

**Programming in Modern C++**  
**Professor Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture 07**  
**Reference & Pointer**

Welcome to Programming in Modern C++, we are in week 2. And now we are going to discuss module 7.

(Refer Slide Time: 0:37)

**Module Recap**

- Revisited manifest constants from C
- Understood `const`-ness, its use and advantages over manifest constants, and its interplay with pointers
- Understood the notion and use of `volatile` data
- Revisited macros with parameters from C
- Understood `inline` functions, their advantages over macros, and their limitations

Programming in Modern C++      Partha Pratim Das      M07.2

In the last module, where we started discussing extensions of C into C++ without the object-oriented features, but other features which makes it a kind of a better C and we discussed about how to replace macros without or with parameter by either const qualifier or by using inline functions. So, we have discussed constant volatile, the both types of cv qualifiers in that context and discussed about how inline functions can really help.

(Refer Slide Time: 1:27)

The slide is titled "Module Objectives" and features a dark blue header with a logo on the left and a small video feed of the presenter on the right. A vertical sidebar on the left lists navigation options. The main content area contains two bullet points. At the bottom, there is a footer with the text "Programming in Modern C++", "Partha Pratim Das", and "M07.3".

## Module Objectives

- Understand References in C++
- Compare and contrast References and Pointers

Programming in Modern C++ Partha Pratim Das M07.3

Now, today we will discuss about references, which is a very key idea in C++, which are similar to pointers, but a different from pointers.

(Refer Slide Time: 1:41)

The slide is titled "Module Outline" and features a dark blue header with a logo on the left and a small video feed of the presenter on the right. A vertical sidebar on the left lists navigation options. The main content area contains a numbered list of seven items. At the bottom, there is a footer with the text "Programming in Modern C++", "Partha Pratim Das", and "M07.4".

## Module Outline

- 1 Reference Variable
  - Pitfalls in Reference
- 2 Call-by-Reference
  - Simple C Program to swap
  - Simple C/C++ Program to swap two numbers
  - const Reference Parameter
- 3 Return-by-Reference
  - Pitfalls of Return-by Reference
- 4 I/O Parameters of a Function
- 5 Recommended Call and Return Mechanisms
- 6 Difference between Reference and Pointer
- 7 Module Summary

Programming in Modern C++ Partha Pratim Das M07.4

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  $\&i = \&j$  ✓  
 $j$  ← alias or reference ✓


Programming in Modern C++ Partha Pratim Das M07.6

This is the module outline as you can see on the left-hand side all the time. So, what is a reference variable? A reference variable is like an alias or a synonym kind of most of us have some name and we have some pet name. Say my name is Partha Pratim Das, my pet name most of my colleagues and students called me as PPD. So, PPD is an alias of my name.


So, this is another identity. So, the alias, if I define a variable, say  $i$  and initialize it with 15, let us say, I can define an alias of  $i$  by doing this where the key point syntax to be noted is before the alias variable or reference  $j$ , I need to put an  $\&$  and then after the initializer symbol, I put the variable that it is an alias 4.

So, if I look into that, this scenario now, in terms of what happens in the representation on the memory,  $i$  is the variable it has been initialized with the value 15. Let us arbitrarily assume that its address is 200,  $j$  is an alias or reference to  $i$  which means  $j$  is just another name for the same location. Therefore, if you look at the address, then the address of  $i$  if you compute and the address of  $j$  if you compute it will be the same value 200. So, this is the basic idea of your alias.

(Refer Slide Time: 3:42)



## Program 07.01: Behavior of Reference



Module M07

Partha Pratim Das

Objectives & Outlines

**Reference**

Review

Call by Reference

Sum of C

Sum of C++

Unit Review

Parthim

1/07 Patterns of a Function

Recommended Mechanisms

Reference in Position

Module Summary

```

#include <iostream>
using namespace std;

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

```

---

```

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

```

---


- a and b have the *same memory location* and hence *the same value*
- Changing one changes the other and vice-versa

Programming in Modern C++
Partha Pratim Das
M07.7


Now, let us look at the behaviour of an alias in the program. So, we have defined a, initialized it with the 10 and an alias of a which is b defined as with this &. So, b is a reference of a. So, a and b are necessarily the same variable, but 2 names of them, that is all. That is all the reference talks about. So, they necessarily have the same memory location.

So, if I print the memory location of a and b, as you will see here, the values are identical and therefore, the value the print are also identical. If I increment a, say I pre-increment a then a changes and therefore b also changes by that because b is just another name. So, when I print a and b after incrementing, both prints as 11. Instead, if I change b, a also changes because it is just b is just another name of a. So, for with another increment when I print a and b, they both become 12. So, a and b enjoyed the same location, they are the same variable, they will have the same value and changing one will necessarily change the other and vice versa. So, that is a basic property of a reference variable.

(Refer Slide Time: 5:13)



## Pitfalls in Reference



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

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

int main() {
    int i = 2; ✓
    int& j = i; ✓
    const int& k = 5; ✓ // const tells compiler to allocate a memory with the value 5
    const int& l = j + k; // Similarly for j + k = 7 for l to refer to

    cout << i << " ", " << &i << endl; // Prints: 2, 0x61fef8 ✓
    cout << j << " ", " << &j << endl; // Prints: 2, 0x61fef8 ✓
    cout << k << " ", " << &k << endl; // Prints: 5, 0x61fefc ✓
    cout << l << " ", " << &l << endl; // Prints: 7, 0x61fff0 ✓
}
```

Programming in Modern C++
Partha Pratim Das
M07.8

Now, there are a few can reference b, what are the kind of pitfalls that you can get into while defining a reference. So, on left I show some of the common wrong declarations that is, here I have tried to define a reference, but without an initialization. So, I am saying that PPD is the nickname of then I do not say who.

So, I am trying to create an alias where the original object, the original variable is not given. So, that is not possible it must be initialized. So, I can say like this, where i is an alias or reference for j. Here, I have created an alias j initialized it with a constant 5 this is also not allowed, why? Because an alias, how will the alias identify itself with the original variable?

Because it needs an address it needs the address of the variable. So, if this is done, then &i is the address and j necessarily uses that address which is computed as &j to be able to track that it is a second name for i, or rather, i is the second name for j, here i is the alias. Now, if I have a constant, constants do not have an address, they are just a value. They do not have they just have an rvalue, they do not have an lvalue.

So, I cannot use that to define an alias I cannot have an alias of a constant value, what they can do, if I want to do that, what I have to do is I have to define that as a constant reference, what does that mean? That means that this is a reference which is referring to an expression which is a constant. So, what the compiler will do?

Compiler will specifically allocate a location for j for 5 that j can track and whenever I talk about j, it will actually get the 5 from that location. Similar thing will happen if I have an expression on the initialization that I use again an expression has only a value, it does not have an address.

It whatever address it is computed in is a temporary address. So, I cannot have a reference to that, but if I make it const then the compiler will actually allocate an address where j plus k is computed and b then value of j plus k will be stored there. So, if j and k changes, that value will not change, and i will become an alias to that value, the expression j plus k.

So, these are some of the basic pitfalls. So, you can see here I have i declared as a variable initialized, j declared as a reference to i, k declared as a constant reference to 5 and l declared as

a constant reference to the expression j plus k. So, when I print the addresses and values I will see that if I print i it is 2, print j it is 2, their alias and they necessarily enjoy the same location, when I print k, I get the value 5 and I do have an address just 5 will not have an address.

But here the compiler has specifically given address to keep 5 which j is referring to actually. Similarly, j plus k has a value 7 and as a reference to that l has an address which is given here again compiler has given a specific address instead of a temporary for keeping the value of j plus k and please note that subsequently even if j plus k changes, the value of l will not change because it is treated as a constant. So, it is referring to a constant always. So, that is the basic, this are some of the basic rules to remember.

(Refer Slide Time: 10:03)

Call-by-Reference

Module M07  
Partha Pratim Das  
Objectives & Outlines  
References  
Pointers  
Call-by-Reference  
Step in C++  
Call Reference Parameter  
Return by Reference  
Pointers  
I/O Parameters of a Function  
Recommended Mechanisms  
References in Pointers  
Module Summary

Programming in Modern C++ Partha Pratim Das M07.9

C++ Program 07.02: Call-by-Reference

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

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

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

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;
}

----- Output -----
a = 20 , &a = 0023FA30
b = 20 , &b = 0023FA30 // Address of b is same as a as b is a reference of a
c = 20 , &c = 0023F95C // Address different from a as c is a copy of a
```

- 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
- Actual param a and formal param c get the *same value* in called function
- Actual param a and formal param b get the *same address* in called function
- However, actual param a and formal param c have *different addresses* in called function

Programming in Modern C++ Partha Pratim Das M07.10

**C++ Program 07.02: Call-by-Reference**

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

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

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

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;
}

----- Output -----
a = 20, &a = 0023FA30 ✓
b = 20, &b = 0023FA30 ✓ // Address of b is same as a as b is a reference of a
c = 20, &c = 0023F95C ✓ // Address different from a as c is a copy of a
```

- 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
- Actual param a and formal param c get the *same value* in called function
- Actual param a and formal param b get the *same address* in called function
- However, actual param a and formal param c have *different addresses* in called function

Programming in Modern C++ Partha Pratim Das M07.10

Now, let us move on to what is known as call by reference? So, you all are familiar with call by value that is what is commonly used in C, what happens is when I call a function passing a parameter, then the actual parameter value is copied on to the formal parameter value and then the function starts proceeding. So, actual and formal parameter enjoy different locations.

So, even if I change the formal parameter, the actual parameter will never get changed, that is the basic scenario. Here in C++, C does not support this, but in C++, I can have a parameter which is of reference type. I write it with this `int &`. It is a reference parameter when I write `int`, it is a value parameter.


So, when I define the function, so, this is my reference parameter and this is my value parameter. So, what it means is when I call the function say with `a` and `a`, the same variable. `b` is a reference parameter. So, `b` necessarily becomes a different name for `a`, `b` does not the value is not copied, the value is not copied.

But just you are saying that in function under parameters in this function, whatever you call `b` is the actual parameter `a` is just a different name for it. So, if you change `b`, `a` will get changed, whereas, this is a copy because this is a call by value parameter. So, C will take the call time value of `a` and that will be copied into `c` in a different location.


So, in the main when I print these addresses, subsequently, say I print the address of `a` and here the address of `b`, `c`, you will find that `a` and `b`. `a` in the main the actual parameter and `b` in the function, the formal parameter have the same address because it is a reference parameter. So, this talking about the same variable whereas `c` which is called by value, therefore, it is a copy it is a different variable in the function as a formal parameter will have a location which is different from `a` and `b`. So, here we have learned that how to pass parameters by reference where they are not copied, but just they become a second name to the actual parameter and accordingly these addresses will be assigned as of you have seen this already.

(Refer Slide Time: 13:24)





## C Program 07.03: Swap in C



Call-by-value - wrong	Call-by-address - right
<pre>#include &lt;stdio.h&gt;  void swap(int, int); // Call-by-value ✓ int main() { int a = 10, b = 15;   printf("a= %d &amp; b= %d to swap\n", a, b);   swap(a, b);   printf("a= %d &amp; b= %d on swap\n", a, b); }  void swap(int c, int d) { int t;   t = c; c = d; d = t; }</pre>	<pre>#include &lt;stdio.h&gt;  void swap(int *, int *); // Call-by-address int main() { int a=10, b=15;   printf("a= %d &amp; b= %d to swap\n", a, b);   swap(&amp;a, &amp;b); // Unnatural call   printf("a= %d &amp; b= %d on swap\n", a, b); }  void swap(int *x, int *y) { int t;   t = *x; *x = *y; *y = t; }</pre>
<ul style="list-style-type: none"> <li>• a= 10 &amp; b= 15 to swap ✓</li> <li>• a= 10 &amp; b= 15 on swap ✓ <span style="color: red;">No swap</span></li> </ul>	<ul style="list-style-type: none"> <li>• a= 10 &amp; b= 15 to swap ✓</li> <li>• a= 15 &amp; b= 10 on swap ✓ <span style="color: green;">Correct swap</span></li> </ul>
<ul style="list-style-type: none"> <li>• Passing values of a=10 &amp; b=15</li> <li>• In callee, c = 10 &amp; d = 15</li> <li>• Swapping the values of c &amp; d</li> <li>• No change for the values of a &amp; b in caller</li> <li>• Swapping the value of c &amp; d instead of a &amp; b</li> </ul>	<ul style="list-style-type: none"> <li>• Passing Address of a &amp; b</li> <li>• In callee x = Addr(a) &amp; y = Addr(b)</li> <li>• Values at the addresses is swapped</li> <li>• Desired changes for the values of a &amp; b in caller</li> <li>• It is correct, but C++ has a better way out +</li> </ul>
Programming in Modern C++	Partha Pratim Das M07.11

Now, let us come to one of the earliest programs you probably have written or earliest functions you probably have written that is swap that is I have 2 values I want to interchange them. Now, let us say what we need to do in writing swap. Now, if we write swap directly by using call by value these are called by value and then this is the swap port which is very straightforward using a local variable t, I can always write this code and then I call swap.

Now, the problem that happens is these are called by value. So, when I call it with a and b, a is copied to c and b is copied to d. Subsequently c and d are swapped, but when the function comes back a and b are different locations. So, they will not be swapped they will remain to be the same.

So, I pass this to swap, but on swap, no changes happen, no swapping has happened in this you have seen in C. So, how do you bypass this in C or what is the hack in C? The hacking C is that instead of passing the parameter directly you pass the address of that. So, you pass the address of that. So, now instead of calling swap(a, b), you are calling swap(&a, &b).

So, when I look into the formal parameter the address of a is copied to x and address of b is copied to y. These copies are by call by value, do not confuse these copies are called by value, but what you are copying is the address. And then what you are doing in the swap? You are actually dereferencing that address and making the swap.

So, the addresses have been copied, they are in different locations, x and y, but they are referring to the original actual parameters a and b. And so, when by dereferencing, you swap effectively a and b gets swapped. That is the basic idea. So, when you swap like this, this will actually take effect that is the correct way to write swap in c.

And when you do this, that is you use call by value, but pass addresses to make changes back into the actual parameter, you typically refer to it by called by address, it is not a very popular name, but this is colloquially, most often we refer to that. So, that is here, you can, you can get the summary of what I have just stated. And this is what happens in C. So, what is becoming is if you look carefully, you will see that the call to swap is becoming somewhat unnatural.



(Refer Slide Time: 16:25)

Program 07.04: Swap in C & C++

C Program: Call-by-value - <b>wrong</b>	C++ Program: Call-by-reference - <b>right</b>
<pre>#include &lt;stdio.h&gt;  void swap(int, int); // Call-by-value int main() { int a = 10, b = 15;   printf("a= %d &amp; b= %d to swap\n", a, b);   swap(a, b);   printf("a= %d &amp; b= %d on swap\n", a, b); }  void swap(int c, int d) { int t;   t = c; c = d; d = t; }</pre>	<pre>#include &lt;iostream&gt; using namespace std; void swap(int&amp;, int&amp;); // Call-by-reference int main() { int a = 10, b = 15;   cout&lt;&lt;"a= "&lt;&lt;a&lt;&lt;" &amp; b= "&lt;&lt;b&lt;&lt;"to swap"&lt;&lt;endl;   swap(a, b); // Natural call   cout&lt;&lt;"a= "&lt;&lt;a&lt;&lt;" &amp; b= "&lt;&lt;b&lt;&lt;"on swap"&lt;&lt;endl; }  void swap(int &amp;x, int &amp;y) { int t;   t = x; x = y; y = t; }</pre>
<ul style="list-style-type: none"><li>• a = 10 &amp; b = 15 to swap ✓</li><li>• a = 10 &amp; b = 15 on swap // <b>No swap</b></li></ul>	<ul style="list-style-type: none"><li>• a = 10 &amp; b = 15 to swap ✓</li><li>• a = 15 &amp; b = 10 on swap ✓ <b>Correct swap</b></li></ul>
<ul style="list-style-type: none"><li>• Passing values of a=10 &amp; b=15</li><li>• In callee; c = 10 &amp; d = 15</li><li>• Swapping the values of c &amp; d</li><li>• <b>No change</b> for the values of a &amp; b in caller</li><li>• Here c &amp; d do not share address with a &amp; b</li></ul>	<ul style="list-style-type: none"><li>• Passing values of a = 10 &amp; b = 15</li><li>• In callee; x = 10 &amp; y = 15</li><li>• Swapping the values of x &amp; y</li><li>• <b>Desired changes</b> for the values of a &amp; b in caller</li><li>• x &amp; y having <b>same address</b> as a &amp; b respectively</li></ul>

Programming in Modern C++ Partha Pratim Das M07.12

I am having to pass &a, &b they will not writing swap is unnatural. So, let us see. Now, let us compare this with C++ and using reference. So, on left again is the original C function, which has called by value, the swap and it cannot swap. Now, I write it in C++, instead of having call by value parameters, now I have reference parameters.

So, instead of int, I write int& int& and then I call swap. Now, what will happen? x and y in this function, x and y in this function are reference parameters. So, what it means that x necessarily becomes a different name for a, y necessarily becomes a different name for b. So, whatever you call a, b in the caller mean is called x and y in the swap.

So, when you swap x and y, actually, we are also swapping a and b. So, this is actually a, this is actually a, this is b, this is b. But in the function, they have different names, and that is possible because of the reference parameter. So, whatever changes you make, that correctly computes the swap, and performs it.

So, it makes it a lot more natural because your call is now not like &a, &b, you can just call them passing the parameters. As they are only thing you need to do is to make sure that the parameters are by reference. So, kind of you can see that call by value only allows you to input a value give it as a copy, whereas call by reference gives you the ability to input as well as output the value. Input because the initial value will always be the value that you called with will always be available in the function. So, it acts as an input like the call by value, but also changes that you making that variable will be reflected back in the caller. So, it is also an output variable.

(Refer Slide Time: 18:55)

Program 07.05: Reference Parameter as const

- A reference parameter may get changed in the called function
- Use `const` to stop reference parameter being changed

const reference - bad	const reference - good
<pre>#include &lt;iostream&gt; using namespace std;  int Ref_const(const int &amp;x) {   ++x; // Not allowed   return (x); }  int main() { int a = 10, b;   b = Ref_const(a);   cout &lt;&lt; "a = " &lt;&lt; a &lt;&lt; " and"   &lt;&lt; " b = " &lt;&lt; b; }</pre>	<pre>#include &lt;iostream&gt; using namespace std;  int Ref_const(const int &amp;x) {   return (x + 1); }  int main() { int a = 10, b;   b = Ref_const(a);   cout &lt;&lt; "a = " &lt;&lt; a &lt;&lt; " and"   &lt;&lt; " b = " &lt;&lt; b; }</pre>
<ul style="list-style-type: none"> <li>• <b>Error:</b> Increment of read only Reference 'x'</li> </ul>	<pre>a = 10 and b = 11</pre> <ul style="list-style-type: none"> <li>• <b>No violation</b></li> </ul>
<ul style="list-style-type: none"> <li>• <b>Compilation Error:</b> Value of x cannot be changed</li> <li>• Implies, a cannot be changed through x</li> </ul>	

Programming in Modern C++ Partha Pratim Das M07.13

Now, so when you use a call by reference, what will happen that a function may inadvertently change that variable. Suppose you want to use something only as an input, but you have used it as a reference parameter. And we will see the reasons of why we want to do that because a big advantage of using reference is that I do not need to make the copy.

If my parameter is big, if that is a huge object, then copying it is a huge overhead and I do not need that I just need that value to compute in the function. So, I would like to pass it as a reference, where no copy will be required. But the side effect of that or the possible danger of that is if the function happens to change that parameter, then the actual parameter will also get changed.

So, to stop that, you can again make use of constant. So, what you say that not only I am passing x as a reference, but I am passing it as a constant reference, which means that it is not allowed in the function to change x. The same concept of const reference we have just seen couple of slides ago.

So, x is a reference, but it is a constant reference, like whatever it is referring to that referent, that value that it is referring to that variable that is referring to cannot be changed. Therefore, if I have this, and I write `++x` and then return x, this will not compile, it will say that x is a constant, and you are not allowed to change x.

So, you cannot do `++x`, because plus plus x will necessarily change x, which effectively will change a, would have changed a if compiler would have allowed or if I did not give this `const`. Whereas I can write it like this, it is a constant reference, instead of doing `++x`, I compute x plus 1, which is necessarily what I was doing on the left column, that it will be allowed, because you are not changing x in the function ref const.

So, in the left, the code will not compile. In the right, it will work properly, and give called with 10, it will print 11, there is no violation. So, this is the constness of the reference parameter that you will have to keep in mind.

(Refer Slide Time: 21:36)

# Return-by-Reference

Module M07  
Partha Pratim Das

Objectives & Outlines  
References  
Call-by-Reference  
Step in C  
Step in C++  
const Reference Parameter

**Return-by-Reference**  
Points  
I/O Parameters of a Function  
Recommended Mechanisms  
References in Pointers  
Module Summary

## Return-by-Reference

Programming in Modern C++ Partha Pratim Das M07.14

# Program 07.06: Return-by-Reference

Module M07  
Partha Pratim Das

Objectives & Outlines  
References  
Call-by-Reference  
Step in C  
Step in C++  
const Reference Parameter

**Return-by-Reference**  
Points  
I/O Parameters of a Function  
Recommended Mechanisms  
References in Pointers  
Module Summary

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

Return-by-value	Return-by-reference
<pre>#include &lt;iostream&gt; using namespace std; int Function_Return_By_Val(int &amp;x) {     cout &lt;&lt; "x = " &lt;&lt; x &lt;&lt; " &amp;x " &lt;&lt; &amp;x &lt;&lt; endl;     return (x); } int main() { int a = 10;     cout &lt;&lt; "a = " &lt;&lt; a &lt;&lt; " &amp;a " &lt;&lt; &amp;a &lt;&lt; endl;     const int&amp; b = // const need why?     Function_Return_By_Val(a);     cout &lt;&lt; "b = " &lt;&lt; b &lt;&lt; " &amp;b " &lt;&lt; &amp;b &lt;&lt; endl; }</pre>	<pre>#include &lt;iostream&gt; using namespace std; int&amp; Function_Return_By_Ref(int &amp;x) {     cout &lt;&lt; "x = " &lt;&lt; x &lt;&lt; " &amp;x " &lt;&lt; &amp;x &lt;&lt; endl;     return (x); } int main() { int a = 10;     cout &lt;&lt; "a = " &lt;&lt; a &lt;&lt; " &amp;a " &lt;&lt; &amp;a &lt;&lt; endl;     const int&amp; b = // const optional     Function_Return_By_Ref(a);     cout &lt;&lt; "b = " &lt;&lt; b &lt;&lt; " &amp;b " &lt;&lt; &amp;b &lt;&lt; endl; }</pre>
<pre>a = 10 &amp;a = 00DCFD18 ✓ x = 10 &amp;x = 00DCFD18 ✓ b = 10 &amp;b = 00DCFD00 / Reference to temporary</pre>	<pre>a = 10 &amp;a = 00A7F8FC ✓ x = 10 &amp;x = 00A7F8FC ✓ b = 10 &amp;b = 00A7F8FC ✓ / Reference to a</pre>
<ul style="list-style-type: none"> <li>Returned variable is <b>temporary</b></li> <li>Has a <b>different address</b></li> </ul>	<ul style="list-style-type: none"> <li>Returned variable is an <b>alias of a</b></li> <li>Has the <b>same address</b></li> </ul>

Programming in Modern C++ Partha Pratim Das M07.15

Now the question is, if I can pass values by reference, can I return values by reference as well? Yes, I can. In C, as I do call by value, similarly, I have returned by value, the value that is returned is actually return through copy, whatever you write in the return statement, will be returned as a copy of that.

So, here I have, I have just made a function which is pathological. It is not doing any computation. But it just takes a variable and returns that same variable. So, return x, and this is the C, return by value mechanism. So, it will simply return a copy of x. So, when I do this, I will get a copy of x. So, a and x will necessarily have the same location, whereas b, which I am making a reference to the return value has a different location.

Now, I can return this value by reference. So, all that I need to do is again, put an & after the int, which means that the x that you have in the function will be returned by itself, not a copy of it.

So, when I set `int& b`, it basically becomes the reference to `x`. And you can see that I pass the parameter also as `x`. So, `x` is an alias of `a` and `b` is an alias of `x`.

So, necessarily `a`, `x`, `b`, all of them are the same location. Whereas here `b` when I reference it is a different location, because it is not `x`, but a copy of `x` a temporary that is created. Now, the question is here, you will note that I have written `const`. Why did I write a `const`? For the same reason, that if I am defining a reference and my initialization is an expression, then the value of the expression is a temporary, which will not be stored otherwise.

So, to give it permanency, I need to make it constant. So, that the compiler has retained that address that location and have kept the value of read function returned by value `a` into that location and `b` is now become a reference to that. So, in here, `x` is a reference of `a` but `b` is not a reference of `x`, but `b` is reference to the temporary carrying this entire expression. In here, when I returned by reference using this constant is not necessary. Because this by itself is a variable.

(Refer Slide Time: 25:11)

Return-by-reference	Return-by-reference - Risky!
<pre>#include &lt;iostream&gt; using namespace std; int&amp; Return_ref(int &amp;x) {     return (x); } int main() { int a = 10, b = Return_ref(a);     cout &lt;&lt; "a = " &lt;&lt; a &lt;&lt; " and b = "     &lt;&lt; b &lt;&lt; endl;     Return_ref(a) = 3; // Changes variable a     cout &lt;&lt; "a = " &lt;&lt; a; }</pre>	<pre>#include &lt;iostream&gt; using namespace std; int&amp; Return_ref(int &amp;x) {     int t = x;     t++;     return (t); } int main() { int a = 10, b = Return_ref(a);     cout &lt;&lt; "a = " &lt;&lt; a &lt;&lt; " and b = "     &lt;&lt; b &lt;&lt; endl;     Return_ref(a) = 3; // Changes local t     cout &lt;&lt; "a = " &lt;&lt; a; }</pre>
<pre>a = 10 and b = 10 a = 3</pre>	<pre>a = 10 and b = 11 a = 10</pre>
<ul style="list-style-type: none"><li>Note how a value is assigned to function call</li><li>This can change a local variable</li></ul>	<ul style="list-style-type: none"><li>We expect a to be 3, but it has not changed</li><li>It returns reference to local. This is risky</li></ul>

So, return by reference can also get tricky, you can see that, for example, you can write something like this, which is kind of some syntax, you can say, no this error, how can you write function on the left-hand side, I cannot write a function on the left-hand side, if it is returning my value. Because it is a value, it does not have an address. But, and on the left-hand side, I always need an address.

But if I am returning by reference, then what I have is a variable, it is an address. So, I can always write it on the left-hand side. So, if I write it like this, then in this case, what will happen? `a` is `a`, `x` is a reference of `x`. And that `x` is what is here. So, this actually is `x`, which means `a`, so the assignment actually is happening to `a`, so `a` becomes 3.

Whereas if I have used a temporary here, and say incremented, that temporary, that is not very important. And I return that temporary, then this is a reference of this local variable `t`, which actually has no existence after the function has returned. So, it is actually dangerous. So, it can be very, very risky. And this assignment will be made to that temporary `t` which have returned by

reference, therefore, a will not change and it will remain to be 10. So, this can get you into really, really big trouble. So, the thumb rule to remember is when you return by reference, never return a local variable.

(Refer Slide Time: 27:05)

**I/O Parameters of a Function**

Module M07  
Partha Pratim Das  
Objectives & Outlines  
Reference  
Pointers  
Call by Reference  
Scope of C  
Scope of C++  
const Reference Parameter  
Return by Reference  
Pointers  
I/O Params of a Function  
Recommended Mechanisms  
References vs. Pointers  
Module Summary

**I/O Parameters of a Function**

Programming in Modern C++ Partha Pratim Das M07.17

**I/O of a Function**

Module M07  
Partha Pratim Das  
Objectives & Outlines  
Reference  
Pointers  
Call by Reference  
Scope of C  
Scope of C++  
const Reference Parameter  
Return by Reference  
Pointers  
I/O Params of a Function  
Recommended Mechanisms  
References vs. Pointers  
Module Summary

- In C++ we can change values with a function as follows:

I/O of Function	Purpose	Mechanism
Value Parameter	Input	Call-by-value
Reference Parameter	In-Out	Call-by-reference
const Reference Parameter	Input	Call-by-reference
Return Value	Output	Return-by-value
		Return-by-reference
		const Return-by-reference

- In addition, we can use the Call-by-address (Call-by-value with pointer) and Return-by-address (Return-by-value with pointer) as in C
- But it is neither required nor advised

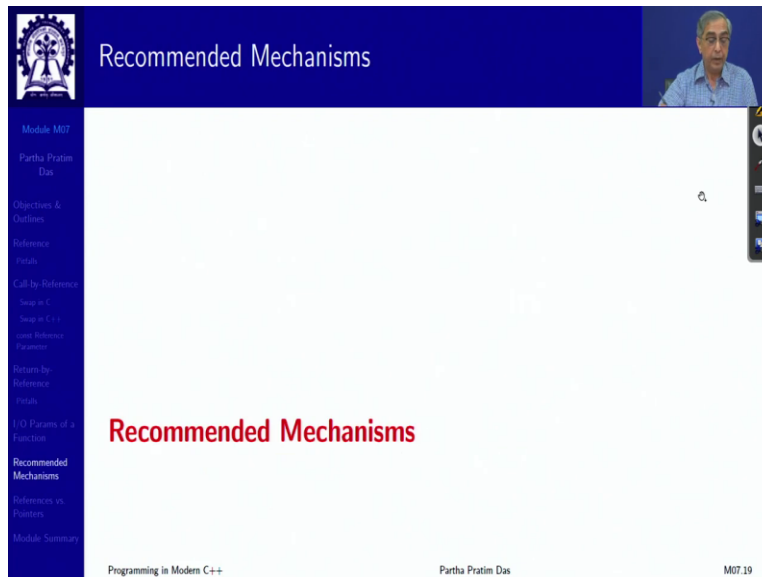
Programming in Modern C++ Partha Pratim Das M07.18

So, if I summarize the input output parameters of a function, then there are these are the different possibilities if it is a value parameter, then the purpose is to input and you can use call by value if it is a reference parameter, then it serves as input as well as output, it is called by reference. If it is a constant reference parameter, then it is input only, call by reference in constant and for return value, you can either have returned by value, you can have returned by reference or you can have also have constant return by reference.

And remember that C has the mechanism in C that we saw call by address which is called by value with pointer and similar corresponding return are also available in C++. But there is no

reason to use them, you will not find any situation where you need to use them. So, always try to avoid them.

(Refer Slide Time: 28:05)



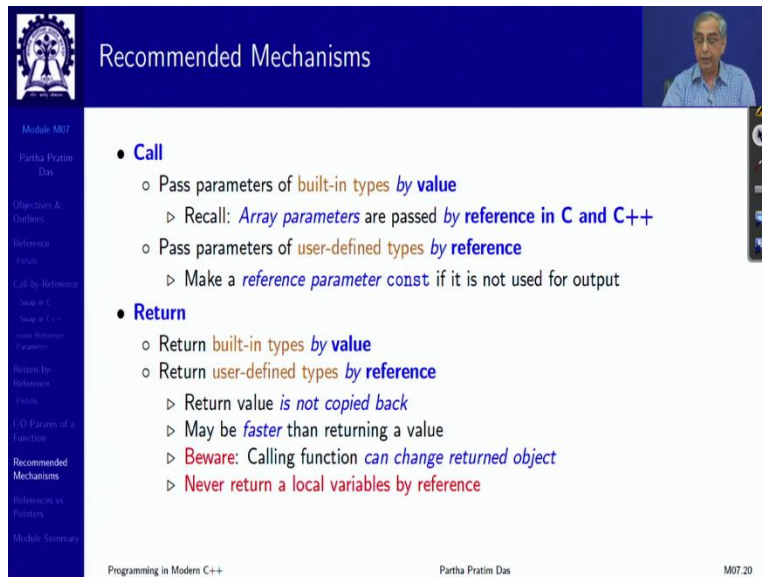
Recommended Mechanisms

Module M07  
Partha Pratim Das

Objectives & Outlines  
References  
Calls  
Call-by-Reference  
Step in C  
Step in C++  
User-Defined Parameters  
Return-by-Reference  
Returns  
I/O Parameters of a Function  
Recommended Mechanisms  
References vs. Pointers  
Module Summary

Recommended Mechanisms

Programming in Modern C++ Partha Pratim Das M07.19



Recommended Mechanisms

Module M07  
Partha Pratim Das

Objectives & Outlines  
References  
Calls  
Call-by-Reference  
Step in C  
Step in C++  
User-Defined Parameters  
Return-by-Reference  
Returns  
I/O Parameters of a Function  
Recommended Mechanisms  
References vs. Pointers  
Module Summary

- **Call**
  - Pass parameters of **built-in types by value**
    - ▷ Recall: *Array parameters* are passed **by reference in C and C++**
  - Pass parameters of **user-defined types by reference**
    - ▷ Make a *reference parameter const* if it is not used for output
- **Return**
  - Return **built-in types by value**
  - Return **user-defined types by reference**
    - ▷ Return value *is not copied back*
    - ▷ May be *faster* than returning a value
    - ▷ **Beware:** Calling function *can change returned object*
    - ▷ **Never return a local variables by reference**

Programming in Modern C++ Partha Pratim Das M07.20

So, instead of the recommended mechanisms, as to how reference can help, when you are making a call, if you are passing a built in type, always pass it by value. The reason for that will become clearer, as we move somewhat forward, it gives you a much better efficiency, because the copy actually is very lightweight.

Only remember that array parameters are always passed by reference even in C. In C that is the only situation where the entire array is never copied, it is just passed by reference, which means the base address only passed. Same thing happens in C++, but any parameter which is a user defined type pass them by reference, because you can avoid unnecessary copy of big objects by passing by reference which is not possible if you pass by value.



Similarly, for return if you are returning built in types do by value if you are returning user defined types do by reference and reference value is not copied back it is faster, but remember that the calling function the caller can now change the returned object. So, you have to be careful about that and never return as shown in a local variable by reference.

(Refer Slide Time: 29:27)

Difference between Reference and Pointer

Module M07  
Partha Pratim Das  
Objectives & Outlines  
References  
Pointers  
Call by Reference  
Swap in C  
Swap in C++  
Using Reference Parameter  
Returns by Reference  
Pointers  
I/O Parameters of a Function  
Recommended Mechanisms  
References vs. Pointers  
Module Summary

Difference between Reference and Pointer

Programming in Modern C++ Partha Pratim Das M07.21

Difference between Reference and Pointer

Pointers	References
<ul style="list-style-type: none"> <li>Refers to an <i>address (exposed)</i></li> <li>Pointers can point to NULL</li> </ul> <pre>int *p = NULL; // p is not pointing</pre> <ul style="list-style-type: none"> <li>Pointers can point to <i>different variables at different times</i></li> </ul> <pre>int a, b, *p; p = &amp;a; // p points to a ... p = &amp;b; // p points to b</pre> <ul style="list-style-type: none"> <li>NULL checking <i>is required</i></li> <li>Allows users to <i>operate on the address</i></li> <li>diff pointers, increment, etc.</li> <li>Array of pointers can be <i>defined</i></li> </ul>	<ul style="list-style-type: none"> <li>Refers to an <i>address (hidden)</i></li> <li>References cannot be NULL</li> </ul> <pre>int &amp;j; // wrong</pre> <ul style="list-style-type: none"> <li>For a reference, its <i>referent is fixed</i></li> </ul> <pre>int a, c, &amp;b = a; // Okay ... &amp;b = c // Error</pre> <ul style="list-style-type: none"> <li><i>Does not require NULL</i> checking</li> <li>Makes code <i>faster</i></li> <li><i>Does not allow</i> users to <i>operate on the address</i></li> <li>All operations are interpreted for the referent</li> <li>Array of references <i>not allowed</i></li> </ul>

Programming in Modern C++ Partha Pratim Das M07.22

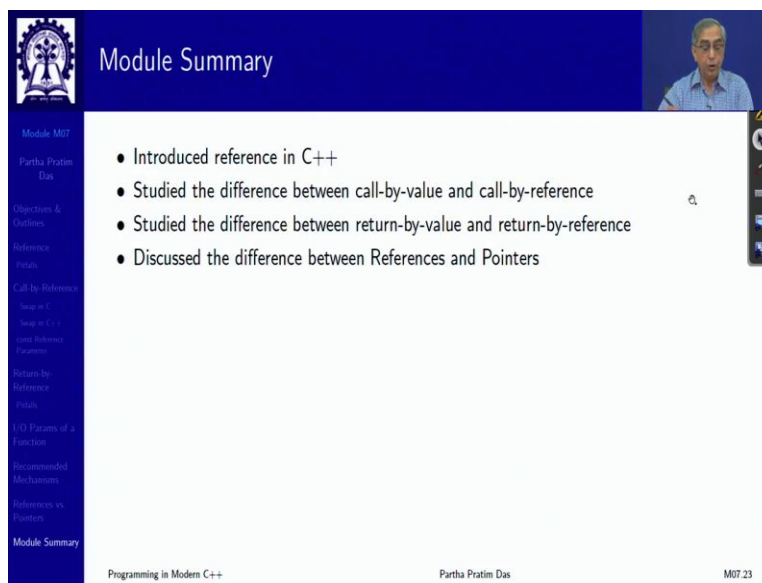
So, to before we summarize, let us just compare the pointer that you knew and the reference both of them reference to an address in some way or other. In pointer that address is exposed you can print that, you can do computation with it. In reference, you cannot do that. So, those of you who know Java, please do not relate. Reference here with the reference in Java. Reference in Java is neither exactly a pointer nor exactly C++ reference. So, please do not confuse that. Pointers can be null reference cannot be null, you will always have to initialize it as we have seen.



References can be pointers can be changed from time to time I can define a pointer and let it point to different objects, but reference cannot be changed it is by birth. So, once you have made a reference to a certain variable you will have to the reference will always be made to that variable only.

For pointer, you need to do a null checking because it may not have been initialized in reference that question does not arise. Pointer allows operations on the address like you can take diff of pointers, you can increment and so on. But in reference, you cannot do anything of that sort whatever you do with the reference. Actually, it means the variable, the referent that the reference is talking about. Finally, arrays of pointers are possible, and arrays of references are not allowed.

(Refer Slide Time: 30:53)



The screenshot shows a presentation slide titled "Module Summary" with a blue header and a white content area. The slide is part of a video lecture, as indicated by the small video feed of the presenter in the top right corner. The slide content includes a list of topics covered in the module:

- Introduced reference in C++
- Studied the difference between call-by-value and call-by-reference
- Studied the difference between return-by-value and return-by-reference
- Discussed the difference between References and Pointers

The slide also features a navigation menu on the left side with the following items: Module M07, Partha Pratim Das, Objectives & Outlines, Reference, Call by Reference, Return by Reference, I/O Params of a Function, Recommended Mechanisms, Reference vs. Pointers, and Module Summary. The footer of the slide contains the text "Programming in Modern C++", "Partha Pratim Das", and "M07.23".

So, this brings us to the end of this module, we have introduced a very important concept of references in C++ and studied the difference between call by value and call by reference or return by value and return by reference. And this will be this idea will be used very heavily all through the course so. Please try out these examples and try to understand it well. Thank you very much for your attention and we will meet in the next module.