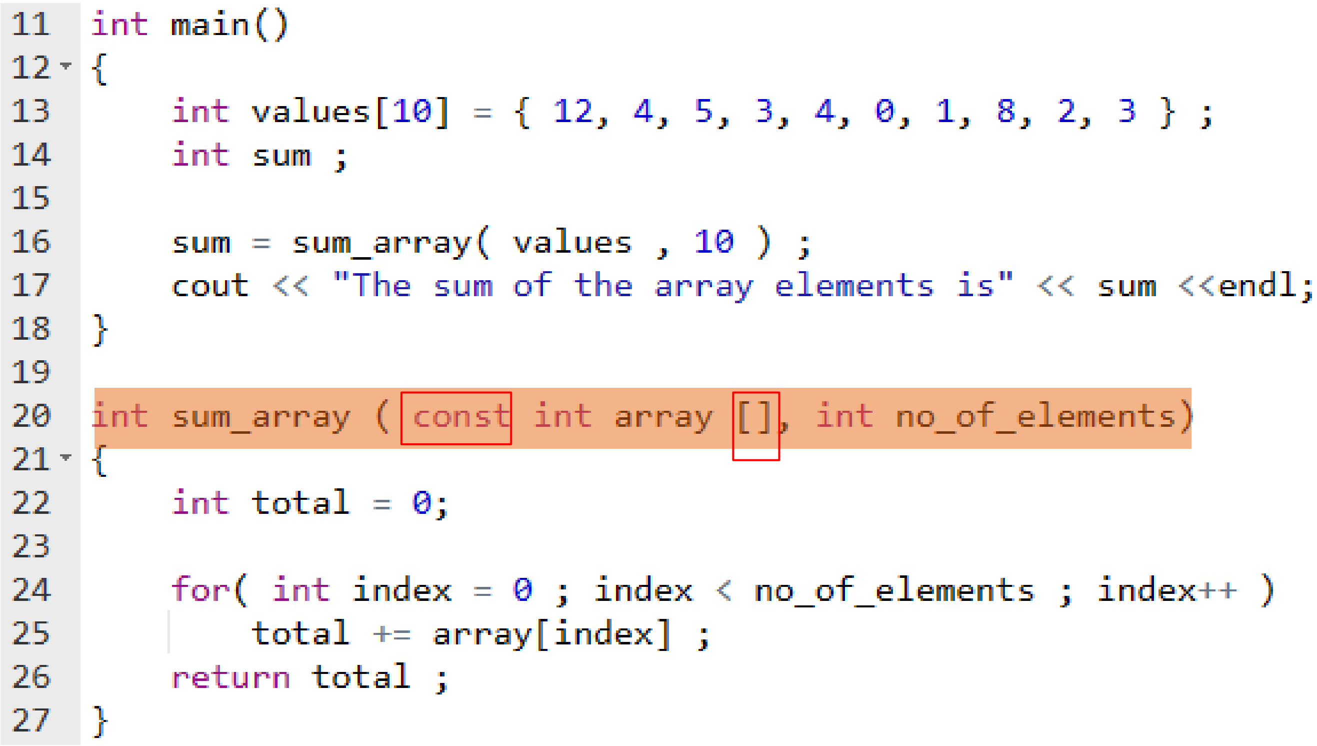
```
11  int main()
12  {
13      int values[10] = { 12, 4, 5, 3, 4, 0, 1, 8, 2, 3 } ;
14      int sum ;
15
16      sum = sum_array( values , 10 ) ;
17      cout << "The sum of the array elements is" << sum <<endl;
18  }
19
20  int sum_array ( const int array [], int no_of_elements)
21  {
22      int total = 0;
23
24      for( int index = 0 ; index < no_of_elements ; index++ )
25          total += array[index] ;
26      return total ;
27  }
```

•Line 16 calls sum_array() to calculate the sum of the values in the array values.

•The arguments are the name and the number of elements in the array.

•In line 20 *array* is a reference to values. Because arrays can only be passed by reference, & is not required.

8.3 Passing arguments



```
int main()
{
    int values[10] = { 12, 4, 5, 3, 4, 0, 1, 8, 2, 3 } ;
    int sum ;

    sum = sum_array( values , 10 ) ;
    cout << "The sum of the array elements is" << sum <<endl;
}

int sum_array ( const int array [], int no_of_elements)
{
    int total = 0;

    for( int index = 0 ; index < no_of_elements ; index++ )
        total += array[index] ;
    return total ;
}
```

- [and] are necessary to indicate that the parameter is a reference to an array.
- The number of elements is not required in the brackets for a one-dimensional array to handle different size array.
- Const informs the compiler that within the function *sum_array*(), array is read-only and cannot be modified.

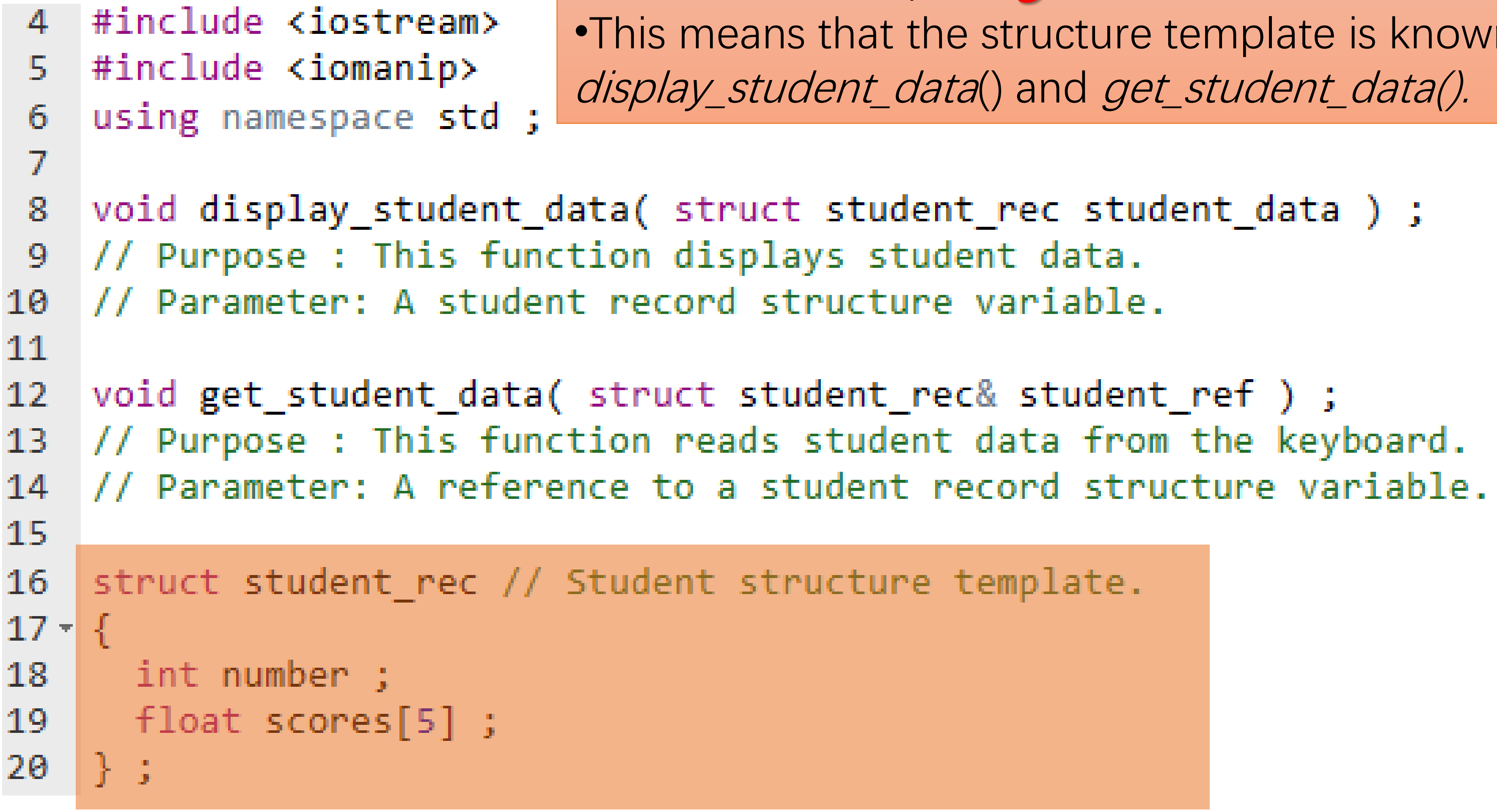
8.3 Passing arguments

- Passing a **structure** variable to a function
  - **pass a copy of the member values** to that function, this means that the values in the (original, outside the function) **structure** variable cannot be changed within the function.
  - The values in a structure variable can be changed from within a function if the variable is passed by reference to the function.

8.3 Passing arguments

- Program Example

- When a structure template is defined **outside** *main()*, it makes the structure template ***global***.
- This means that the structure template is known in *main()* and in *display_student_data*() and *get_student_data().*



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

void display_student_data( struct student_rec student_data ) ;
// Purpose : This function displays student data.
// Parameter: A student record structure variable.

void get_student_data( struct student_rec& student_ref ) ;
// Purpose : This function reads student data from the keyboard.
// Parameter: A reference to a student record structure variable.

struct student_rec // Student structure template.
{
  int number ;
  float scores[5] ;
} ;
```

8.3 Passing arguments

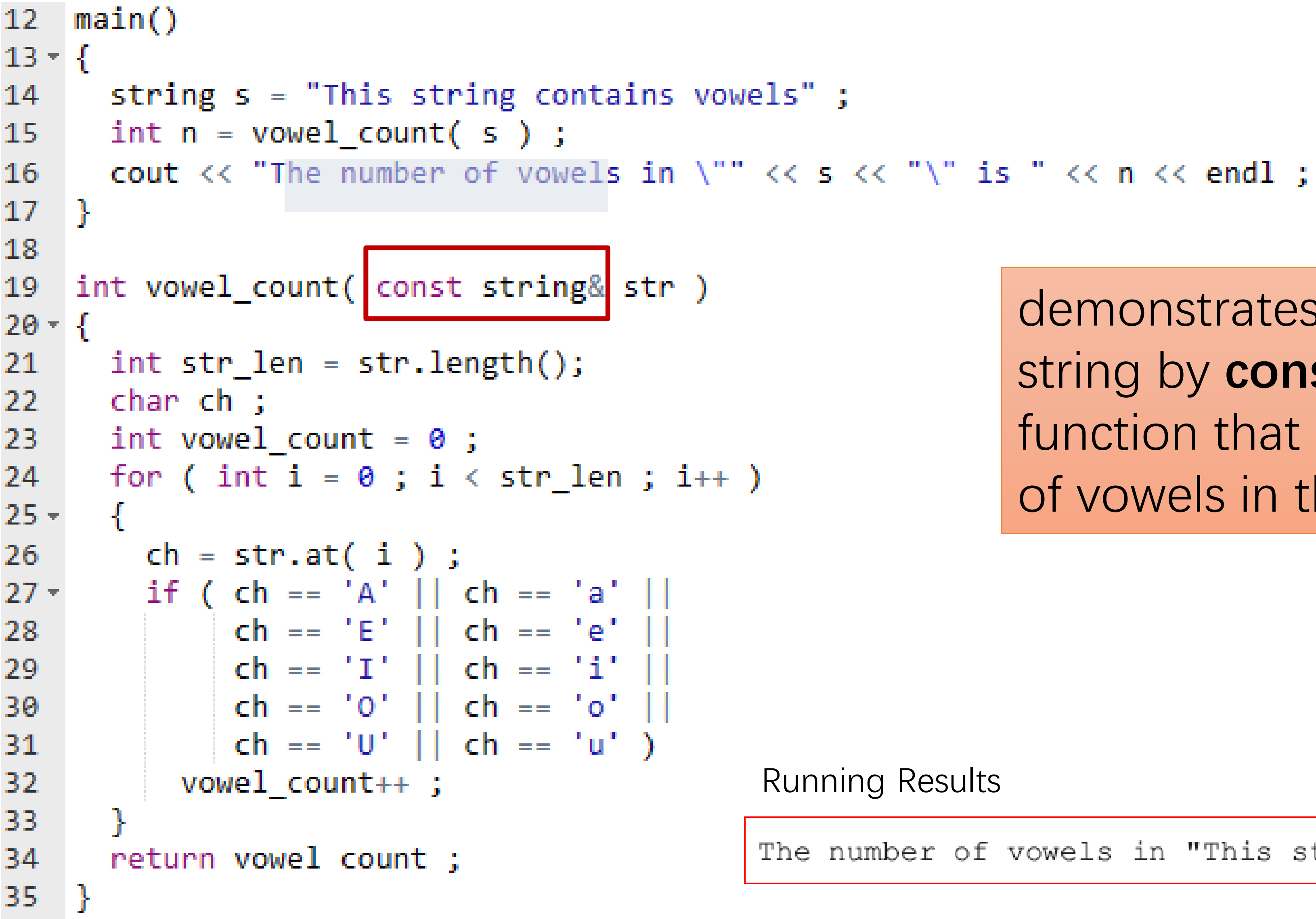
- The same considerations should be kept in mind when using **strings** as arguments as when using **structure** variables as arguments, i.e. passing a string by value means copying all the characters of the string to a function parameter.
  - To avoid this overhead it is preferable to pass strings **by reference**.

8.3 Passing arguments

The same considerations should be kept in mind when using strings as arguments as when using structure variables as arguments, i.e. **passing a string by value** means copying all the characters of the string to a function parameter.

To avoid this overhead it is preferable to pass strings **by reference**.

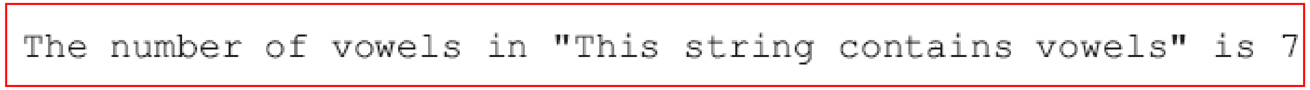
8.3 Passing arguments



```
main()
{
  string s = "This string contains vowels" ;
  int n = vowel_count( s ) ;
  cout << "The number of vowels in \"" << s << "\" is " << n << endl ;
}

int vowel_count( const string& str )
{
  int str_len = str.length();
  char ch ;
  int vowel_count = 0 ;
  for ( int i = 0 ; i < str_len ; i++ )
  {
    ch = str.at( i ) ;
    if ( ch == 'A' || ch == 'a' ||
         ch == 'E' || ch == 'e' ||
         ch == 'I' || ch == 'i' ||
         ch == 'O' || ch == 'o' ||
         ch == 'U' || ch == 'u' )
      vowel_count++ ;
  }
  return vowel_count ;
}
```

demonstrates passing a C++ string by **const reference** to a function that counts the number of vowels in the string.

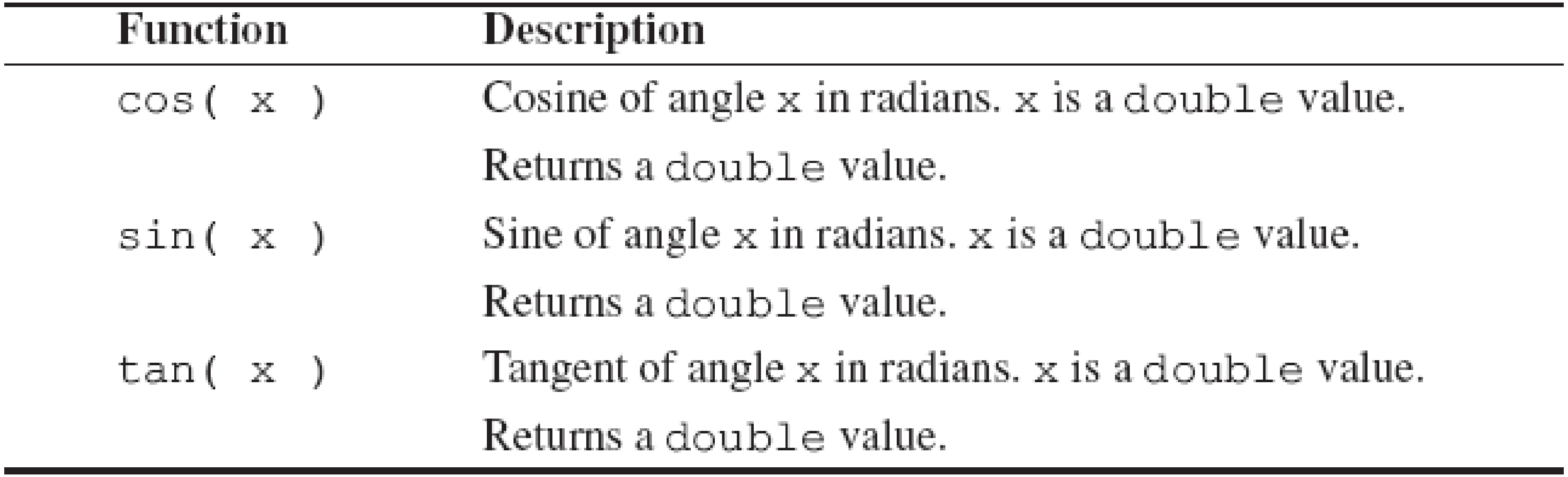
Running Results



```
The number of vowels in "This string contains vowels" is 7
```

8.4 Mathematical functions

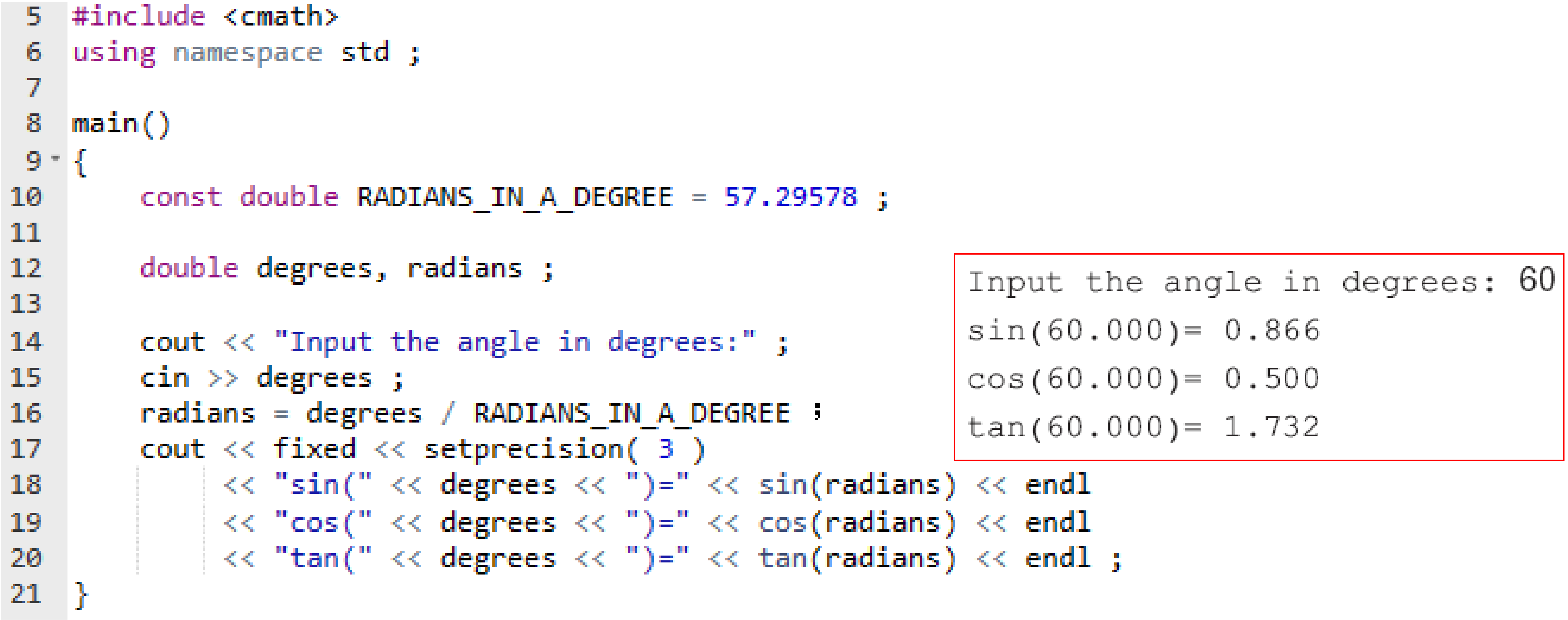
- To use any of the mathematical functions place the statement *#include <cmath>* at the start of the program.
- **Some trigonometric functions**



Function	Description
`cos( x )`	Cosine of angle `x` in radians. `x` is a `double` value. Returns a `double` value.
`sin( x )`	Sine of angle `x` in radians. `x` is a `double` value. Returns a `double` value.
`tan( x )`	Tangent of angle `x` in radians. `x` is a `double` value. Returns a `double` value.

8.4 Mathematical functions

- Program Example: demonstrates sin(), cos() and tan() functions.

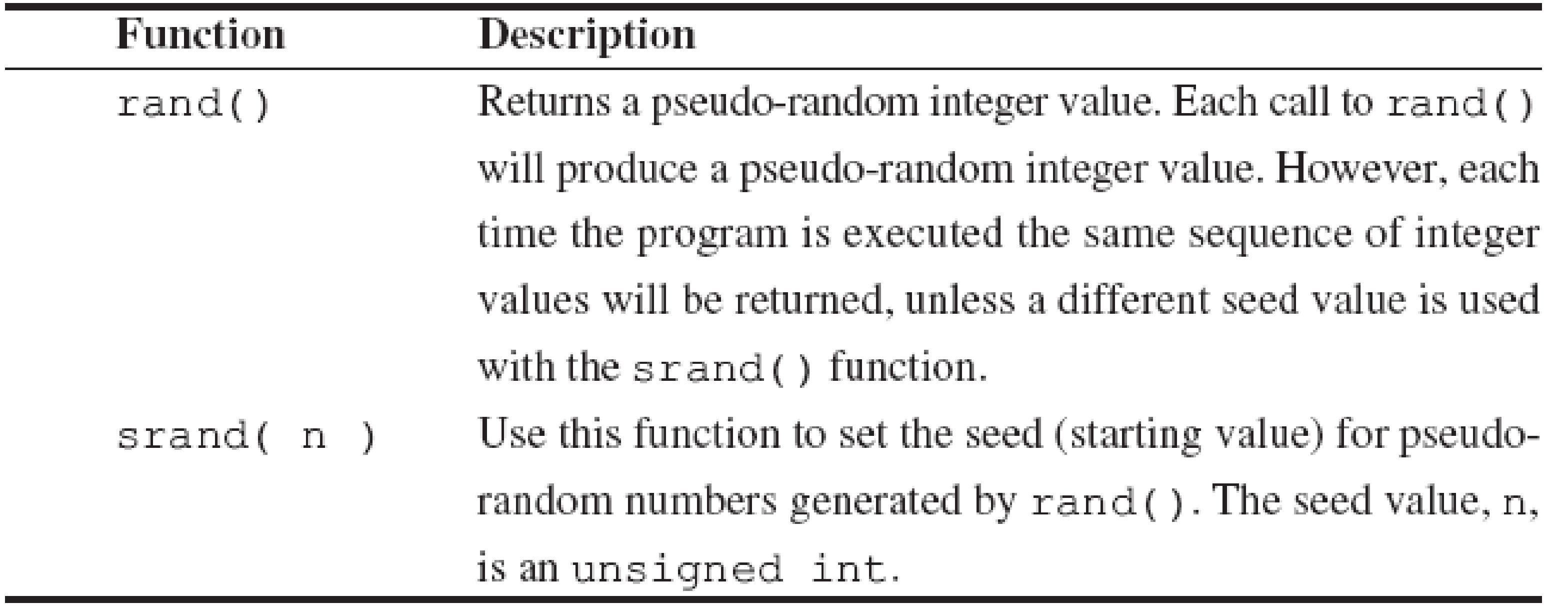


```
#include <cmath>
using namespace std ;

main()
{
    const double RADIANS_IN_A_DEGREE = 57.29578 ;

    double degrees, radians ;

    cout << "Input the angle in degrees:" ;
    cin >> degrees ;
    radians = degrees / RADIANS_IN_A_DEGREE ;
    cout << fixed << setprecision( 3 )
        << "sin(" << degrees << ")=" << sin(radians) << endl
        << "cos(" << degrees << ")=" << cos(radians) << endl
        << "tan(" << degrees << ")=" << tan(radians) << endl ;
}
```

```
Input the angle in degrees: 60
sin(60.000)= 0.866
cos(60.000)= 0.500
tan(60.000)= 1.732
```

8.4 Mathematical functions

- **Pseudo-random number functions**
  - To use the **pseudo-random generating functions** *rand()* and *srand()*, place the statement *#include <cstdlib>* at the start of the program.



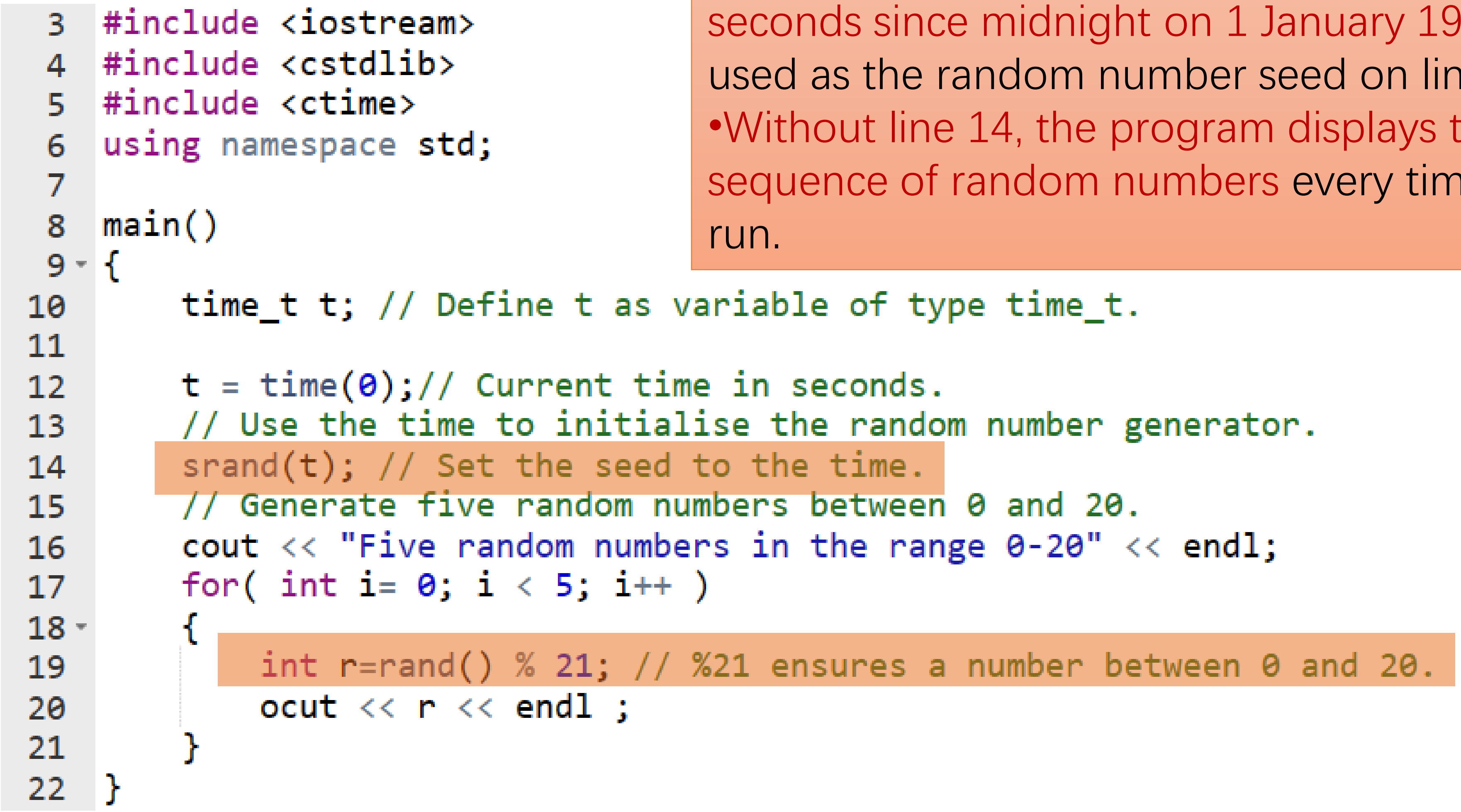
Function	Description
`rand()`	Returns a pseudo-random integer value. Each call to `rand()` will produce a pseudo-random integer value. However, each time the program is executed the same sequence of integer values will be returned, unless a different seed value is used with the `srand()` function.
`srand( n )`	Use this function to set the seed (starting value) for pseudo-random numbers generated by `rand()`. The seed value, `n`, is an `unsigned int`.

8.4 Mathematical functions

- Program Example

•Line 12 assigns to t the current time (measured in seconds since midnight on 1 January 1970, GMT) which is used as the random number seed on line 14.

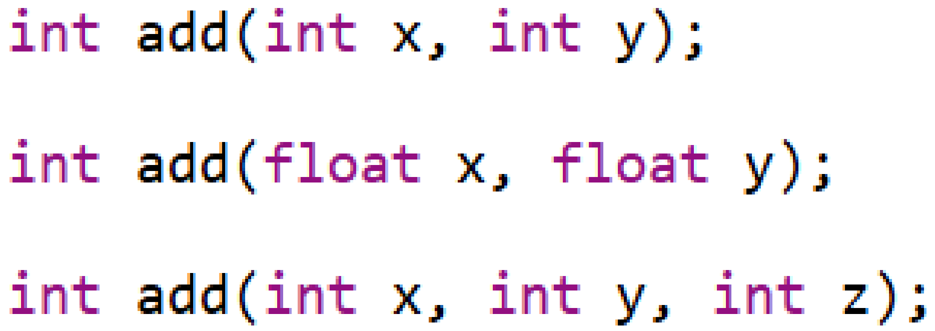
•Without line 14, the program displays the same sequence of random numbers every time the program is run.



```
 3  #include <iostream>
 4  #include <cstdlib>
 5  #include <ctime>
 6  using namespace std;
 7
 8  main()
 9  {
10      time_t t; // Define t as variable of type time_t.
11
12      t = time(0);// Current time in seconds.
13      // Use the time to initialise the random number generator.
14      srand(t); // Set the seed to the time.
15      // Generate five random numbers between 0 and 20.
16      cout << "Five random numbers in the range 0-20" << endl;
17      for( int i= 0; i < 5; i++ )
18      {
19          int r=rand() % 21; // %21 ensures a number between 0 and 20.
20          ocut << r << endl ;
21      }
22  }
```

8.5 Function overloading

- **Function overloading** is used when there is a need for two or more functions to perform similar tasks, but where each function requires a different number of arguments and/or different argument data types.



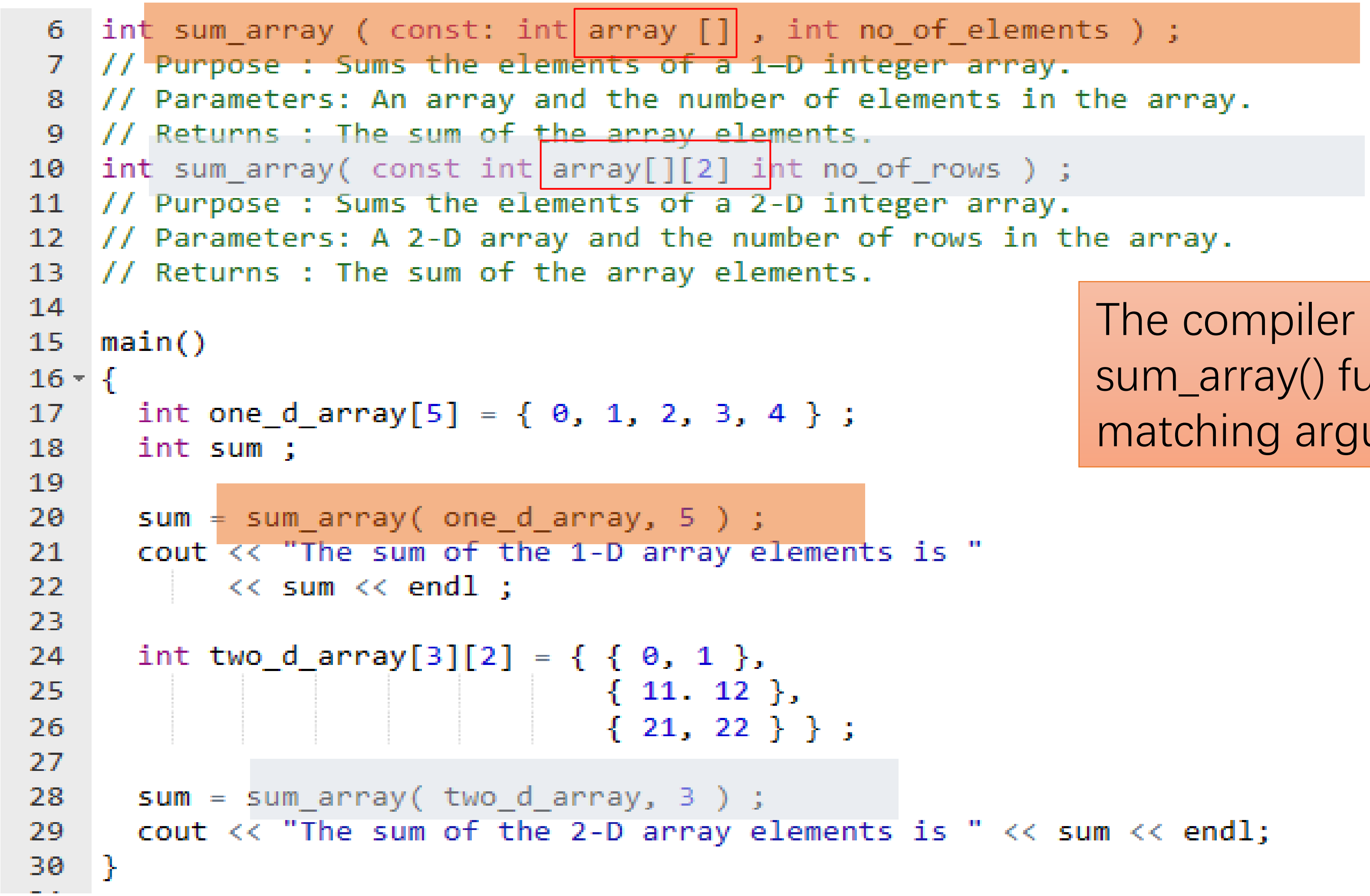
```
int add(int x, int y);

int add(float x, float y);

int add(int x, int y, int z);
```

8.5 Function overloading

- Using **different functions** with the **same name** in a program is called *function overloading* and the functions are called *overloaded functions.*
  - Function overloading requires that each overloaded function have **a different parameter list**, i.e. a **different number** of parameters or <u>at least one parameter with a **different data type**</u>.

8.5 Function overloading

- Program Example

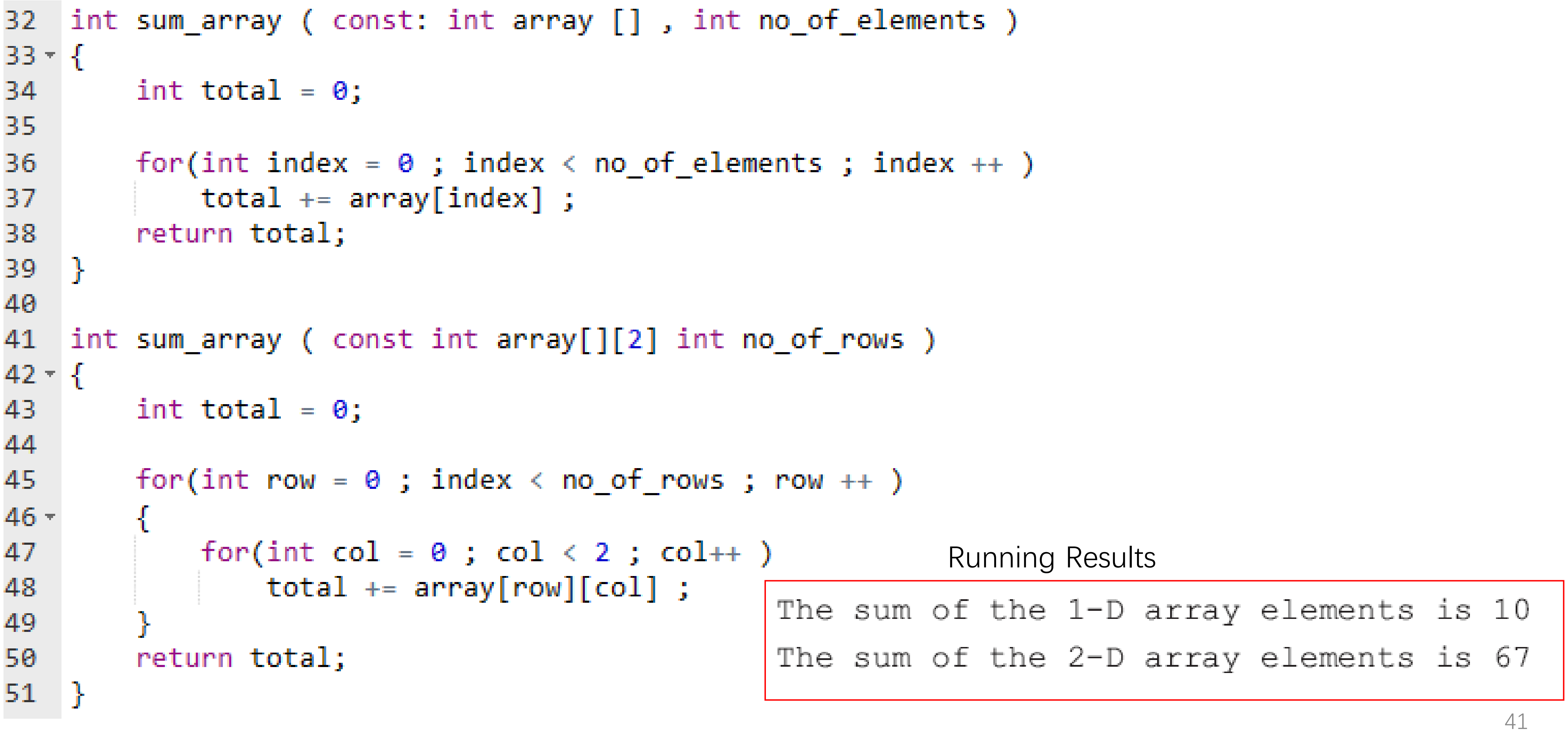


```
int sum_array ( const: int array [] , int no_of_elements ) ;
// Purpose : Sums the elements of a 1—D integer array.
// Parameters: An array and the number of elements in the array.
// Returns : The sum of the array elements.
int sum_array( const int array[][2] int no_of_rows ) ;
// Purpose : Sums the elements of a 2-D integer array.
// Parameters: A 2-D array and the number of rows in the array.
// Returns : The sum of the array elements.

main()
{
  int one_d_array[5] = { 0, 1, 2, 3, 4 } ;
  int sum ;

  sum = sum_array( one_d_array, 5 ) ;
  cout << "The sum of the 1-D array elements is "
       << sum << endl ;

  int two_d_array[3][2] = { { 0, 1 },
                            { 11. 12 },
                            { 21, 22 } } ;

  sum = sum_array( two_d_array, 3 ) ;
  cout << "The sum of the 2-D array elements is " << sum << endl;
}
```

The compiler decides which of the two sum_array() functions to call based on matching arguments with parameters.

8.5 Function overloading

- Program Example…continued



```cpp
int sum_array ( const: int array [] , int no_of_elements )
{
    int total = 0;

    for(int index = 0 ; index < no_of_elements ; index ++ )
        total += array[index] ;
    return total;
}

int sum_array ( const int array[][2] int no_of_rows )
{
    int total = 0;

    for(int row = 0 ; index < no_of_rows ; row ++ )
    {
        for(int col = 0 ; col < 2 ; col++ )
            total += array[row][col] ;
    }
    return total;
}
```

Running Results

```
The sum of the 1-D array elements is 10
The sum of the 2-D array elements is 67
```

8.6 Recursion

- **Recursion** is a programming technique in which a problem can be defined in terms of itself. The technique involves solving a problem by reducing the problem to smaller versions of itself.

8.6 Recursion

- A mathematical example

The **factorial** of a positive integer is the **product of the integers from 1 through to that number:**

$$n! \text{ is } \begin{cases} 1 \text{ when n is } 0 \\ n * (n-1)! \text{ when } n>0 \end{cases}$$

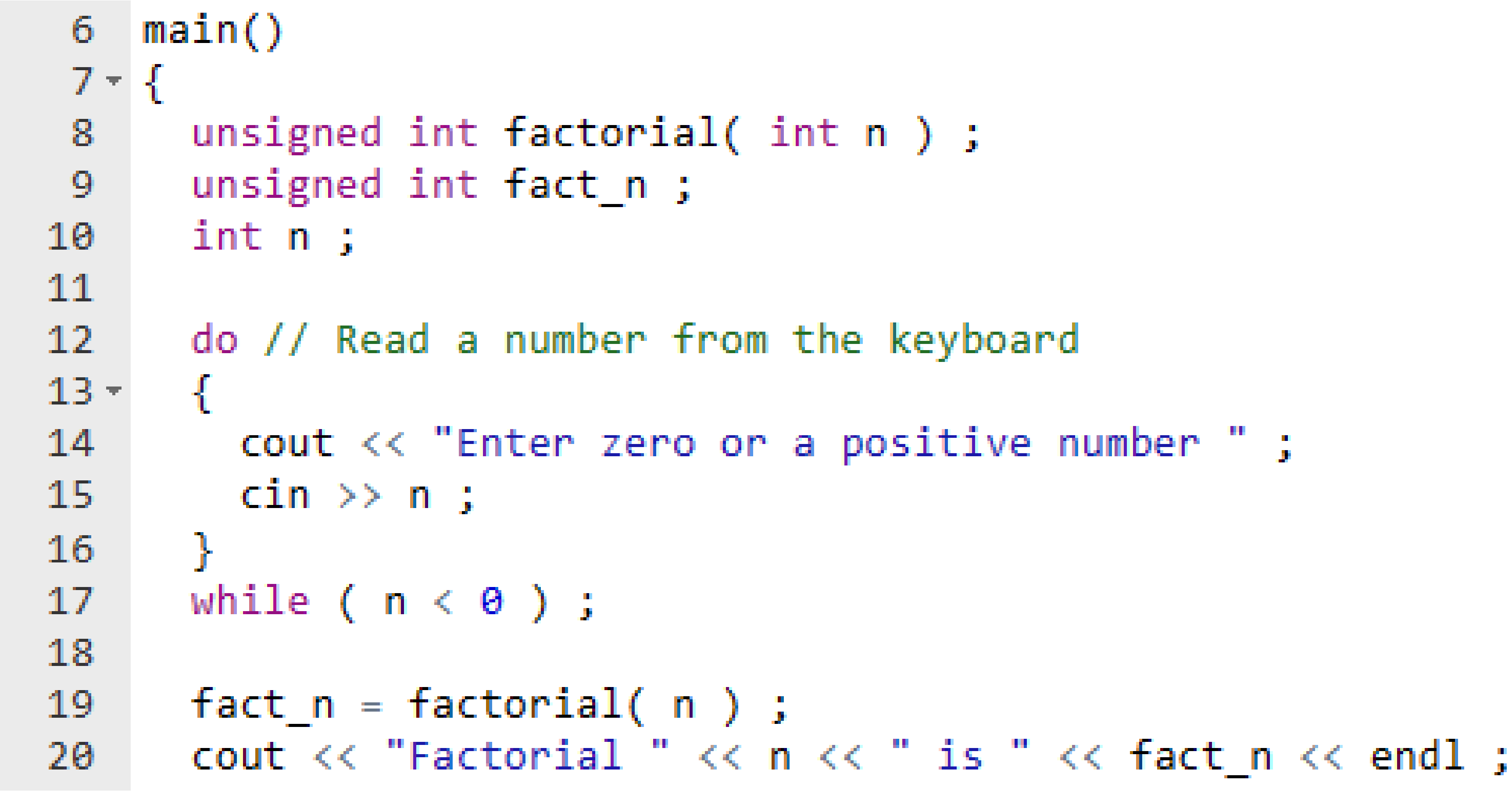
- (a) 0!=1. This is called the *base case.*
- (b) For a positive integer n, factorial n is n times the factorial of n-1.This is called the *general case* clearly indicates that factorial is defined in terms of itself.

8.6 Recursion

- Using the definition, factorial 3 is calculated as follows:
  - The value of n is 3 so, using (b) above, 3! = 3 * 2!
    - Next find 2! Here n = 2 so, using (b) again, 2! = 2 * 1!
      - Next find 1! Here n = 1 so, using (b) again, 1! = 1 * 0!
        - Next find 0! In this case using (a), 0! is defined as 1.
      - Substituting for 0! gives 1! = 1 * 1 = 1.
    - Substituting for 1! gives 2! = 2 * 1! = 2 * 1 = 2.
  - Finally, substituting for 2! gives 3! = 3 * 2! = 3 * 2 = 6.

8.6 Recursion

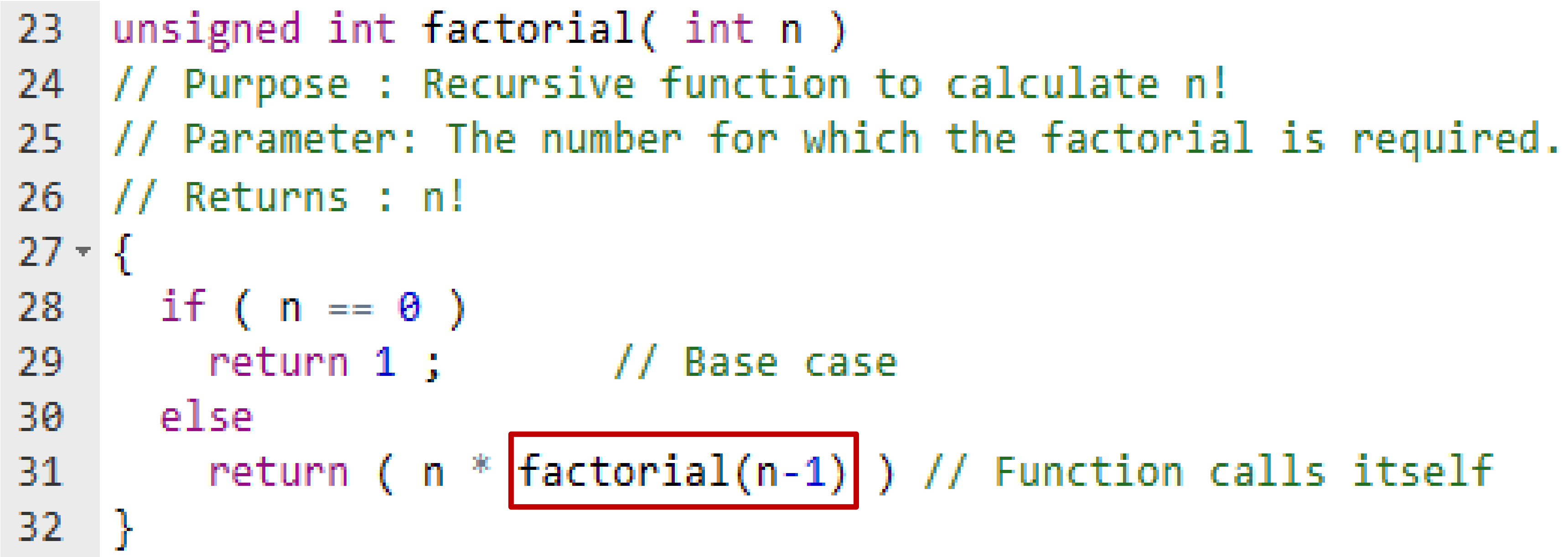
- Program Example



```
main()
{
  unsigned int factorial( int n ) ;
  unsigned int fact_n ;
  int n ;

  do // Read a number from the keyboard
  {
    cout << "Enter zero or a positive number " ;
    cin >> n ;
  }
  while ( n < 0 ) ;

  fact_n = factorial( n ) ;
  cout << "Factorial " << n << " is " << fact_n << endl ;
```

8.6 Recursion

- Program Example



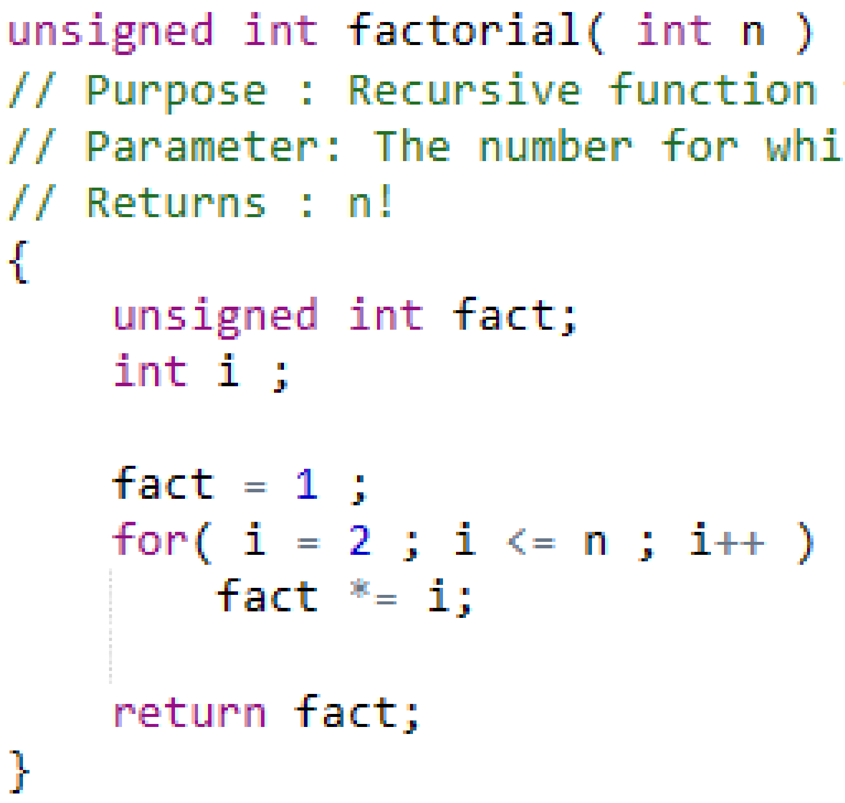
```
unsigned int factorial( int n )
// Purpose : Recursive function to calculate n!
// Parameter: The number for which the factorial is required.
// Returns : n!
{
  if ( n == 0 )
    return 1 ;        // Base case
  else
    return ( n * factorial(n-1) ) // Function calls itself
}
```

Note that

- Every recursive function must have at least one base case which stops the recursion
- The general case eventually reduces to a base case.

8.6 Recursion

- The factorial function could be written using **iteration**



```
unsigned int factorial( int n )
// Purpose : Recursive function to calculate n!
// Parameter: The number for which the factorial is required.
// Returns : n!
{
    unsigned int fact;
    int i ;

    fact = 1 ;
    for( i = 2 ; i <= n ; i++ )
        fact *= i;

    return fact;
}
```

•The recursive version will execute more **slowly** than the iterative equivalent because of the added overhead of the function calls.

•The advantage of the recursive version is that it is clearer because it follows the actual mathematical definition of factorial.

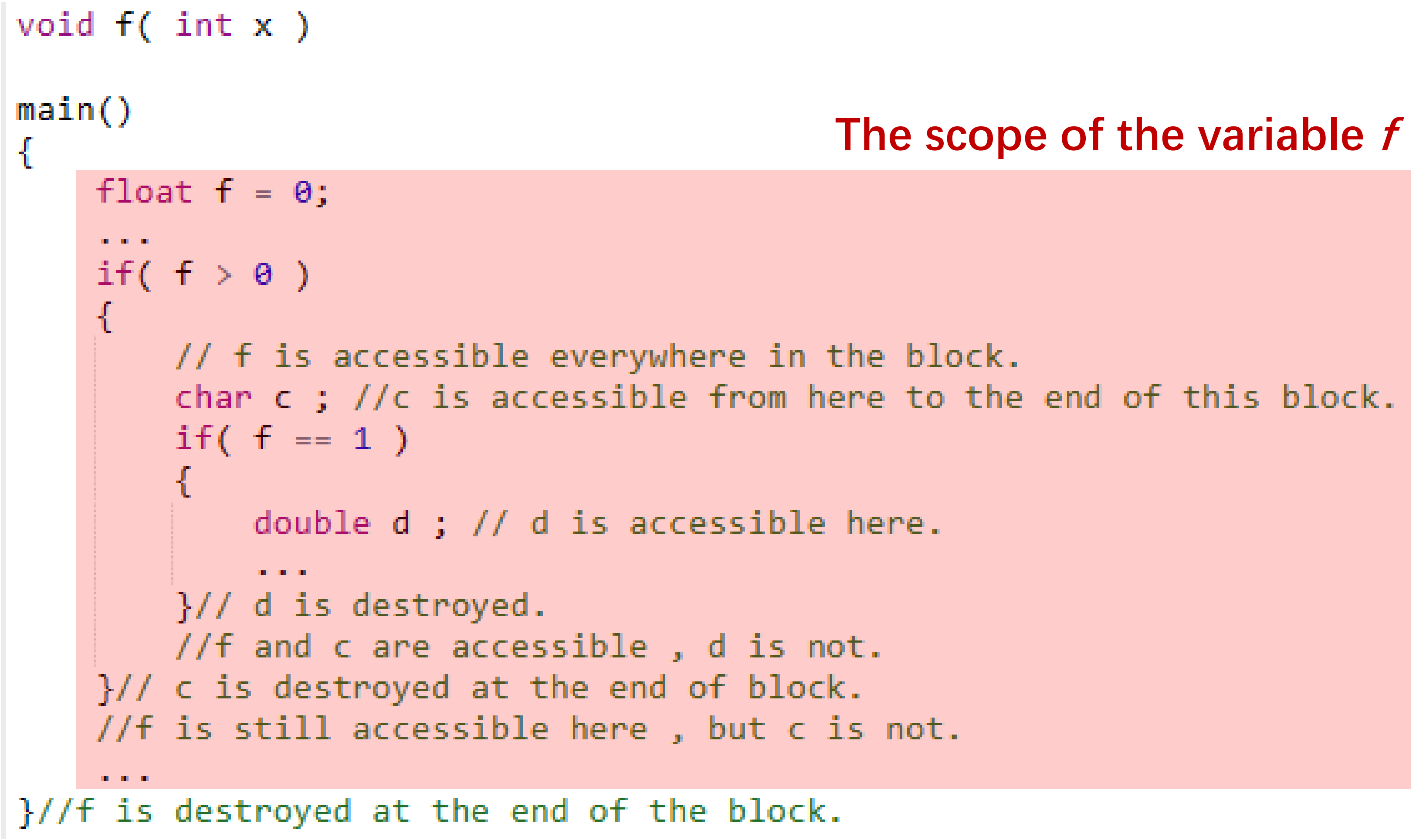
8.7 The scope of a variable

- The scope of a variable refers to the part of the program in which a variable can be accessed.
  - *block scope*
  - *global scope*

- **Block scope**
  - A block is one or more statements enclosed in braces { and } that also includes variable declarations.
  - A variable declared in a block is accessible only within that block.

8.7 The scope of a variable

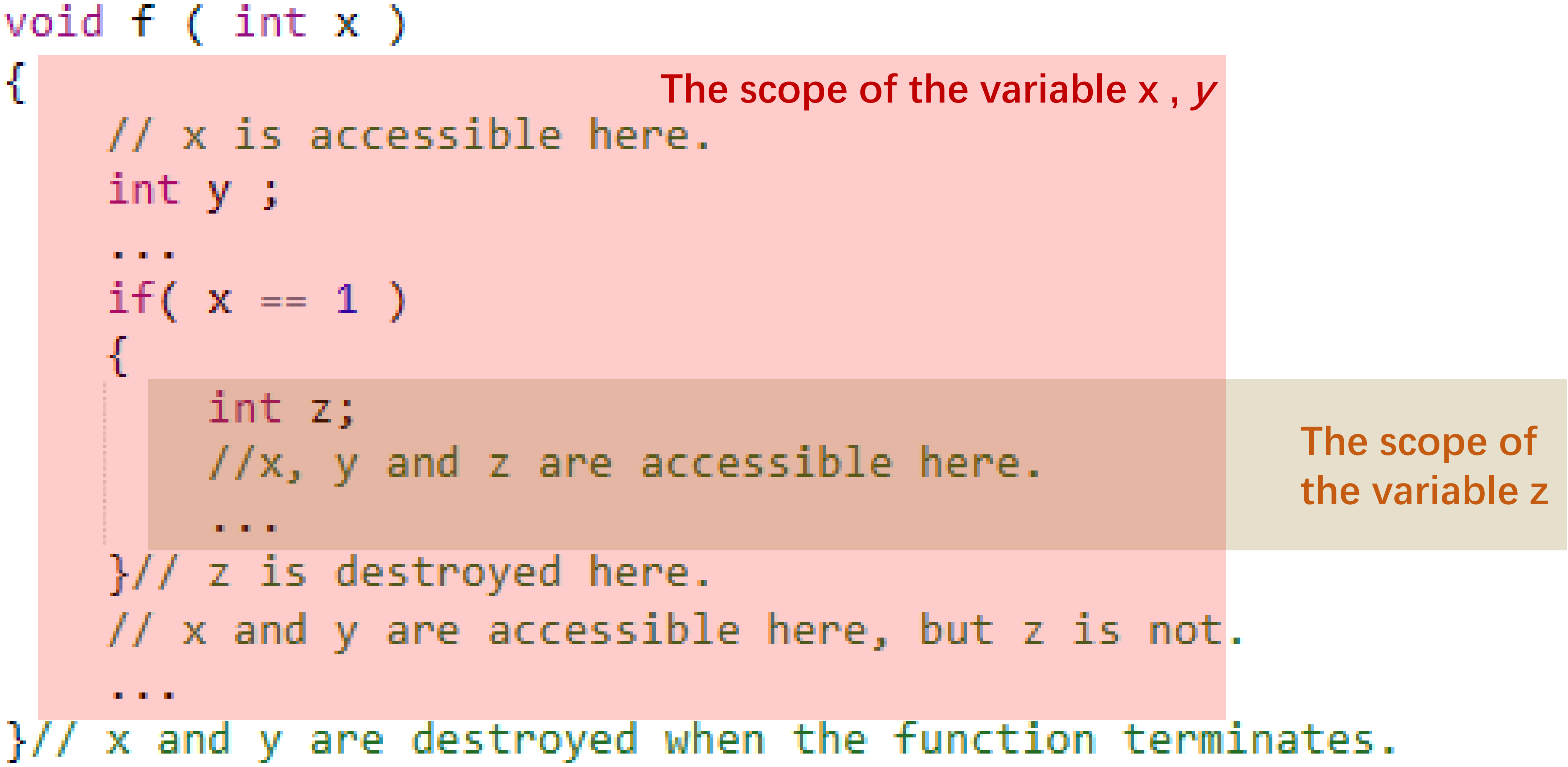
- **Block scope**



```
void f( int x )

main()
{
    float f = 0;
    ...
    if( f > 0 )
    {
        // f is accessible everywhere in the block.
        char c ; //c is accessible from here to the end of this block.
        if( f == 1 )
        {
            double d ; // d is accessible here.
            ...
        }// d is destroyed.
        //f and c are accessible , d is not.
    }// c is destroyed at the end of block.
    //f is still accessible here , but c is not.
    ...
}//f is destroyed at the end of the block.
```

The scope of the variable *f*

8.7 The scope of a variable



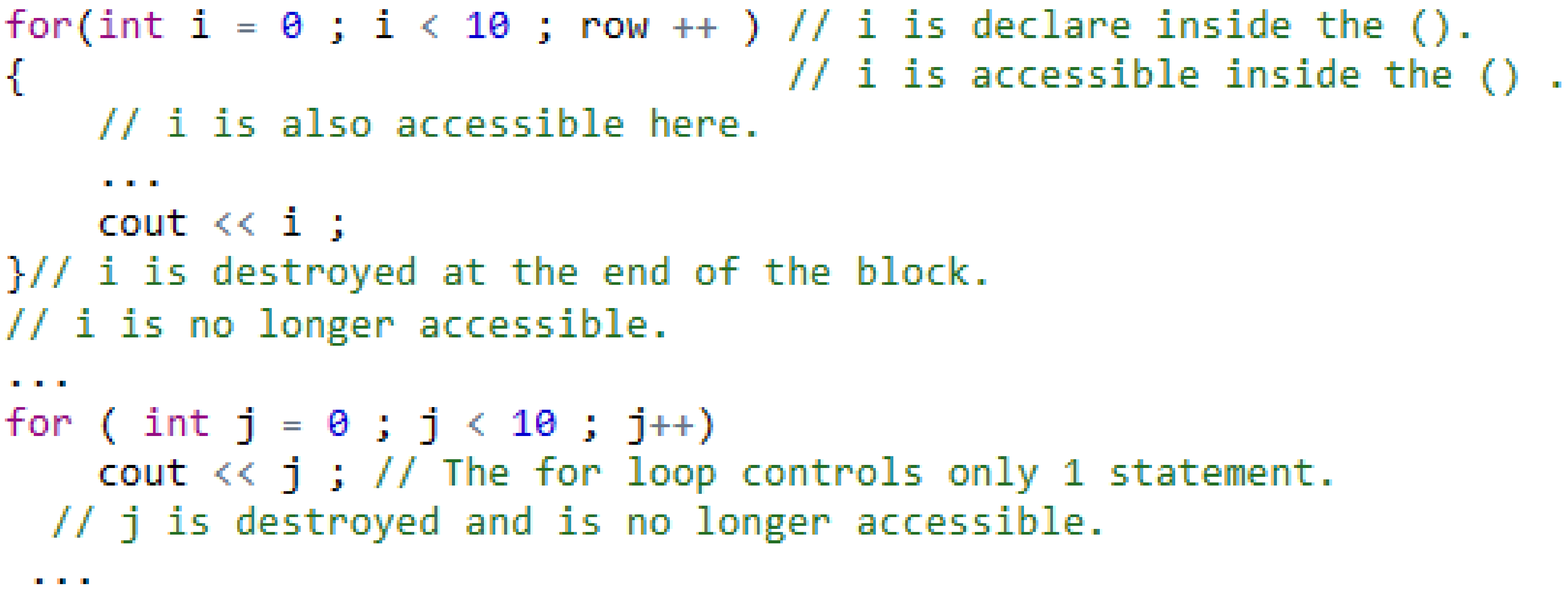
```
void f ( int x )
{
    // x is accessible here.
    int y ;
    ...
    if( x == 1 )
    {
        int z;
        //x, y and z are accessible here.
        ...
    }// z is destroyed here.
    // x and y are accessible here, but z is not.
    ...
}// x and y are destroyed when the function terminates.
```

The scope of the variable x , *y*

The scope of the variable z

8.7 The scope of a variable

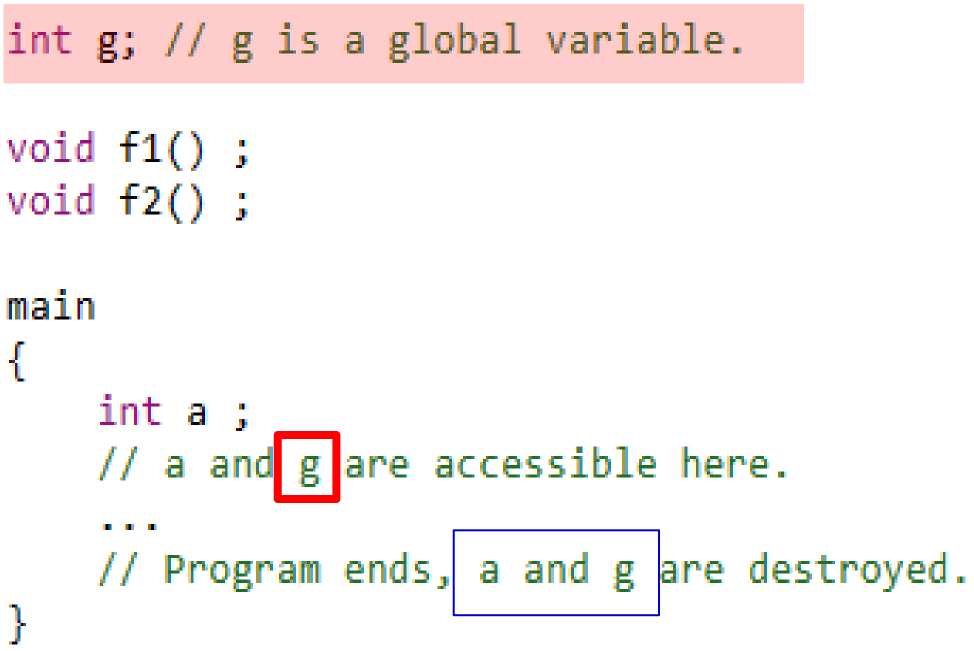
- Variables declared inside the parentheses of a for are accessible within the parentheses, as well as in the statement(s) contained in the for loop.



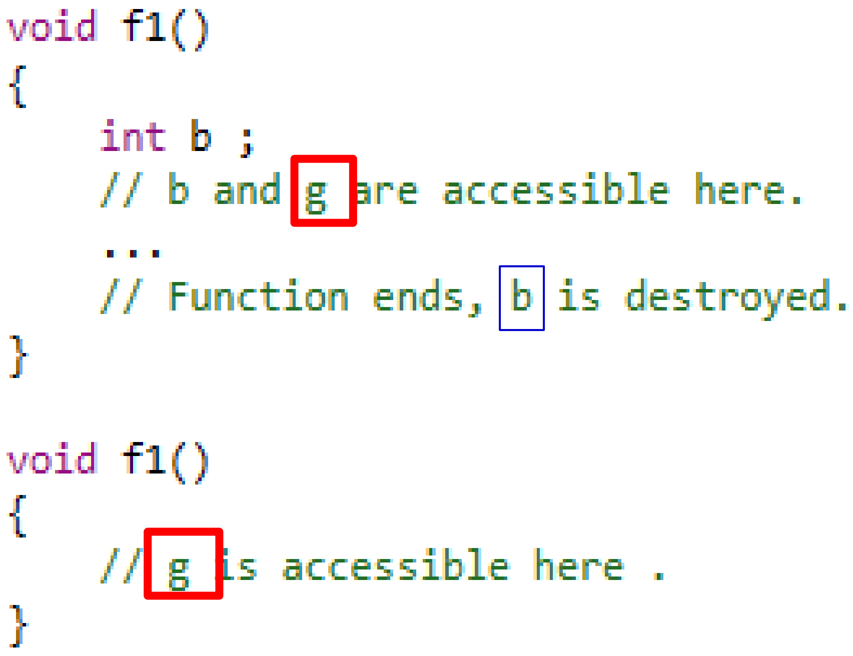
```
for(int i = 0 ; i < 10 ; row ++ ) // i is declare inside the ().
{                                 // i is accessible inside the () .
    // i is also accessible here.
    ...
    cout << i ;
}// i is destroyed at the end of the block.
// i is no longer accessible.
...
for ( int j = 0 ; j < 10 ; j++)
    cout << j ; // The for loop controls only 1 statement.
  // j is destroyed and is no longer accessible.
 ...
```

8.7 The scope of a variable

- **Global scope**
  - A variable declared **outside** main() is **accessible from anywhere within the program** and is known as a global variable.



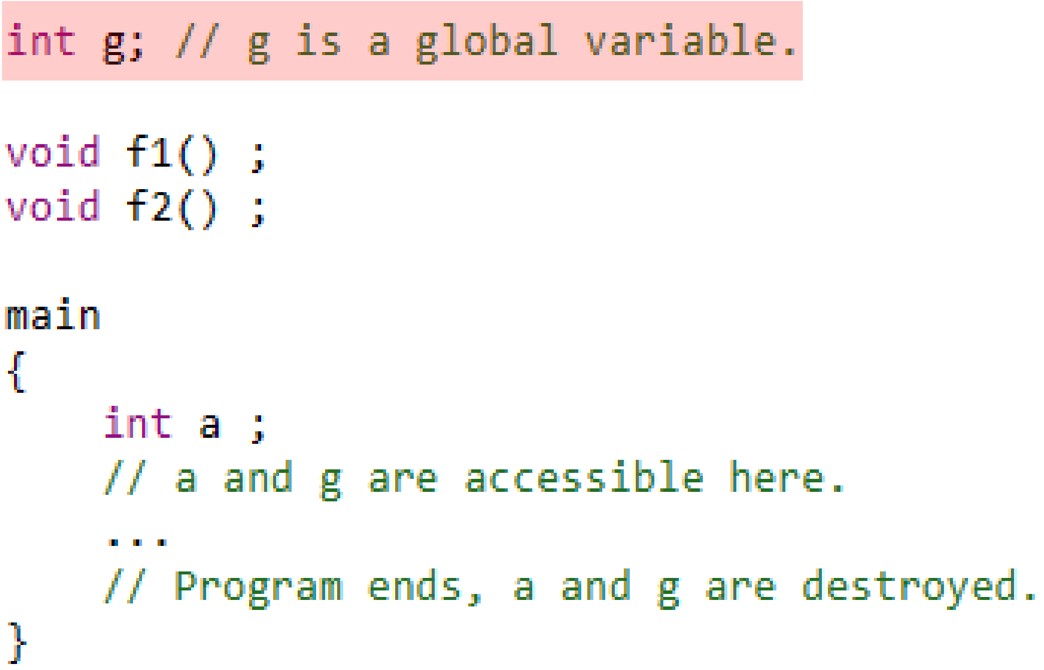
```
int g; // g is a global variable.

void f1() ;
void f2() ;

main
{
    int a ;
    // a and g are accessible here.
    ...
    // Program ends, a and g are destroyed.
}
```



```
void f1()
{
    int b ;
    // b and g are accessible here.
    ...
    // Function ends, b is destroyed.
}

void f1()
{
    // g is accessible here .
}
```

8.7 The scope of a variable

- **Global scope**
  - A variable declared outside main() is accessible from anywhere within the program and is known as a global variable.



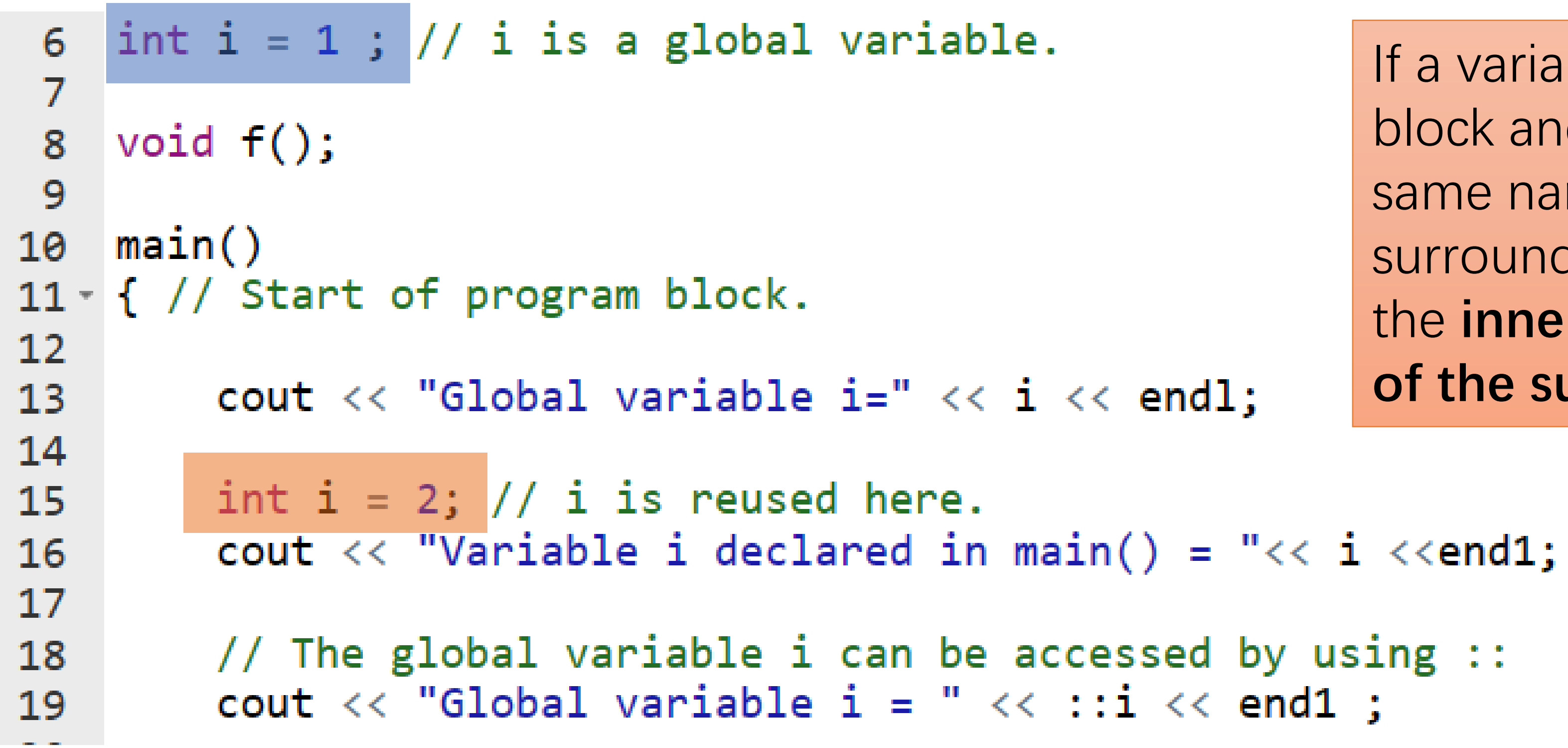
```
int g; // g is a global variable.

void f1() ;
void f2() ;

main
{
    int a ;
    // a and g are accessible here.
    ...
    // Program ends, a and g are destroyed.
}
```

•Because global variables are known, and therefore **can be modified within every function**, they can make a program **difficult to debug and maintain**.

•Global variables are **not a substitute for function arguments**. Apart from its own local variables, a function should have access only to the data specified in the function parameter list.

8.7 The scope of a variable

- Reusing a variable name
  - It is permissible to give a variable the same name as another variable in another block. This is known as *name reuse*.

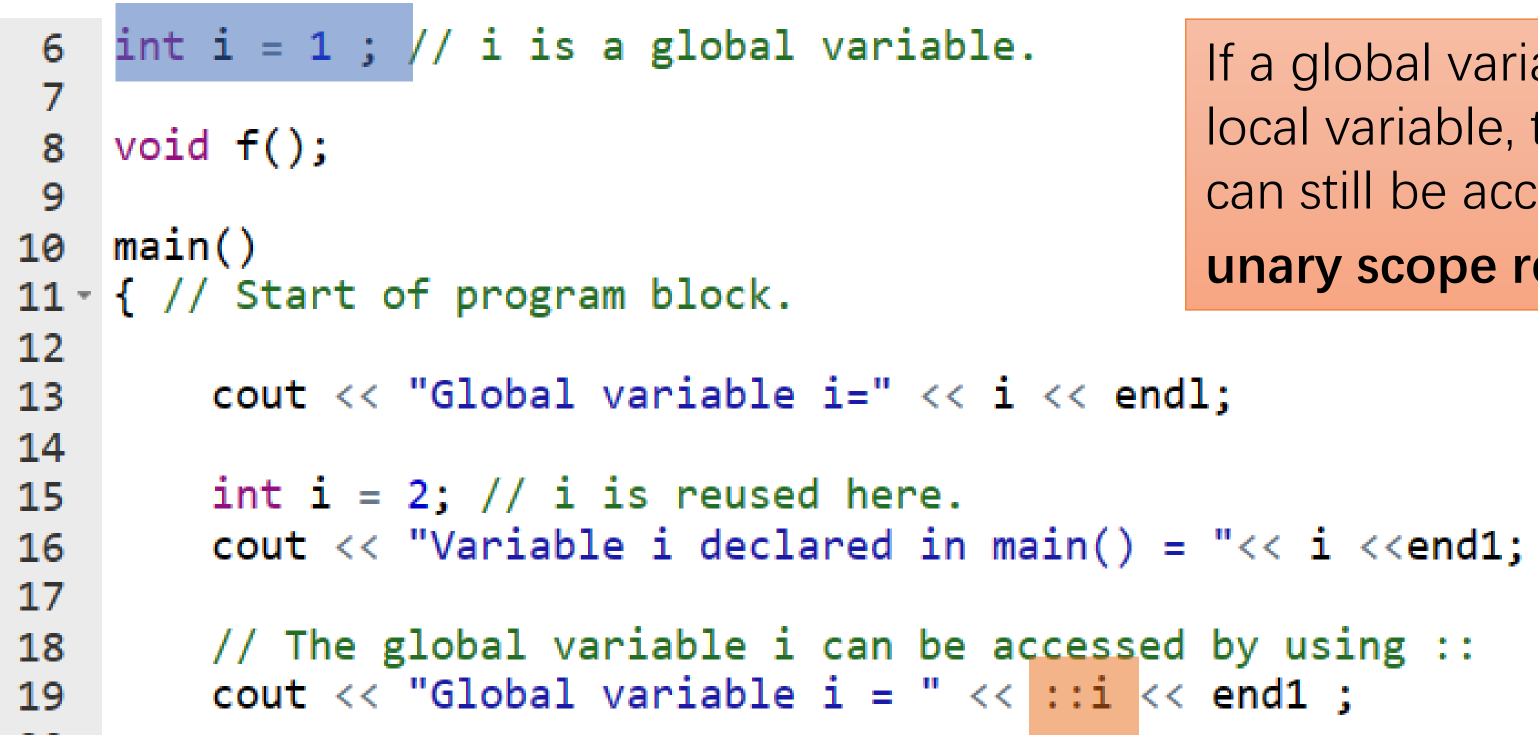


```
int i = 1 ; // i is a global variable.

void f();

main()
{ // Start of program block.

    cout << "Global variable i=" << i << endl;

    int i = 2; // i is reused here.
    cout << "Variable i declared in main() = "<< i <<end1;

    // The global variable i can be accessed by using ::
    cout << "Global variable i = " << ::i << end1 ;
```

If a variable is declared in an inner block and if a variable with the same name is declared in a surrounding block, the variable in the **inner block** ***hides*** **the variable of the surrounding block**.

8.7 The scope of a variable

- Reusing a variable name
  - It is permissible to give a variable the same name as another variable in another block. This is known as *name reuse*.



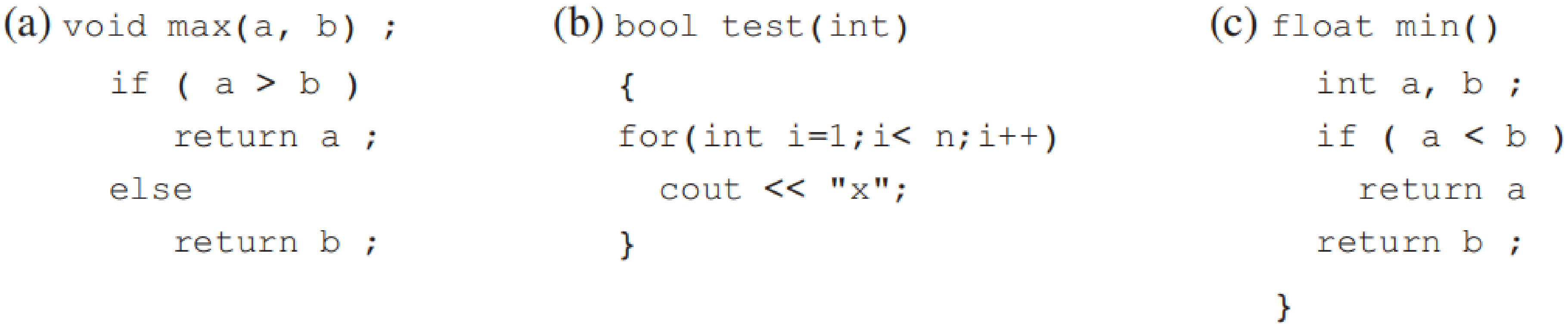
```
int i = 1 ; // i is a global variable.

void f();

main()
{ // Start of program block.

    cout << "Global variable i=" << i << endl;

    int i = 2; // i is reused here.
    cout << "Variable i declared in main() = "<< i <<end1;

    // The global variable i can be accessed by using ::
    cout << "Global variable i = " << ::i << end1 ;
```

If a global variable is hidden by a local variable, the global variable can still be accessed using the **unary scope resolution operator ::**

HOMEWORK

Homework 8

- 1. Identify the errors of the following functions:



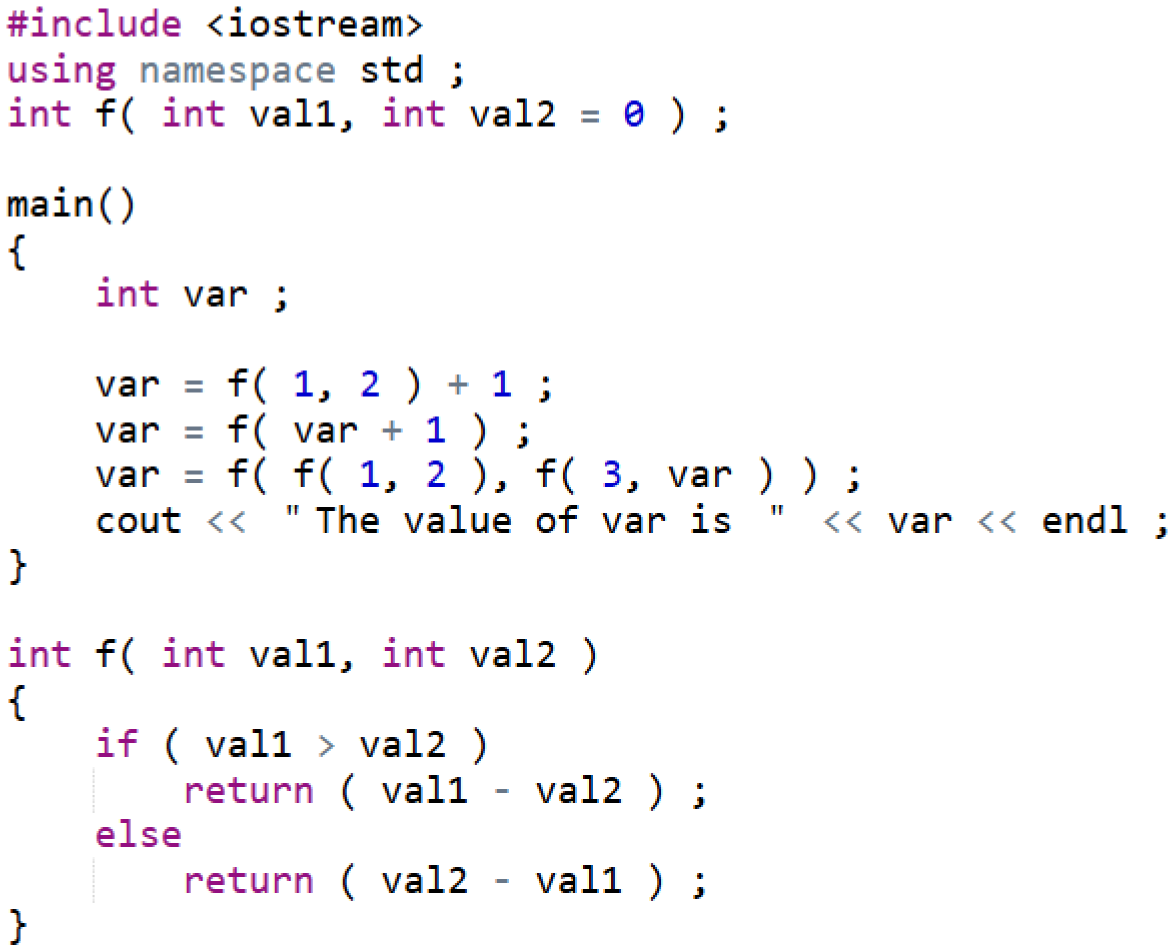
(a)
```
void max(a, b) ;
    if ( a > b )
        return a ;
    else
        return b ;
```

(b)
```
bool test(int)
{
for(int i=1;i< n;i++)
    cout << "x";
}
```

(c)
```
float min()
    int a, b ;
    if ( a < b )
      return a
    return b ;
}
```

Homework 8

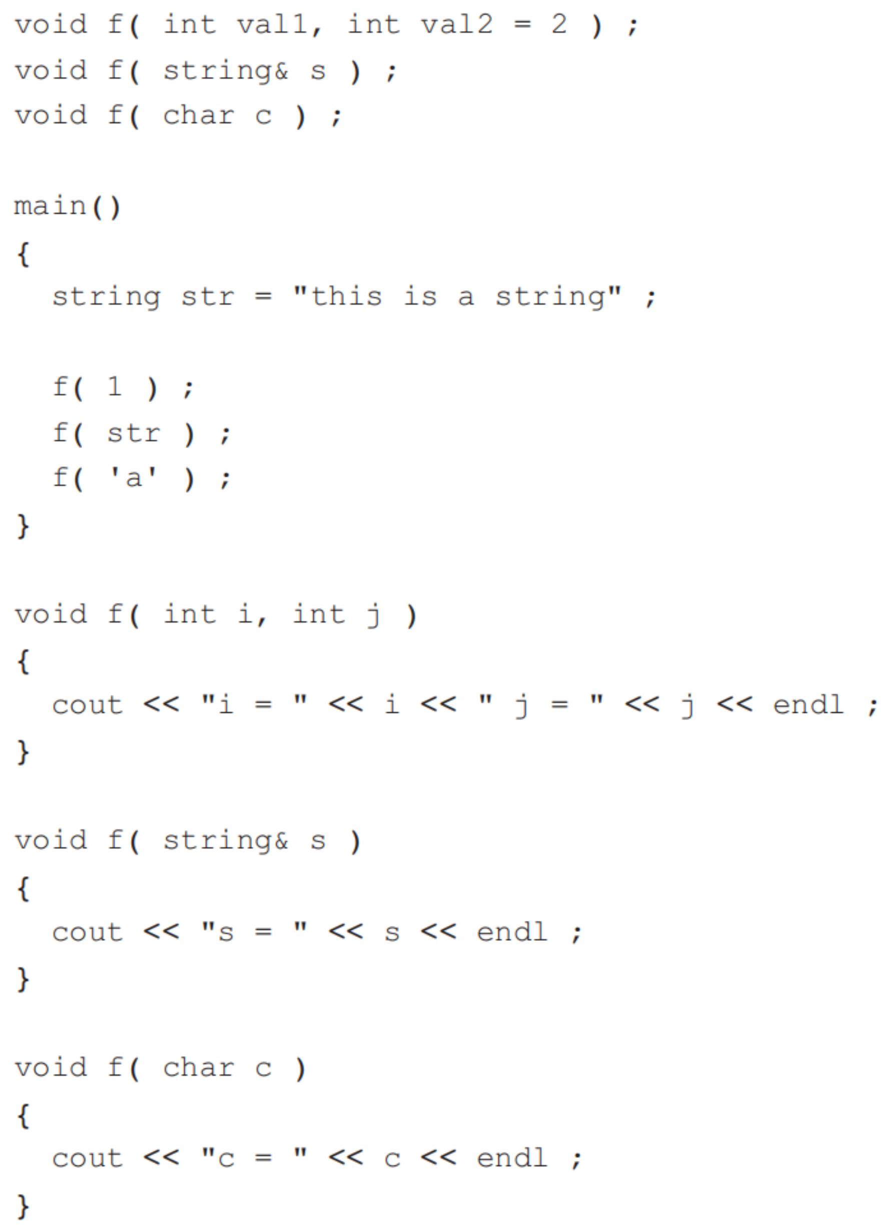
- 2. What is the output from the following?



```
#include <iostream>
using namespace std ;
int f( int val1, int val2 = 0 ) ;

main()
{
    int var ;

    var = f( 1, 2 ) + 1 ;
    var = f( var + 1 ) ;
    var = f( f( 1, 2 ), f( 3, var ) ) ;
    cout <<  " The value of var is  "  << var << endl ;
}

int f( int val1, int val2 )
{
    if ( val1 > val2 )
        return ( val1 - val2 ) ;
    else
        return ( val2 - val1 ) ;
}
```

Homework 8

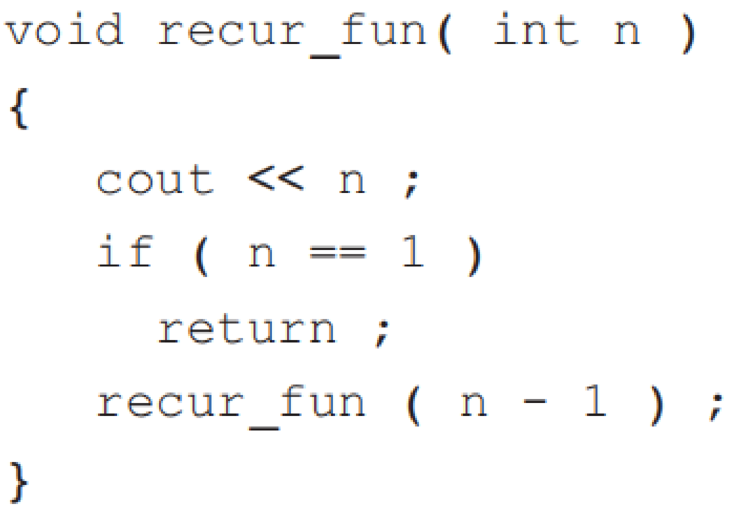
- 3. What is the output from the following?



```
void f( int val1, int val2 = 2 ) ;
void f( string& s ) ;
void f( char c ) ;

main()
{
  string str = "this is a string" ;

  f( 1 ) ;
  f( str ) ;
  f( 'a' ) ;
}

void f( int i, int j )
{
  cout << "i = " << i << " j = " << j << endl ;
}

void f( string& s )
{
  cout << "s = " << s << endl ;
}

void f( char c )
{
  cout << "c = " << c << endl ;
}
```

Homework 8

- 4. (a) Write a function to return the minimum value in an integer array. (b) Overload the function in (a) with a function to return the minimum value in a floating-point array.
- 5. What does this recursive function do?



```
void recur_fun( int n )
{
   cout << n ;
   if ( n == 1 )
     return ;
   recur_fun ( n - 1 ) ;
}
```