Tishk International University
Science Faculty
IT Department

Programming II - IT-118

Global variables - Pointers

1st Grade - Fall Semester

Lecture #3

Instructor: Hemin Ibrahim

Email: hemin.ibrahim@tiu.edu.iq

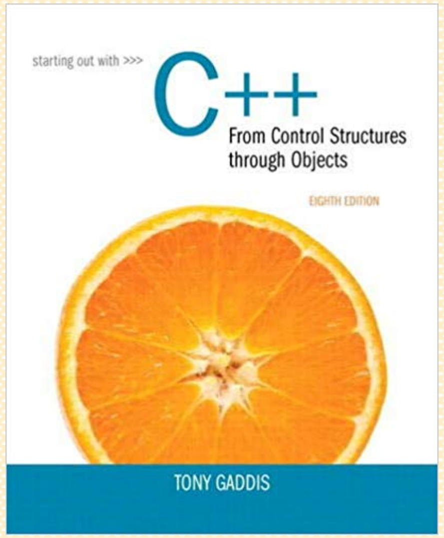
Overview

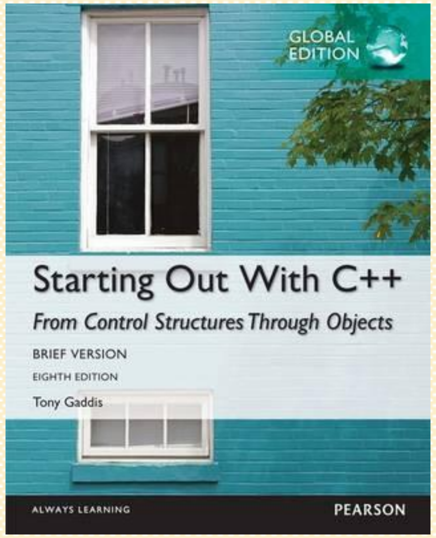
- ✓ Local & Global variables
- ✓ Introduction to Pointer
- ✓ Pointers and arrays
- ✓ Pointers and functions

Textbook Source

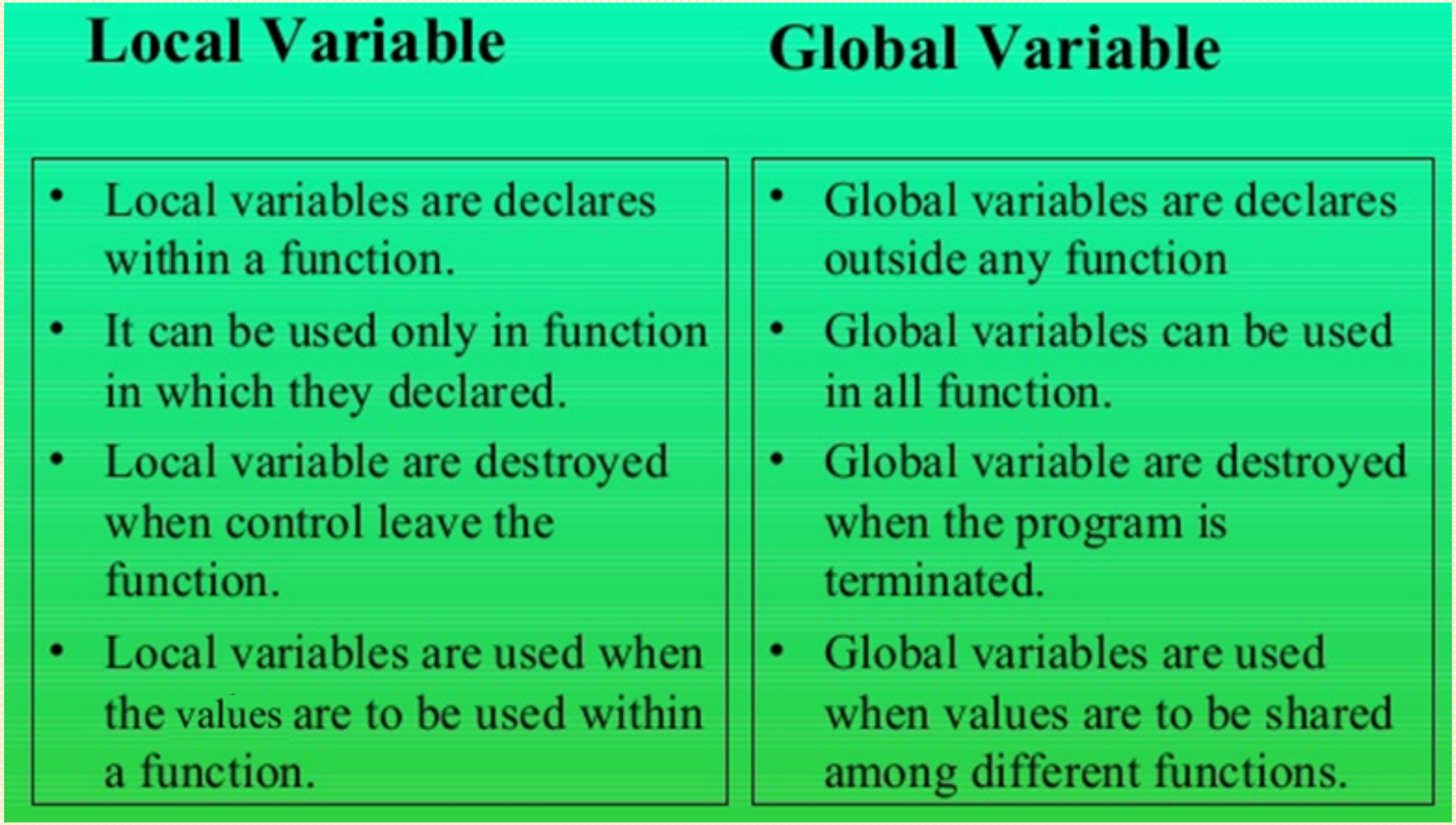
Tony Gaddis, Starting Out with C++: From Control Structures through Objects, 8th Edition

✓ **Chapter 8 (from page 305 to 363)**





Local and Global Variables

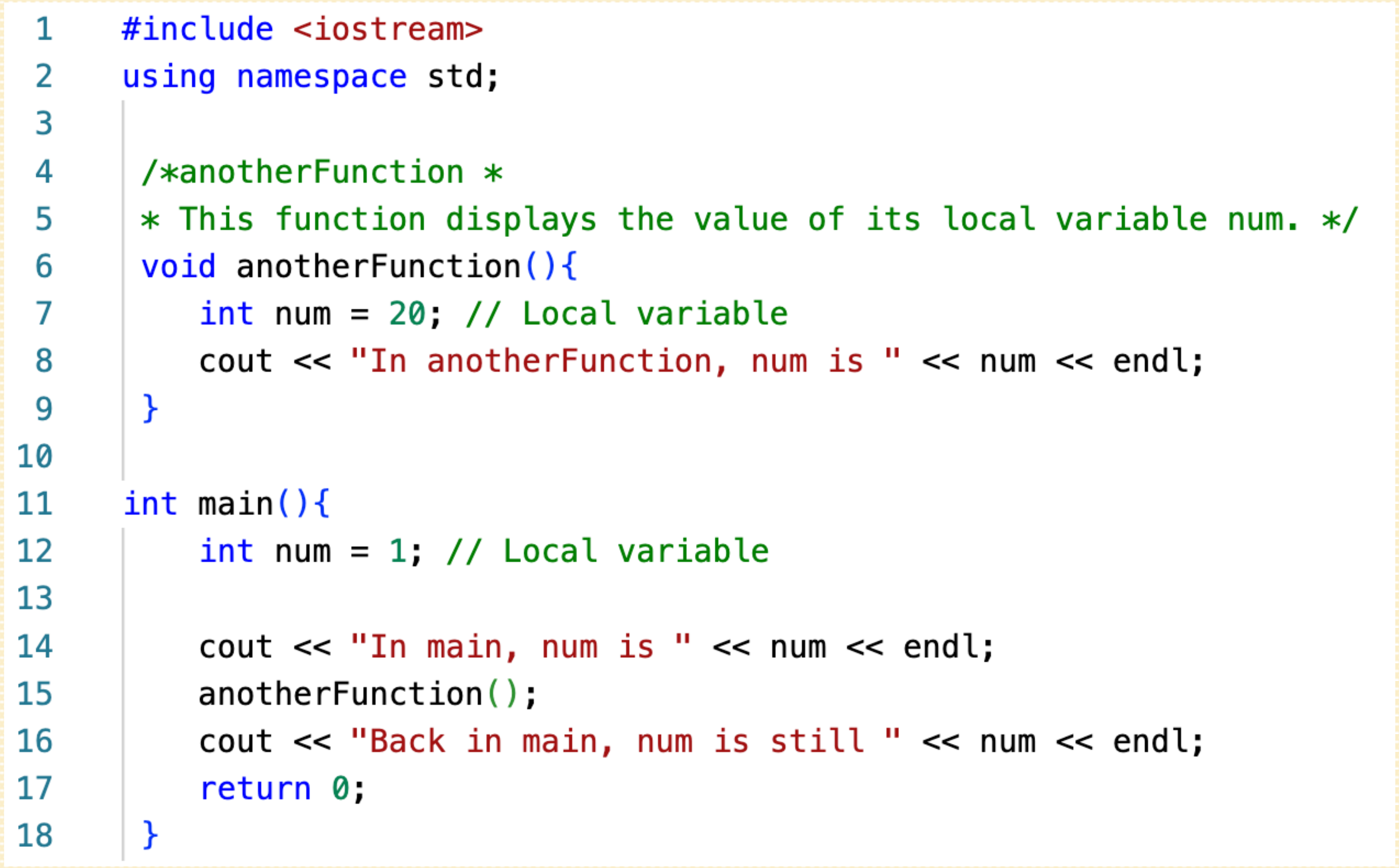


Local Variable	Global Variable
• Local variables are declares within a function.	• Global variables are declares outside any function
• It can be used only in function in which they declared.	• Global variables can be used in all function.
• Local variable are destroyed when control leave the function.	• Global variable are destroyed when the program is terminated.
• Local variables are used when the values are to be used within a function.	• Global variables are used when values are to be shared among different functions.

Local and Global Variable Names

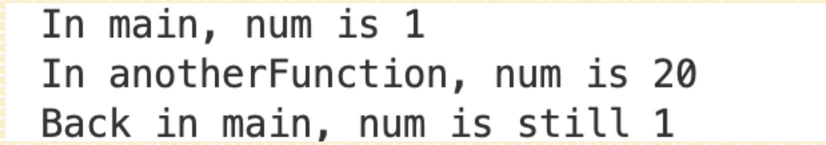
- Avoid using same names for Local and Global variables
- In case by mistake, a function contains a local variable that has the <u>same name as a global variable</u>, the global variable is **unavailable** from within the function.

Local Variable



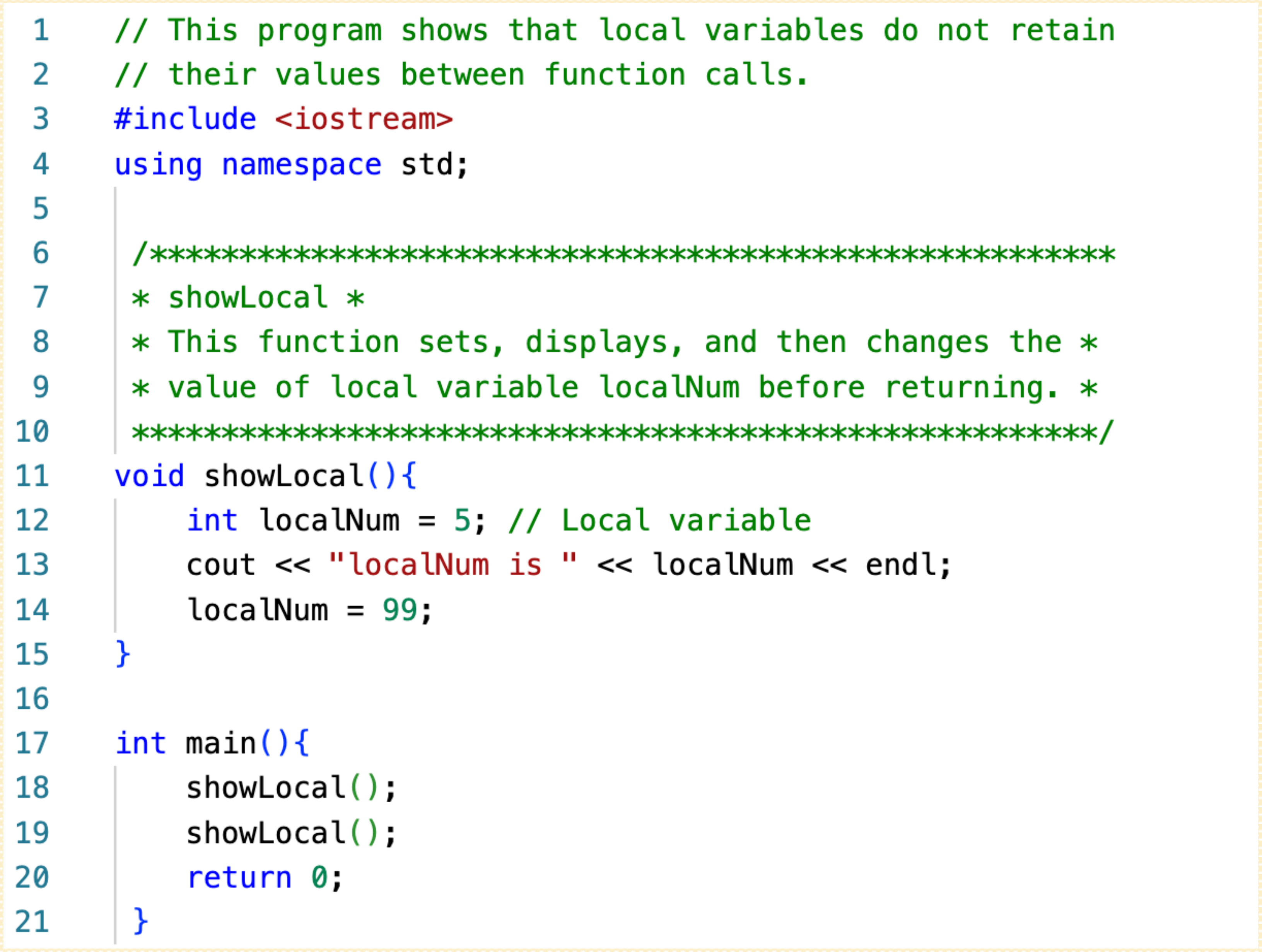
```cpp
#include <iostream>
using namespace std;

 /*anotherFunction *
 * This function displays the value of its local variable num. */
 void anotherFunction(){
    int num = 20; // Local variable
    cout << "In anotherFunction, num is " << num << endl;
 }

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

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

Output



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

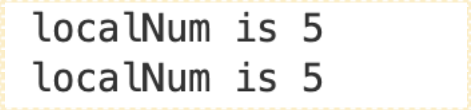
Local Variable



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

 /******************************************************
 * showLocal *
 * This function sets, displays, and then changes the *
 * value of local variable localNum before returning. *
 ******************************************************/
void showLocal(){
    int localNum = 5; // Local variable
    cout << "localNum is " << localNum << endl;
    localNum = 99;
}

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

local variables do not retain their values between function calls

Output



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

Static local variable

- A **static** local variable in C++ is only initialized once
- Declaring a local variable as `static` means it will *remember* it’s last value (it’s not destroyed and recreated each time it’s scope is entered).

Static local variable

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

 /*showStatic *
 * This function keeps track of how many times it *
 * has been called by incrementing a static local *
 * variable, numCalls, each time it is called. */
void showStatic(){
    static int numCalls = 0; // Static local variable
    cout << "This function has been called "
    << ++numCalls << " times. " << endl;
}

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

Output

```
This function has been called 1 times.
This function has been called 2 times.
This function has been called 3 times.
This function has been called 4 times.
This function has been called 5 times.
```

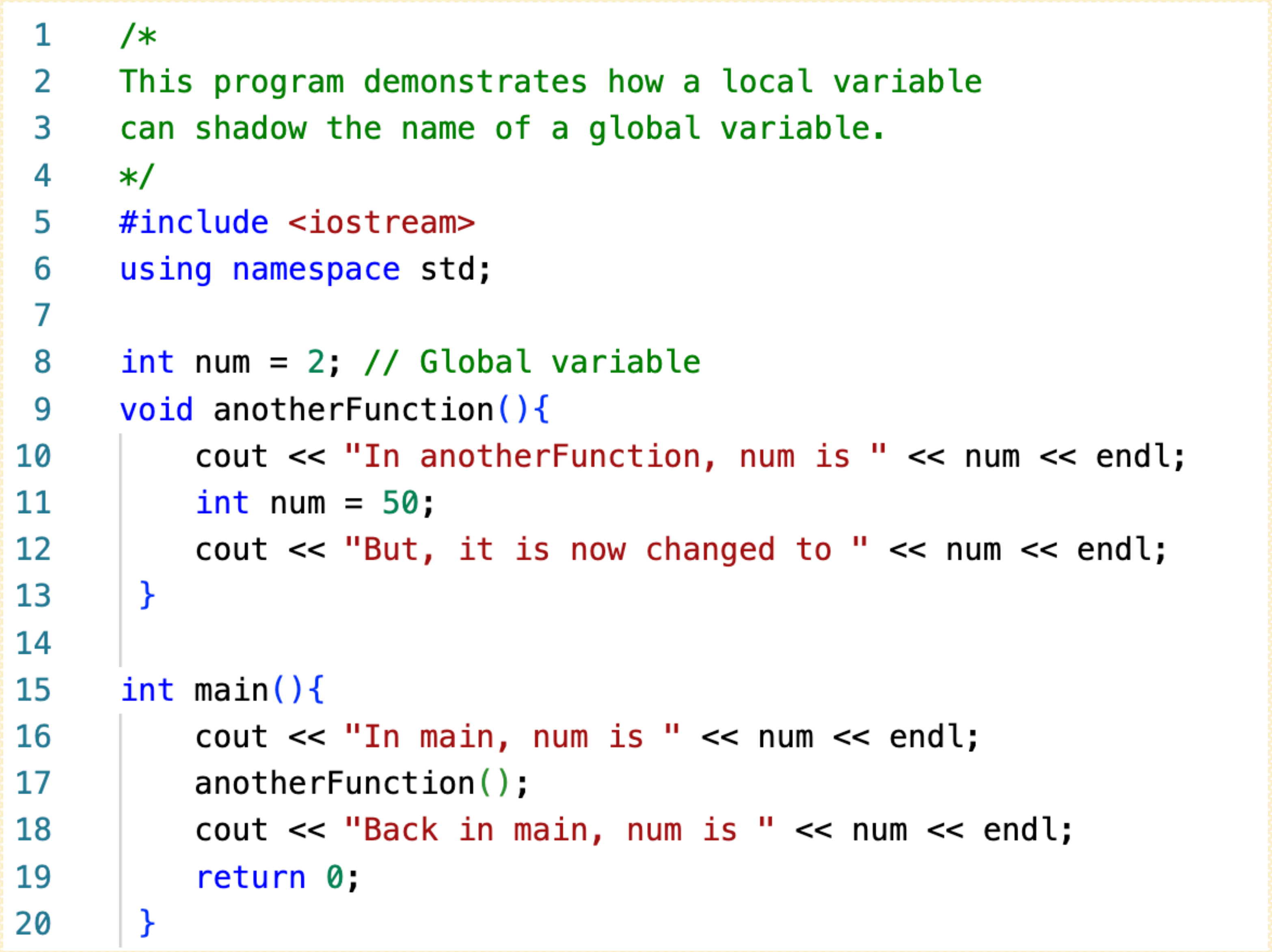
Global Variable

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

int num = 2; // Global variable

/*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;
}

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

Output

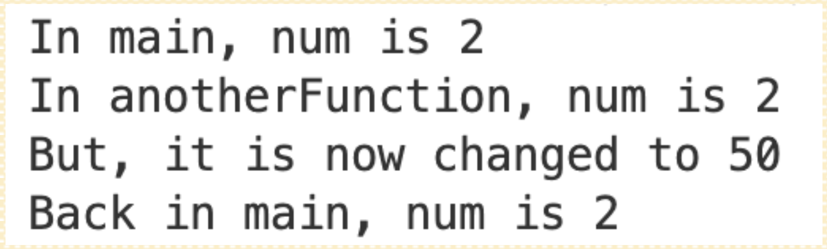
```
In main, num is 2
In anotherFunction, num is 2
But, it is now changed to 50
Back in main, num is 50
```

Shadowing or overwriting



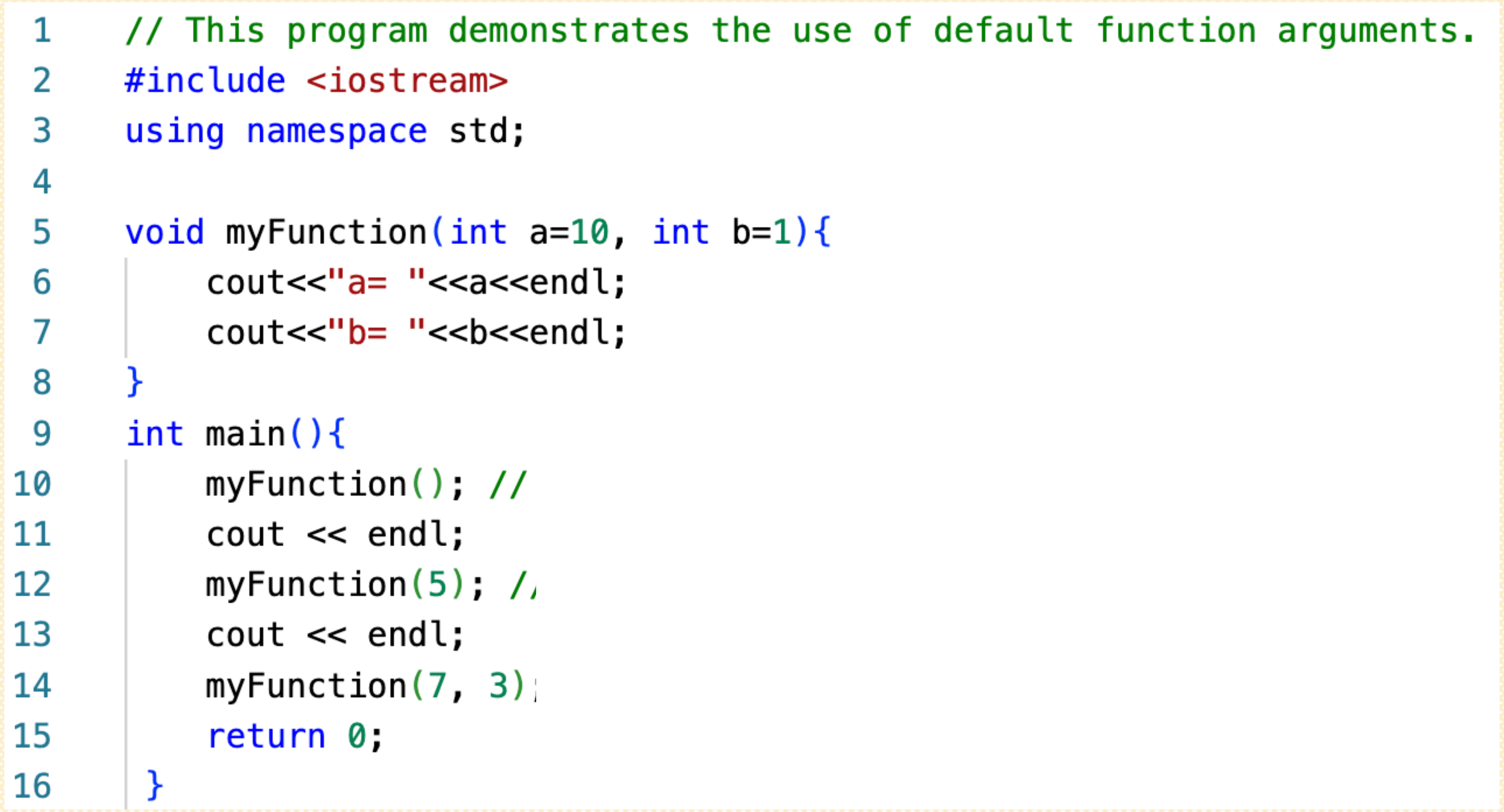
```cpp
/*
This program demonstrates how a local variable
can shadow the name of a global variable.
*/
#include <iostream>
using namespace std;

int num = 2; // Global variable
void anotherFunction(){
    cout << "In anotherFunction, num is " << num << endl;
    int num = 50;
    cout << "But, it is now changed to " << num << endl;
 }

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

Output



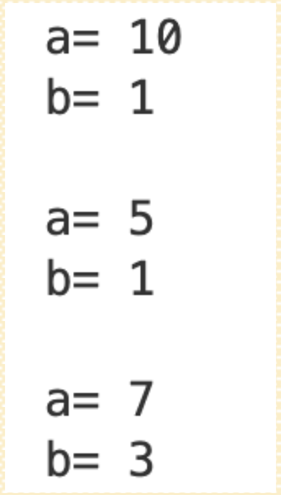
```
In main, num is 2
In anotherFunction, num is 2
But, it is now changed to 50
Back in main, num is 2
```

Default Parameters



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

void myFunction(int a=10, int b=1){
    cout<<"a= "<<a<<endl;
    cout<<"b= "<<b<<endl;
}
int main(){
    myFunction(); //
    cout << endl;
    myFunction(5); //
    cout << endl;
    myFunction(7, 3);
    return 0;
 }
```

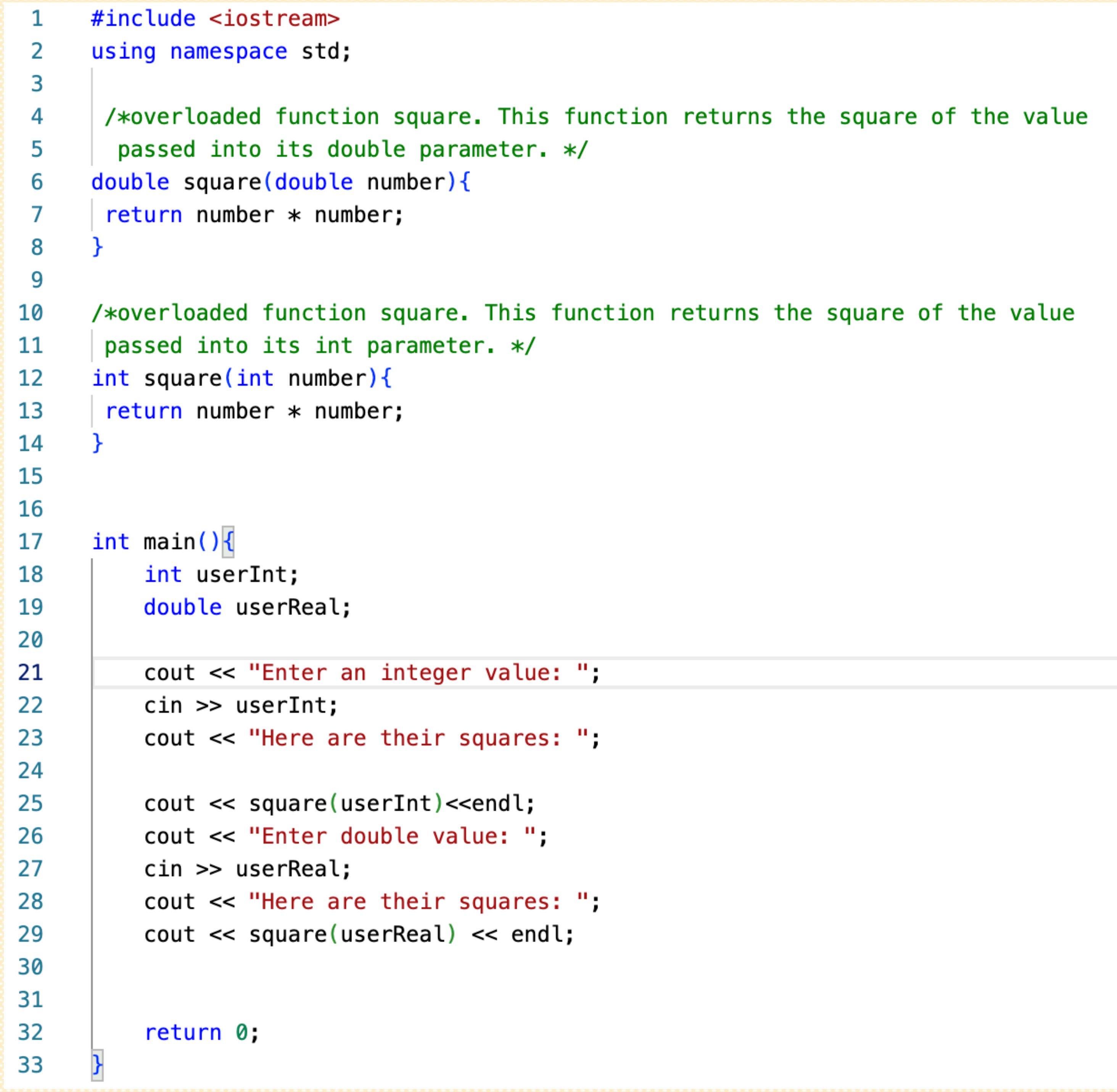
Output



```
a= 10
b= 1

a= 5
b= 1

a= 7
b= 3
```

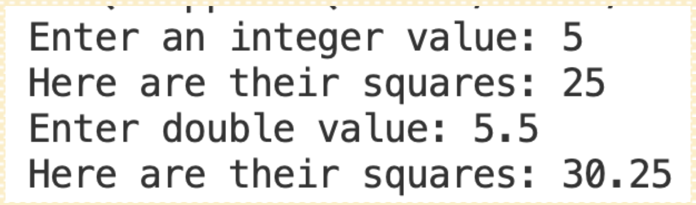
Overloading Function

Two or more functions may have the same name, as long as their parameter lists are different.



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

 /*overloaded function square. This function returns the square of the value
  passed into its double parameter. */
double square(double number){
 return number * number;
}

/*overloaded function square. This function returns the square of the value
 passed into its int parameter. */
int square(int number){
 return number * number;
}


int main(){
    int userInt;
    double userReal;

    cout << "Enter an integer value: ";
    cin >> userInt;
    cout << "Here are their squares: ";

    cout << square(userInt)<<endl;
    cout << "Enter double value: ";
    cin >> userReal;
    cout << "Here are their squares: ";
    cout << square(userReal) << endl;


    return 0;
}
```

Output



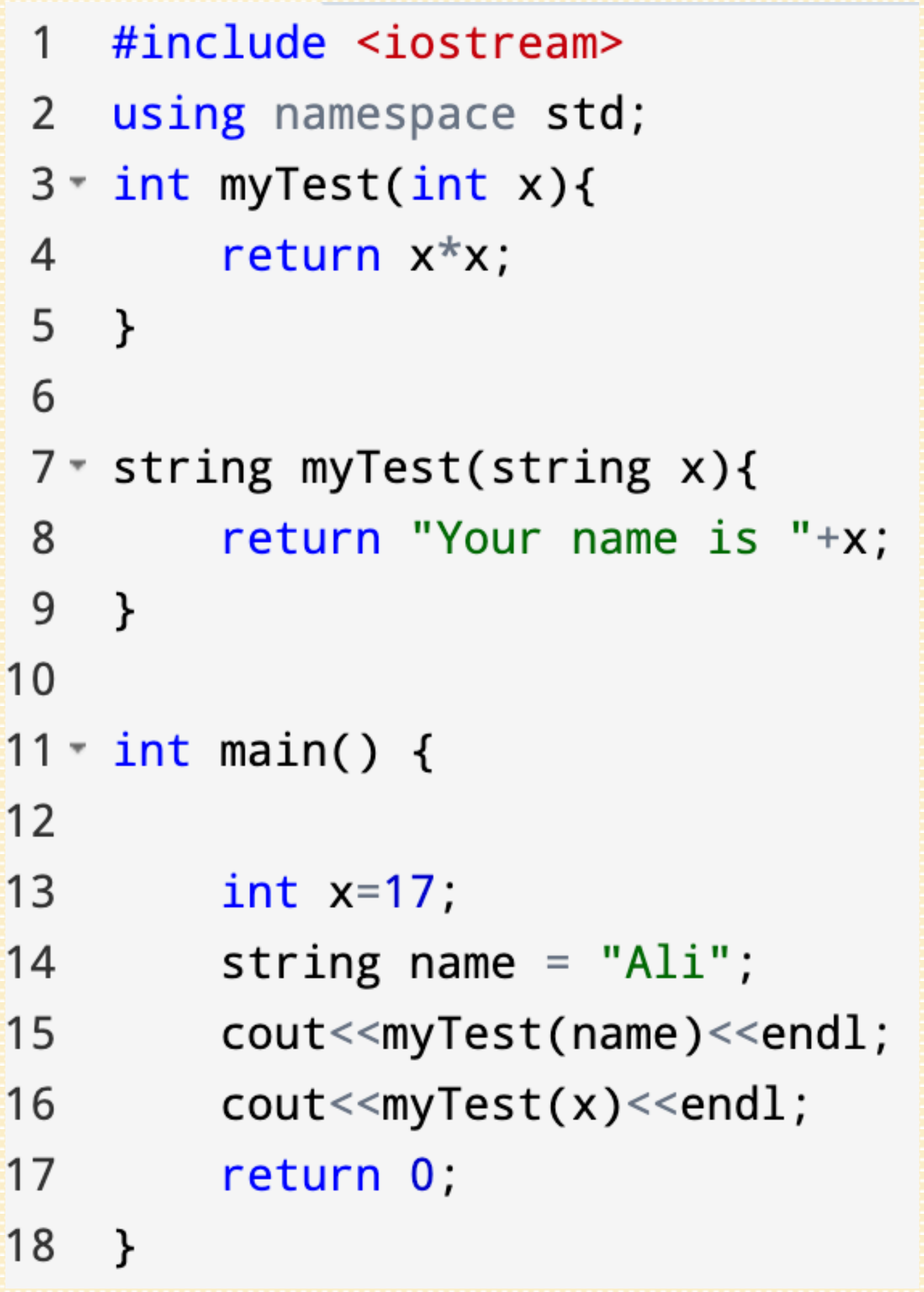
```
Enter an integer value: 5
Here are their squares: 25
Enter double value: 5.5
Here are their squares: 30.25
```

Overloading Function

- How does the compiler know which version of "square() " to call?
  - It looks at the arguments passed in each case.

  - then It looks to find the right version of square() that match the correct argument.

Overloading Function



```cpp
#include <iostream>
using namespace std;
int myTest(int x){
    return x*x;
}

string myTest(string x){
    return "Your name is "+x;
}

int main() {

    int x=17;
    string name = "Ali";
    cout<<myTest(name)<<endl;
    cout<<myTest(x)<<endl;
    return 0;
}
```

Pointer

Pointer

- A **pointer** is a variable that holds the memory address of another variable of same type.
- OR
- A **pointer** in C++, is a data type that holds the memory address of another variable.
- Pointers allow you to manipulate and access memory locations, which is useful for tasks like passing variables by reference and working with data structures.
- Pointers provide the right access to data by accessing to memory access, rather than copying data between variables.

Address in C++

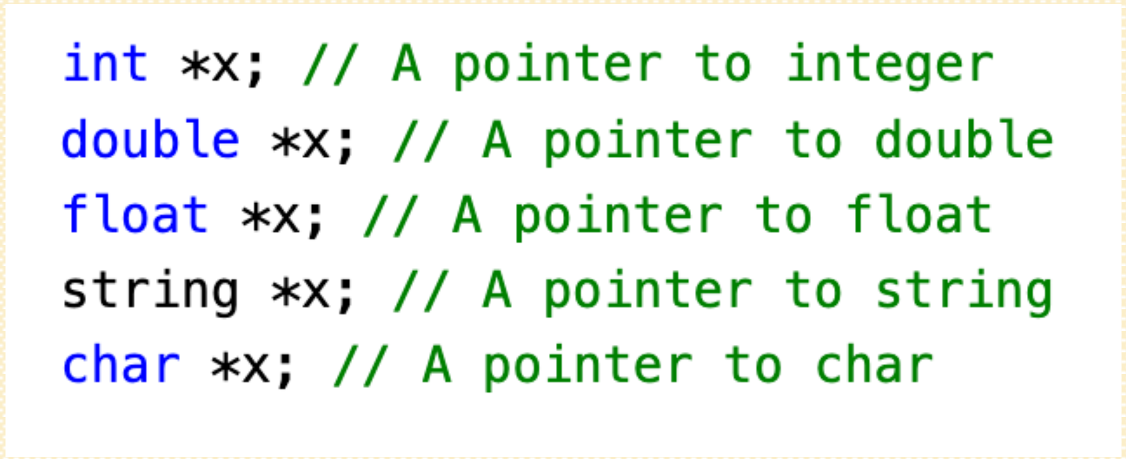
- To understand C++ pointer, we must understand how computers store data.
- When the variable is created in C++, it is assigned a space in computer **memory**. The **value** of this variable is stored in the assigned location.
- To know the location in the computer memory, C++ provides the & (reference) operator. This operator return the address that this variable stored.

Declaring Pointer Variables

❑ Pointers declared like other types
  - Add "*" before variable name
  - Produces "pointer to" that type

❑ "*" must be before each variable

- **int *ptr;**
- ptr holds pointers to int variables
  (i.e. ptr is a pointer to an integer)



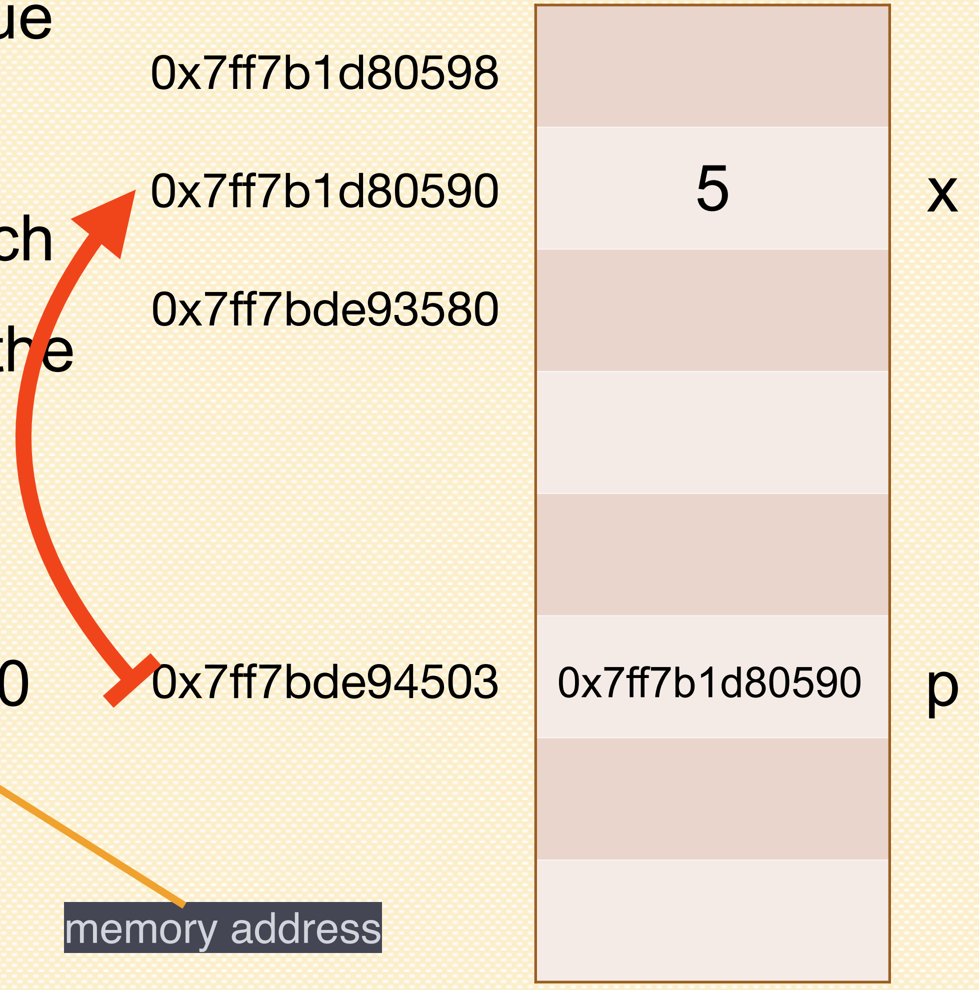
```
int *x; // A pointer to integer
double *x; // A pointer to double
float *x; // A pointer to float
string *x; // A pointer to string
char *x; // A pointer to char
```

Memory address

- In a computer system, memory address is a unique identifier that refers to a specific location in the system's memory such as RAM or hard drive. Each memory address represents a unique location in the memory hierarchy.

```
int x = 5;
cout<<&x<<endl;
int p=&x; ✗
int *p=&x; ✓
cout<<p<<endl;
```

Output: 0x7ff7b1d80590

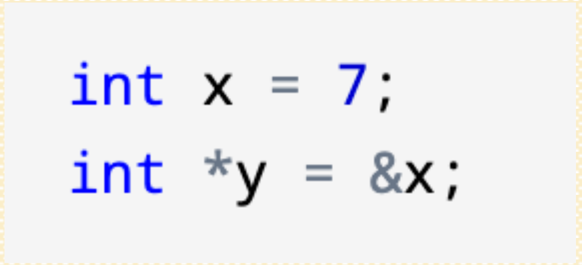


Declaring Pointer Variables

- Sets pointer variable **y** to "point to" int variable **x**
- Reference operator, & (ampersand): Determines "address of" variable
- Read like

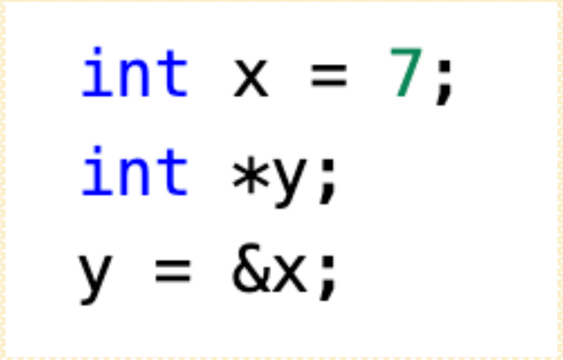
  y equals address of **x**

  Or "**y** points to **x**"



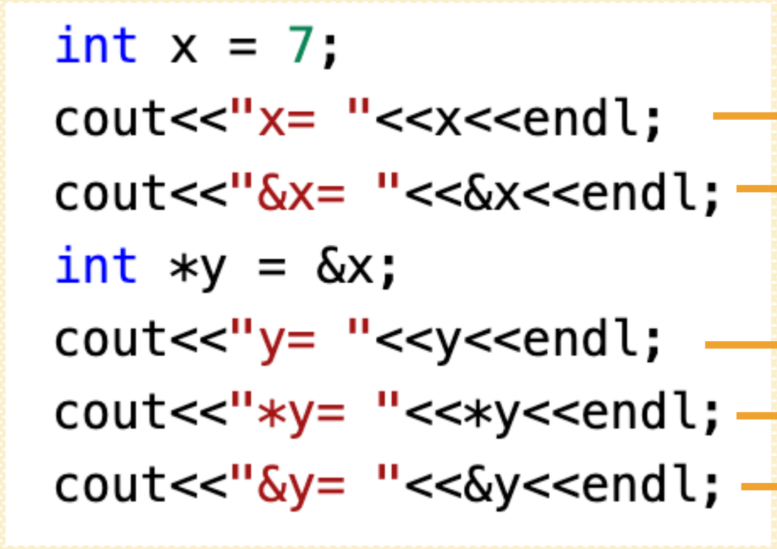
```
int x = 7;
int *y = &x;
```

Same



```
int x = 7;
int *y;
y = &x;
```

Declaring Pointer Variables - example

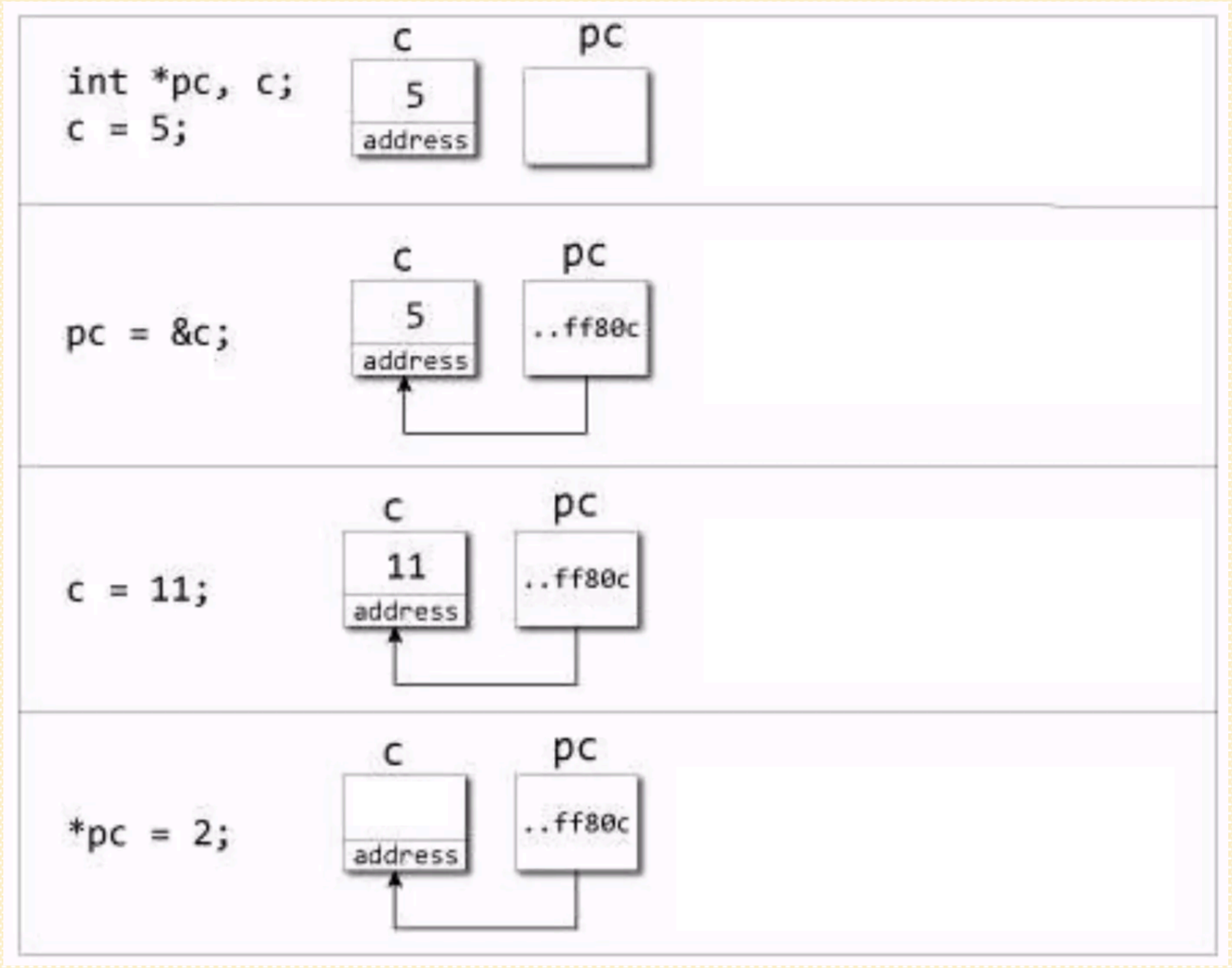


```cpp
int x = 7;
cout<<"x= "<<x<<endl;
cout<<"&x= "<<&x<<endl;
int *y = &x;
cout<<"y= "<<y<<endl;
cout<<"*y= "<<*y<<endl;
cout<<"&y= "<<&y<<endl;
```

Output

```
x= 7
&x= 0x7ff7b9fe4598
y= 0x7ff7b9fe4598
*y= 7
&y= 0x7ff7b9fe4590
```

Declaring Pointer Variables - example



```
int *pc, c;
c = 5;
```

```
pc = &c;
```

```
c = 11;
```

```
*pc = 2;
```

Reference operator (&) and Dereference Operator (*)

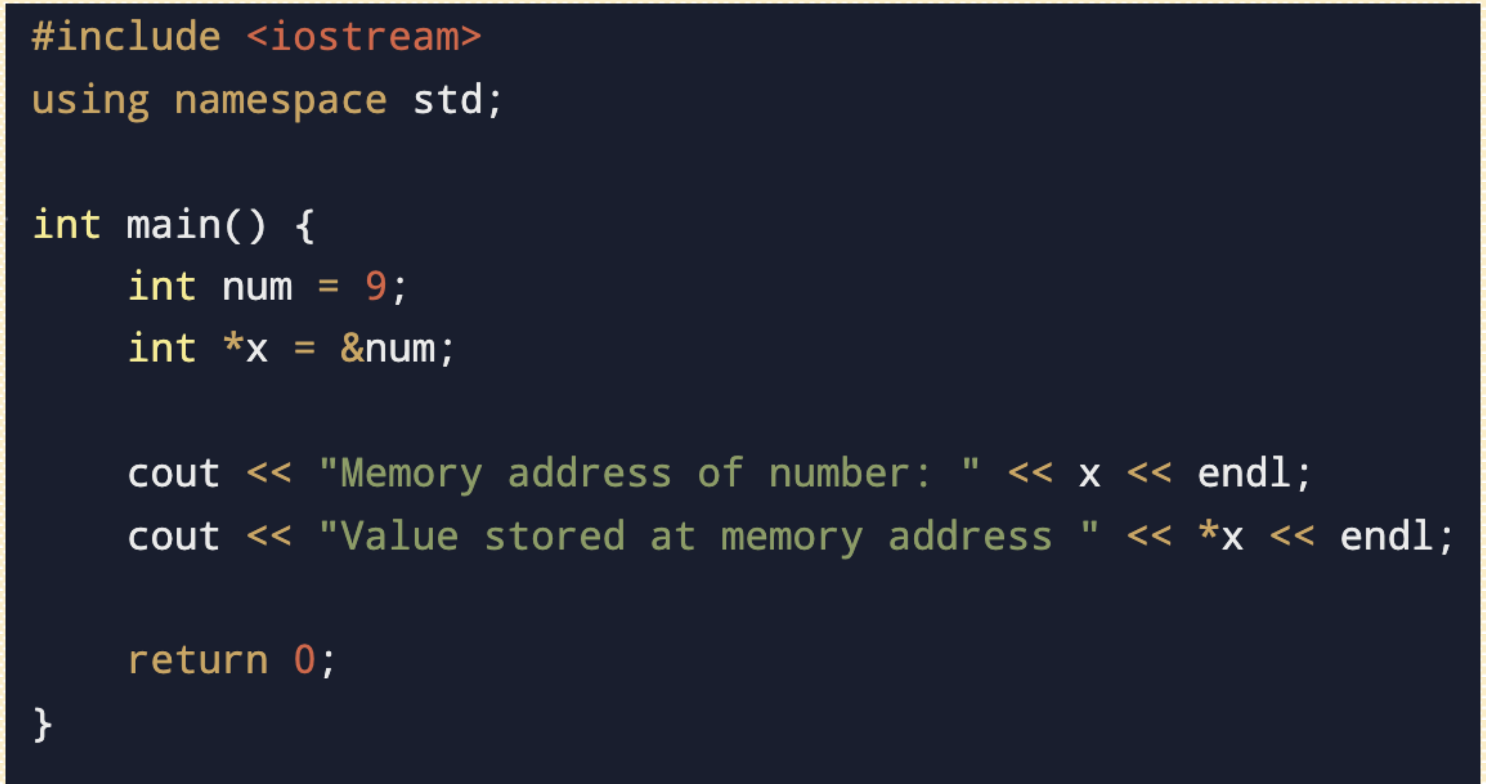
- **&** (ampersand) operator return the address of the variable.
- ***** operator return the value that stored in the memory.

- **For example:** If a number variable is stored in the memory address **`0x7ff7b3a74598`** and it contains a value 7.
- The reference (&) operator gives the value **`0x7ff7b3a74598`**. While the dereference operator (*) gives the value 7.

Reference operator (&) and Dereference Operator (*)

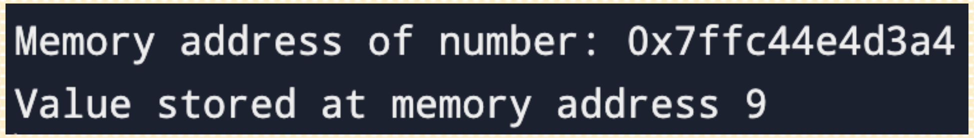
- **&** operator return the address of the variable.
- * operator return the value that stored in the memory.

- **For example:** If a number variable is stored in the memory address `0x7ff7b3a74598` and it contains a value 7.
- The reference (&) operator gives the value `0x7ff7b3a74598`. While the dereference operator (*) gives the value 7.



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

int main() {
    int num = 9;
    int *x = &num;

    cout << "Memory address of number: " << x << endl;
    cout << "Value stored at memory address " << *x << endl;

    return 0;
}
```

Output



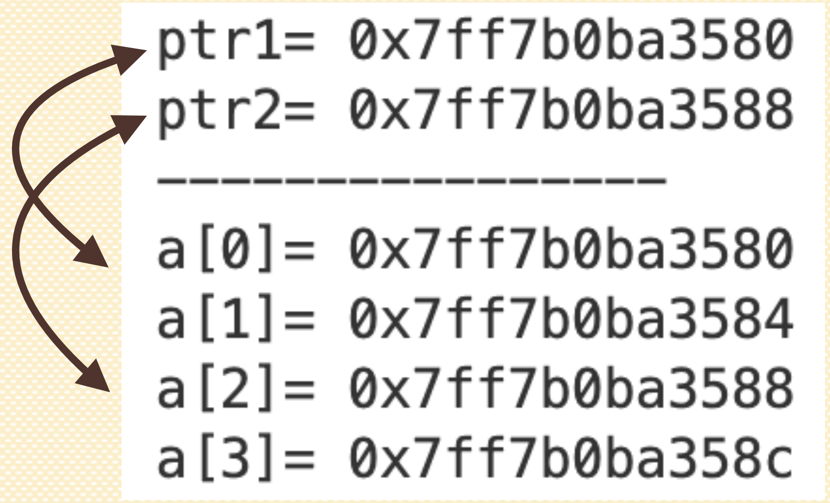
```
Memory address of number: 0x7ffc44e4d3a4
Value stored at memory address 9
```

Pointers and arrays

- Pointers are more efficient in handling array.
- Pointers provide an alternative way to access array's elements.

```cpp
int *ptr1, *ptr2;
int a[4];
ptr1 = a;
ptr2 = &a[2];

cout<<"ptr1= "<<ptr1<<endl; // pointer to a[0]
cout<<"ptr2= "<<ptr2<<endl; // pointer to a[2]
cout<<"----------------"<<endl;
for(int i=0;i<4;i++){
    cout<<"a["<<i<<"]= "<<&a[i]<<endl;
}
```

Output

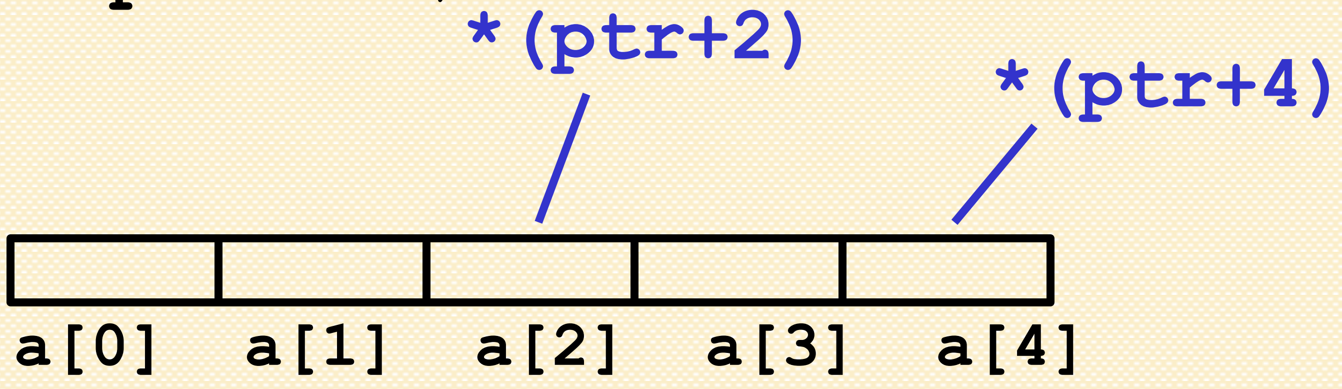


```
ptr1= 0x7ff7b0ba3580
ptr2= 0x7ff7b0ba3588
----------------
a[0]= 0x7ff7b0ba3580
a[1]= 0x7ff7b0ba3584
a[2]= 0x7ff7b0ba3588
a[3]= 0x7ff7b0ba358c
```

Pointer arithmetic

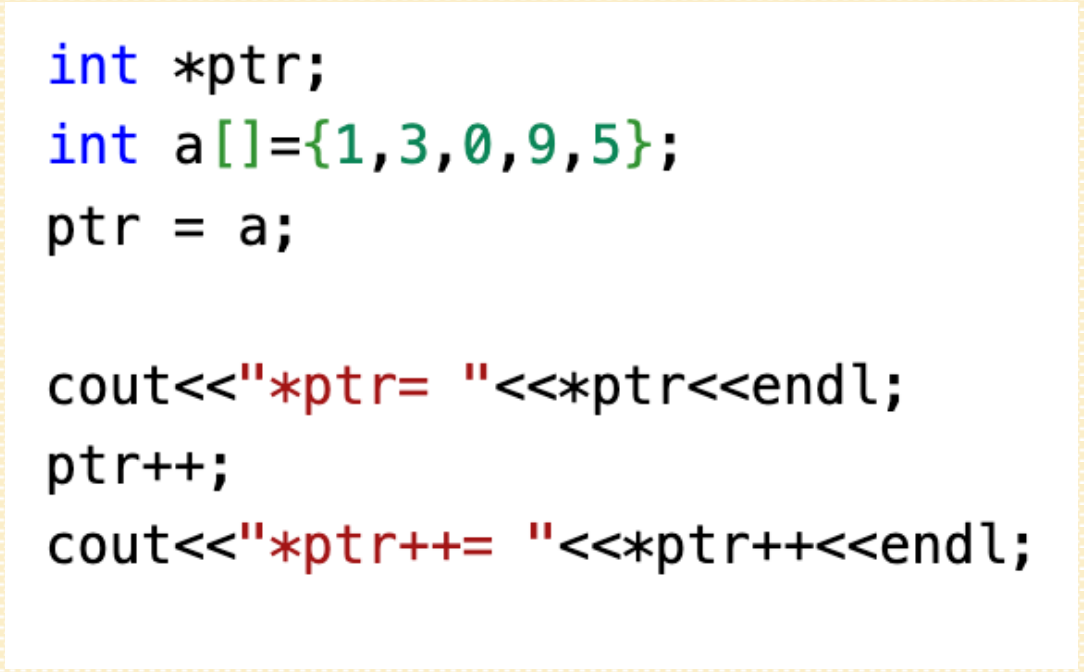
- Integer math operations can be used with pointers.
- If you increment a pointer, it will be increased by the size of whatever it points to

```
int *ptr = a;
```



```
int a[5];
```

Pointer arithmetic - Example

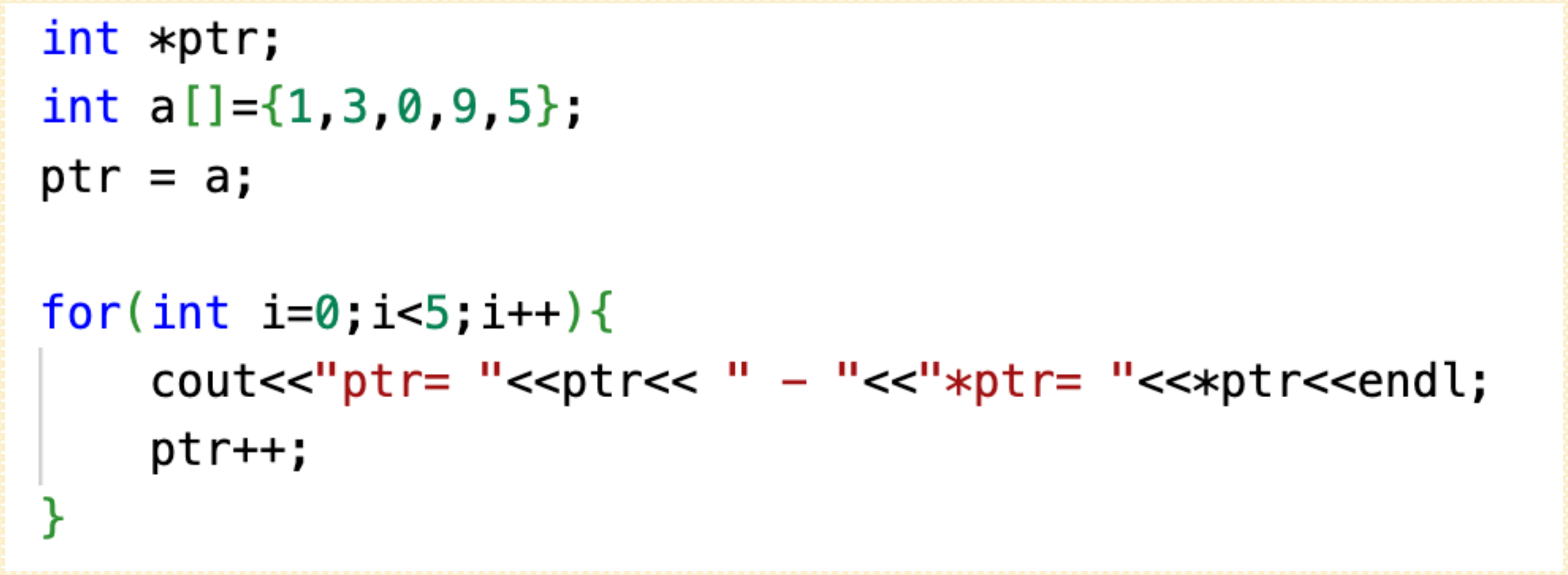


```
int *ptr;
int a[]={1,3,0,9,5};
ptr = a;

cout<<"*ptr= "<<*ptr<<endl;
ptr++;
cout<<"*ptr++= "<<*ptr++<<endl;
```

Output:

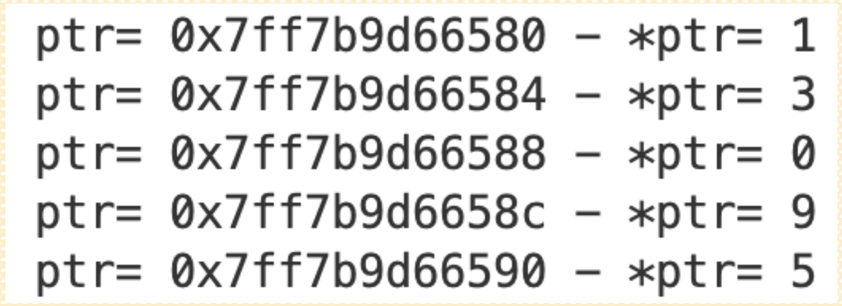
```
*ptr= 1
*ptr++= 3
```

Pointer arithmetic - Example



```
int *ptr;
int a[]={1,3,0,9,5};
ptr = a;

for(int i=0;i<5;i++){
    cout<<"ptr= "<<ptr<< " – "<<"*ptr= "<<*ptr<<endl;
    ptr++;
}
```

Output



```
ptr= 0x7ff7b9d66580 – *ptr= 1
ptr= 0x7ff7b9d66584 – *ptr= 3
ptr= 0x7ff7b9d66588 – *ptr= 0
ptr= 0x7ff7b9d6658c – *ptr= 9
ptr= 0x7ff7b9d66590 – *ptr= 5
```

Pointers and functions

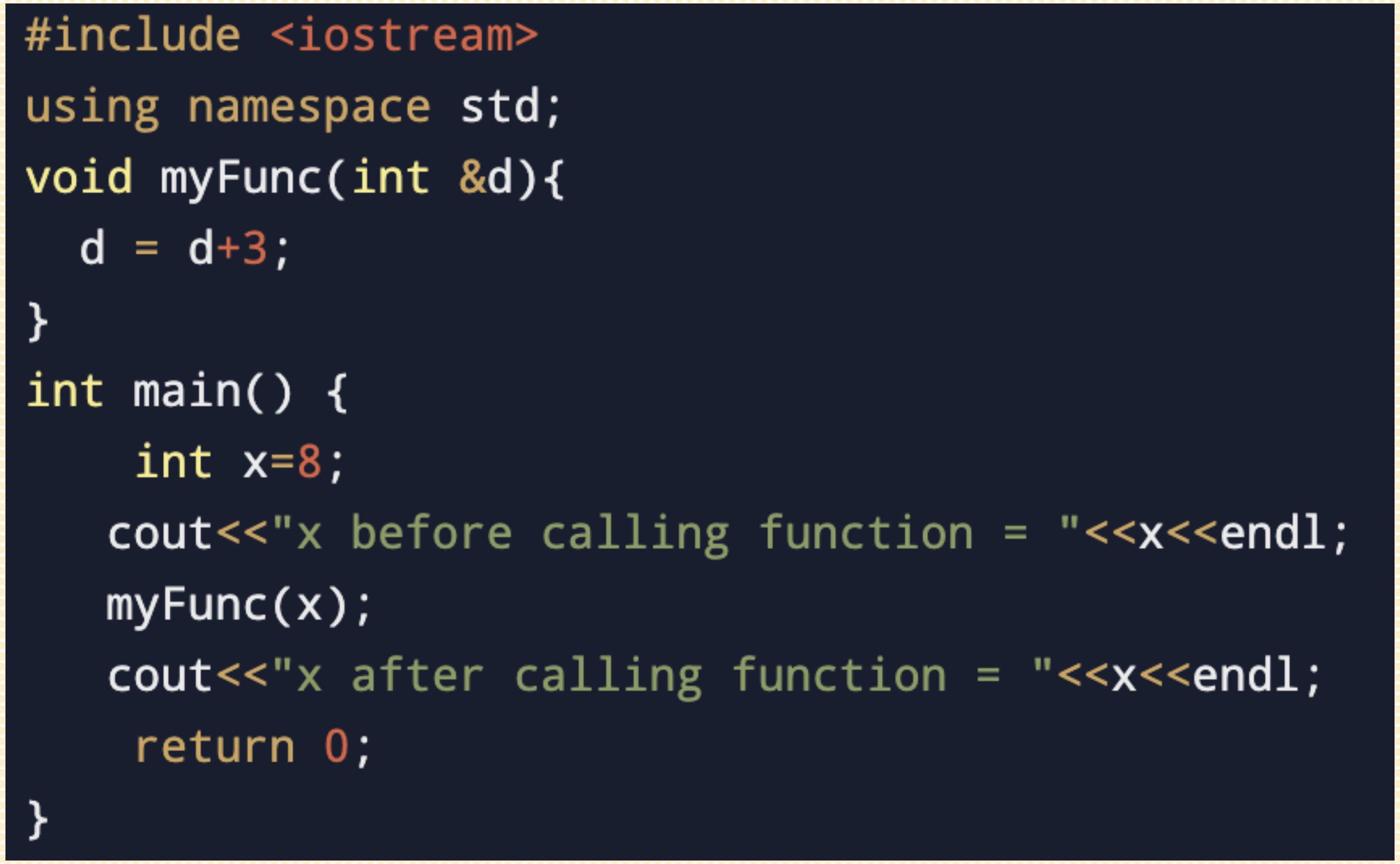
- Pointers variable can pass as a function argument and function can return pointer.
- There are two approaches to passing argument to a function:
  - Call by value (Previous lecture)
  - Call by reference

Pass by Reference

- Mechanism that allows a function to work with the original variable from the function call, not a copy of the value
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to ‘return’ more than 1 value

Pass by Reference

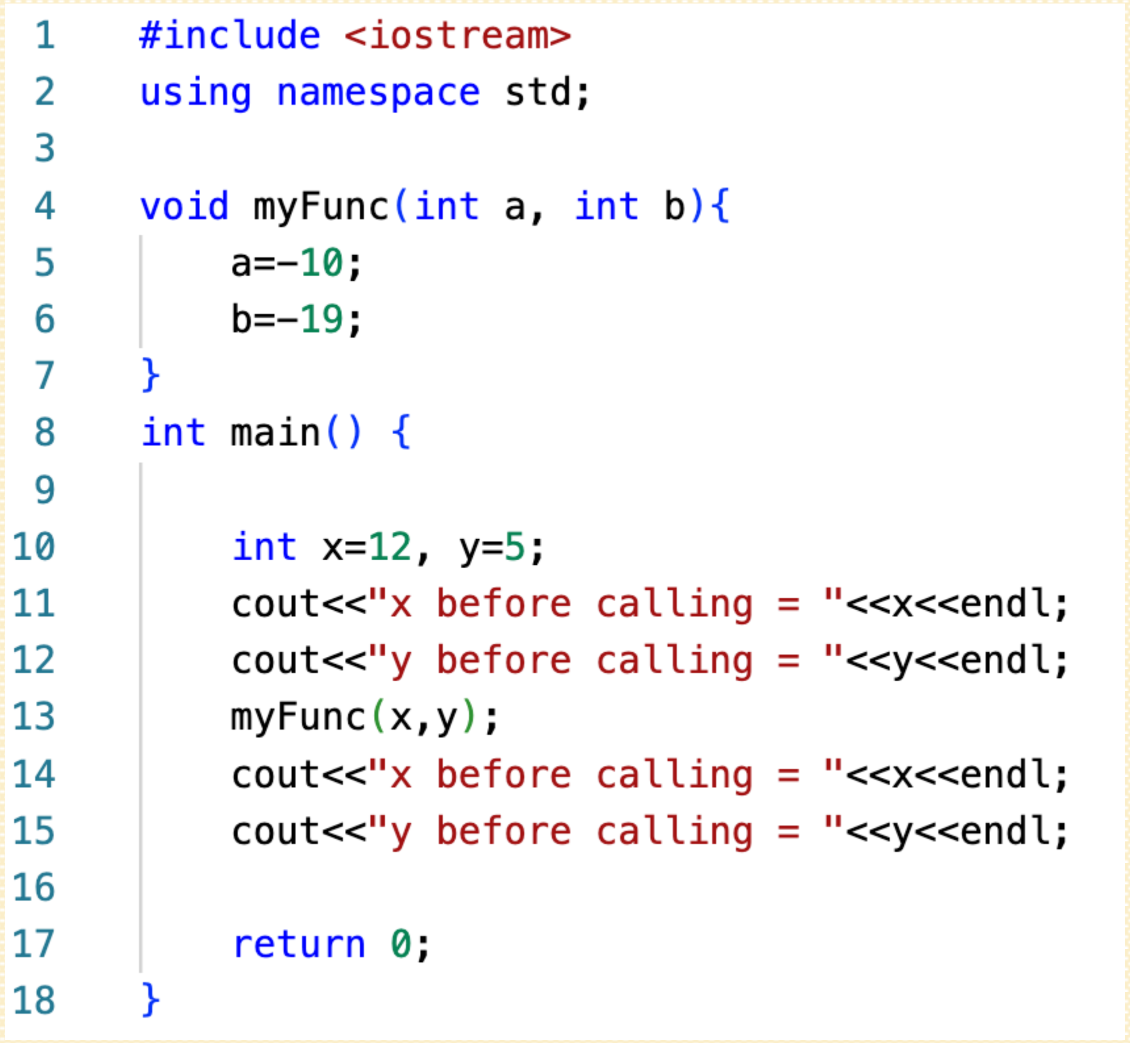
- Pass-by-Reference: Pointers can be used to pass variables to functions by reference, which means that the function can modify the variable's value directly in memory. This can be useful for functions that need to modify the input variables.



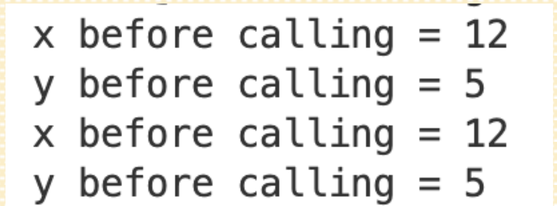
```
#include <iostream>
using namespace std;
void myFunc(int &d){
  d = d+3;
}
int main() {
    int x=8;
   cout<<"x before calling function = "<<x<<endl;
   myFunc(x);
   cout<<"x after calling function = "<<x<<endl;
    return 0;
}
```

```
x before calling function = 8

x after calling function = 11
```

Call by value

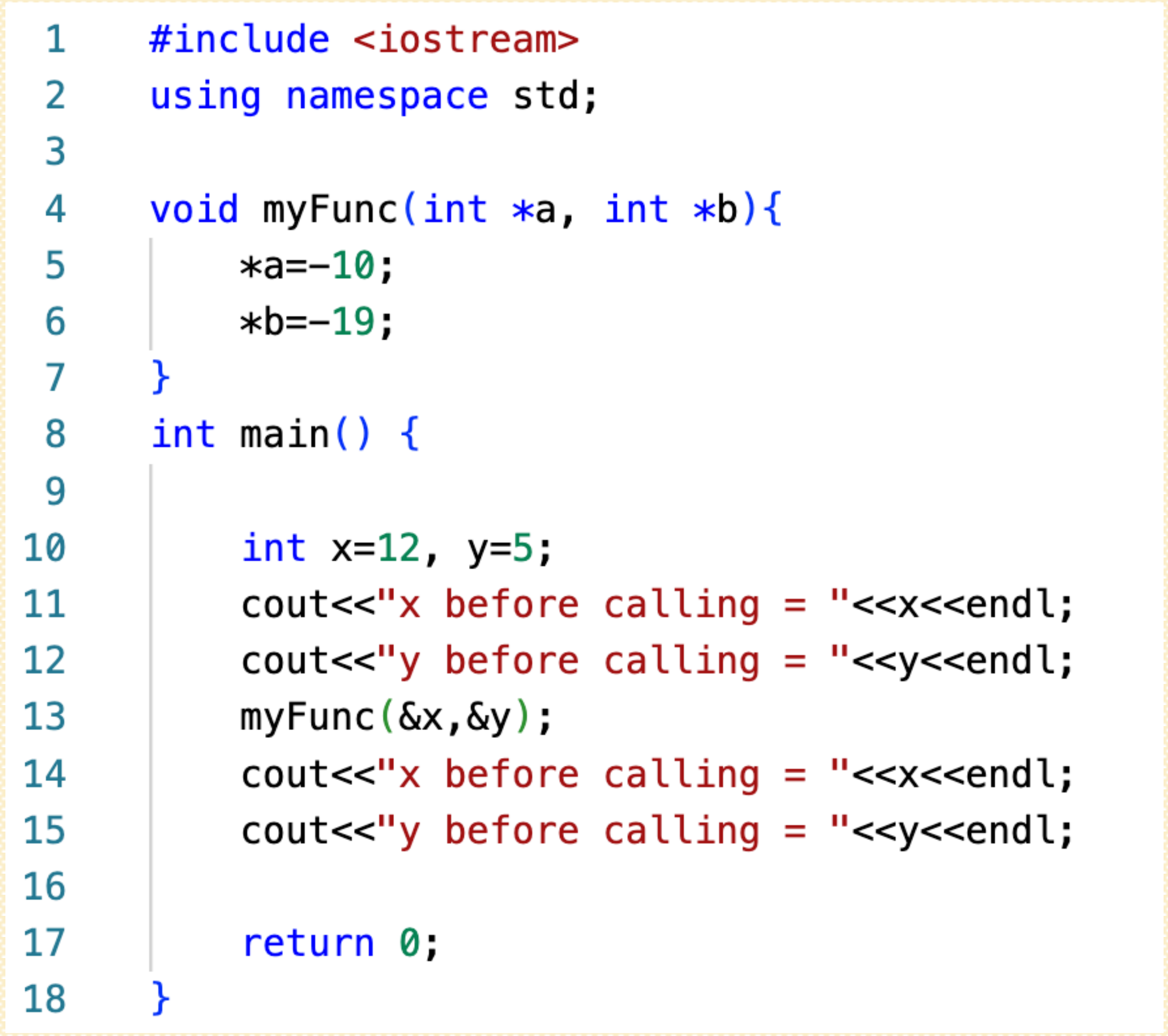


```cpp
#include <iostream>
using namespace std;

void myFunc(int a, int b){
    a=-10;
    b=-19;
}
int main() {

    int x=12, y=5;
    cout<<"x before calling = "<<x<<endl;
    cout<<"y before calling = "<<y<<endl;
    myFunc(x,y);
    cout<<"x before calling = "<<x<<endl;
    cout<<"y before calling = "<<y<<endl;

    return 0;
}
```

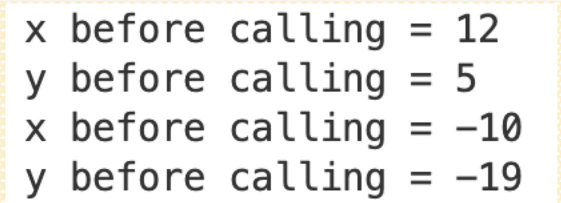


```
x before calling = 12
y before calling = 5
x before calling = 12
y before calling = 5
```

Call by reference

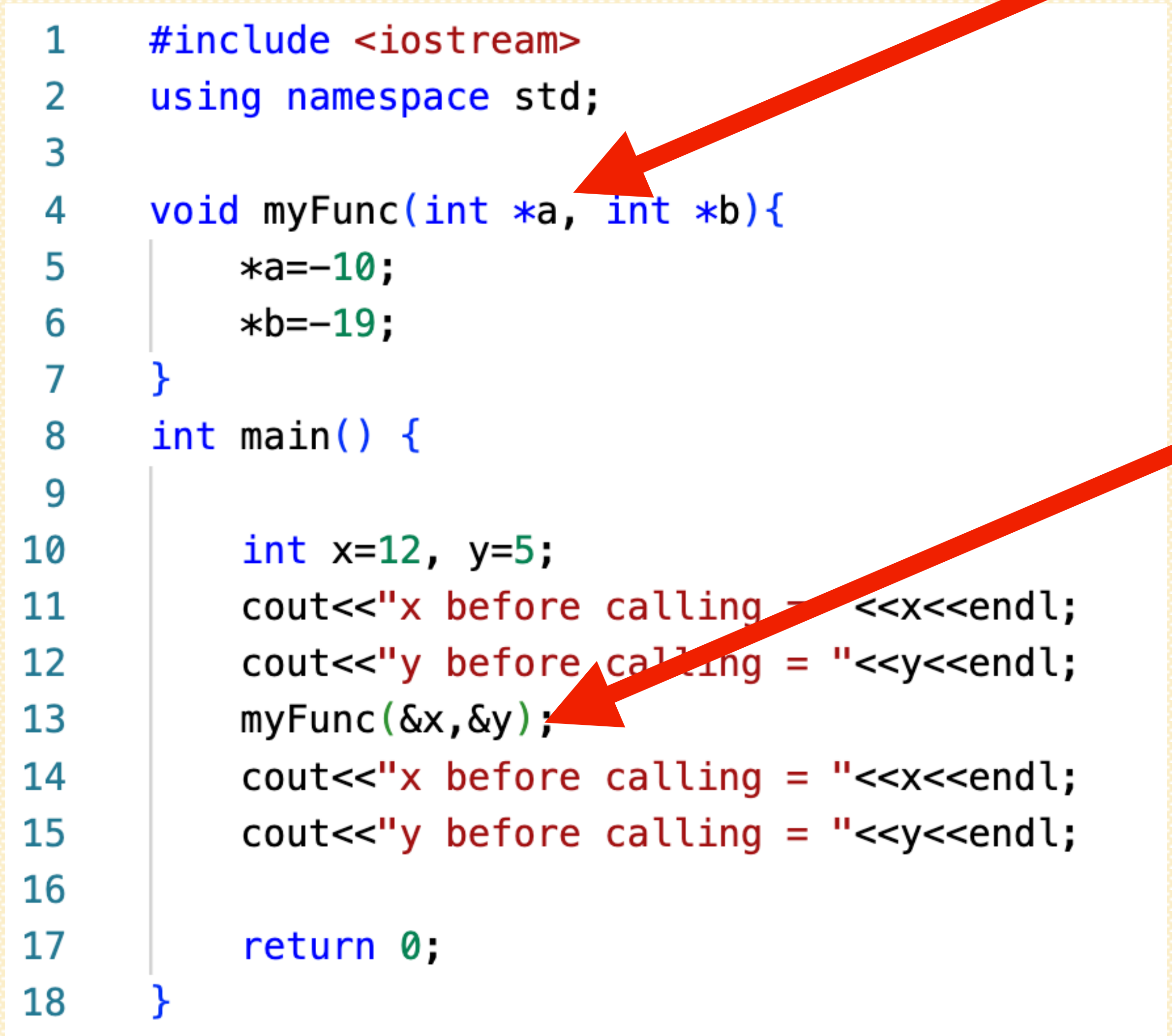


```cpp
#include <iostream>
using namespace std;

void myFunc(int *a, int *b){
    *a=-10;
    *b=-19;
}
int main() {

    int x=12, y=5;
    cout<<"x before calling = "<<x<<endl;
    cout<<"y before calling = "<<y<<endl;
    myFunc(&x,&y);
    cout<<"x before calling = "<<x<<endl;
    cout<<"y before calling = "<<y<<endl;

    return 0;
}
```

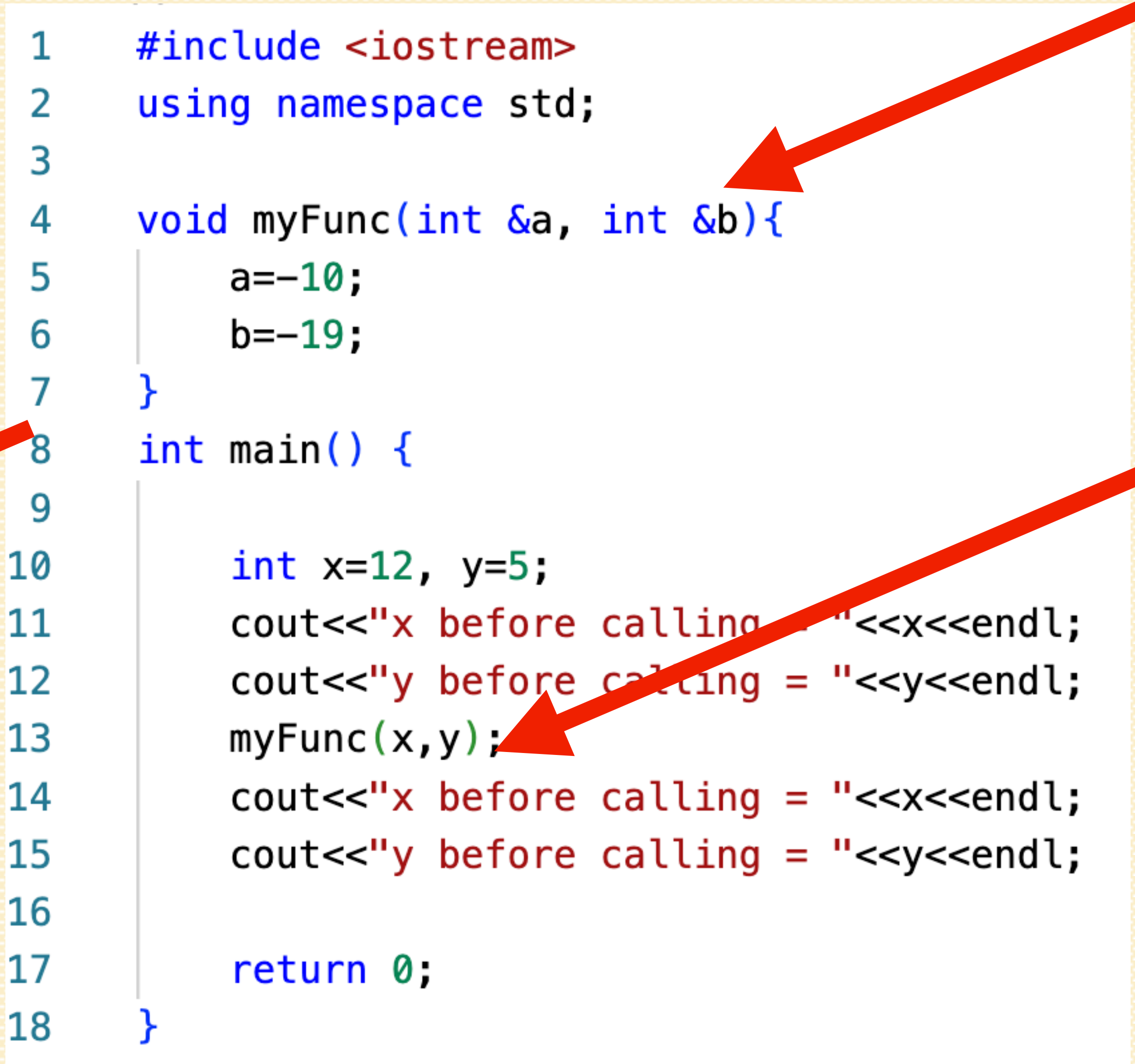


```
x before calling = 12
y before calling = 5
x before calling = -10
y before calling = -19
```

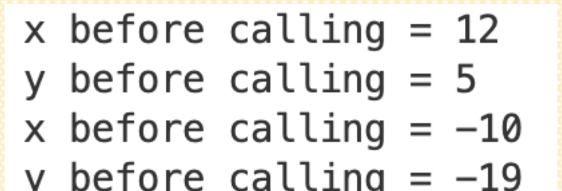
They are the same



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

void myFunc(int *a, int *b){
    *a=-10;
    *b=-19;
}
int main() {

    int x=12, y=5;
    cout<<"x before calling = "<<x<<endl;
    cout<<"y before calling = "<<y<<endl;
    myFunc(&x,&y);
    cout<<"x before calling = "<<x<<endl;
    cout<<"y before calling = "<<y<<endl;

    return 0;
}
```



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

void myFunc(int &a, int &b){
    a=-10;
    b=-19;
}
int main() {

    int x=12, y=5;
    cout<<"x before calling = "<<x<<endl;
    cout<<"y before calling = "<<y<<endl;
    myFunc(x,y);
    cout<<"x before calling = "<<x<<endl;
    cout<<"y before calling = "<<y<<endl;

    return 0;
}
```



```
x before calling = 12
y before calling = 5
x before calling = -10
y before calling = -19
```

Swap Function with **Reference Variables**

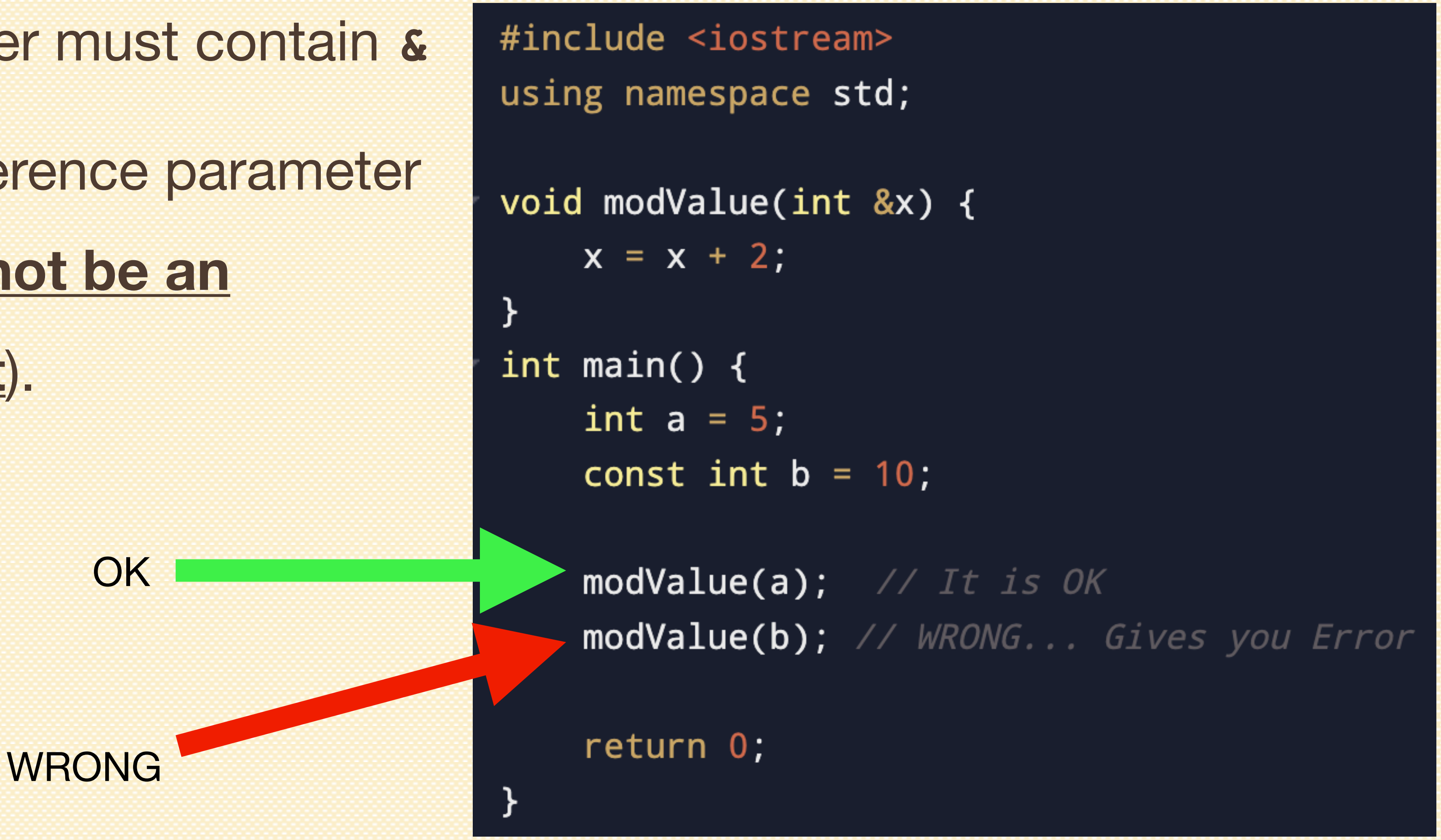
```cpp
#include <iostream>
using namespace std;
void swap_func(int &a, int &b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}
int main(){

    int v = 5, x = 10;
    cout << "v= "<<v << ", x= " << x << endl;
    swap_func(v,x);
    cout << "v= "<<v << ", x= " << x << endl;

    return 0;
}
```

Output

```
v= 5, x= 10
v= 10, x= 5
```

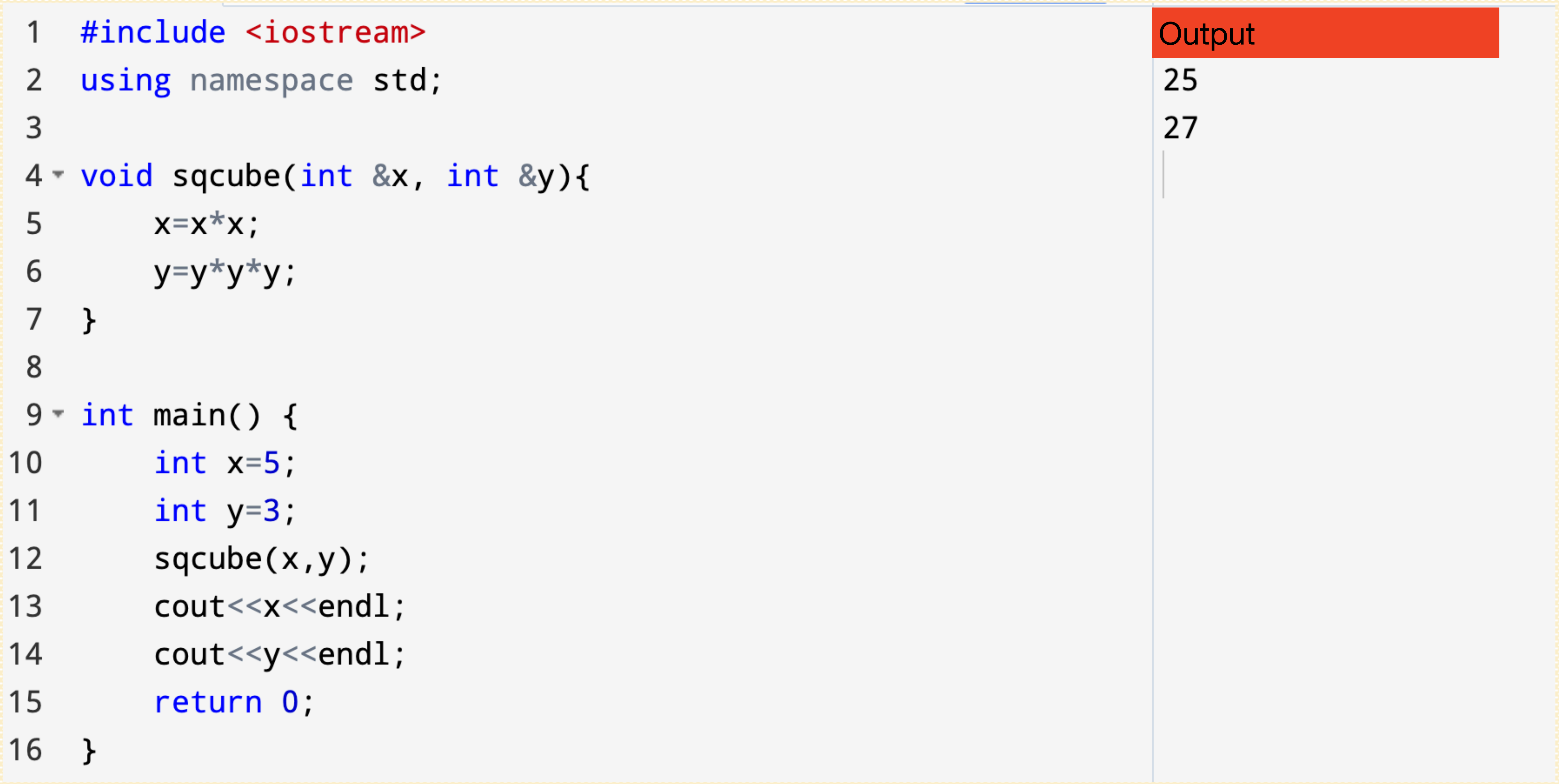
Reference Variable Notes

- Each reference parameter must contain **&**
- Argument passed to reference parameter must be a variable (**<u>cannot be an expression or constant</u>**).



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

void modValue(int &x) {
    x = x + 2;
}
int main() {
    int a = 5;
    const int b = 10;

    modValue(a);  // It is OK
    modValue(b); // WRONG... Gives you Error

    return 0;
}
```

Returning Multiple Values using Pointers



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

void sqcube(int &x, int &y){
     x=x*x;
     y=y*y*y;
}

int main() {
     int x=5;
     int y=3;
     sqcube(x,y);
     cout<<x<<endl;
     cout<<y<<endl;
     return 0;
}
```

```
25
27
```

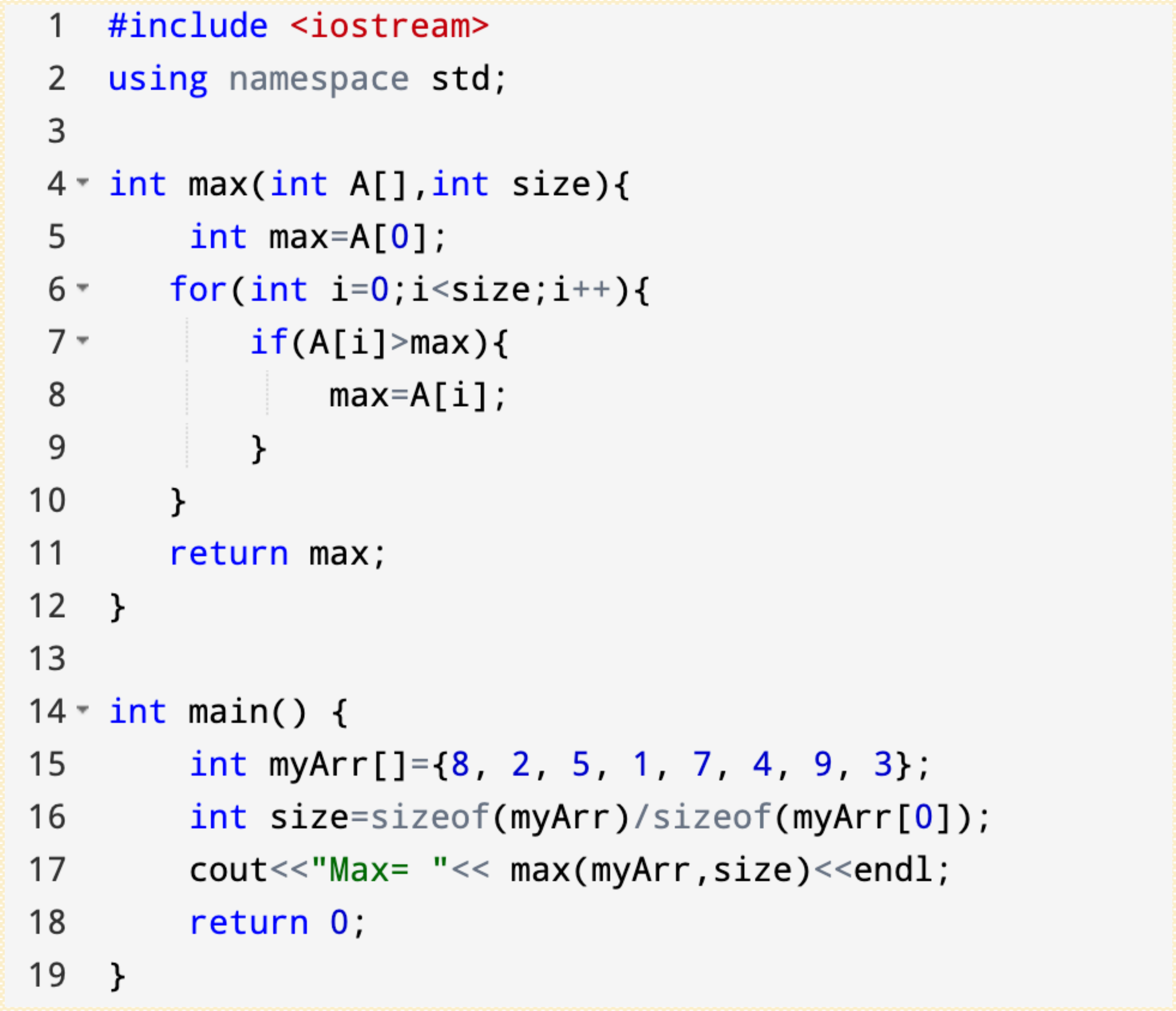
Exercise #1

find the maximum numbers in the following array, using function.

myArr[]={8, 2, 5, 1, 7, 4, 9, 3};

Exercise #1

find the maximum numbers in the following array, using function.
myArr[]={8, 2, 5, 1, 7, 4, 9, 3};



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

int max(int A[],int size){
     int max=A[0];
    for(int i=0;i<size;i++){
        if(A[i]>max){
            max=A[i];
        }
    }
    return max;
}

int main() {
     int myArr[]={8, 2, 5, 1, 7, 4, 9, 3};
     int size=sizeof(myArr)/sizeof(myArr[0]);
     cout<<"Max= "<< max(myArr,size)<<endl;
     return 0;
}
```

Exercise #2

find the minimum and maximum numbers in the following array, using function.

myArr[]={8, 2, 5, 1, 7, 4, 9, 3};

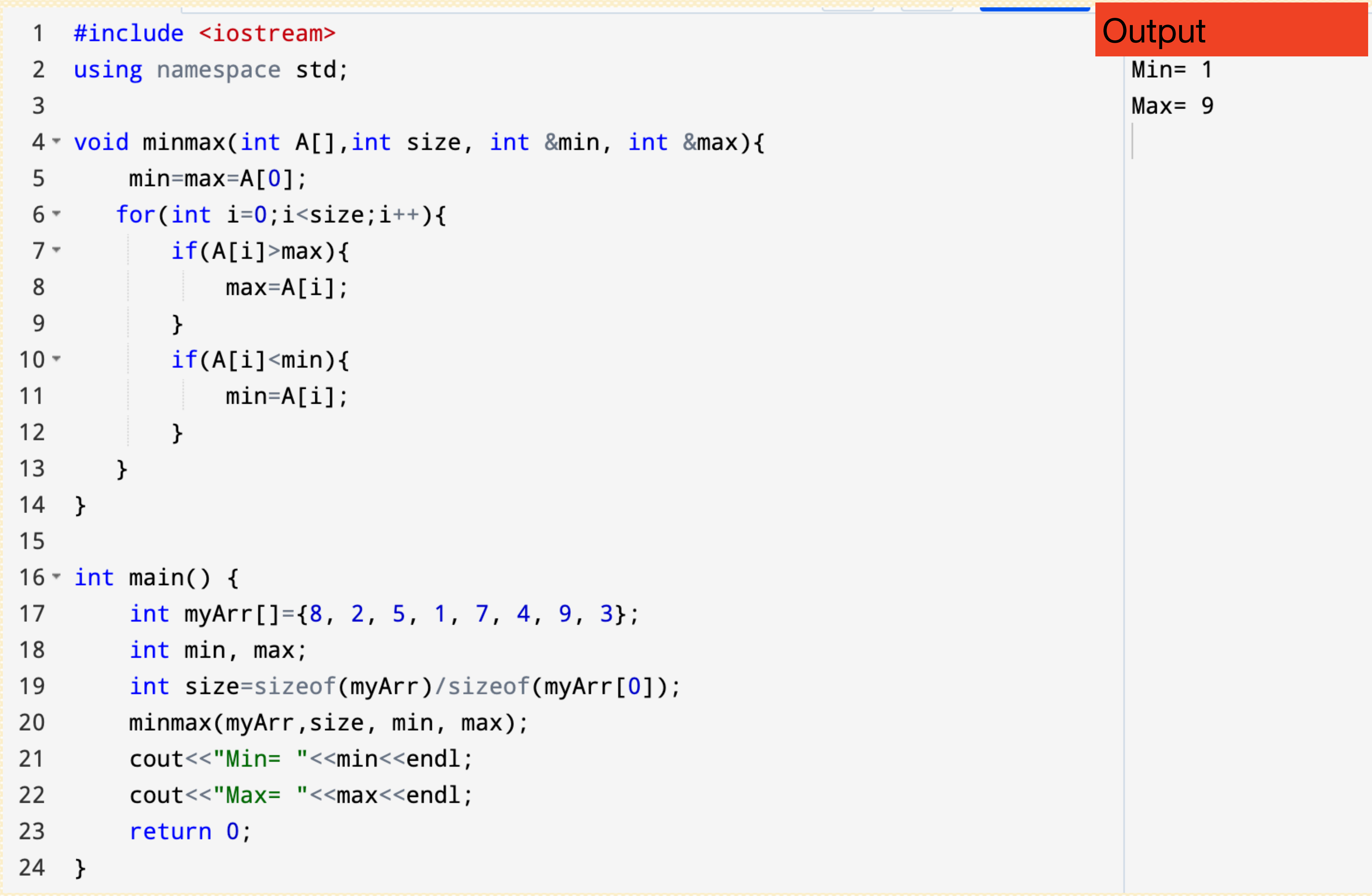
Exercise #2

find the minimum and maximum numbers in the following array, using pointer and function.

myArr[]={8, 2, 5, 1, 7, 4, 9, 3};

Exercise #2

find the minimum and maximum numbers in the following array, using pointer and function.
myArr[]={8, 2, 5, 1, 7, 4, 9, 3};



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

void minmax(int A[],int size, int &min, int &max){
     min=max=A[0];
    for(int i=0;i<size;i++){
        if(A[i]>max){
            max=A[i];
        }
        if(A[i]<min){
            min=A[i];
        }
    }
}

int main() {
    int myArr[]={8, 2, 5, 1, 7, 4, 9, 3};
    int min, max;
    int size=sizeof(myArr)/sizeof(myArr[0]);
    minmax(myArr,size, min, max);
    cout<<"Min= "<<min<<endl;
    cout<<"Max= "<<max<<endl;
    return 0;
}
```

Output

```
Min= 1
Max= 9
```