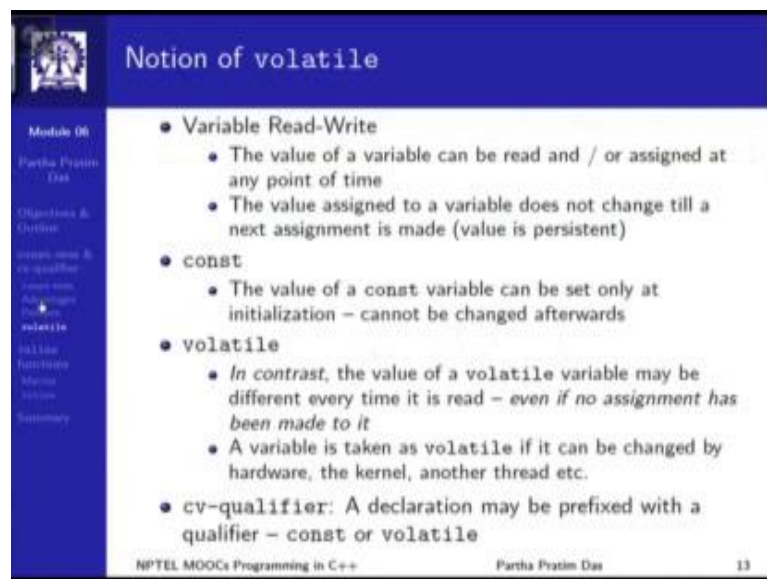


Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 09
Constants and Inline Functions (Contd.)

Welcome to module 6 in a programming in C++ again. We have already discussed the notion const-ness in this module, we have discussed how in place of using manifest constant we can use the const declaration to qualify a variable declaration and how it does not allow us to change the value of a variable once it has been declare and initialized. We have also shown how const-ness work with pointers and talked about the constant pointer ans const-ness of the pointed data. We will continue on that.

(Refer Slide Time: 01:02)



The slide is titled "Notion of volatile" and contains the following content:

- Variable Read-Write
 - The value of a variable can be read and / or assigned at any point of time
 - The value assigned to a variable does not change till a next assignment is made (value is persistent)
- const
 - The value of a const variable can be set only at initialization – cannot be changed afterwards
- volatile
 - *In contrast*, the value of a volatile variable may be different every time it is read – *even if no assignment has been made to it*
 - A variable is taken as volatile if it can be changed by hardware, the kernel, another thread etc.
- cv-qualifier: A declaration may be prefixed with a qualifier – const or volatile

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

Next, we will talk about a related notion which is known as the Volatile. This volatile is a less commonly known concept so let us try to understand it little bit. Let us think about a variable, so what can we do with a variable, after variable has been declared defined possibly has been initialized then we can read the value of the variable or we can assign a new value at any point of time. The basic property that we always program with is if I have read the value of the variable then if I read the value again I am expected to get the

earlier value itself unless some new value has been assigned in between. So if I assign a value and keep on reading a number of times I will always get the same value till I make the next assignment, these are the basic concept of read write of a variable.

Now, look at const-ness in this context. Const-ness what we are saying that we are allowed to assign or in that sense initialized the value only once and not allowed to change it afterwards. So, for the whole lifetime of the program the value that I read of this variable will be the same that is the const-ness. Volatile in contrast is saying that it is a volatile value which means that if I read a variable at different points of time there is no guarantee that I will get the same value I may get different values.

So, even when I have not done any assignment to it so as a volatile refers volatile is something that evaporates. Suppose, the value of the variable was 5 you read it once then you need not make an assignment, but you read it after may be ten statements or after it certain time the value may be found to be 7. Earlier value 5 has disappeared has evaporated. So that is the notion of the volatile variable.

Why is this important to have this kind of a variable behavior in program because there are some system situations where this can happen, for example, suppose I am writing a network programming code whose purpose is to keep listening to a port to find, if some data has arrived. So what are you doing you are expected to write anything to that port because you are expecting data from the outside the system, so you are only reading, reading, reading. What will happen? When some data arrives then your value will change, but earlier when you have read the value was possibly null then suddenly you read it after may be 100 millisecond you find that some value has come, you read it after another 100 millisecond then there may be at different value that has come because another different packet has come.

It is possible that a variable can be chased by the hardware by the kernel of the operating system by another thread and so on, so it is required to module that behavior and in C++ this is module in the conjunction with the const concept because one which kind gives you absolute truth and the other which gives you kind of no guarantee that you just do not know what are the value of the variable is. So they are club together and we called

then as CV qualifier, C for const V for volatile, CV qualifier and any declaration of a variable can be prefix with a CV qualifier const or volatile.

(Refer Slide Time: 05:01)

The slide is titled "Using volatile" and features a blue header. On the left, there is a vertical navigation menu with items like "Module 06", "Partha Pratim Das", "Objectives & Overview", "C++ Basics & C++ Qualifier", "C++ Operators", "C++ Control Flow", "C++ Arrays", "C++ Strings", "C++ Pointers", "C++ References", "C++ Namespaces", "C++ Templates", "C++ STL", and "C++ Summary". A small circular video feed of a man is positioned at the bottom left of the slide content. The main text area contains the following:

Consider:

```
static int i;
void fun(void) {
    i = 0;
    while (i != 100);
}
```

This is an infinite loop! Hence the compiler should optimize as:

```
static int i;
void fun(void) {
    i = 0;
    while (1); // Compiler optimizes
}
```

Now qualify i as volatile:

```
static volatile int i;
void fun(void) {
    i = 0;
    while (i != 100); // Compiler does not optimize
}
```

Being volatile, i can be changed by hardware anytime. It waits till the value becomes 100 (possibly some hardware writes to a port).

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

We will show example of that. Here, we are trying to show how volatile could be useful. This a code very simple code which assigns the value 0 to i and then tries to look on the condition that i is not equal to 100. Now, if you just had shown this code you will immediately identify that this is an infinite loop. Why it is an infinite loop? Because if the value of i is a 0 and then I am checking if i is 100 and i will continue as long as i is not hundred suddenly the value of i will never become 100. So this condition will always remain true so this loop will continue in definitely, so the what the compiler will do compiler will optimize and say that this is true, this is while one simply that whole expression will go away.

Now let us say you qualify this by as a volatile. You say this is a volatile variable i and you read the same code now it does neither gets optimize, and what you expect is this code will actually work because you being a volatile variable you expect that there is some other agent, there some hardware, some port, some kernel system, some thread possibly is changing its through some other means. So, you keep on waiting till its value becomes 100 and it is possible that at some point it become 100 and then that a particular

condition becomes false and the function will be able to return. This is the use of the volatile variable in C, C++ as well.

(Refer Slide Time: 06:43)

Program 06.03: Macros with Parameters

- Macros with Parameters are defined by #define
- Macros with Parameters are replaced by CPP

Source Program	Program after CPP
<pre>#include <iostream> using namespace std; #define SQUARE(x) x * x int main() { int a = 3, b; b = SQUARE(a); cout << "Square = " << b << endl; return 0; }</pre>	<pre>// Contents of <iostream> header replaced by CPP using namespace std; // #define of SQUARE(x) consumed by CPP int main() { int a = 3, b; b = a * a; // Replaced by CPP cout << "Square = " << b << endl; return 0; }</pre>
Square = 9	Square = 9

• SQUARE(x) is a macro with one param
• SQUARE(x) looks like a function

• CPP replaces the SQUARE(x) substituting x with a
• Compiler does not see it as function

NPTEL MOOCs Programming in C++ Partha Pratim Das 15

Let us now move on and a talk about a different kind of use for hash define which is again processed by the c p processor we call them macros. The difference being that we still define them with a hash define word we have a name, but the main thing is we now have a parameter into this. So what happens is when I use, I use it with a parameters so we are saying that a square is the defined name and I am putting a parameter to it the effect of that is, the C preprocessor will directly go, match a with x and replace all excess in the define expression by a. It is simple x base substitution. So, wherever it will find x in that expression it will substitute it with the parameter a.

Now, macros are very commonly used featured in C and are also useful in C++, but it kind of allows us to write a function like notation. So if you are not told to hash define line and if you just reading the main you will not know where the square is a macro or it is a function it could pretty well be a function. But when it is come to the compiler, again like in the case of manifest constant the compiler was not been able to see that it is a variable it was just seeing the constant value that you have written or the constant expression that you have written, here again the compiler will not able to see any kind of

a function notation he will simply be able to see the expression that the CPP has replaced.

(Refer Slide Time: 08:52)

Pitfalls of macros

Consider the example:

```
#include <iostream>
using namespace std;
#define SQUARE(x) x * x

int main() {
    int a = 3, b;

    b = SQUARE(a + 1); // Wrong macro expansion

    cout << "Square = " << b << endl;

    return 0;
}
```

Output is 7 in stead of 16 as expected. On the expansion line it gets:

```
b = a + 1 * a + 1;
```

To fix:

```
#define SQUARE(x) (x) * (x)
```

Now:

```
b = (a + 1) * (a + 1);
```

NPTEL MOOCs Programming in C++ Partia Pratin Das 16

Macros have been very often widely used in a C programming and it has advantages particularly in C you could not have done without macros for several reasons. It gives you efficiency also because macros do not need a function called over it, but they have a very serious pitfalls also we show some of the pitfalls here in the same square a example and we are trying to, this is a macro and here I am just trying to use it with a plus 1. Earlier, I was using it with a I am trying to use it a plus one, certainly if somebody reads it the mental notion would be a 1 will be added to a so it will become 4 and then it will be squared, so it will become 16.

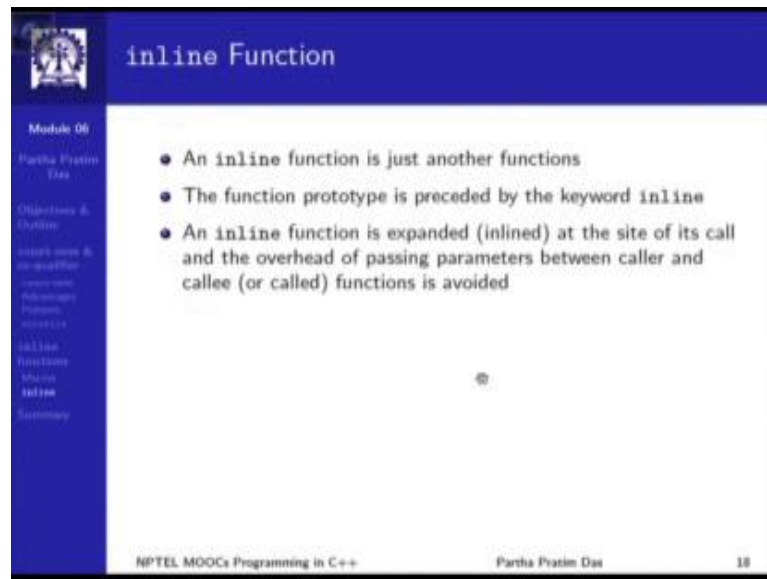
But, when you run this program we will actually get an output 7 and to understand that you will have to expand the macro, now expand this macro to this line. You can see that in the macro x is replaced by a plus 1, so in place of x if you just write a plus 1 it becomes a plus 1 star a plus one and then it becomes an expression were the precedence and associativity apply. So, 1 into 1 times a which is in the middle gets the precedence so that is gets a operated first, so this practically becomes 2 a plus 1. So it becomes 7. So, certainly this is was all the problem.

Fortunately, this problem which can be fixed, if you just put parenthesis around every x or around every instance of the parameter in the macro definition, if you do that, how it will help? We can just see the expansion, this will help because now I have a sorry, since a the parenthesis have been put after the macro is expanded in the last line of this slide you can see that there is parenthesis around a plus 1. So, it is says a plus 1 into a plus 1 now the BODMAS rule say that a plus 1 has to happen first which is what we had expected, so this can still be fixed.

Let us go ahead and this is really disturbing that if you have to remember that every time you write a (Refer Time: 11:08) parenthesis around that otherwise you might get surprise. Next, let us see an example were the situation is actually worse because, here now we have fixed the definition of the macro now we are trying to use it again and I want to use it with plus plus a, a plus plus is a pre increment. If a is 3, as in here if do plus plus I expected to become 4, 4 should go to the square it should get squared it should become 4 times 4 it should result should be 16. You try that the result is 25. Why?

Again look into the expansion of this macro, if it is expanded it looks like this, because there are two x so plus plus a, is written for each one of them. So what happens, plus plus has the highest precedence over is a higher precedence than multiplication, so both plus plus has happen before multiplication has happened so, a was 3 it first becomes 4 then it becomes 5 and then the multiplication happens. So the result is expectedly 25. The unfortunate part of the story is there is no easy fix for this in C. So, you will have to live with these kinds of possible pitfalls in the macros.

(Refer Slide Time: 12:30)



The slide is titled "inline Function" and is part of a presentation. It features a blue header with the title and a small logo on the left. A vertical navigation menu on the left side lists various topics, with "inline" highlighted. The main content area contains three bullet points:

- An **inline** function is just another functions
- The function prototype is preceded by the keyword **inline**
- An **inline** function is expanded (inlined) at the site of its call and the overhead of passing parameters between caller and callee (or called) functions is avoided

At the bottom of the slide, there is a footer with the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "18".

So, there is a new feature in C++ which is called as Inline Function. Let me first define how to do this and then we will explain, how does it relate to the macros? An Inline function is just another function is no special kind of function only difference is in the prototype of the function in the header you write the keyword `inline` before the return type. If you write this keyword `inline` then what happens is when the function is called the actual function call does not happen, but whatever code the function has that code the compiler puts at the call site that is the basic. So, the over rate of function is call a avoided,

(Refer Slide Time: 13:11)

The slide is titled "Program 06.04: Macros as inline Functions". It lists three bullet points: "Define the function", "Prefix function header with inline", and "Compile function body and function call together". It is divided into two columns: "Using macro" and "Using inline".

Using macro	Using inline
<pre>#include <iostream> using namespace std; #define SQUARE(x) x * x int main() { int a = 3, b; b = SQUARE(a); cout << "Square = " << b << endl; return 0; }</pre>	<pre>#include <iostream> using namespace std; inline int SQUARE(int x) { return x * x; } int main() { int a = 3, b; b = SQUARE(a); cout << "Square = " << b << endl; return 0; }</pre>
Output: Square = 9	Output: Square = 9
<ul style="list-style-type: none">• SQUARE(x) is a macro with one param• Macro SQUARE(x) is efficient• SQUARE(a + 1) fails• SQUARE(++a) fails• SQUARE(++a) does not check type	<ul style="list-style-type: none">• SQUARE(x) is a function with one param• inline SQUARE(x) is equally efficient• SQUARE(a + 1) works• SQUARE(++a) works• SQUARE(++a) checks type

NPTEL MOOCs Programming in C++ Partha Pratim Das 19

So, we define the function, we prefix the function header with inline it just focus here on the right hand side which is a C++ code left hand side is the original C code of audio reference on the right hand side we have not doing a hash define macro we are saying square is the function which takes x returns x times x as an integer and we are prefixing it with the inline keyword in front. The use code for this here and here remains same.

Now, the advantage is in C++ this is truly a function. So you cannot have any of the pitfalls that we were showing earlier very truly first evaluate the parameter and then takes that evaluated value and calls the function. So if you pass here a plus 1 it will first make plus 1 as 4 and then pass that, if you do plus plus a it will increment a from 3 to 4 and pass 4. So, you are not going to get any of the pitfalls. But you get the advantage of the macro that is with the hash define macros you were able to avoid the over rate of function call all the parameter coping, then transfer of control, then the computation, and then again transfer of control back and the copy of the return value all this can be avoid because of compiler would try to actually put the x times x that I am doing here right at the site were the function has been called. So, that is the basic feature of the inlining.

(Refer Slide Time: 15:06)

The slide is titled "Macros & inline Functions: Compare and Contrast". It features a blue header with the title and a small logo on the left. Below the header, there is a table comparing Macros and Inline Functions. The table has two columns: "Macros" and "Inline Functions". Each column contains a list of bullet points. On the left side of the slide, there is a vertical navigation menu with the following items: "Module 09", "Partha Prasin Das", "Objectives & Prerequisites", "C++ Basics & Compilation", "C++ Data Types", "C++ Operators", "C++ Control Flow", "C++ Functions", "C++ Macros", "C++ Inlines", and "Summary". At the bottom left, there is a circular portrait of Partha Prasin Das. At the bottom center, it says "NPTEL MOOCs Programming in C++". At the bottom right, it says "Partha Prasin Das" and "30".

Macros	Inline Functions
<ul style="list-style-type: none">• Expanded at the place of calls• Efficient in execution• Code bloats• Has syntactic and semantic pitfalls• Type checking for parameters is not done• Helps to write <code>max / swap</code> for all types• Errors are not checked during compilation• Not available to debugger	<ul style="list-style-type: none">• Expanded at the place of calls• Efficient in execution• Code bloats• No pitfall• Type checking for parameters is robust• Needs template for the same purpose• Errors are checked during compilation• Available to debugger in DEBUG build

Inlining helps to get the benefit of macro to a good extent, while it protects the basic properties and the types of functions. If we can just compare them site by site then macros are expanded at the place of called inline are also in some way expanded the place of call. But it is not possible to show that to you because macro are expanded in text form so I could illustrate that to you, but inline functions are expanded in terms of the assembly code or the binary code at times so it is not possible to understand it so easily, but we can we can just take it that it does the job right of the site were you have called. Both of them are efficient in execution both of them bloats the codes. The code becomes fatter, because what is happening you have one function defined and if you are doing a macro that function that macro may have been invoked at 10 places so the whole macro code will be copied ten times.

Similarly, inline function if I have function in line and that has been called at ten places the function body will occur at ten places so code will become bigger. Which may not be concern for some of the soft common software system that we do, but may be a concern in other cases like when he write program for a mobile phones and handled devices were the memory is really small if the program becomes larger then it becomes difficult to fit that app, because you are concern with the size of the app also. But we will not get much deeper into it.

Next point is the macros have syntactic and semantic pitfalls we have shown two of them, the a plus 1 was a syntactic pitfall so we could manage it by putting parenthesis around them, plus plus a was a syntactic pitfall because the way the plus plus is executed, so we could not manage that we have to leave it, but inline function will not have any such pitfalls.

Certainly macros do not check types for parameters, were as inline functions do that. So, their parameters are robust, so it is possible that a macro which intendedly written for a say an integer type of value may inadvertently be called or invoked with some double variable, the compiler would not able to understand, whereas in inline function that is not possible because it is just the function that we have. However, my macros do have advantages for example, the fact that it does not check for type help us to write some code were I do not know the type. For example, I want to write a code to swop two variables. Now, the code to swop two variables could swop int variables, could swop double variables, could swop cad variables and so on.

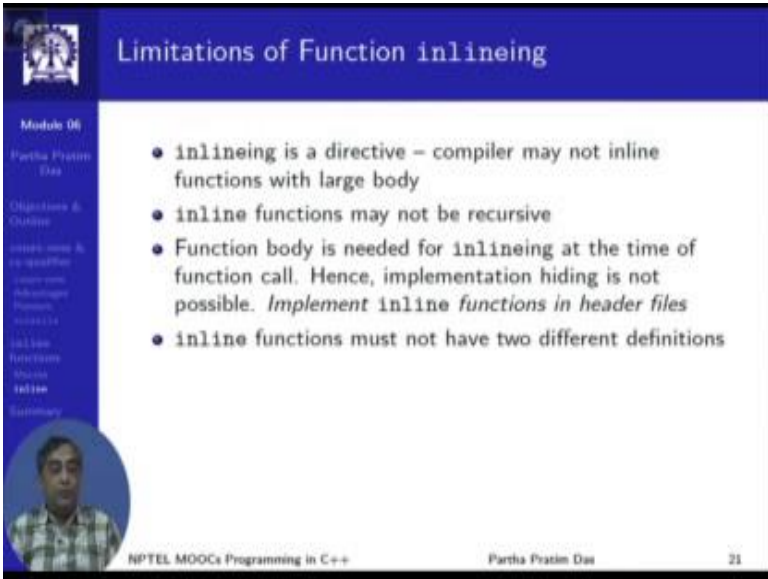
But if you have to write in c plus plus the swop function, one swop function cannot do all this because if I have to swop two int variable then my parameters would be of int type, if I want to swop two double variables my parameters will be of two types, if I want to swop two pointers my parameters would be of pointer type. So I cannot write that function easily in C++ and inline that. Whereas, I could write one macro and swop it because macro is not checking that type it will just check that there are two variables and third variables three assignments we know how to do swop it will just put that variable here. So, macros do provide some advantages and going forwards towards the end of the course we will show that you can also do that with inline functions, but with the support of very later features of C++ known as templates so when you discussed that we will show how do this.

Finally, in for macros there is no error checking done for compilation in inline function the errors are checked and certainly like the manifest constant macros are also not visible to the debugger. You will not able to see the square macro if you try to debug because actually the compiler has never seen that square because it was replaced by the pre processor. Whereas, for inline functions you will be able to see all of that in the debug

build. If you do a debug build you will be able to see that.

The difference that I should highlight here, that with inline function 1 feature that the compiler do is if you do a debug build the inlining is not done, if you do a release build then the inlining is done. If you do a debug building an inline function is just like any other function, so you can debug into that function. Because in the debug build you are saying I want to debug I want to look inside what is happening whereas, in a release or production build you really want the efficiency to be there you want that the code should run the fastest. So, you are not interested to debug any more you have already debug you know it is correct, that is when the inlining actually happens it is little bit subtle point, but please keep that this in mind over time you will slowly understand these factors. So, we would suggest that you always use inlining.

(Refer Slide Time: 20:29)



The slide is titled "Limitations of Function inlining" and is part of a presentation on C++ programming. It features a blue header and a white main content area. On the left side, there is a vertical navigation menu with a small portrait of the speaker at the bottom. The main content area contains a list of four bullet points detailing the limitations of the `inline` keyword.

- `inline` is a directive – compiler may not inline functions with large body
- `inline` functions may not be recursive
- Function body is needed for `inline`ing at the time of function call. Hence, implementation hiding is not possible. *Implement inline functions in header files*
- `inline` functions must not have two different definitions

NPTEL MOOCs Programming in C++ Partia Pratim Das 21

There are however some limitations that you should be aware off. Inlining is called a Directive. A directive in a language is a suggestion to the compiler, you are telling the compiler look I think it is useful to inline this function, but it is not mandatory, it is not a binding on the compiler, it must inline. So you must say `inline`, but the compiler may not inline the function. If the compiler finds that inlining it has problems or inlining it does not really help the efficiency. For example as very simple example is, if a function body

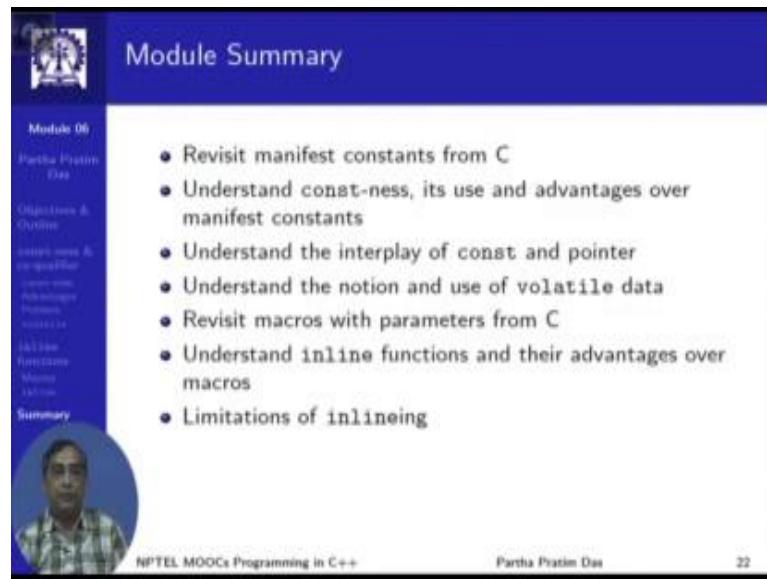
is very large then what again you are doing by inlining is that the function call is not required function written is not required, rest of it the computation has to be done any way.

If the function body is very large, then the additional overhead of call and return are very small, so you do not want to get it all this trouble of inlining, but if the function body is very small inlining release. The compiler decides whether it wants to do inlining or not. In many cases, in C++ the reverse is also true that you may not have said that a function to be inlined, but the compiler might inline it finding that it is efficient to inline.

Second point to be noted is inline functions cannot be recursive. What we are saying we are saying that at the place of inline, at the phase of function call we will put the body of the function if a function is recursive the body itself have another call. So, at that call you will again have to put the body of the function that will have another call. Now how many times you will put it that depends on how deep the recursion goes which you do not know till you have the values till you know whether you are trying to do factorial 3 or you are trying to do factorial 77. You do not know how many times the inlining has to happen, so recursion function cannot be inline they will necessarily have to be a normal function.

Since, inlining is replacing the body. In the third point I highlight that fact that if you want to inline the function, then the function body will also have to be in header file. In the earlier in module 1 we have talked about source organization, where he said that all functions header must be prototype must be in headers files, dot h files function bodies implementation should be in separate dot CPP files, but for inline functions this is not possible because when an application sees the function prototype unless it can see the body how does it inline, how does it replace the body. So for inline function the body should also have to be in header. Certainly the inline functions must not have two different definitions because it does have two different definitions then the two invocations will have two different behavior.

(Refer Slide Time: 23:48)



The slide is titled "Module Summary" and is part of the NPTEL MOOCs Programming in C++ course. It features a blue header with the title and a small logo on the left. A vertical sidebar on the left contains a navigation menu with items: "Module 06", "Partha Pratim Das", "Objectives & Prerequisites", "Learning Goals & Outcomes", "Learning Resources", "Lecture Functions", "Macro", "Inline", and "Summary". A circular portrait of the instructor, Partha Pratim Das, is positioned at the bottom left of the sidebar. The main content area is white and contains a bulleted list of seven items. At the bottom of the slide, the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" are visible on the left, and the number "22" is on the right.

Module Summary

- Revisit manifest constants from C
- Understand `const`-ness, its use and advantages over manifest constants
- Understand the interplay of `const` and pointer
- Understand the notion and use of `volatile` data
- Revisit macros with parameters from C
- Understand `inline` functions and their advantages over macros
- Limitations of `inline`ing

NPTEL MOOCs Programming in C++ Partha Pratim Das 22

So, these are some of the limitations or restrictions of function inlining that needs to be kept in mind. In summary, we have revisited the manifest constant for C; I am talking about the whole of module 6. So, we have revisited the manifest constants and we have understood the notion of `const`-ness as is available in C++ and we have seen what the advantages of `const` over manifest constant are. We have also seen how `const` and pointer interplay. We have introduced the notion for volatility of data and seen how volatile data can be used in a C++ program.

Next, we have revisited macros with parameters from C and shown how inline functions or function inlining can be used with advantage in place of macros which solves a number of syntactic and semantic problems that the macros offer. Finally, we have also looked at the restrictions on inline.