Call by reference, Return by reference, Inline function.

Return by reference in C++ with Examples:

Pointers (https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/) and **References** (https://www.geeksforgeeks.org/references-in-c/) in C++ held close relation with one another. The major difference is that the pointers can be operated on like adding values whereas references are just an **alias** for another variable.

- **Functions in C++** (https://www.geeksforgeeks.org/functions-in-c/) can return a **reference** as it's returns a **pointer**.
- When function returns a **reference** it means it returns a **implicit** pointer.

Return by reference is very different from **Call by reference** (https://www.geeksforgeeks.org/difference-between-call-by-value-and-call-by-reference/). Functions behaves a very important **role** when variable or pointers are returned as reference.

See this function signature of Return by Reference Below:

> **dataType& functionName(parameters);**
> where,
> **dataType** is the **return type** of the **function**,
> and **parameters** are the passed **arguments** to it.

Below is the code to illustrate the Return by reference:

- CPP

```
// C++ program to illustrate return by reference
#include <iostream>
using namespace std;

// Function to return as return by reference
int& returnValue(int& x)
{

    // Print the address
    cout << "x = " << x
```

```
        << " The address of x is "
        << &x << endl;

    // Return reference
    return x;
}

// Driver Code
int main()
{
    int a = 20;
    int& b = returnValue(a);

    // Print a and its address
    cout << "a = " << a
        << " The address of a is "
        << &a << endl;

    // Print b and its address
    cout << "b = " << b
        << " The address of b is "
        << &b << endl;

    // We can also change the value of
    // 'a' by using the address returned
    // by returnValue function

    // Since the function returns an alias
    // of x, which is itself an alias of a,
    // we can update the value of a
    returnValue(a) = 13;

    // The above expression assigns the
    // value to the returned alias as 3.
    cout << "a = " << a
        << " The address of a is "
        << &a << endl;
    return 0;
}
```

Output:

```
x = 20 The address of x is 0x7fff3025711c
a = 20 The address of a is 0x7fff3025711c
```

```
b = 20 The address of b is 0x7fff3025711c
x = 20 The address of x is 0x7fff3025711c
a = 13 The address of a is 0x7fff3025711c
```

Explanation:

Since **reference** is nothing but an **alias**(synonym) of another variable, the address of **a**, **b** and **x** never changes.

> **Note:** We should never return a **local variable** as a **reference**, reason being, as soon as the functions returns, local variable will be **erased**, however, we still will be left with a **reference** which might be a **security bug** in the code.

Below is the code to illustrate the Return by reference:

- C++

```
// C++ program to illustrate return
// by reference
#include <iostream>
using namespace std;

// Global variable
int x;

// Function returns as a return
// by reference
int& retByRef()
{
    return x;
}

// Driver Code
int main()
{
    // Function Call for return
    // by reference
    retByRef() = 10;

    // Print X
    cout << x;
    return 0;
}
```

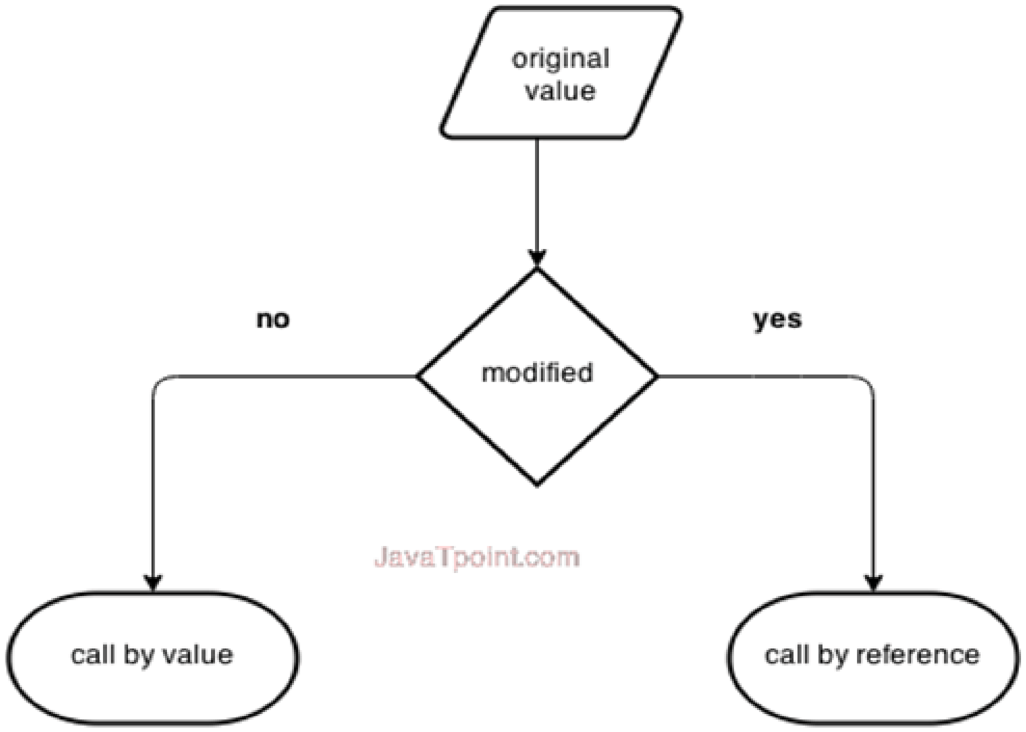
Output:

```
10
```

Explanation:
Return type of the above function **retByRef()** is a reference of the variable **x** so value **10** will be assigned into the **x.**

Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.



Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in C++ language by the example given below:

(https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#) (https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#) (https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#)

```
using namespace std;
int main()
int data = 3;
cout << "Value of the data is: " << data<< endl;
}
{
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

(https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#) (https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#) (https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#)

```
using namespace std;
```

2. {
3. swap=*x;
4. *y=swap;
5. int main()
6. int x=500, y=100;
7. cout<<"Value of x is: "<<x<<endl;
8. return 0;

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

Pointers (https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/) and **References** (https://www.geeksforgeeks.org/references-in-c/) in C++ held close relation with one another. The major difference is that the pointers can be operated on like adding values whereas references are just an **alias** for another variable.

- **Functions in C++** (https://www.geeksforgeeks.org/functions-in-c/) can return a **reference** as it's returns a **pointer**.
- When function returns a **reference** it means it returns a **implicit** pointer.

Return by reference is very different from **Call by reference** (https://www.geeksforgeeks.org/difference-between-call-by-value-and-call-by-reference/) . Functions behaves a very important **role** when variable or pointers are returned as reference.

See this function signature of Return by Reference Below:

> **dataType& functionName(parameters);**
> where,
> **dataType** is the **return type** of the **function**,
> and **parameters** are the passed **arguments** to it.

Below is the code to illustrate the Return by reference:

- CPP

```
// C++ program to illustrate return by reference
#include <iostream>
```

```
using namespace std;

// Function to return as return by reference
int & returnValue( int & x)
{

    // Print the address
    cout << "x = " << x
         << " The address of x is "
         << &x << endl;

    // Return reference
    return x;
}

// Driver Code
int main()
{
    int a = 20;
    int & b = returnValue(a);

    // Print a and its address
    cout << "a = " << a
         << " The address of a is "
         << &a << endl;

    // Print b and its address
    cout << "b = " << b
         << " The address of b is "
         << &b << endl;

    // We can also change the value of
    // 'a' by using the address returned
    // by returnValue function

    // Since the function returns an alias
    // of x, which is itself an alias of a,
    // we can update the value of a
    returnValue(a) = 13;

    // The above expression assigns the
    // value to the returned alias as 3.
    cout << "a = " << a
         << " The address of a is "
         << &a << endl;
    return 0;
}
```

Output:

```
x = 20 The address of x is 0x7fff3025711c
a = 20 The address of a is 0x7fff3025711c
b = 20 The address of b is 0x7fff3025711c
x = 20 The address of x is 0x7fff3025711c
a = 13 The address of a is 0x7fff3025711c
```

Explanation:

Since **reference** is nothing but an **alias**(synonym) of another variable, the address of **a**, **b** and **x** never changes.

> **Note:** We should never return a **local variable** as a **reference**, reason being, as soon as the functions returns, local variable will be **erased**, however, we still will be left with a **reference** which might be a **security bug** in the code.

Below is the code to illustrate the Return by reference:

- C++

```
// C++ program to illustrate return
// by reference
#include <iostream>
using namespace std;

// Global variable
int x;

// Function returns as a return
// by reference
int& retByRef()
{
    return x;
}

// Driver Code
int main()
{
    // Function Call for return
    // by reference
    retByRef() = 10;

    // Print X
    cout << x;
    return 0;
}
```

Output:

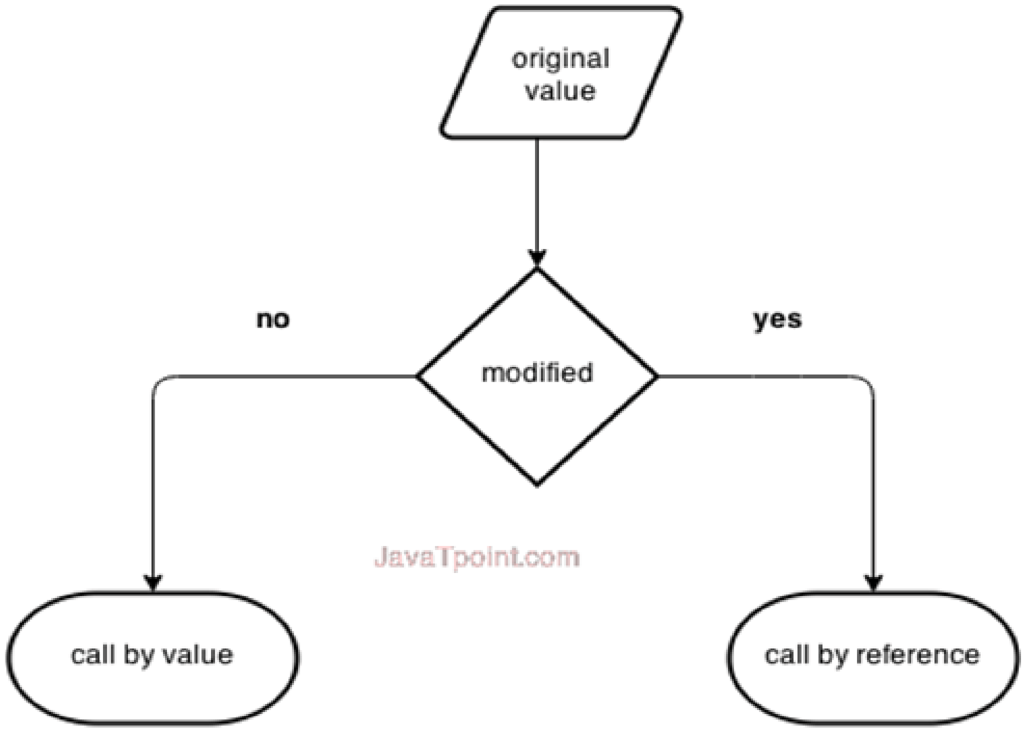
```
10
```

Explanation:

Return type of the above function **retByRef()** is a reference of the variable **x** so value **10** will be assigned into the **x.**

Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference. Original value is not modified in call by value but it is modified in call by reference.



Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in C++ language by the example given below:

(https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#) (https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#) (https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#)

```
using namespace std;
int main()
int data = 3;
cout << "Value of the data is: " << data<< endl;
}
{
}
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Here, address of the value is passed in the function, so actual and formal arguments share the same address space. Hence, value changed inside the function, is reflected inside as well as outside the function.

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

(https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#) (https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#) (https://www.javatpoint.com/call-by-value-and-call-by-reference-in-cpp#)

1. using namespace std;
2. {
3. swap=*x;
4. *y=swap;
5. int main()
6. int x=500, y=100;
7. cout<<"Value of x is: "<<x<<endl;
8. return 0;

Output:

```
Value of x is: 100
Value of y is: 500
```

Difference between call by value and call by reference in C++

Inline Functions in C++

Inline function is one of the important feature of C++. So, let’s first understand why inline functions are used and what is the purpose of inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function’s code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small.
The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
    // function code
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:
1) If a function contains a loop. (for, while, do-while)
2) If a function contains static variables.
3) If a function is recursive.
4) If a function return type is other than void, and the return statement doesn’t exist in function body.
5) If a function contains switch or goto statement.

Inline functions provide following advantages:
1) Function call overhead doesn’t occur.
2) It also saves the overhead of push/pop variables on the stack when function is called.
3) It also saves overhead of a return call from a function.
4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.
5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:
1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

```
#include <iostream>
using namespace std;
inline int cube( int s)
{
    return s*s*s;
}
int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
} //Output: The cube of 3 is: 27
```

Inline function and classes:
It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.
For example:

```
class S
{
public:
    inline int square( int s) // redundant use of inline
    {
        // this function is automatically inline
        // function body
    }
};
```

The above style is considered as a bad programming style. The best programming style is to just write the prototype of function inside the class and specify it as an inline in the function definition.
For example:

```
class S
```

```
{
public :
    int square( int s); // declare the function
};

inline int S::square( int s) // use inline prefix
{

}
```

The following program demonstrates this concept:

```
#include <iostream>
using namespace std;
class operation
{
    int a,b,add,sub,mul;
    float div ;
public :
    void get();
    void sum();
    void difference();
    void product();
    void division();
};
inline void operation :: get()
{
    cout << "Enter first value:" ;
    cin >> a;
    cout << "Enter second value:" ;
    cin >> b;
}

inline void operation :: sum()
{
    add = a+b;
    cout << "Addition of two numbers: " << a+b << "\n" ;
}

inline void operation :: difference()
{
    sub = a-b;
    cout << "Difference of two numbers: " << a-b << "\n" ;
}

inline void operation :: product()
{
    mul = a*b;
    cout << "Product of two numbers: " << a*b << "\n";
}

inline void operation ::division()
{
    div =a/b;
    cout<< "Division of two numbers: " <<a/b<< "\n" ;
}

int main()
{
    cout << "Program using inline function\n";
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}
```

Output:

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

What is wrong with macro?

Readers familiar with the C language knows that C language uses macro. The preprocessor replace all macro calls directly within the macro code. It is recommended to always use inline function instead of macro. According to Dr. Bjarne Stroustrup the creator of C++ that macros are almost never necessary in C++ and they are error prone. There are some problems with the use of macros in C++. Macro cannot access private members of class. Macros looks like function call but they are actually not.

Example:

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

```
class S
{
    int m;
public :
#define MAC(S::m)    // error
};
```

C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. Preprocessor macro is not capable for doing this. One other thing is that the macros are managed by preprocessor and inline functions are managed by C++ compiler.

Remember: It is true that all the functions defined inside the class are implicitly inline and C++ compiler will perform inline call of these functions, but C++ compiler cannot perform inlining if the function is virtual. The reason is call to a virtual function is resolved at runtime instead of compile time. Virtual means wait until runtime and inline means during compilation, if the compiler doesn't know which function will be called, how it can perform inlining?

One other thing to remember is that it is only useful to make the function inline if the time spent during a function call is more compared to the function body execution time. An example where inline function has no effect at all:

```
inline void show()
{
    cout << "value of S = " << S << endl;
}
```

The above function relatively takes a long time to execute. In general function which performs input output (I/O) operation shouldn't be defined as inline because it spends a considerable amount of time. Technically inlining of show() function is of limited value because the amount of time the I/O statement will take far exceeds the overhead of a function call.

Depending upon the compiler you are using the compiler may show you warning if the function is not expanded inline. Programming languages like Java & C# doesn't support inline functions.
But in Java, the compiler can perform inlining when the small final method is called, because final methods can't be overridden by sub classes and call to a final method is resolved at compile time. In C# JIT compiler can also optimize code by inlining small function calls (like replacing body of a small function when it is called in a loop).

Last thing to keep in mind that inline functions are the valuable feature of C++. An appropriate use of inline function can provide performance enhancement but if inline functions are used arbitrarily then they can't provide better result. In other words don't expect better performance of program. Don't make every function inline. It is better to keep inline functions as small as possible.