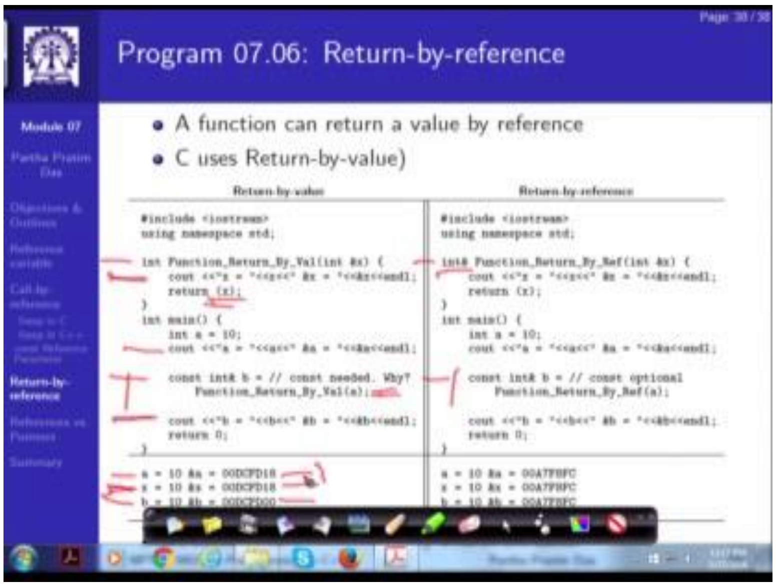
Programming in C++

Prof. Partha Prathim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 11
Reference and Pointer (Contd.)

Welcome to module 7 in Programming in C++. We have discussed the basic concept of reference, and using that we have introduced the notion of call by reference. And, we have shown that how certain functions like swapping can be written in a better way in C++ by using the call by reference mechanism. We have also discussed that a reference parameter can be a general input/output parameter for a function. Therefore, if I do not want the function to change the formal parameter and make subsequent changes to the consequent changes to the actual parameter, then we also have the option of defining a reference parameter to be a const parameter by which it can only be initialized by the actual parameter. But, any changes made to that will not be allowed by the compiler. We will now next continue and talk about the other side of the function mechanisms.

(Refer slide Time: 01:29)



Program 07.06: Return-by-reference

- A function can return a value by reference
- C uses Return-by-value)

Return-by-value

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

int Function_Return_By_Val(int &x) {
    cout <<"x = "<<x<<" &x = "<<&x<<endl;
    return (x);
}
int main() {
    int a = 10;
    cout <<"a = "<<a<<" &a = "<<&a<<endl;

    const int& b = // const needed. Why?
        Function_Return_By_Val(a);

    cout <<"b = "<<b<<" &b = "<<&b<<endl;
    return 0;
}
```

```
a = 10 &a = 00DCFD18
x = 10 &x = 00DCFD18
b = 10 &b = 00DCFD00
```

Return-by-reference

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

int& Function_Return_By_Ref(int &x) {
    cout <<"x = "<<x<<" &x = "<<&x<<endl;
    return (x);
}
int main() {
    int a = 10;
    cout <<"a = "<<a<<" &a = "<<&a<<endl;

    const int& b = // const optional
        Function_Return_By_Ref(a);

    cout <<"b = "<<b<<" &b = "<<&b<<endl;
    return 0;
}
```

```
a = 10 &a = 00A7F8FC
x = 10 &x = 00A7F8FC
b = 10 &b = 00A7F8FC
```

In C, we use the mechanism to get a value back from a function; is called return by value mechanism. That is, as parameters are copied from actual to formal, the return value is also copied from the return value expression, which we write in the return statement back to wherever we are assigning that function value. So, in contrast in C++ it is possible to return a value by reference. Let us see what we are saying.

So, first let us focus on the two sides. On left, we have return by value; on right, we have return by reference. So, on left we have a function which is a typical c return form in C++. It is only using a call by reference. And on right, please note that after the return type we have the reference symbol. So, if it says that we are returning the reference of the return expression. And then, the use is similar. Here, we invoke this function; here we again invoke this function. And to keep the result of the function, and this is primarily for the purpose of illustrating the effect to you.

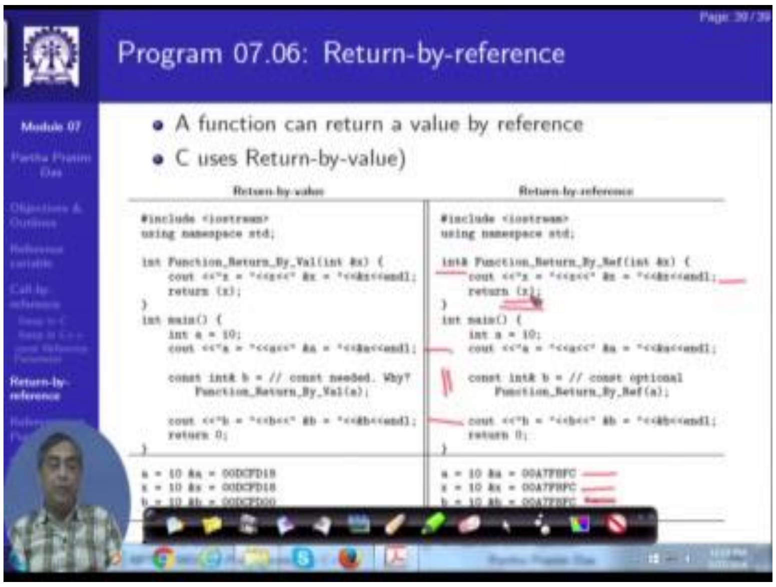
I have used another reference variable b. So, b will keep the reference to the value that is returned. If you look into the return by value part that is on this side, you will need to understand that this reference b must be a constant. Why should this be a constant because if you refer to the equal; the pitfalls of reference definition, we had shown that if I have an expression j plus k, I cannot create a reference to that because j plus k is computed in a temporary.

Now, here what do I have on the right hand side? I have a function invocation. What is a function invocation? It is an expression. So, I cannot write a difference to that. I can have to write a constant reference to it, which is the temporary location which will be preserved through this reference. On this, writing the constant is just for the sake of uniformity. It is not actually required. You will understand that once you understand the mechanism.

Let us look at now the output. So, the first output that will come the cout which is here, which prints a, and its address. Next is within the function. The function has been called. After this print, the function has been called here. So, next is this cout, which is this output where you print x. It is a call by reference. So as expected a, and x have the same address.

Finally, you look at this third output which will come from main, after the function has returned. And, what does a function do? Function simply returns a value that you had passed it. So, it is expected to return the same value. So, b returns the same value. But, if you print address of b here, it is different from the address of a, or address of x. And, this is expected; because it is returning by value. So, what it is returning is a copy of x; which is in a temporary, and I have hold that temporary. I have held that temporary is a part of b; as a reference in b.

(Refer slide Time: 05:52)



Program 07.06: Return-by-reference

- A function can return a value by reference
- C uses Return-by-value)

Return-by-value

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

int Function_Return_By_Val(int &x) {
    cout <<"x = "<<x<<" &x = "<<&x<<endl;
    return (x);
}
int main() {
    int a = 10;
    cout <<"a = "<<a<<" &a = "<<&a<<endl;

    const int& b = // const needed. Why?
        Function_Return_By_Val(a);

    cout <<"b = "<<b<<" &b = "<<&b<<endl;
    return 0;
}
```

```
a = 10 &a = 00DCFD18
x = 10 &x = 00DCFD18
b = 10 &b = 00DCFD00
```

Return-by-reference

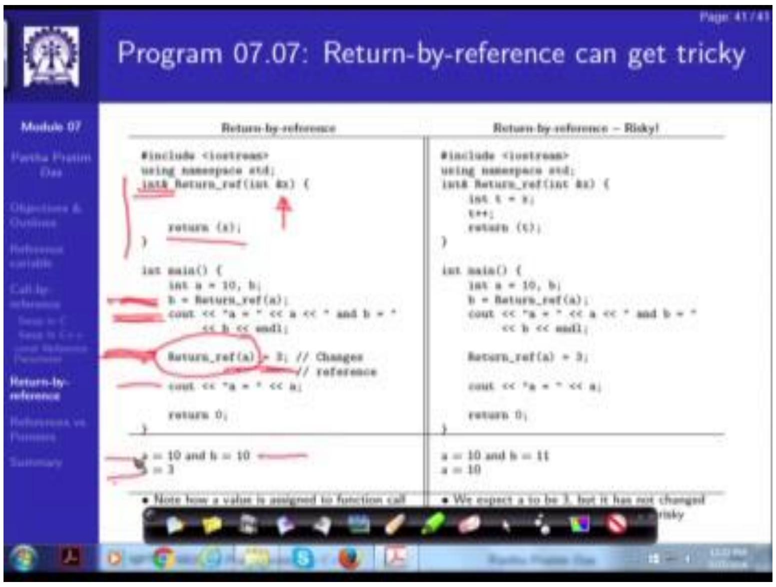
```
#include <iostream>
using namespace std;

int& Function_Return_By_Ref(int &x) {
    cout <<"x = "<<x<<" &x = "<<&x<<endl;
    return (x);
}
int main() {
    int a = 10;
    cout <<"a = "<<a<<" &a = "<<&a<<endl;

    const int& b = // const optional
        Function_Return_By_Ref(a);

    cout <<"b = "<<b<<" &b = "<<&b<<endl;
    return 0;
}
```

```
a = 10 &a = 00A7FBFC
x = 10 &x = 00A7FBFC
b = 10 &b = 00A7FBFC
```

Think about the similar on the reference side. Here, we are looking at call by, return by reference. The first output comes from here which is, a and its address. Second is from the function which is x and its address. Call by reference; they have to be identical. they are indeed identical. The third output comes from here after the function has returned. You see that it is not only b is same, this address of b is also same; because what has actually been returned, what has been returned is not the value of x.

But, what has been returned is a reference that is the address of x itself; the reference of x, which is an alias of x. So, b now becomes an alias of what was x. And, what was x? x was an alias of a. So, b has in this process become an alias of a. So, that is the difference between return by value and return by reference. Again if you do return by reference, then it will show some interesting and tricky issues. But if you do return by reference, then again we can avoid copying of large structures if that is what we want to return. So, in many places you may want to return by reference.

(Refer slide Time: 07:22)



Page 41/41

Program 07.07: Return-by-reference can get tricky

Module 07

Partha Pratim Das

Objectives & Outlines

Reference variable

Call-by-reference

Swap in C

Swap in C++

Const Reference Parameter

Return-by-reference

References vs Pointers

Summary

Return-by-reference	Return-by-reference – Risky!

`#include <iostream>`
`using namespace std;`
`int& Return_ref(int &x) {`
 	
`    return (x);`
`}`
 	
`int main() {`
`    int a = 10, b;`
`    b = Return_ref(a);`
`    cout << "a = " << a << " and b = "`
`         << b << endl;`
 	
`    Return_ref(a) = 3; // Changes`
`                       // reference`
`    cout << "a = " << a;`
 	
`    return 0;`
`}`
	`#include <iostream>`
	`using namespace std;`
	`int& Return_ref(int &x) {`
	`    int t = x;`
	`    t++;`
	`    return (t);`
	`}`
	
	`int main() {`
	`    int a = 10, b;`
	`    b = Return_ref(a);`
	`    cout << "a = " << a << " and b = "`
	`         << b << endl;`
	
	`    Return_ref(a) = 3;`
	
	`    cout << "a = " << a;`
	
	`    return 0;`
	`}`

`a = 10 and b = 10`
`a = 3`
	`a = 10 and b = 11`
	`a = 10`

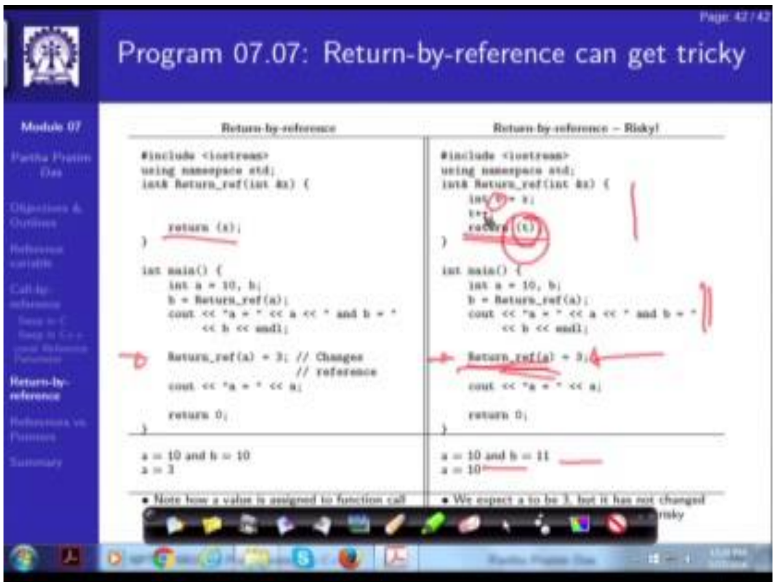
• Note how a value is assigned to function call	• We expect a to be 3, but it has not changed [illegible] risky

Now as I said, return by reference can get tricky at times. Look at here. There is a function which takes a call, takes a parameter by reference and returns that same parameter by reference. And, look at this line and first we will have to believe your eyes that what you are seeing is correct program. You have never seen a function invocation occurring on the left hand side of an assignment. Function invocation is always on the right hand side. So, here this is valid; because what is the function returning? The function is actually returning an alias. It is actually returning an address; it is actually returning a variable. So, if I say that I am making an assignment free to it, then I am actually making a assignment to that alias.

So, let us see what it means. Let us look at the output. We have the outputs here. So, this is the cout, the function was called at this point with a, x is 10, which is a is 10. So, it returned the same x. So, b has become 10. So, when I look at this output, it is a is 10, b is 10. No surprise. I call it once more here. And, assign 3 to the return reference. So, what will it mean? if I am calling it with a, then a and x become alias. x is an alias of a. I am returning by reference. So, what I returned is an alias of x, which has to be an alias of a. So, what I have got here as return is actually an alias of a. So, you can always make an assignment 3 to it. Where will that assignment happen? It is an alias of a. So, assignment will happen in a. So, check a out of this. a has become 3. So, this is what gets possible if you do a return by reference.

Now, we will subsequently as we go forward, to get to later features, we will show how this kind of a tricky thing and little bit confusing thing can be used for advantage in some places in the program. I am not advising that you write this kind of code very frequently. But, there are places where this can be used to advantage in a good way.

(Refer slide Time: 10:44)



Program 07.07: Return-by-reference can get tricky

Return-by-reference

```
#include <iostream>
using namespace std;
int& Return_ref(int &x) {


    return (x);
}

int main() {
    int a = 10, b;
    b = Return_ref(a);
    cout << "a = " << a << " and b = "
         << b << endl;

    Return_ref(a) = 3; // Changes
                       // reference
    cout << "a = " << a;

    return 0;
}
```

```
a = 10 and b = 10
a = 3
```

- Note how a value is assigned to function call

Return-by-reference – Risky!

```
#include <iostream>
using namespace std;
int& Return_ref(int &x) {
    int t = x;
    t++;
    return (t);
}

int main() {
    int a = 10, b;
    b = Return_ref(a);
    cout << "a = " << a << " and b = "
         << b << endl;

    Return_ref(a) = 3;

    cout << "a = " << a;

    return 0;
}
```

```
a = 10 and b = 11
a = 10
```

- We expect a to be 3, but it has not changed

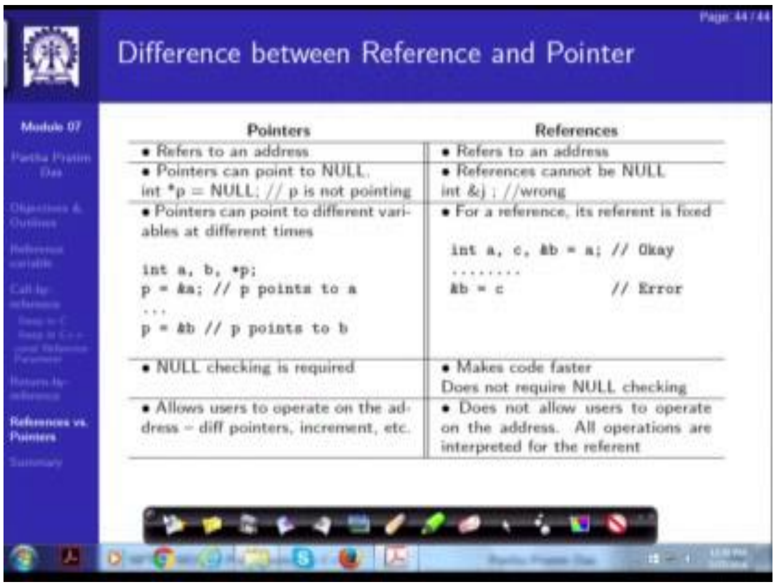
And, using this on the right column which you had not seen so far, I show that if you are not skilled, you might shoot at your foot in trying to do this. Look at this code, which is very similar to what the earlier code was. The only difference being that it now takes the parameter x, which is naturally an alias of a, because it is call by reference and it has a local variable t and it initializes t with x and then increments t.

And then, look at here, earlier it was returning x. Now, it returns t which is this local variable as a reference. And, you did a same thing here. The first two lines are at similar. They produce a same output. So, there is nothing to look at. Look at this line. You have done this here. So, if you do this and try to expect that, some change will happen to a, or something like that. You will be surprised that nothing will happen; because what you have done? You have actually returned a local variable. So, what gets returned here is a reference to the local variable t. And that is very risky; because once you have returned the reference, the function call is over. So, the local variable has disappeared, but your reference is still alive. So, your reference says that I have the variable. The function call has terminated. So, it actually does not exist. This variable t no more exists, that is dead.

But, you hold a reference to that. So, the results of this kind of program could be very unpredictable.

So, the bottom line prescription is if you are using return by reference, never return a local variable. If you are returning by reference, always return a variable which is available, which should logically be live after the function call has ended. So, those variables could be global, those variables as we will see could be static members, those could be the parameters that you got as alias in the formal parameter, those could be dynamically allocated values and so on. But, not the automatic local variables that a function has because that can get you into real difficulty and these are bugs which are very difficult to detect; because you will not see anything in the code. On the code, it looks everything clean and simple. But, still your results will become unpredictable.

(Refer slide Time: 13:32)



Page 44 / 44

Difference between Reference and Pointer

Module 07

Partha Pratim Das

Objectives & Outlines

Reference variable

Call-by-reference

Swap in C

Swap in C++

const Reference Parameter

Return-by-reference

References vs. Pointers

Summary

Pointers	References
• Refers to an address	• Refers to an address
• Pointers can point to NULL. int *p = NULL; // p is not pointing	• References cannot be NULL int &j ; //wrong

• Pointers can point to different variables at different times
`int a, b, *p;`
`p = &a; // p points to a`
`...`
`p = &b // p points to b`
	• For a reference, its referent is fixed
	`int a, c, &b = a; // Okay`
	`.........`
	`&b = c            // Error`

• NULL checking is required	• Makes code faster Does not require NULL checking
• Allows users to operate on the address – diff pointers, increment, etc.	• Does not allow users to operate on the address. All operations are interpreted for the referent

But now, finally as we have seen through this module, reference is talking about an alias of a variable it is a mechanism which allows you to change the value of a variable without actually taking the name of the variable. So, in that way it has got lot of similarity and differences with the pointers.

So, I will just summarize this difference in these two columns. So, between the pointers and references, both refer to addresses. Pointers refer to address; reference also refers to an address. So, they are similar to that extent, but they differ in multiple ways. For example, look at the next point. Pointers can point to null; I can assign a pointer to null.

What it means? Semantically, it means that I am not pointing anywhere. It has no pointed data that I carry. But, reference cannot be null; because it is an alias. It is just an alias. So, it has to have a variable to effect. So, that is a basic difference.

Since pointer points to different other data, unless a pointer is constant, I can actually change the data that it is pointing to. So, the example clearly shows that you can do that. If p was initially pointing to a, then it at some point statements, it can point to b. But, an reference; for a reference, what you are referring to is fixed by the definition; because it is an alternate name for a variable. So, certainly I cannot change that reference and make the name different.

For example, if we just look into, if we just look into this line, if we just look into, if you are trying to do like this, then you will not be able to achieve that if you write it as ampersand b assigned c, thinking that you will change the reference of b to from a to c. You will not able to do that because a moment you write ampersand b. Since, it is an alias for a ampersand b will be understood as ampersand a because this is an alias. So, whatever you write to as b is what should apply to a, because this is an alias of a. And then, ampersand a is a address of a; ampersand is the address operator.

So, what do you saying that you trying to assign c to the address of a, which is the meaningless thing. So, you can try in all possibilities. For example, if you try to say okay, am, I would like to change the reference by putting c to b, but then b is a. So, if you try to assign c to b, you are basically assigning c to a. There is no way that the language does not give you any mechanism, any operator to do anything with the reference. You can only refer. And, whatever you do actually is interpreted in terms of the data that is referred, the referent. So, that is a basic difference.

For pointers, since it is possible that I am not pointing anywhere. Before using a pointer, we need to check for null. Reference does not need that because it is, if it exists, then it is alias of something or it does not exist. So, it makes the code much easier to write. You do not have to bother about that and certainly it makes it faster in that sense because those checks are not required.

Finally, if you look into exactly if both pointers and references are addresses, and exactly what is the difference? The difference is not in the empowerment, not in terms of what you get for reference. The difference is in terms of what you do not get for reference. If you have a pointer variable, then you are given with a number of operators that you can operate with that variable. You can actually use that address as a value and do different things. You can add an integer to it and advance a pointer. You can take two pointer values and make a difference and see how many elements exist between these two pointers in an array.

So, all these operators are given to you, by which you can change the pointer in whatever way you want; which makes pointer so powerful. In reference, also the address is stored. But, there is no way that you can catch hold of that address. You cannot hold that address. There is no operation that is given on the reference. Any operation that you try to do, actually boils down to operating on the referred object or the referent. So, that is the basic difference.

And, in the way C++ is designed, both pointers and references have their own respective place. It is not possible to do away with pointers and reference really had a value, but there are languages where you do not have both these mechanisms. So, it is good to understand. Particularly, if some you know java you would know that Java has a reference, does not have a pointer. And, I am just raising this point because I want you to note that the reference in java is not like the reference in C++. So, if you thought that let me have more clarification in the concept of reference and read the favorite java book, then you will only get more utterly confused. And, you will get. So, there is no real parallel of the reference of java in C++, but it is mostly similar to constant pointer. What you have in java is a reference is actually a pointer, but it is a constant pointer that you cannot change it. But, it is not. It is neither fully a pointer nor fully a reference in C++. And, there are languages which do not; like c it does not have reference.

There are languages which do not have pointers and so on. In C++ we have both. So, we have a big responsibility to decide in a given context, whether we need to use the pointer or we need to use a reference. And as we go along, different features and different programming examples will keep on highlighting that the choice has to be judicious and right for you to become an efficient C++ programmer.

So in the module, in this module 7, we have introduced the concept of reference in C++ and we have studied the difference between call by value and call by reference. We have also introduced the concept of return by reference and studied the difference between the two mechanisms. We have shown some interesting pitfalls, tricks, tricky situations that may arise out of this and we have discussed about the differences between references and pointers.