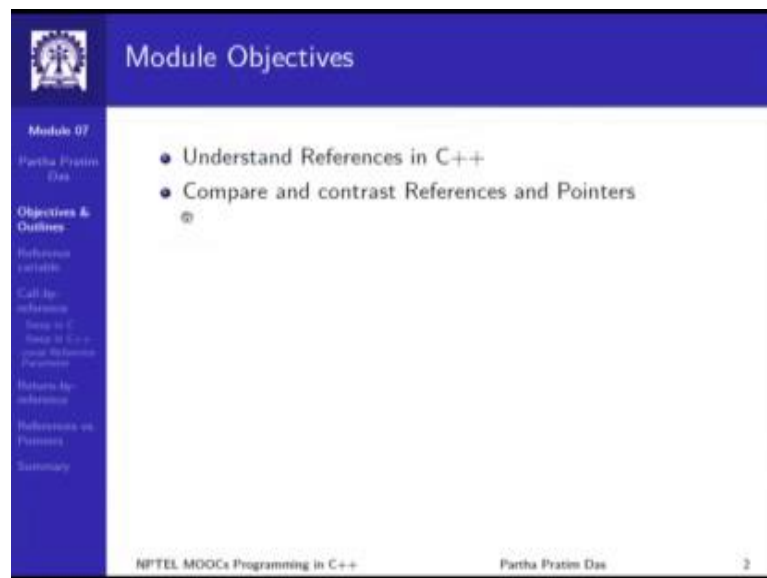


Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 10
Reference and Pointer

Welcome to module 7 of programming in C++. We will continue to discuss the procedural extension of C into C++. We have introduced that in module 6 and discussed the two basic concepts of cv-qualifier, constant and volatiles qualifier and the use of inline functions.

(Refer slide Time: 01:21)



The slide is titled "Module Objectives" and features a blue header with the IIT Kharagpur logo. A left sidebar lists the module's content, with "Objectives & Outlines" selected. The main content area contains two bullet points: "Understand References in C++" and "Compare and contrast References and Pointers". The footer includes "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the page number "2".

In this module, we will introduce another concept called reference. This concept of reference is a closely related, but is very different from the concept of pointer. So, we will also; as we go through the module, we will also try to compare and contrast between the reference and the pointer. So, their objective is to understand reference and understand this contrast.

(Refer slide Time: 01:29)

Reference

Module 07
Pointer Program
Day

Objectives & Outcomes
Reference variable

- Call for reference
- Step in C
- Step in C++
- Step in C++
- Reference

• A reference is an alias / synonym for an existing variable

```
int i = 15; // i is a variable
int &j = i; // j is a reference to i
```

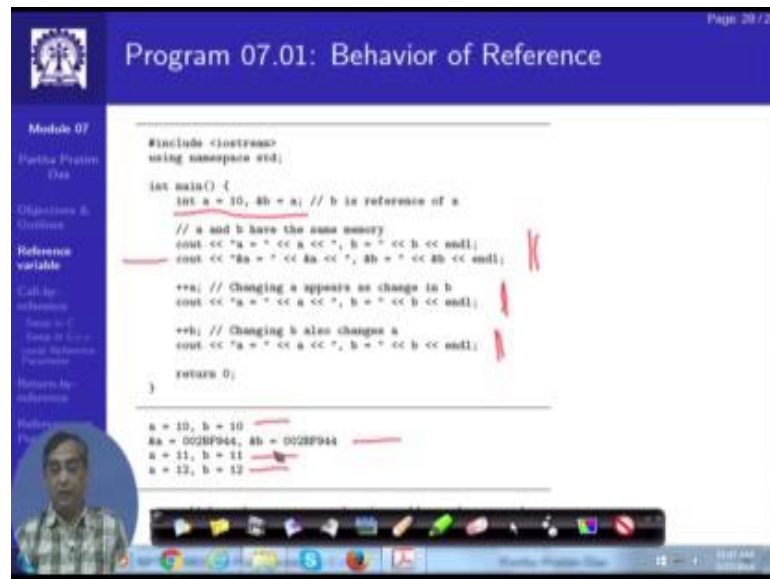
i ← variable
15 ← memory content
200 ← address
j ← alias or reference

11:04 AM

These are the different specific topics that we will go through. A reference is like an alias or a synonym for an existing variable. So, alias is like what we have in terms of our own names. We have some given name. We have some pet names and we can be referred, called it by either of the names. So, here reference variable also has that similar purpose. So, `i` is a variable which is declared here. It is initialized to 15 and in this context we have a variable `i`, which is defined. It has an initial value 15. Now, I define another variable and particularly look into this use of the ampersand symbol here. We define a variable `j` and initialize it with `i`. Such a variable `j` is called a reference to `i` or a reference variable for `i`.

So, this reference variable is actually an alternate name and alias name for `i`. So, if I look into the memory, both `i` and `j` will actually represent the same memory location. So, if `i` happens to have a memory address 200, as I see, show here below if this is the address of `i` and its content is 15, the address of `j` will also be, this will also be 200 as of `i`. So, it is immaterial. After this, this particular reference `j` has been introduced it is immaterial as to whether I refer `i` as `i` or I refer `i` as `j`. That is the basic concept of a reference.

(Refer slide Time: 03:40)



The screenshot shows a presentation slide with a blue header containing the text "Program 07.01: Behavior of Reference" and "Page 28/28". On the left side, there is a sidebar with a logo and a list of topics: "Module 07", "Hello Program", "Day", "Objectives & Overview", "Reference variable", "Call by reference", "Step by Step", "How to Compile", "Programs", "Return by reference", and "Reference". The main content area displays C++ code for a program that demonstrates reference variables. The code includes comments and output statements. Red annotations highlight the memory addresses of variables 'a' and 'b' in the output, showing they are identical. A small video inset of a man is visible in the bottom-left corner of the slide.

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

int main() {
    int a = 10, &b = a; // b is reference of a
    // a and b have the same memory
    cout << "a = " << a << ", b = " << b << endl;
    cout << "&a = " << &a << ", &b = " << &b << endl;

    ++a; // Changing a appears as change in b
    cout << "a = " << a << ", b = " << b << endl;

    ++b; // Changing b also changes a
    cout << "a = " << a << ", b = " << b << endl;

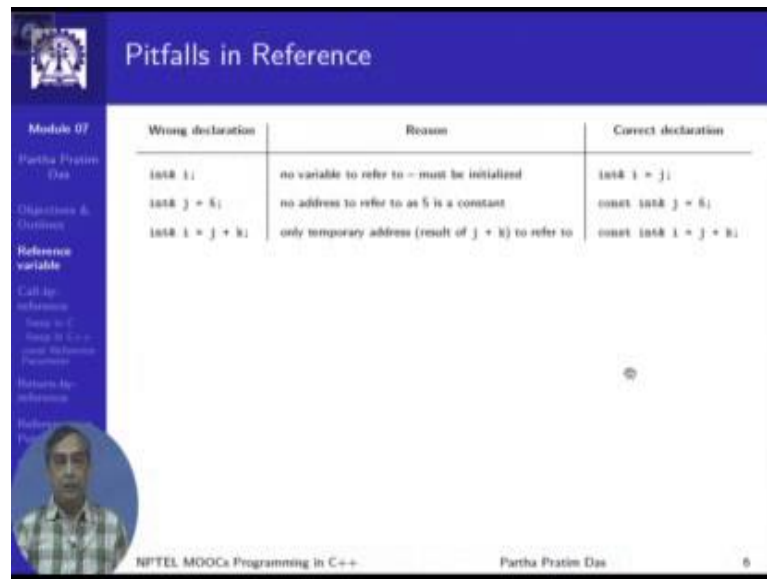
    return 0;
}
```

a = 10, b = 10
&a = 002BF944, &b = 002BF944
a = 11, b = 11
a = 12, b = 12

So, let us concentrate on a small program to understand the behavior of reference. In this program, I will show you there is a variable a and b is set as a reference of a. Then in the next two output statements, cout, we first print the value of a and b. we can see the output here; the 10 and 10; because a is 10 and b is a alias of a. So, if I print b, also I will print 10. And in the second line of the cout, we print that is the line here. We print the address of a, and we print the address of b. These are the addresses this line being printed. We can again check that they are identical addresses. That is they indeed are the same variable just that they have two different names.

So, let us try to change the value of the variable. So, here we increment a, and then output it again. If we increment a, it will become 11, so you can see here that a, has become 11. And b, even though now operation was done with b, b also has become eleven. And, you can do it other way. If you increment b, then b becomes 12. And a, the variable to which b is a reference has also become 12. That is, they are very strongly coupled together and any one can be used for the purpose of the other.

(Refer slide Time: 05:36)



The slide is titled "Pitfalls in Reference" and features a table with three columns: "Wrong declaration", "Reason", and "Correct declaration". The table lists three common errors in C++ reference declarations. A sidebar on the left contains navigation links for "Module 07", "Partha Pratim Das", "Objectives & Outcomes", "Reference variable", "Call by reference", "Pass by C", "Pass by C++", "Pass by Reference", "Return by reference", and "Reference". A small circular portrait of Partha Pratim Das is located in the bottom left corner of the slide content. The footer includes "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the slide number "6".

Wrong declaration	Reason	Correct declaration
<code>int& i;</code>	no variable to refer to – must be initialized	<code>int& i = j;</code>
<code>int& j = 5;</code>	no address to refer to as 5 is a constant	<code>const int& j = 5;</code>
<code>int& i = j + k;</code>	only temporary address (result of <code>j + k</code>) to refer to	<code>const int& i = j + k;</code>

So, this is the basic motion. Now, certainly if you try to do this you will have to be careful that there are certain very typical pitfalls that you can get yourself into. So, three common pitfalls I illustrate here; that could be more. That is, if we just try to define a reference, but without initializing it with a variable. Then, the compiler will give you an error because a reference is an alias for some other variable. So, unless you define a variable along with it, initialize a variable along with it, there is no referent to refer to. Therefore, it is wrong.

So, if you just see on the table, on the left I show the erroneous declaration and on the right most I show the corresponding correct version, and you can understand the reason as given. If I look into the next one, that is, if I look in here, then you are trying to do a reference to a constant value. This is also is an error because a constant value is just a value. It does not have an address to reside.

So, you cannot have reference to a value, but you can have a constant reference to a value; because it is a constant value. So, the reference also will have to be a constant one. Otherwise, you can you can think of the danger that will happen; is this is related to the dangerous as we showed in conciseness. That if I, if this were correct, suppose this were correct, then `j` is 5.

Now, what will happen if I do plus plus j? Whatever it is referring to will get incremented. So, this will become; so, a constant 5 will become a constant 6, which is not possible. So, it has to be defined as const, so that you cannot make any changes to it.

Similarly, in the next one if you look in here I have expression j plus k. and, I am trying to create a reference to that. But, the expression again does not have an address. The computation of j plus k is stored only as a temporary location and those temporary locations are not retained. So, again I cannot create a reference to that. If I want to have a reference, that reference will have to be a constant, which will refer to the value of j plus k as computed as the value of the reference.

So, it will not be; you will not be able to change that because the expression j plus k cannot be changed. So, if I make; if we allow a reference, then we would be able to change that; which is not semantically valid, if the reference is being made to an expression. So, all that you can say that. At this point of time, j plus k had some value; which is treated as a constant. And, I have reference to that. So, all those reference will have to be constant. There could be more pitfalls, but these are the common one. So, I just choose to discuss them.

(Refer slide Time: 08:48)

The screenshot shows a C++ program titled "C++ Program 07.02: Call-by-reference". The code is as follows:

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

void Function_under_param_test(int &b, int c) // Function prototype
int &b; // Reference parameter
int c; // Value parameter

int main() {
    int a = 20;
    cout << "a = " << a << ", &a = " << &a << endl << endl;
    Function_under_param_test(a, a); // Function call
    return 0;
}

void Function_under_param_test(int &b, int c) // Function definition
cout << "b = " << b << ", &b = " << &b << endl << endl;
cout << "c = " << c << ", &c = " << &c << endl << endl;
}
```

The output of the program is:

```
----- Output -----
a = 20, &a = 0023FA30
b = 20, &b = 0023FA30
c = 20, &c = 0023F96C
```

Red arrows in the original image point to the memory addresses for &a, &b, and &c, showing that &a and &b have the same address (0023FA30) while &c has a different address (0023F96C).

Below the output, there are two bullet points:

- Param b is call-by-reference while param c is call-by-value
- Actual param a and formal param b get the same value in called function

Now, why are we doing this? So, let me introduce a totally new concept of passing parameters to functions. We know how to pass parameters to function from c. It is called call by value. Just for a quick recap, the function as defined has formal parameters. When it is called, I have actual parameters. They correspond in the order of position and at the time of call, each actual parameter's value is copied to the formal parameter and then function is called. So, when the call is done, the function is in call. Actual parameters reside in some memory corresponding formal parameters reside in distinctly different memory. In contrast, we can do what is known as call by reference.

So, I would like you to focus on this particular line. To a trying, we have given a function header. Function under param test is a prototype. Look at this first parameter where we write the parameter name prefixed with ampersand, which is the notation for reference. Such a parameter is called a reference parameter. I have also another parameter c in that function, which is the typical value parameter and we will follow the call by value rules.

Now, let us look into this part of use. So, use that and we choose to have one variable a, and call the function. That is, as actual parameter we pass a in place of both the formal parameters. Then, this is the definition of the function; where given the two parameters, we just print their value and we just print their addresses. Now, let us look into the output. So, if I, if we do this, then the first output will come from this slide before the function call, which is this output, which tells me a is at this location. And, this is the address. The second output will come from this cout, which prints b and gives the b's address.

Look at something very interesting. The address of b is exactly same as address of a, something that we do not expect in a call by value mechanism. And, to just show that what call by value would have done, you would look at the third output line c out and the corresponding output here of the parameter c. And, you do find that even though c also has the same value as of a, which it should, the address of c is different. So, this clearly show that between the two parameters b and c, c is following the original call by value rule by which is just the value of a which is copied to c. a and c continue to exist in two different memory locations, whereas, b has basically become a different name for a. It has become an alias for a.

These symptoms as we have just seen at the symptoms of alias that when I have two variables differently named, but they enjoy the same memory location and hence the same value. So, that is a reference variable. So when we use parameters, reference variables in parameters, we call them as call by reference mechanism. So, in call by reference the actual parameter and the formal parameter will have different names in the caller and the callee, but they will have the same memory location. So, this is the new feature that C++ allows us to do. Here, the all that I said are written at the bottom part of the slide. You can just read that.

(Refer slide Time: 13:15)

C Program 07.03: Swap in C

Call-by-value	Call-by-address
<pre>#include <stdio.h> void swap(int, int); // Call-by-value int main() { int a = 10, b = 15; printf("a = %d & b = %d to swap\n", a, b); swap(a, b); printf("a = %d & b = %d on swap\n", a, b); } void swap(int a, int b) { int t; t = a; a = b; b = t; }</pre>	<pre>#include <stdio.h> void swap(int *, int *); // Call-by-address int main() { int a=10, b=15; printf("a = %d & b = %d to swap\n", a, b); swap(&a, &b); printf("a = %d & b = %d on swap\n", a, b); return 0; } void swap(int *x, int *y) { int t; t = *x; *x = *y; *y = t; }</pre>
<ul style="list-style-type: none"> • a = 10 & b = 15 to swap • a = 10 & b = 15 on swap 	<ul style="list-style-type: none"> • a = 10 & b = 15 to swap • a = 15 & b = 10 on swap
<ul style="list-style-type: none"> • Passing values of a=10 & b=15 • In callee; c = 10 & d = 15 • Swapping the values of c & d 	<ul style="list-style-type: none"> • Passing Address of a & b • In callee x = Addr(a) & y = Addr(b) • Value of the addresses is swapped

Handwritten notes on the slide:
 - Red circles around the parameter lists in both functions.
 - Red arrows pointing from the parameter lists to the function calls.
 - Red text: "x = a", "y = b" next to the pointer parameters in the call-by-address version.
 - Red text: "in caller the way out" at the bottom right.

Now, we will just preside for a while. We have just learnt, what is call by reference mechanism. But, we still, you will still wonder as to why we are trying to do this. So, to appreciate why we are trying to do this, I take an example from C. And, this is one example which every one of you, who have done little bit of C would know I am trying to write a swap function. It will, which will take two variables a, and b and try to swap them. So, this is what we are focusing on. This is a signature.

If I write this function, if I write this code I am calling swap here and this print out at to show what is the value of a and b. So, this is a first print which comes from this particular printf; which shows that a, and b to swap have values 10 and 15, as they are

initialized. Then, I go to swap. So, c and d becomes 10 and 15, the code of swap swaps them and back, I print again. But, unfortunately the values, actual parameters have not been swapped. So, the swap did not work and this is to be expected. Because what is the mechanism? The mechanism is call by value; which means that when I have call the function swap, the function has taken the value of a copied it to c, taken value of b copied it to d. And then, the function has done whatever it had to do. It had swapped c and d. But, these c and d have different locations than a, and b. So, nothing has happened to a, and b. They are well protected as actual parameters. So, when the function comes back a, and b are as same. They have not been swapped. So, swapping does not work this way.

So, we have learnt in C++, in C we cannot write the swap this way. So, what do I have to do? I have to do some tricking around; some tricks. So, what trick do we do? The trick that we try to do is define swap in this way. I have mentioned this as called by address; call by address is not a very formally accepted name. It is actually called by value, but the only difference is that here the parameter that we pass is a pointed type parameter.

So, we say instead of swapping two integers, now we will swap two pointers to integers. Therefore, this code is also written with the pointers where, and since these are the pointers. Whenever I have to refer to the first, I have to it star x now and the second by star y and we will do that. So, since these are pointers if I have to call them at this slide, the two actual parameters a, and b will have to be passed as the address of a, and address of b. So, we pass the two addresses.

Now, what is happening is x. If I look at, x is a pointer to a. y is a pointer to b. Now, what it is doing? It is trying to swap the value of start x and start y. It is not changing the value of x and y. It is not swapping these. It is swapping the values of star x and star y. What is star x? Star x is actually a. If x is the address of a, then star x is actually a. So, when I swap star x and similarly star y, it is actually b. So, when I swap star x with star y, I am actually swapping a with b.

So, what I am doing is basically since call by value will not allow me to make changes to the actual parameter, I am sending their addresses out and remotely I am allowing the

swap function to use the address to actually refer to the actual parameters, and then make the change. So, in a way I am breaking the rule of call by value because without that I cannot bring the changes back.

So, here we will have; we see that. Since we are not being able to swap the values directly, we use the call by address kind of mechanism, where we pass the addresses and access those values through those addresses. Certainly, it is kinds of a back door to get achieve the result. And, why are we doing this? What is the fundamental symptom for which we have to do this? Swap may be one specific instance. The symptom is if we do call by value, then the actual parameter is copied to the formal parameter. So, whatever you do in the function with the formal parameter, those effects will never come back to the actual parameter. This is what we wanted. But, in swap we need the changes in the formal parameter to come back to the actual parameter. Only then, swap can happen. If I am swapping a, and b, unless a can change and unless b can change, swap will not happen.

So to generalize, call by value allows us to have only input kind of parameters; whether values can go from the caller function to the called function. But, call by value does not allow me to do output kind of parameter; that is, I cannot compute a value in the function and get it back to the parameter. So, as such function returns only one value. So, if I want a function where from which I need more than one output, I have no mechanism, straight mechanism in C. Call by value fails. So, the only other mechanism is to use the addresses and do it in a roundabout way.

Now, let us see. Now, let us try to combine the two factors we have talked up. One, we have introduced the concept of reference, call by reference and we have talked about the difficulty of having multiple output from a function, from a C function. So, let us look at particularly in the swap example again. Now, let us. On the left hand side, we have the call by value example. The example that we saw is actually wrong because it cannot change because it should not change. This is the call by value prototype.

Now, here in C++ all the only change that we have made is instead of having call by reference, call by value. We are now saying that we have two parameters, which are

called by reference. That is the parameters are not usual parameters. They are reference parameters. And, now you write the whole thing with the reference parameter. Rest of it, rest of the code between what you see in the C and what you see in C++ are same. These are the just two places where changes are made.

What is the effect? If it is a call by reference, then when this particular call happens, the actual parameter a, and the formal parameter x here, they are, they enjoy the same location; x is another name for a, y is another name for b. So, if I interchange x and y it is same as interchanging a, and b. So, I can get the whole effect in my code. So, what I gain by doing this? Several things; one is I do not need to take the back door. If I want a parameter to be input, I make a call by value; if I want the parameter to be output as well, then I will do a call by reference because then the changes done in that formal parameter within the function will also be available in the actual parameters; because the actual parameter and the formal corresponding formal parameter enjoys the same address.

So, that is the advantage of using reference and doing the call by reference mechanism in C++. It certainly makes the code much cleaner. It makes it more easy and reliable to write programs. And, as we will see later on, it also often saves a lot of effort because in call by value you have to make a copy of that value. Now, as long as your value is an integer, may be making a copy is not costly. But, think of, if you are, if the parameter that you are passing is possibly a structure which may have say 10 kilo byte of size having 200 different members, 200 different components, then copying that itself may be a lot of cost. So, you would not like to incur that cost. You could just use reference.

There is a side effect of using reference parameter. Now, said that if we do a reference parameter, that is if we use call by reference, then the parameter actually is input and output parameter. It is input because if I, when the moment I am calling whatever the value that the actual parameter has, the formal parameter will also have the same value. So, it serves the purpose of input. So, actually I do not need to do call by value. I can only use call by reference. But, if I do call by reference, then the potential is that any changes made to the formal parameter will also get reflected in the actual parameter. So, any inadvertent change done in the formal parameter or intentional change is done in the formal parameter, will spoil my actual parameter. So, can we do something so that I can

use call by reference, but just make it an input parameter. So, to do that, you make use of the const.

So, what here? Just look at the code here. Particularly focus on this function, which is taking one formal parameter x of type int. But, what we have done? Before that we have said it is const. So, what does that mean? Because it is a call by reference because it is a reference parameter, so when I call it at this point as a, a, and x refer to the same address. But, I am saying that the reference x is constant; which means that no change of x is possible. This is a situation which is very similar to what we discussed in case of conciseness of pointer and the conciseness of pointed data. We are sure that the data itself may not be constant, but if I am holding a constant pointed to that data, then certainly that pointer will not allow me to change the data. So, similarly here I have a constant reference or rather I have a reference to the conciseness of the data. So, x will not allow me to change.

So, if within the function, I try to write something like plus plus x. This code will not compile because plus plus x being that I am trying to change the value of x. But, x is a constant reference. x is a constant value. So, whatever it got initialized with. What did it get initialized with? It got initialized with a. When the call was made that cannot be changed. So, now what happens? At the time of call the value of a will be available as a value of x. There is the same address. But x, you cannot change within the function. So, no way changes in a x can affect the actual parameter a. So result, the effect cannot come back. So, now, the parameter becomes purely an input parameter.

So, using a reference we can make either input output parameter or we can make input only parameter, if we make that a constant reference parameter. So, instead of using call by value, in a large number of cases we will try to use just constant reference parameter; because you would not need to do copy. And, we will still be protected that our actual parameter will not get affected.

So, on the right hand side we just show that, what is a proper way of doing it, where x is a constant reference and you do not try to. Here, we were trying to increment x and then return that. Here, we do not do that. We compute x with adding one with it and then

return that so. The code on the right hand side will compile and run well. There is no violation. The code on the left hand side will have compilation error and we will not be able to proceed with that any further. At this point, we will stop and we will continue in the next part.