Value, Pointer and Reference Variables Parameter Passing to Functions Comprehensive Guide

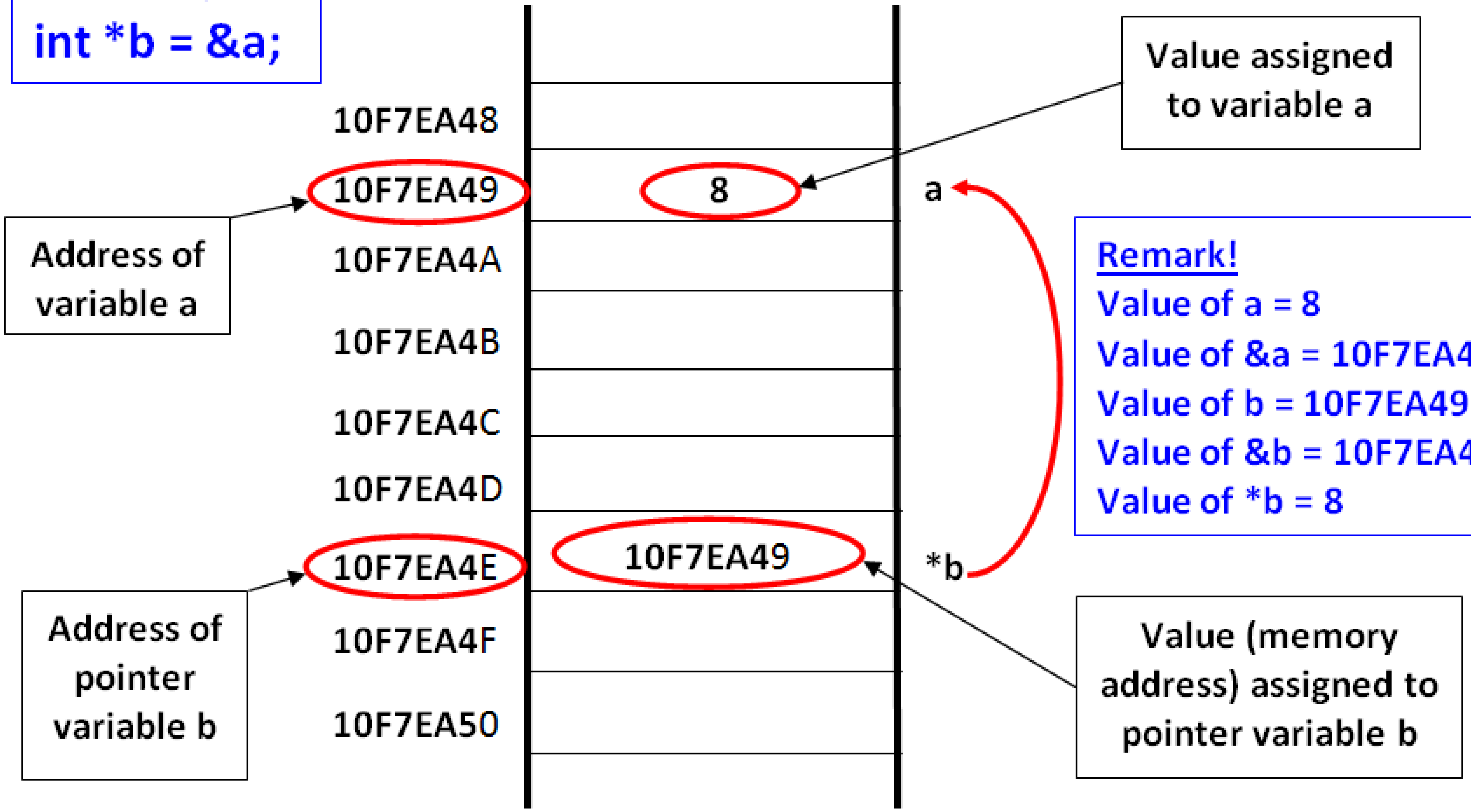
- Variables in C++
- Usage of Variables
- Parameter passing to functions
- Returning values from functions

Variable??? What?

- A **NAME** given **by a programmer** to **a specific memory location**
- Can store a data value of some specifid data type
- Data type can not be changed once declared
- Can store
  - ➢ An actual literal value: **Value Variable**
  - ➢ A memory address of another memory location: **Pointer Variable**
- Can also be an alias (second name) to an existing variable: **Reference Variable**

Value and Pointer Variables

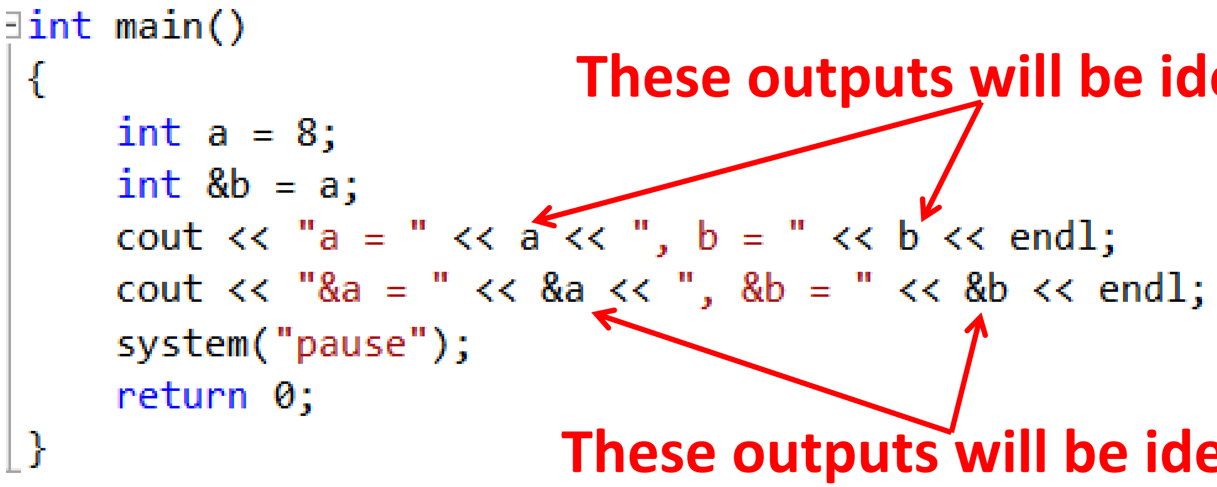
```
int a = 8;
int *b = &a;
```



10F7EA48
10F7EA49
10F7EA4A
10F7EA4B
10F7EA4C
10F7EA4D
10F7EA4E
10F7EA4F
10F7EA50
8
a
10F7EA49
*b
Value assigned to variable a
Address of variable a
Address of pointer variable b
Value (memory address) assigned to pointer variable b

Remark!
Value of a = 8
Value of &a = 10F7EA49
Value of b = 10F7EA49
Value of &b = 10F7EA4E
Value of *b = 8

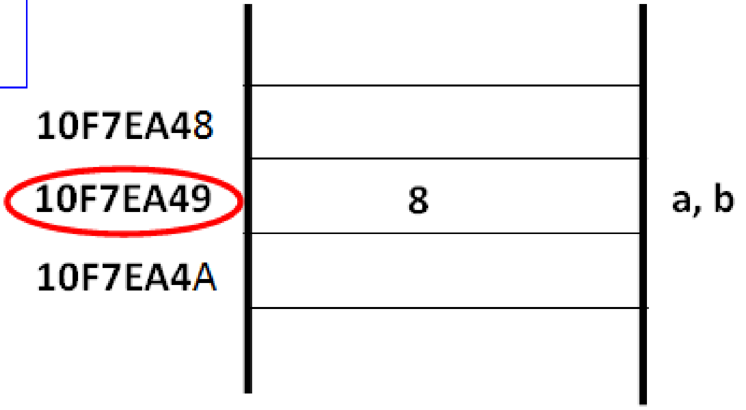
Reference Variables



```
int main()
{
    int a = 8;
    int &b = a;
    cout << "a = " << a << ", b = " << b << endl;
    cout << "&a = " << &a << ", &b = " << &b << endl;
    system("pause");
    return 0;
}
```

These outputs will be identical
These outputs will be identical

int a = 8;
int &b = a;

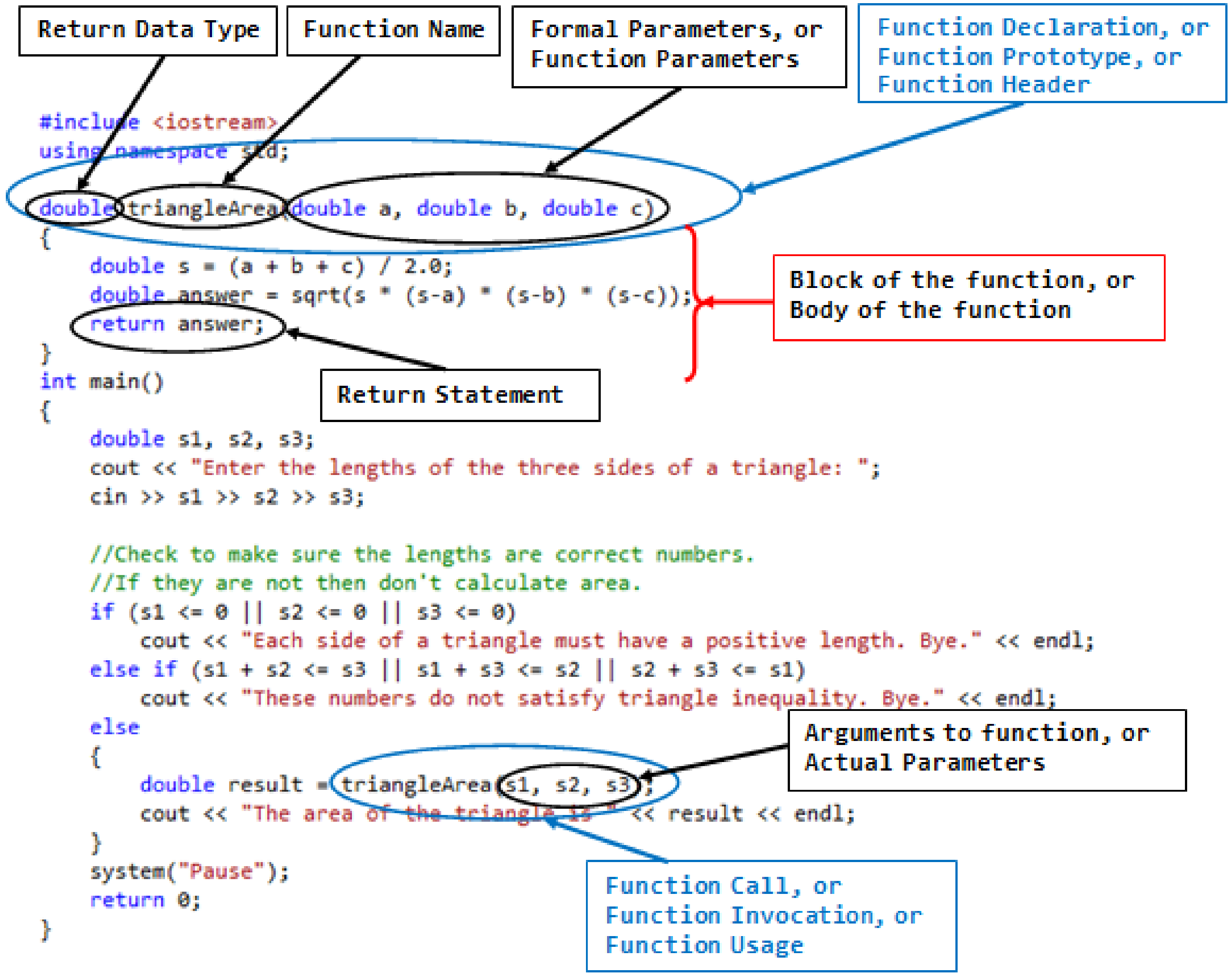


Remark!
Value of a = 8
Value of &a = 10F7EA49
Value of b = 8
Value of &b = 10F7EA49

More on Reference Variables

- Practically speaking there are **only two types of variables**: **Value and Pointer variables**
- What is Reference variable then?
  - ➢ Depending who you are referencing, a Reference variable is either a
    - ➢ **Value variable**: **When referencing to a Value variable** (see variable **b** in the previous program), or
    - ➢ **Pointer variable**: **When referencing to a Pointer variable** (see variable **d** in the previous program)

C++ Functions: Terminologies

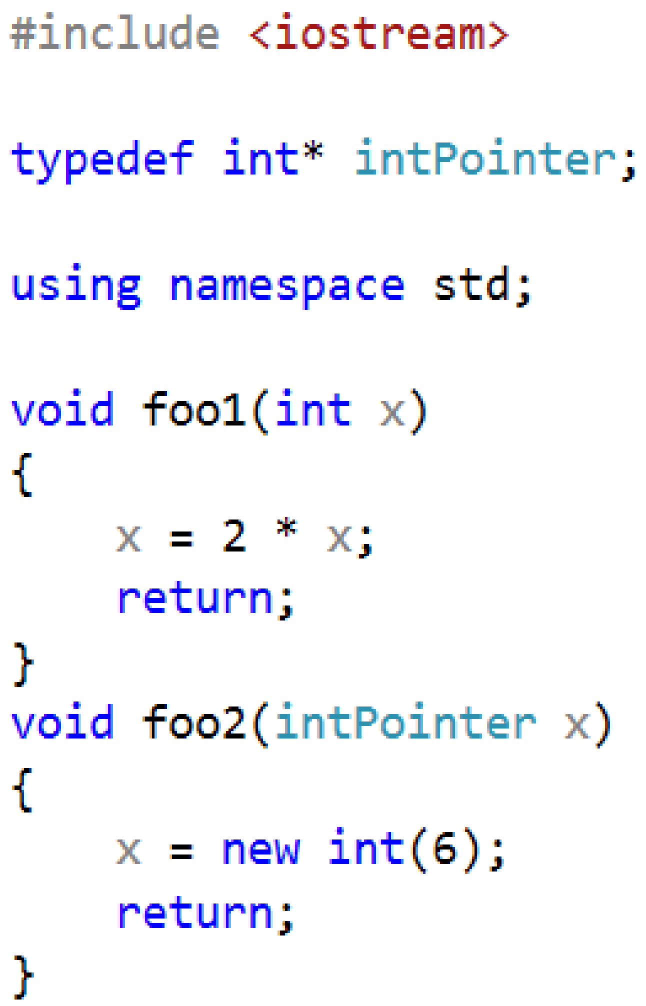

Return Data Type
Function Name
Formal Parameters, or Function Parameters
Function Declaration, or Function Prototype, or Function Header
Block of the function, or Body of the function
Return Statement
Arguments to function, or Actual Parameters
Function Call, or Function Invocation, or Function Usage

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

double triangleArea(double a, double b, double c)
{
    double s = (a + b + c) / 2.0;
    double answer = sqrt(s * (s-a) * (s-b) * (s-c));
    return answer;
}
int main()
{
    double s1, s2, s3;
    cout << "Enter the lengths of the three sides of a triangle: ";
    cin >> s1 >> s2 >> s3;

    //Check to make sure the lengths are correct numbers.
    //If they are not then don't calculate area.
    if (s1 <= 0 || s2 <= 0 || s3 <= 0)
        cout << "Each side of a triangle must have a positive length. Bye." << endl;
    else if (s1 + s2 <= s3 || s1 + s3 <= s2 || s2 + s3 <= s1)
        cout << "These numbers do not satisfy triangle inequality. Bye." << endl;
    else
    {
        double result = triangleArea(s1, s2, s3);
        cout << "The area of the triangle is " << result << endl;
    }
    system("Pause");
    return 0;
}
```

Parameter Passing to Functions Part I: Pass by Value

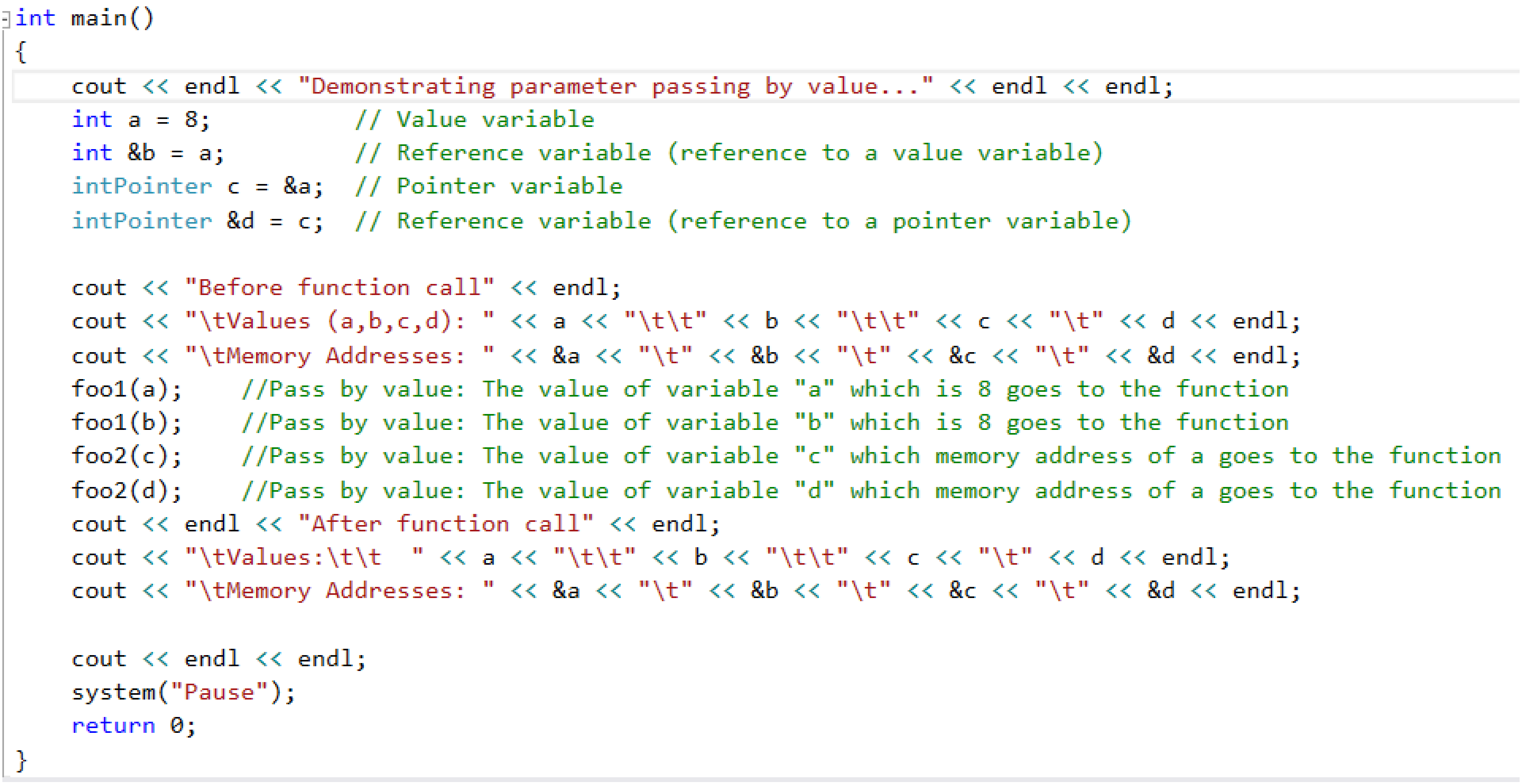
- Parameter passing by Value entails
  - Passing **a copy of** a literal value or the value of a variable to a function as an argument
- Then
  - The formal parameter in the function header corresponding to this argument must be the same data type variable (that is Value or Pointer variable)
  - Any modification made to the formal parameter in the function is not reflected in the calling function
  - **This is TRUE even if the argument is a Value variable, Pointer variable or Reference variable!!!**

Parameter Passing to Functions Part I: Pass by Value



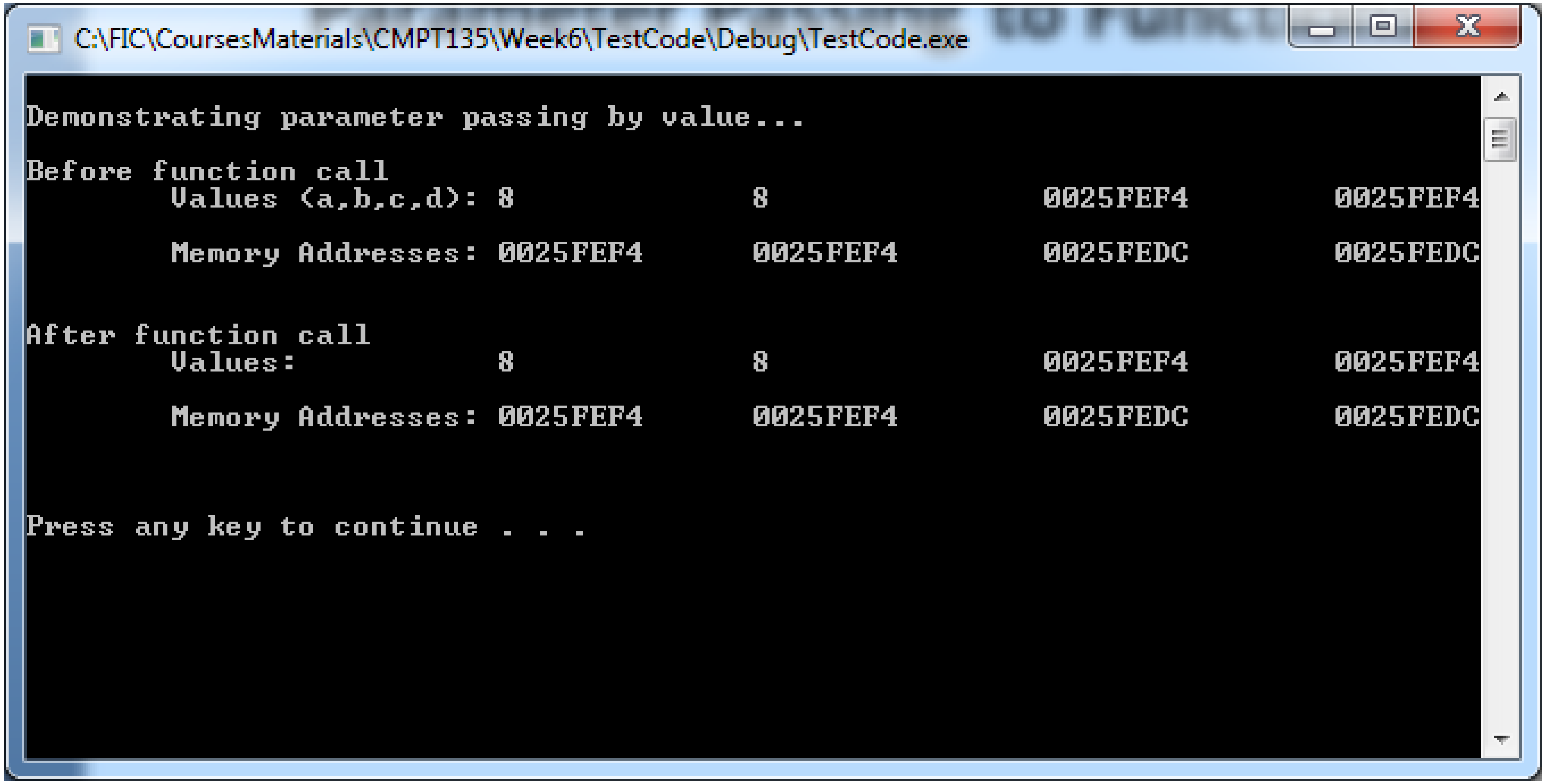
```
#include <iostream>

typedef int* intPointer;

using namespace std;

void foo1(int x)
{
    x = 2 * x;
    return;
}
void foo2(intPointer x)
{
    x = new int(6);
    return;
}
```

Parameter Passing to Functions Part I: Pass by Value



```
int main()
{
    cout << endl << "Demonstrating parameter passing by value..." << endl << endl;
    int a = 8;          // Value variable
    int &b = a;         // Reference variable (reference to a value variable)
    intPointer c = &a;  // Pointer variable
    intPointer &d = c;  // Reference variable (reference to a pointer variable)

    cout << "Before function call" << endl;
    cout << "\tValues (a,b,c,d): " << a << "\t\t" << b << "\t\t" << c << "\t" << d << endl;
    cout << "\tMemory Addresses: " << &a << "\t" << &b << "\t" << &c << "\t" << &d << endl;
    foo1(a);    //Pass by value: The value of variable "a" which is 8 goes to the function
    foo1(b);    //Pass by value: The value of variable "b" which is 8 goes to the function
    foo2(c);    //Pass by value: The value of variable "c" which memory address of a goes to the function
    foo2(d);    //Pass by value: The value of variable "d" which memory address of a goes to the function
    cout << endl << "After function call" << endl;
    cout << "\tValues:\t\t  " << a << "\t\t" << b << "\t\t" << c << "\t" << d << endl;
    cout << "\tMemory Addresses: " << &a << "\t" << &b << "\t" << &c << "\t" << &d << endl;

    cout << endl << endl;
    system("Pause");
    return 0;
}
```

Parameter Passing to Functions
Part I: Pass by Value


C:\FIC\CoursesMaterials\CMPT135\Week6\TestCode\Debug\TestCode.exe

```
Demonstrating parameter passing by value...

Before function call
        Values (a,b,c,d): 8             8               0025FEF4        0025FEF4

        Memory Addresses: 0025FEF4      0025FEF4        0025FEDC        0025FEDC


After function call
        Values:           8             8               0025FEF4        0025FEF4

        Memory Addresses: 0025FEF4      0025FEF4        0025FEDC        0025FEDC



Press any key to continue . . .
```

Parameter Passing to Functions Part I: Pass by Value

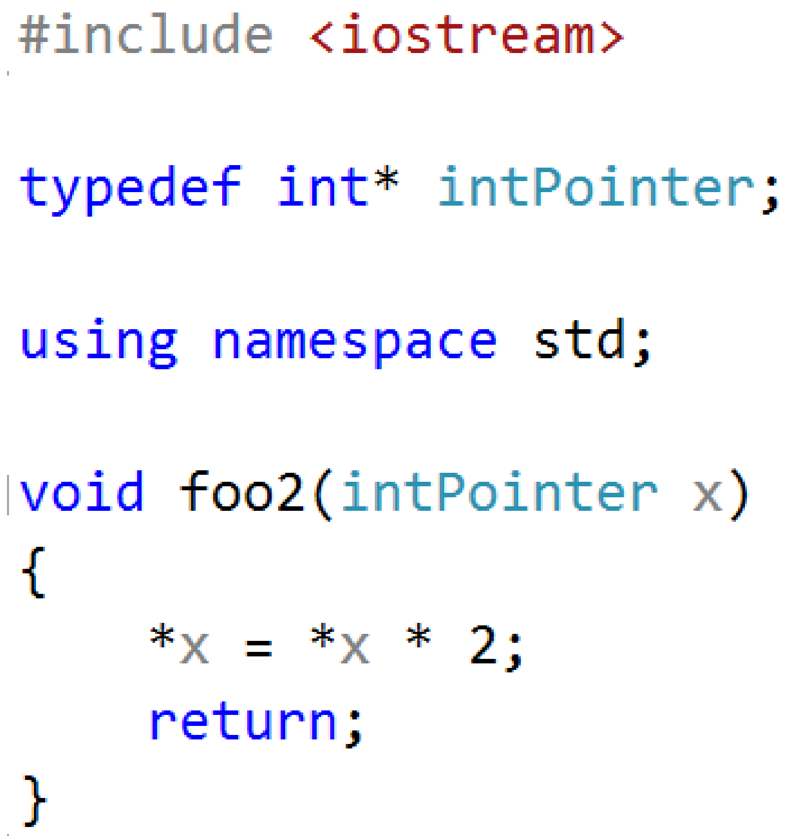
- What is the output of the program shown?
- In order to answer this question, observe that
  - Function foo2 **does not modify de-referenced value** of the formal parameter x
  - Instead it **modifies the value** of the formal parameter x
  - Thus… parameter passing by value
  - Hence **not only the values** of the variables **a, b, c,** and **d but also their memory address do NOT change!!!**

Parameter Passing to Functions Part II: Pass by Pointer

- Parameter passing by Pointer entails
  - ➢ Passing **a copy of** memory address value to a function
    - ➢ This can be
      - ➢ A memory address of a value or reference variable or
      - ➢ The value of a Pointer variable
  - ➢ The formal parameter corresponding to this argument must be a Pointer data type
  - ➢ If the formal parameter modifies its de-referenced value, then
    - ➢ The value stored in the memory whose address is passed to the function is also modified

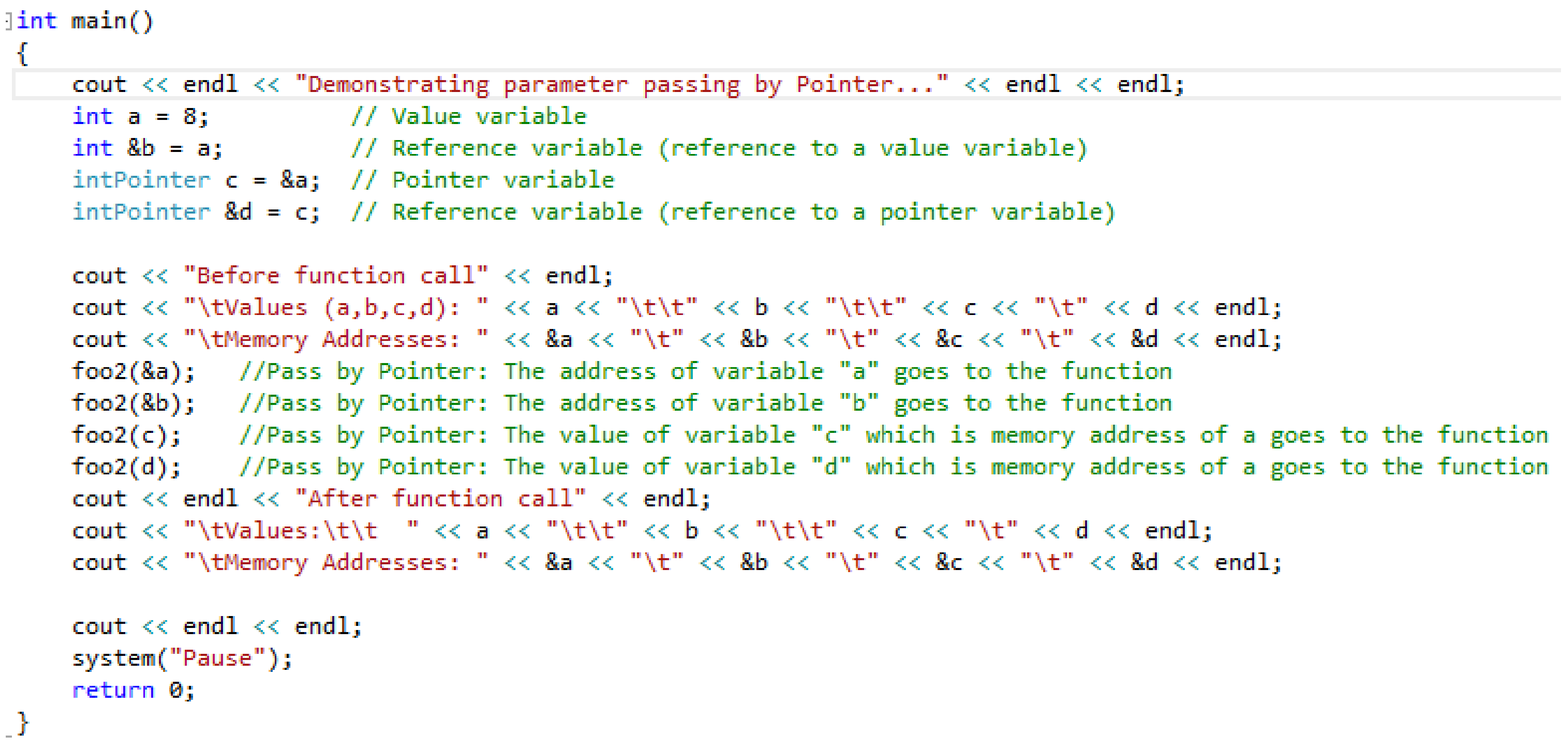
Parameter Passing to Functions Part II: Pass by Pointer

- Analyze the following program and determine its output



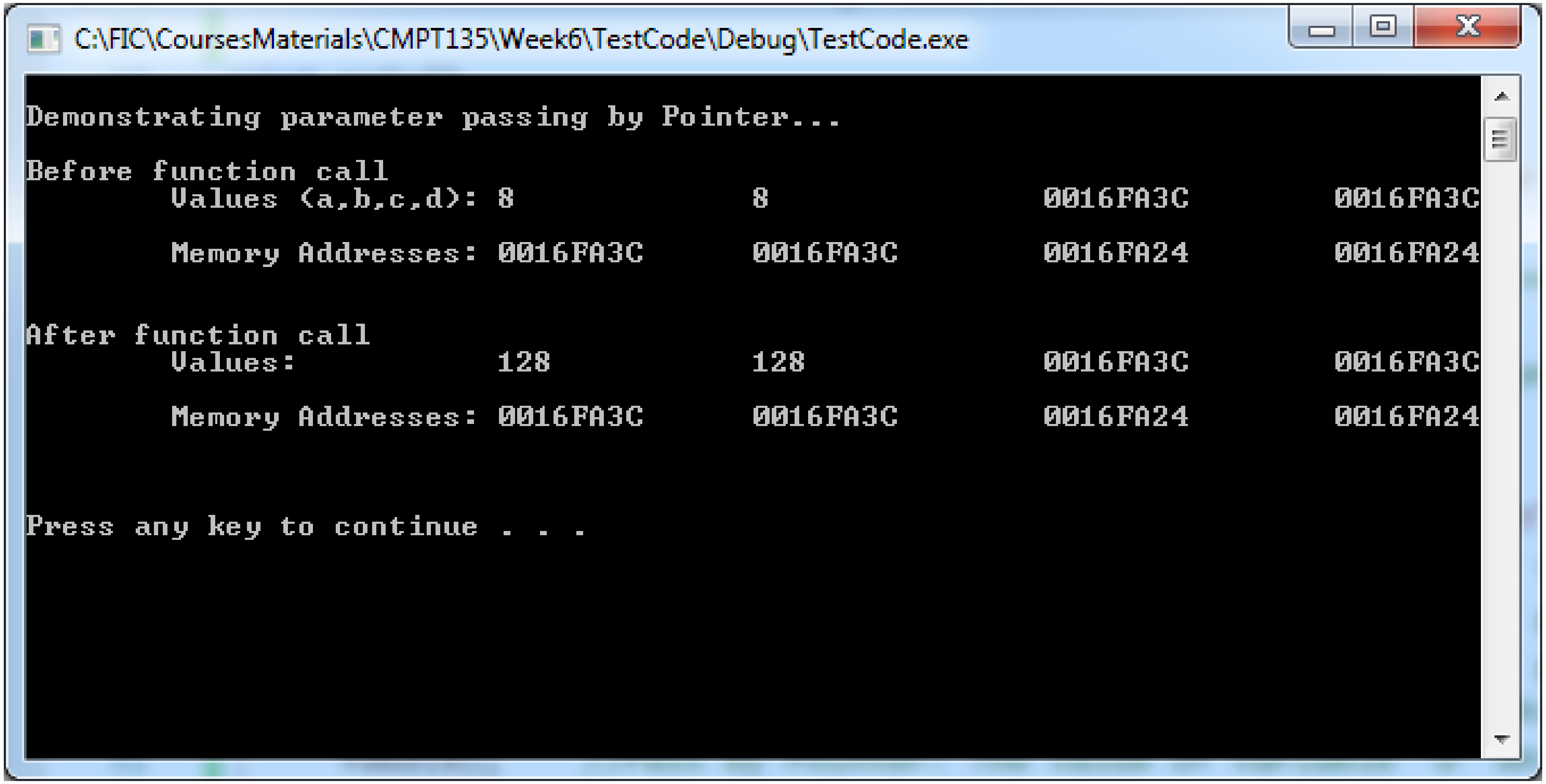
```
#include <iostream>

typedef int* intPointer;

using namespace std;

void foo2(intPointer x)
{
    *x = *x * 2;
    return;
}
```

Parameter Passing to Functions
Part II: Pass by Pointer



```
int main()
{
    cout << endl << "Demonstrating parameter passing by Pointer..." << endl << endl;
    int a = 8;          // Value variable
    int &b = a;         // Reference variable (reference to a value variable)
    intPointer c = &a;  // Pointer variable
    intPointer &d = c;  // Reference variable (reference to a pointer variable)

    cout << "Before function call" << endl;
    cout << "\tValues (a,b,c,d): " << a << "\t\t" << b << "\t\t" << c << "\t" << d << endl;
    cout << "\tMemory Addresses: " << &a << "\t" << &b << "\t" << &c << "\t" << &d << endl;
    foo2(&a);   //Pass by Pointer: The address of variable "a" goes to the function
    foo2(&b);   //Pass by Pointer: The address of variable "b" goes to the function
    foo2(c);    //Pass by Pointer: The value of variable "c" which is memory address of a goes to the function
    foo2(d);    //Pass by Pointer: The value of variable "d" which is memory address of a goes to the function
    cout << endl << "After function call" << endl;
    cout << "\tValues:\t\t  " << a << "\t\t" << b << "\t\t" << c << "\t" << d << endl;
    cout << "\tMemory Addresses: " << &a << "\t" << &b << "\t" << &c << "\t" << &d << endl;

    cout << endl << endl;
    system("Pause");
    return 0;
}
```

Parameter Passing to Functions Part II: Pass by Pointer


C:\FIC\CoursesMaterials\CMPT135\Week6\TestCode\Debug\TestCode.exe

```
Demonstrating parameter passing by Pointer...

Before function call
        Values (a,b,c,d): 8             8               0016FA3C        0016FA3C

        Memory Addresses: 0016FA3C      0016FA3C        0016FA24        0016FA24


After function call
        Values:           128           128             0016FA3C        0016FA3C

        Memory Addresses: 0016FA3C      0016FA3C        0016FA24        0016FA24



Press any key to continue . . .
```

Parameter Passing to Functions Part II: Pass by Pointer

- In parameter passing by Pointer
  - The value of the argument (which is a memory address) does not change
  - What changes is; the de-referenced value of the argument
- This is why, in the previous example
  - The the values of variables **c** and **d** **did NOT change**
  - BUT the values of variables **a** and **b** **changed**

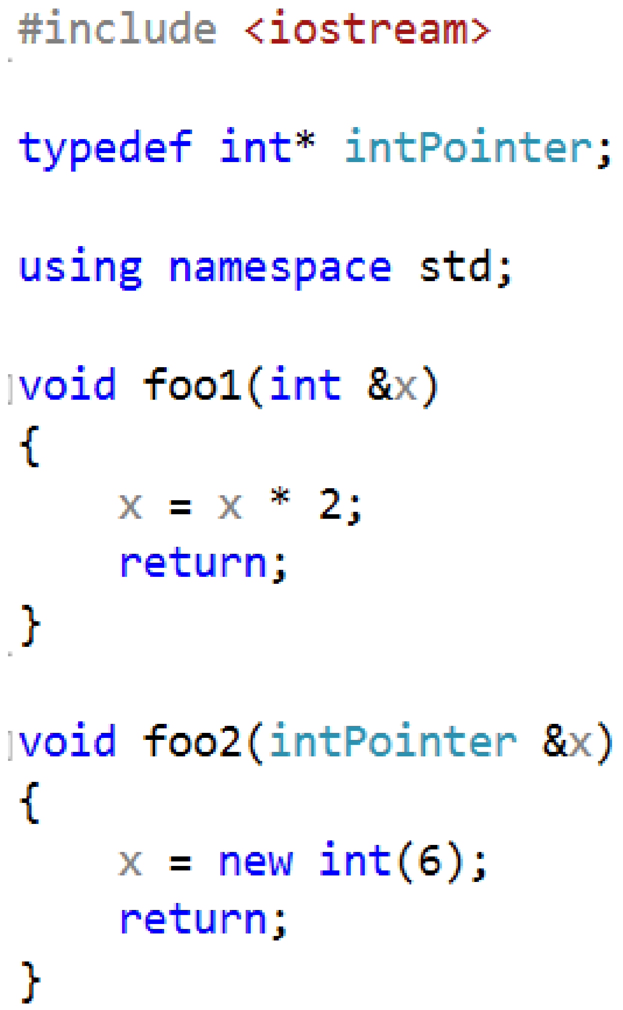
Parameter Passing to Functions Part III: Pass by Reference

- Parameter passing by reference means
  - ➢ Creating a Reference formal parameter to the argument you are passing to a function
  - ➢ The formal parameter is therefore just a second name to the actual memory location of the argument
  - ➢ Therefore any modification to the value of the formal parameter also modifies the value of the argument

Parameter Passing to Functions Part III: Pass by Reference

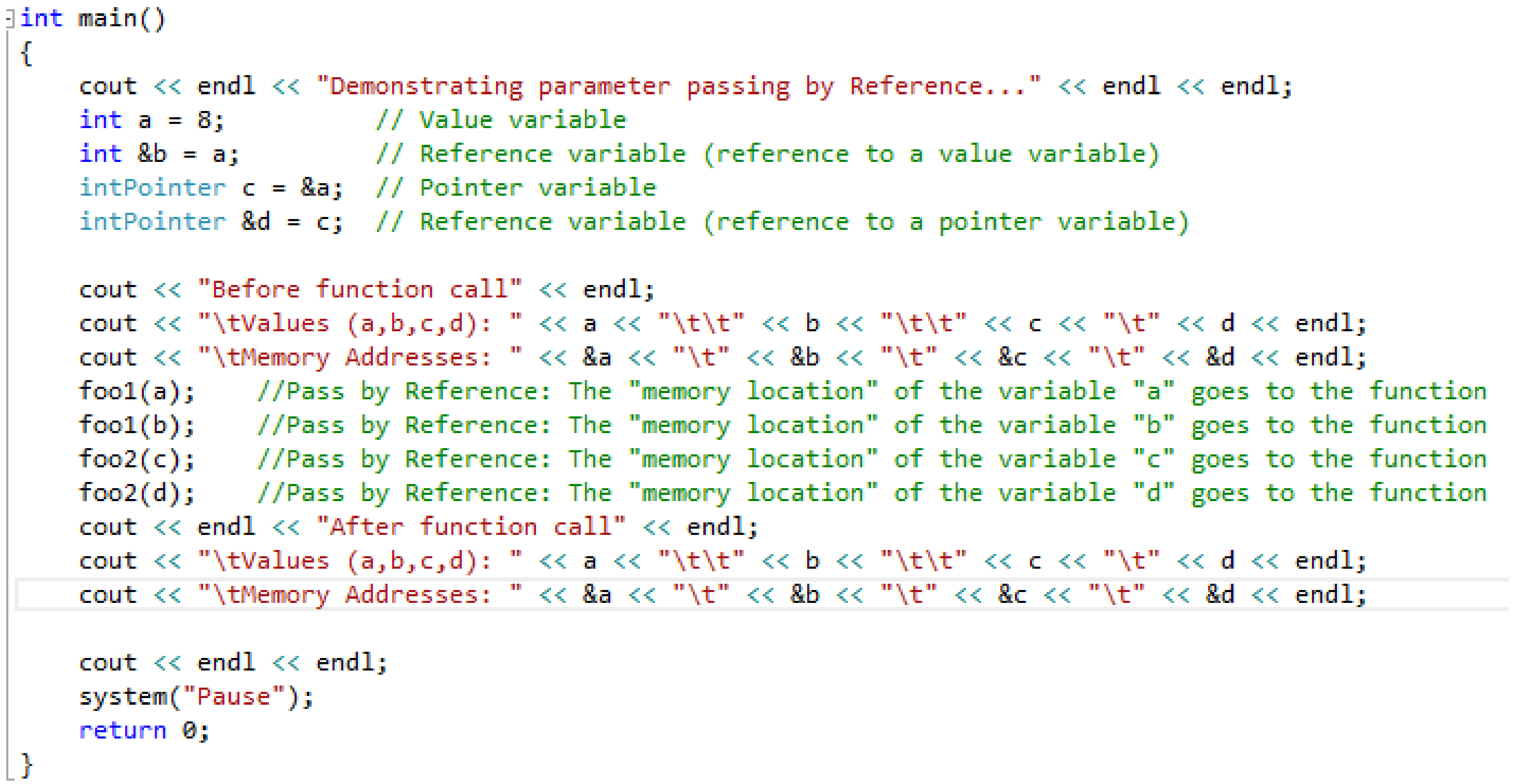
- Consider the following program
  - ➢ Analyze the program carefully and determine its output
  - ➢ Use a SCHEMATIC diagram of memory locations to help you understand the relationship between the variables in the program?
  - ➢ Does the memory address of any of the variables change after the function call? Why or why not?
  - ➢ Does the value of any of the variables change after the function call? Why or why not?

Parameter Passing to Functions Part III: Pass by Reference



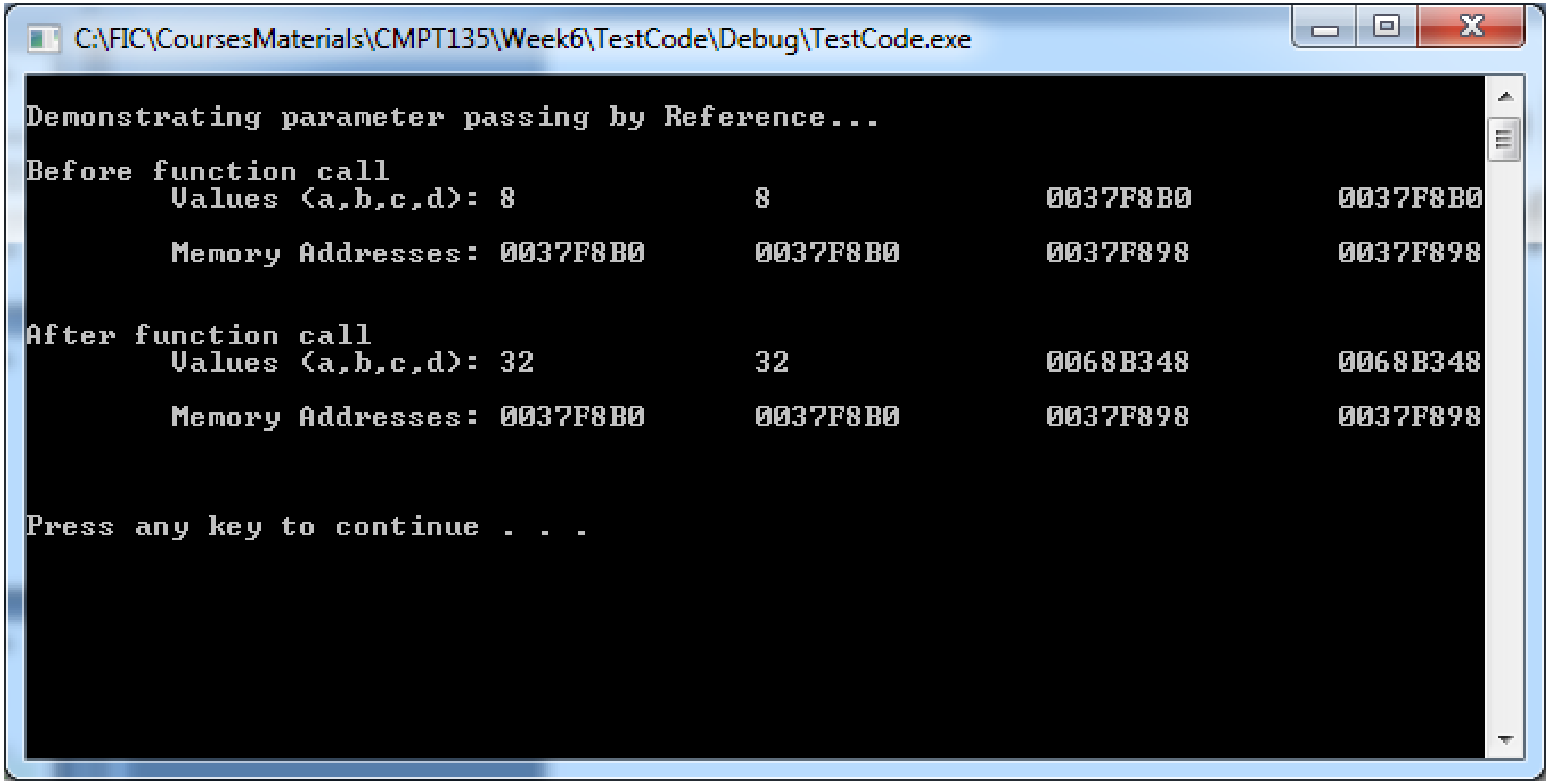
```
#include <iostream>

typedef int* intPointer;

using namespace std;

void foo1(int &x)
{
	x = x * 2;
	return;
}

void foo2(intPointer &x)
{
	x = new int(6);
	return;
}
```

Parameter Passing to Functions Part III: Pass by Reference



```
int main()
{
    cout << endl << "Demonstrating parameter passing by Reference..." << endl << endl;
    int a = 8;          // Value variable
    int &b = a;         // Reference variable (reference to a value variable)
    intPointer c = &a;  // Pointer variable
    intPointer &d = c;  // Reference variable (reference to a pointer variable)

    cout << "Before function call" << endl;
    cout << "\tValues (a,b,c,d): " << a << "\t\t" << b << "\t\t" << c << "\t" << d << endl;
    cout << "\tMemory Addresses: " << &a << "\t" << &b << "\t" << &c << "\t" << &d << endl;
    foo1(a);    //Pass by Reference: The "memory location" of the variable "a" goes to the function
    foo1(b);    //Pass by Reference: The "memory location" of the variable "b" goes to the function
    foo2(c);    //Pass by Reference: The "memory location" of the variable "c" goes to the function
    foo2(d);    //Pass by Reference: The "memory location" of the variable "d" goes to the function
    cout << endl << "After function call" << endl;
    cout << "\tValues (a,b,c,d): " << a << "\t\t" << b << "\t\t" << c << "\t" << d << endl;
    cout << "\tMemory Addresses: " << &a << "\t" << &b << "\t" << &c << "\t" << &d << endl;

    cout << endl << endl;
    system("Pause");
    return 0;
}
```

Parameter Passing to Functions
Part III: Pass by Reference



C:\FIC\CoursesMaterials\CMPT135\Week6\TestCode\Debug\TestCode.exe

```
Demonstrating parameter passing by Reference...

Before function call
        Values (a,b,c,d): 8             8               0037F8B0        0037F8B0

        Memory Addresses: 0037F8B0      0037F8B0        0037F898        0037F898


After function call
        Values (a,b,c,d): 32            32              0068B348        0068B348

        Memory Addresses: 0037F8B0      0037F8B0        0037F898        0037F898



Press any key to continue . . .
```

Parameter Passing to Functions Summary

- Parameter passing by Value and by Pointer are almost identical
  - ➢ In both cases, we are passing **a copy** of some VALUE
  - ➢ In the case of parameter passing by value; we are passing **a copy** of either a literal value or the value of a variable (which can be either a value or a pointer variable)
  - ➢ In the case of parameter passing by Pointer; we are passing **a copy** of a memory address value
- Parameter passing by Reference however means nothing is really passed to a function; instead the formal parameter of the function becomes an alias of the argument (value or pointer variable)

Returning from Functions

- A C++ function can return
  - A **literal value**
  - A **memory address** of an existing variable
  - A **memory address** of a memory space on the heap
  - A **reference** to some existing variable
  - A **reference** to a memory space on the heap memory

Returning from Functions

- **When a function returns a literal value**
  - A **copy of** the literal value is returned
  - The returned value is an R-value
  - The returned value is NOT a memory location
  - The returned value can be used in arithmetic or boolean expressions
  - The returned value CAN NOT be used as a left hand side operand in an assignment operator
  - The ++ and -- unary operators CAN NOT be applied to the returned value! Why?

Returning from Functions

- **When a function returns a memory address**
  - A **copy of** the memory address is returned
  - We say this is a function that returns a POINTER
  - It is advisable the returned value to the calling function first be assigned to a Pointer variable and then work with the Pointer variable
  - The returned value MUST NOT be a memory address of a local variable in the function; rather it should be obtained by the **new** operator or memory address of a variable with a scope in the calling function

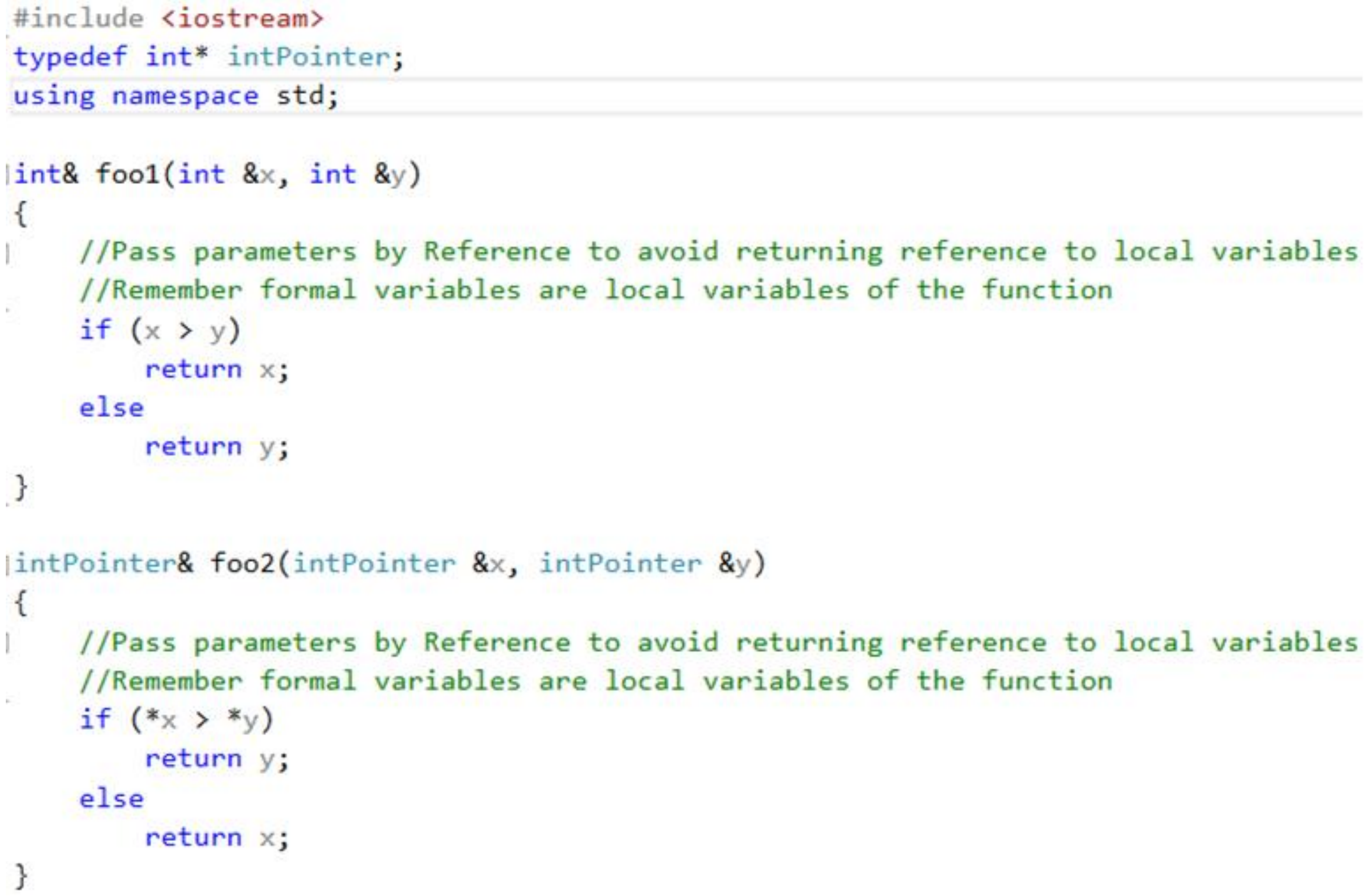
Returning from Functions

- **When a function returns a reference,**
  - No value is returned; instead an information about a memory space (i.e. a reference) is returned
  - The memory space MUST NOT have only a local scope in the function; rather it MUST have a scope in the calling function
  - The returned **thing** is an L-value
  - Hence it can be on the left side of assignment operator!!!
  - For simplicity, the returned **thing** can practically be considered as an actual physical memory location

Returning References

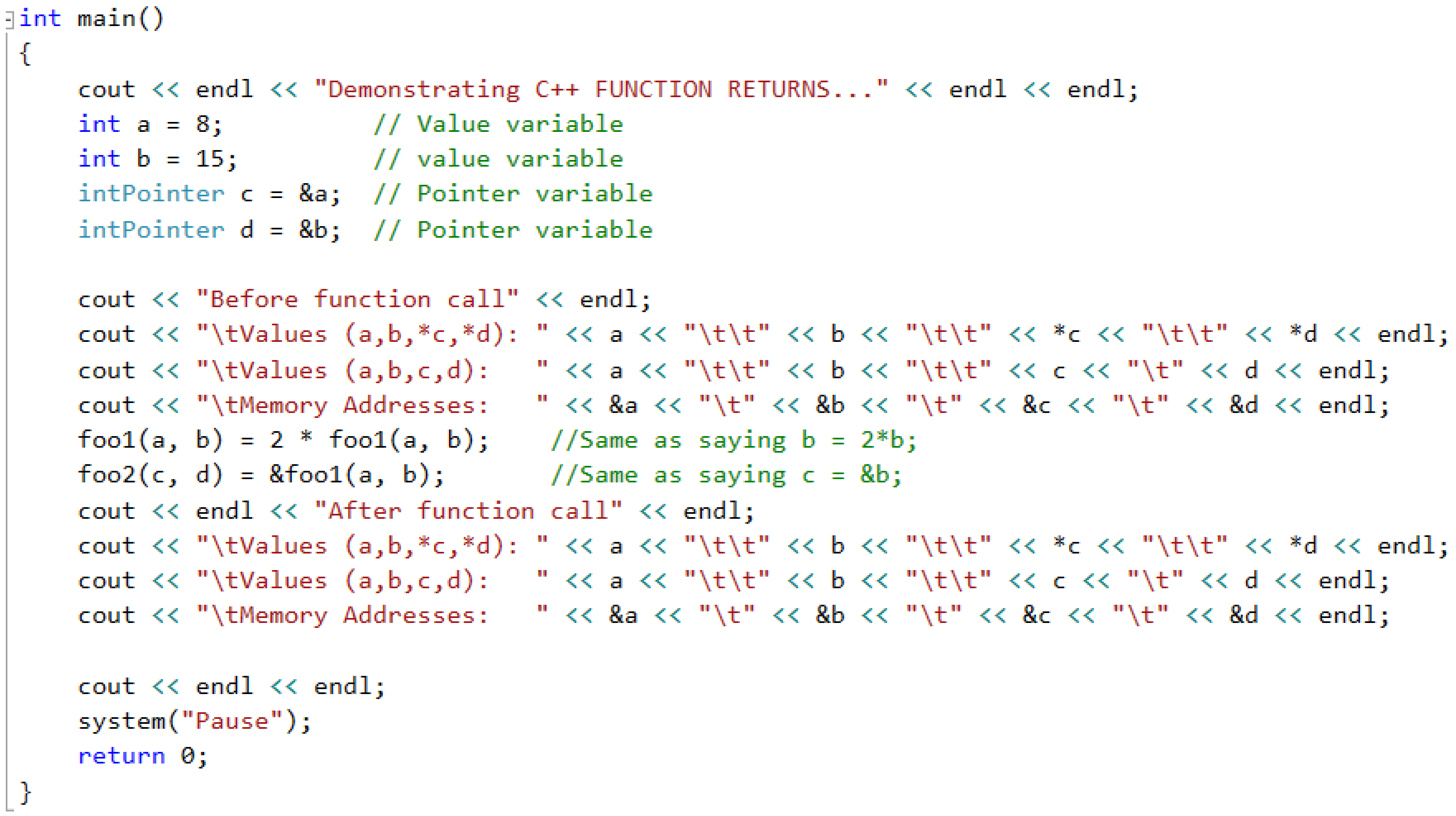
- Equipped with our discussion so far, now let us demonstrate returning References from functions
- All we need to remember is; returning a reference practically means returning a memory location whose data type corresponds to the return data type of the function
- Analyze the following program and determine its output?

Returning References



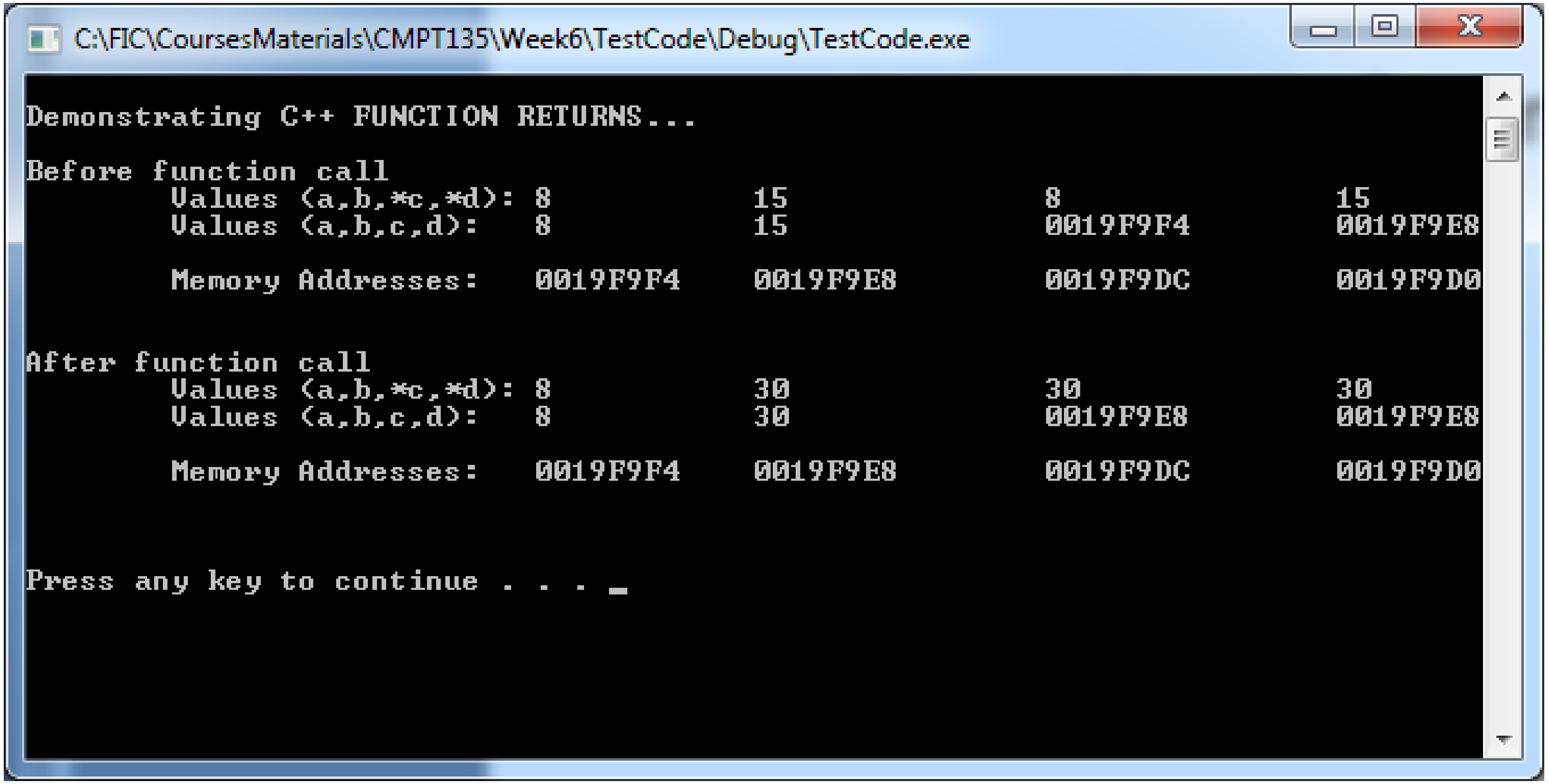
```
#include <iostream>
typedef int* intPointer;
using namespace std;

int& foo1(int &x, int &y)
{
    //Pass parameters by Reference to avoid returning reference to local variables
    //Remember formal variables are local variables of the function
    if (x > y)
        return x;
    else
        return y;
}

intPointer& foo2(intPointer &x, intPointer &y)
{
    //Pass parameters by Reference to avoid returning reference to local variables
    //Remember formal variables are local variables of the function
    if (*x > *y)
        return y;
    else
        return x;
}
```

Returning References



```
int main()
{
    cout << endl << "Demonstrating C++ FUNCTION RETURNS..." << endl << endl;
    int a = 8;          // Value variable
    int b = 15;         // value variable
    intPointer c = &a;  // Pointer variable
    intPointer d = &b;  // Pointer variable

    cout << "Before function call" << endl;
    cout << "\tValues (a,b,*c,*d): " << a << "\t\t" << b << "\t\t" << *c << "\t\t" << *d << endl;
    cout << "\tValues (a,b,c,d):   " << a << "\t\t" << b << "\t\t" << c << "\t" << d << endl;
    cout << "\tMemory Addresses:   " << &a << "\t" << &b << "\t" << &c << "\t" << &d << endl;
    foo1(a, b) = 2 * foo1(a, b);    //Same as saying b = 2*b;
    foo2(c, d) = &foo1(a, b);       //Same as saying c = &b;
    cout << endl << "After function call" << endl;
    cout << "\tValues (a,b,*c,*d): " << a << "\t\t" << b << "\t\t" << *c << "\t\t" << *d << endl;
    cout << "\tValues (a,b,c,d):   " << a << "\t\t" << b << "\t\t" << c << "\t" << d << endl;
    cout << "\tMemory Addresses:   " << &a << "\t" << &b << "\t" << &c << "\t" << &d << endl;

    cout << endl << endl;
    system("Pause");
    return 0;
}
```

Returning References


C:\FIC\CoursesMaterials\CMPT135\Week6\TestCode\Debug\TestCode.exe

```
Demonstrating C++ FUNCTION RETURNS...

Before function call
        Values (a,b,*c,*d): 8           15              8               15
        Values (a,b,c,d):   8           15              0019F9F4        0019F9E8

        Memory Addresses:   0019F9F4    0019F9E8        0019F9DC        0019F9D0


After function call
        Values (a,b,*c,*d): 8           30              30              30
        Values (a,b,c,d):   8           30              0019F9E8        0019F9E8

        Memory Addresses:   0019F9F4    0019F9E8        0019F9DC        0019F9D0



Press any key to continue . . . _
```