Practical Lecture 2: Function

LOVELY PROFESSIONAL UNIVERSITY
PUNJAB (INDIA)

Quick Recap

Let’s take a quick recap of previous lecture –

A) Function

B) Overloading of a function

C) Inline Function

D) Manipulators

Today's

Today we are going to cover -

- Friend Function
- Reference Variables
- Difference b/w call by value,call by reference and call by address
- Recursion

Let’s Get Started-

Friend Class

As we know that a class cannot access the private members of other class. Similarly a class that doesn’t inherit another class cannot access its protected members.

A friend class is a class that can access the private and protected members of a class in which it is declared as friend. This is needed when we want to allow a particular class to access the private and protected members of a class.

Friend Class Example

```
#include <iostream>
using namespace std;
class XYZ {
private:
   char ch='A';
   int num = 11;
public:
                                                            /* This statement would make class ABC
                                                             * a friend class of XYZ, this means that
                                                              * ABC can access the private and protected
                                                              * members of XYZ class.
                                                              */
   friend class ABC;
};
```

Friend Class Example

```
class ABC {
public:
   void disp(XYZ obj){
      cout<<obj.ch<<endl;
      cout<<obj.num<<endl;
   }
};
int main() {
   ABC obj;
   XYZ obj2;
   obj.disp(obj2);
   return 0;
}
```

Friend Class Example

Output:-

```
A
11
```

Friend Class Example

In the above example we have two classes XYZ and ABC. The XYZ class has two private data members ch and num, this class declares ABC as friend class. This means that ABC can access the private members of XYZ, the same has been demonstrated in the example where the function disp() of ABC class accesses the private members num and ch. In this example we are passing object as an argument to the function.

Friend Function

Similar to friend class, this function can access the private and protected members of another class. A global function can also be declared as friend as shown in the example below:

```
#include <iostream>
using namespace std;
class XYZ {
private:
   int num=100;
   char ch='Z';
public:
   friend void disp(XYZ obj);
};
```

Friend Function

```
//Global Function
void disp(XYZ obj){
   cout<<obj.num<<endl;
   cout<<obj.ch<<endl;

}
int main() {
   XYZ obj;
   disp(obj);
   return 0;
}
```

Output:-

```
100
Z
```

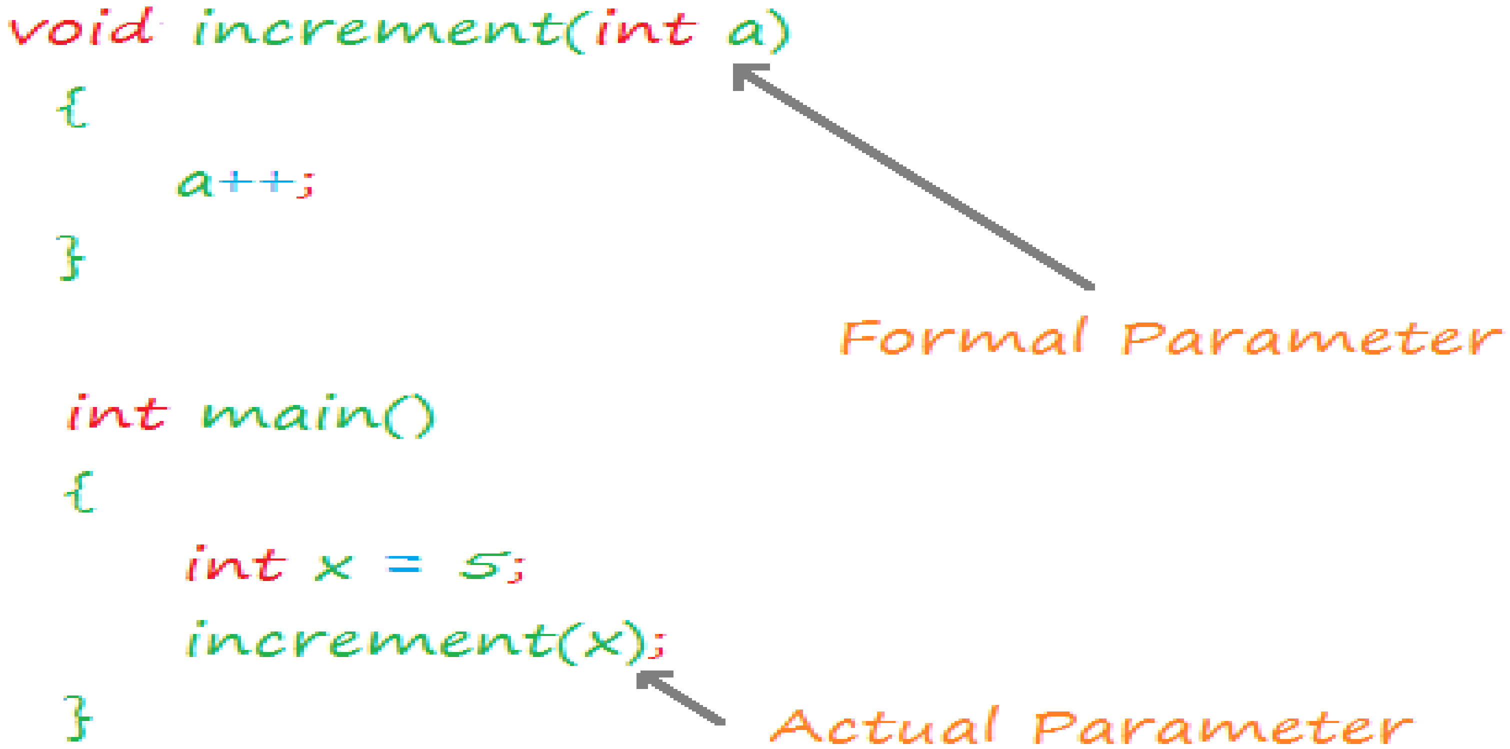
Call by value, Call by reference and Call by address

To understand the importance of each type of call and their difference, we must know, what the actual and formal parameters are:

Actual Parameters are the parameters that appear in the function call statement.

Formal Parameters are the parameters that appear in the declaration of the function which has been called.

Call by value, Call by reference and Call by address


```
void increment(int a)
 {
      a++;
 }

 int main()
 {
      int x = 5;
      increment(x);
 }
```

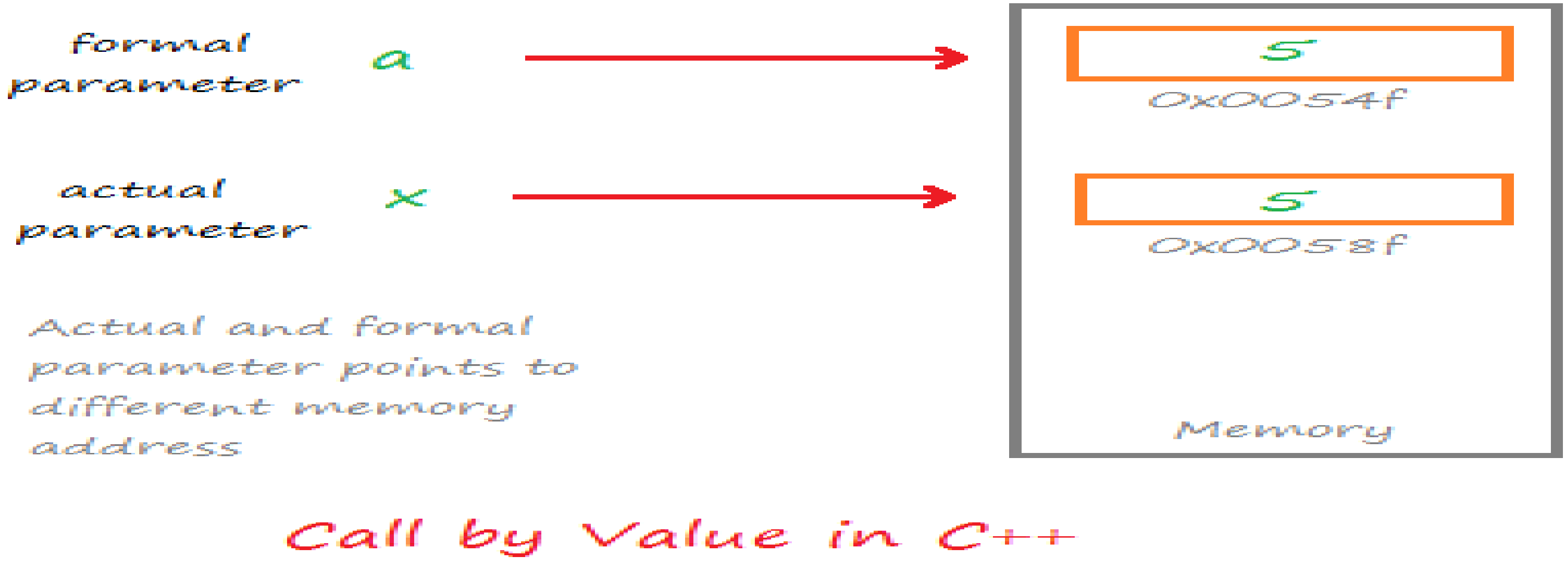
Call by value

When a function is called in the call by value, the value of the actual parameters is copied into formal parameters.

Both the actual and formal parameters have their own copies of values, therefore any change in one of the types of parameters will not be reflected by the other.

This is because both actual and formal parameters point to different locations in memory (i.e. they both have different memory addresses).

Call by value

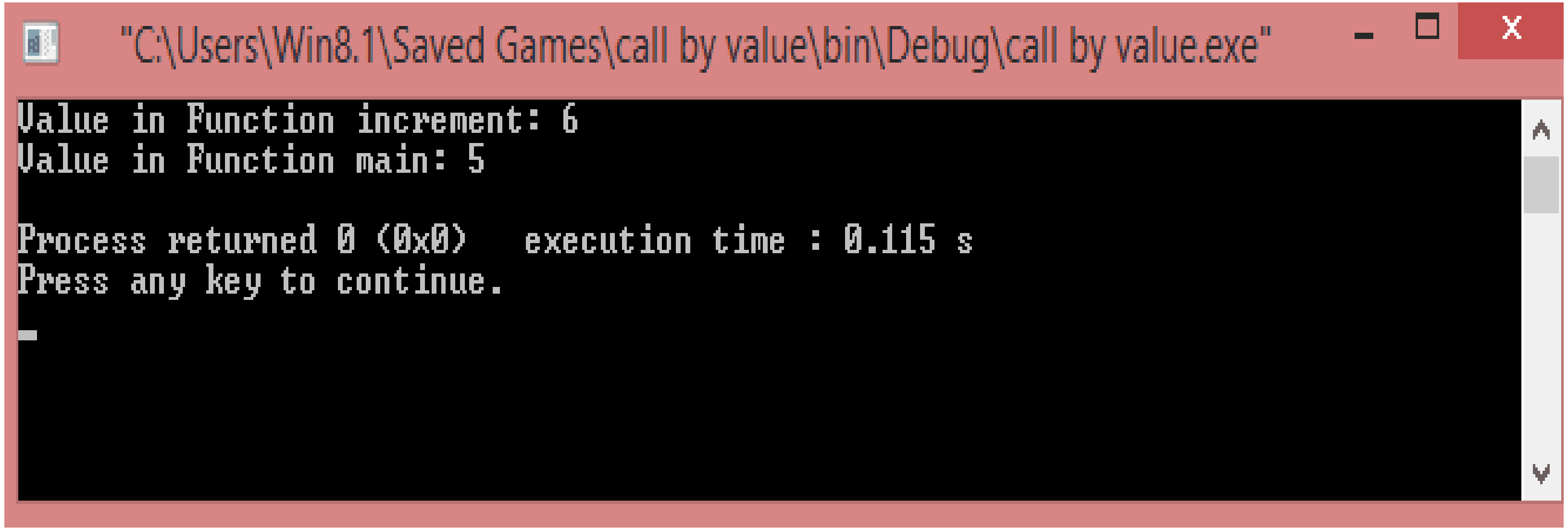


Call by value method is useful when we do not want the values of the actual parameters to be changed by the function that has been invoked.

Call by value

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

//Value of x gets copied into a
void increment(int a){
    a++;
    cout << "Value in Function increment: "<< a <<endl;
}
int main()
{
    int x = 5;
    increment(x);
    cout << "Value in Function main: "<< x <<endl;
    return 0;
}
```

Call by value


"C:\Users\Win8.1\Saved Games\call by value\bin\Debug\call by value.exe"

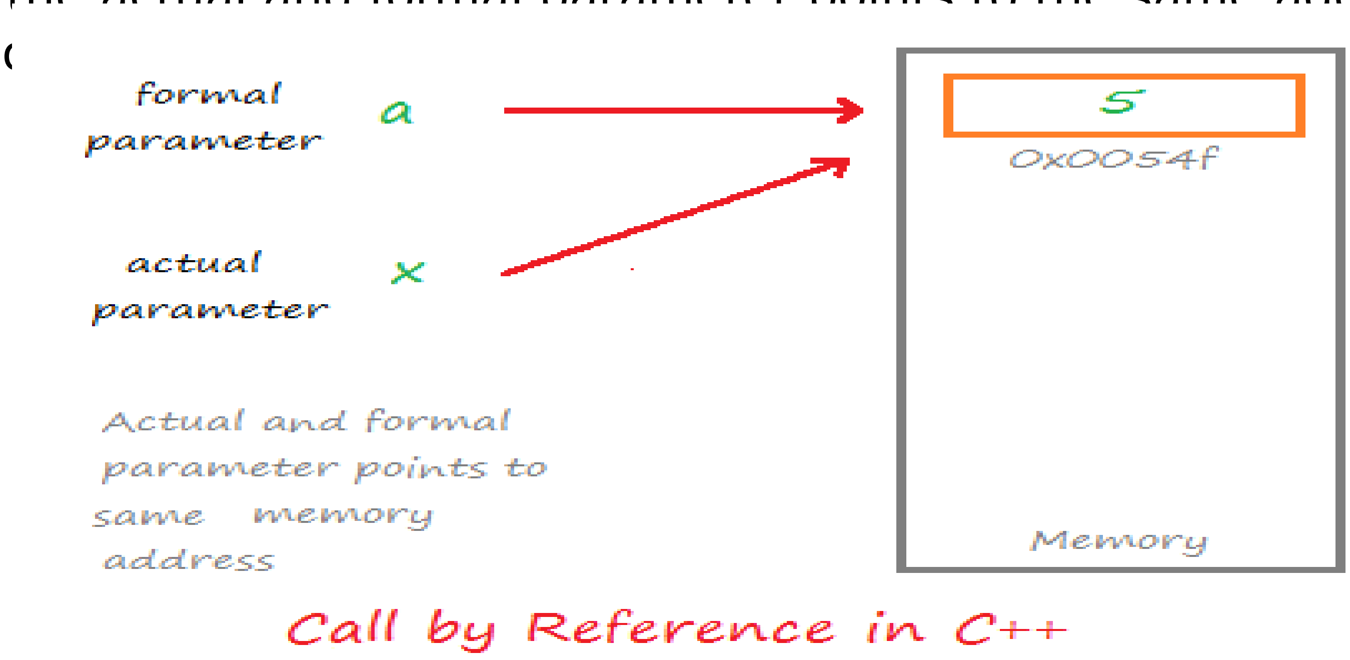
```
Value in Function increment: 6
Value in Function main: 5

Process returned 0 (0x0)   execution time : 0.115 s
Press any key to continue.
```

Note the output of the program. The value of ‘a’ has been increased to 6, but the value of ‘x’ in the main method remains the same.

This proves that the value is being copied to a different memory location in the call by value.

Call by Reference

In the call by reference, both formal and actual parameters share the same value.
Both the actual and formal parameter points to the same address in the memo



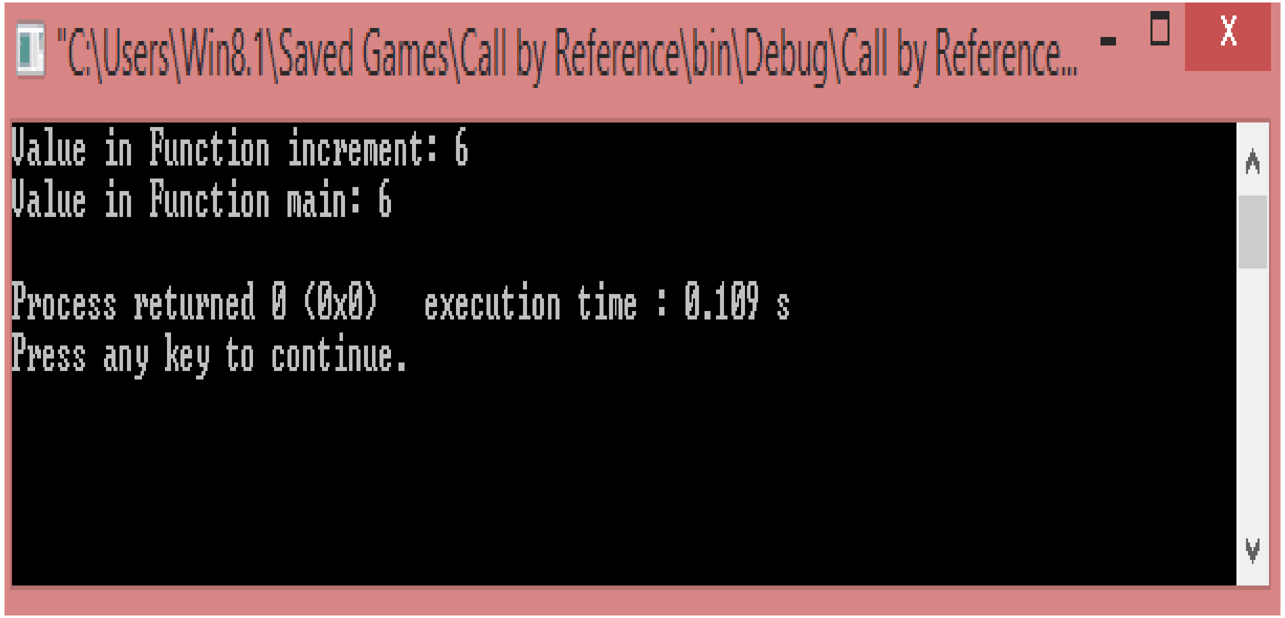
Call by Reference

That means any change on one type of parameter will also be reflected by other.
Calls by reference are preferred in cases where we do not want to make copies of objects or variables, but rather we want all operations to be performed on the same copy.

Call by Reference Example

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

//Value of x is shared with a
void increment(int &a){
    a++;
    cout << "Value in Function increment: "<< a <<endl;
}

int main()
{
    int x = 5;
    increment(x);
    cout << "Value in Function main: "<< x <<endl;
    return 0;
}
```

Call by Reference Example



"C:\Users\Win8.1\Saved Games\Call by Reference\bin\Debug\Call by Reference...

```
Value in Function increment: 6
Value in Function main: 6

Process returned 0 (0x0)   execution time : 0.109 s
Press any key to continue.
```

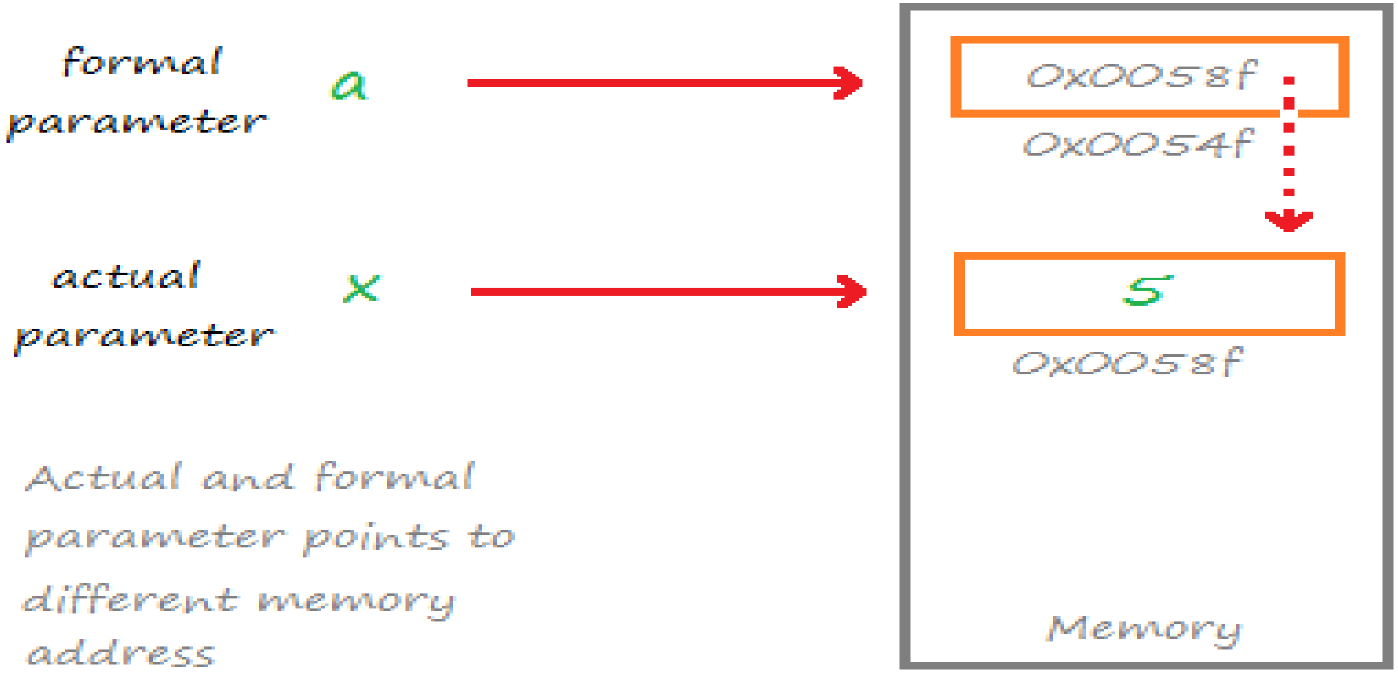
Call by Address

In the call by address method, both actual and formal parameters indirectly share the same variable.

In this type of call mechanism, pointer variables are used as formal parameters.

The formal pointer variable holds the address of the actual parameter, hence the changes done by the formal parameter is also reflected in the actual parameter.

Call by Address


formal parameter
a
actual parameter
x
0x0058f
0x0054f
5
0x0058f
Memory
Actual and formal parameter points to different memory address

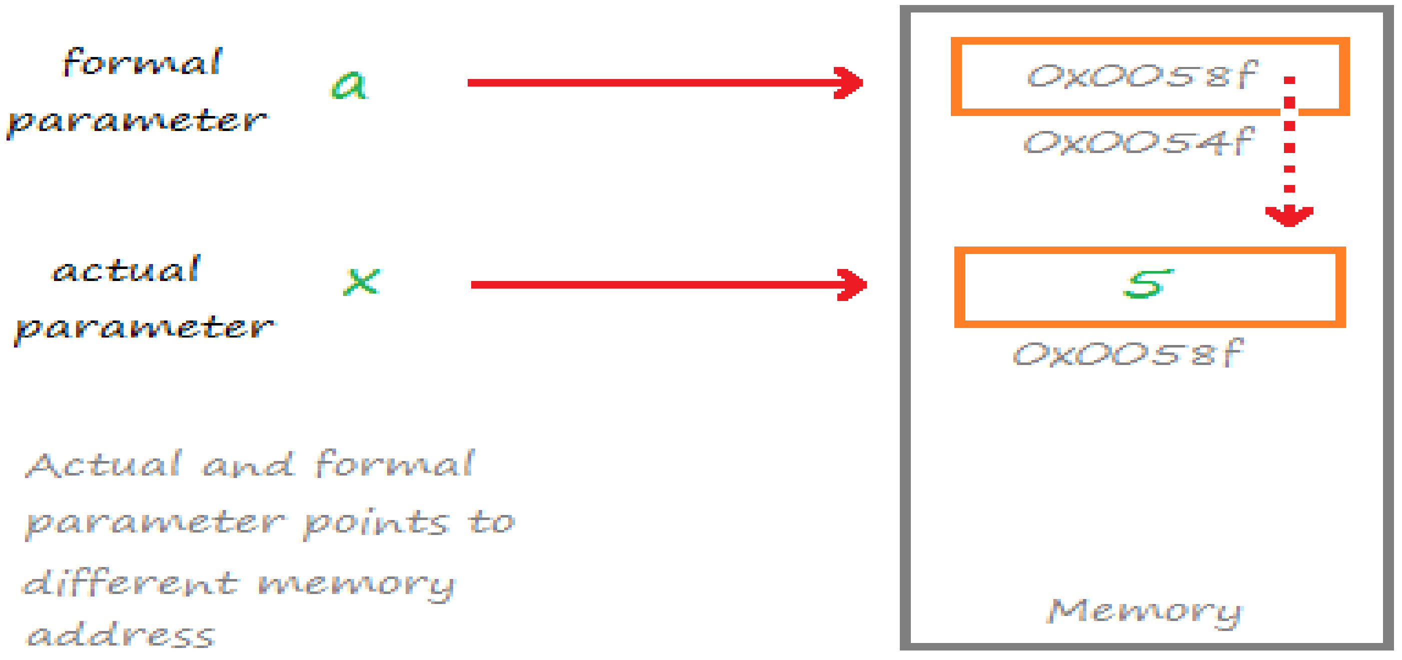
Call by Address in C++

Call by Address

As demonstrated in the diagram, both parameters point to different locations in memory, but since the formal parameter stores the address of the actual parameter, they share the same value.

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

//a stores the address of x
void increment(int *a){
    (*a)++;
    cout << "Value in Function increment: "<< *a <<endl;
}
```

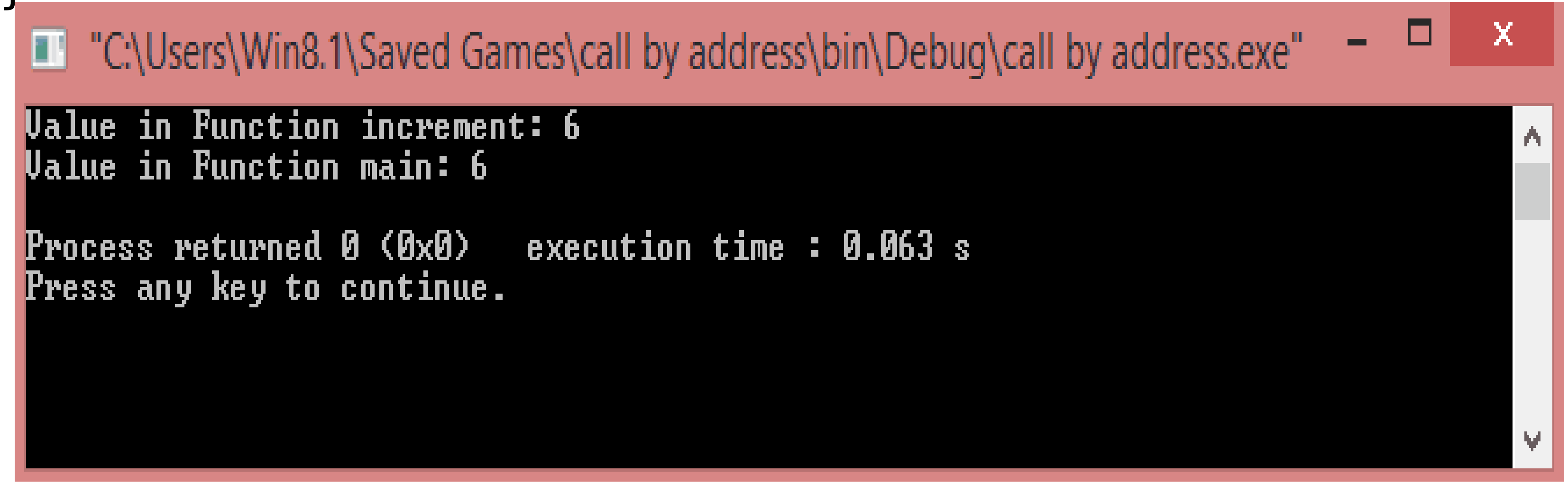
Call by Address


formal parameter
a
0x0058f
0x0054f
actual parameter
x
5
0x0058f
Actual and formal parameter points to different memory address
Memory

Call by Address in C++

Call by Address

```
int main()
{
    int x = 5;
    increment(&x); //Passing address of x //int *a=&x;
    cout << "Value in Function main: "<< x <<endl;
    return 0;
}
```

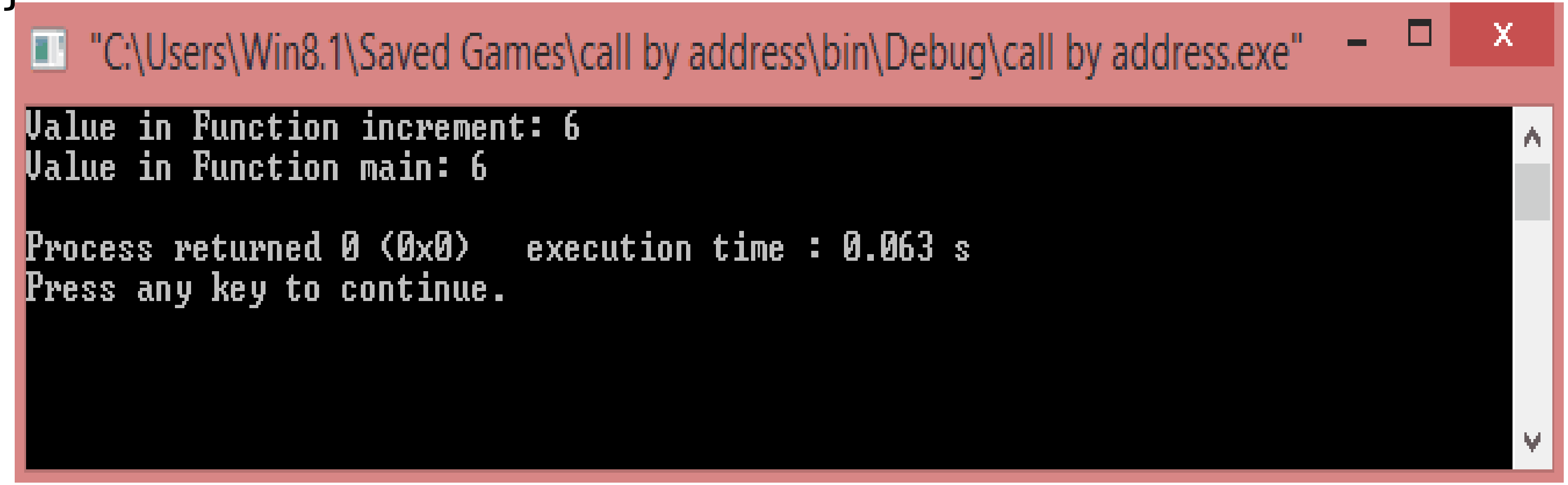


"C:\Users\Win8.1\Saved Games\call by address\bin\Debug\call by address.exe"

```
Value in Function increment: 6
Value in Function main: 6

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

Call by Address

```
int main()
{
    int x = 5;
    increment(&x); //Passing address of x
    cout << "Value in Function main: "<< x <<endl;
    return 0;
}
```



"C:\Users\Win8.1\Saved Games\call by address\bin\Debug\call by address.exe"

```
Value in Function increment: 6
Value in Function main: 6

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.
```

Conclusion

As a conclusion, we can say that call by value should be used in cases where we do not want the value of the actual parameter to be disturbed by other functions and call by reference and call by address should be used in cases where we want to maintain a variable or a copy of the object throughout the program.

Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.
A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.
A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

```
recursionfunction(){
recursionfunction(); //calling self function
}
```

Recursion

When function is called within the same function, it is known as recursion in C++. The function which calls the same function, is known as recursive function.
A function that calls itself, and doesn't perform any task after function call, is known as tail recursion. In tail recursion, we generally call the same function with return statement.

```
recursionfunction(){
recursionfunction(); //calling self function
}
```

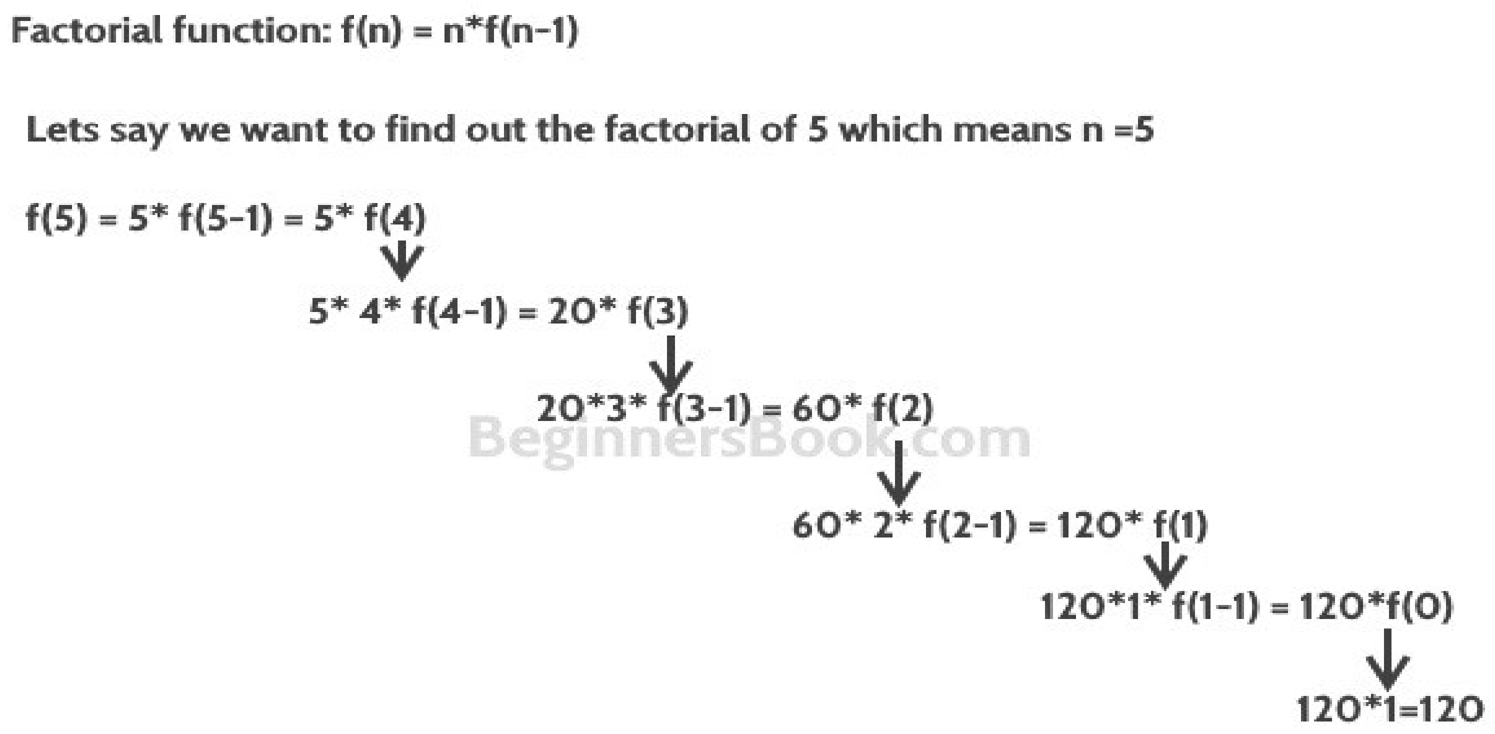
Recursion

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

int main()
{
int factorial(int);
int fact,value;
cout<<"Enter any number: ";
cin>>value;
fact=factorial(value);
cout<<"Factorial of a number is: "<<fact<<endl;
return 0;
}
```

Recursion

```
int factorial(int n)
{
if(n<0)
return(-1); /*Wrong value*/
if(n==0)
return(1);  /*Terminating condition*/
else
{
return(n*factorial(n-1));
}
}
```

Recursion



Factorial function: f(n) = n*f(n-1)

Lets say we want to find out the factorial of 5 which means n =5

f(5) = 5* f(5-1) = 5* f(4)

↓

5* 4* f(4-1) = 20* f(3)

↓

20*3* f(3-1) = 60* f(2)

↓

60* 2* f(2-1) = 120* f(1)

↓

120*1* f(1-1) = 120*f(0)

↓

120*1=120

BeginnersBook.com

Stack Overflow in Recursion

If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
    // wrong base case (it may cause
    // stack overflow).
    if (n == 100)
        return 1;

    else
        return n*fact(n-1);
}
```

Stack Overflow in Recursion

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

Advantage of Recursion

1. It makes our code shorter and cleaner.

2. Recursion is required in problems concerning data structures and advanced algorithms, such as Graph and Tree Traversal.

Disadvantage of Recursion

1. It takes a lot of stack space compared to an iterative program.

2. It uses more processor time.

3. It can be more difficult to debug compared to an equivalent iterative program.

Any

Questions ??

Thank You!

See you guys in next class.