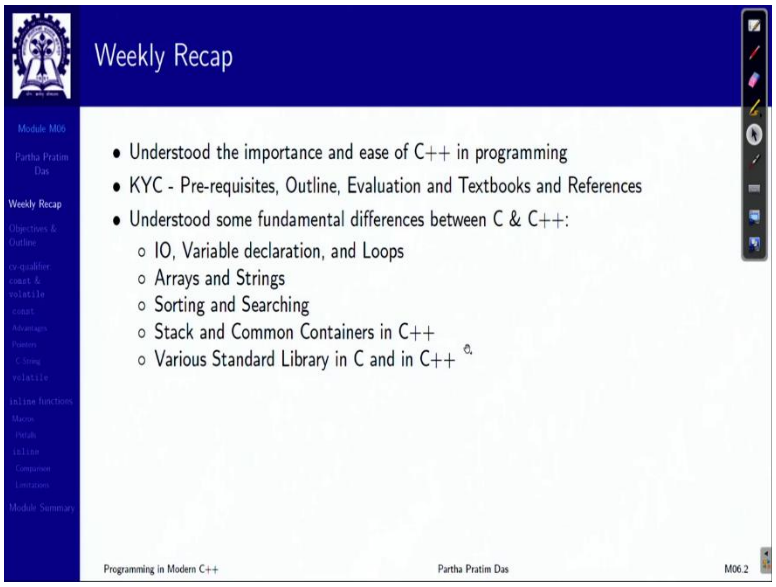
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture No. 10
Constants and Inline Functions

Welcome to Programming in Modern C++, we are in week 2 and we are going to discuss module 6.

(Refer Slide Time: 0:36)



Weekly Recap

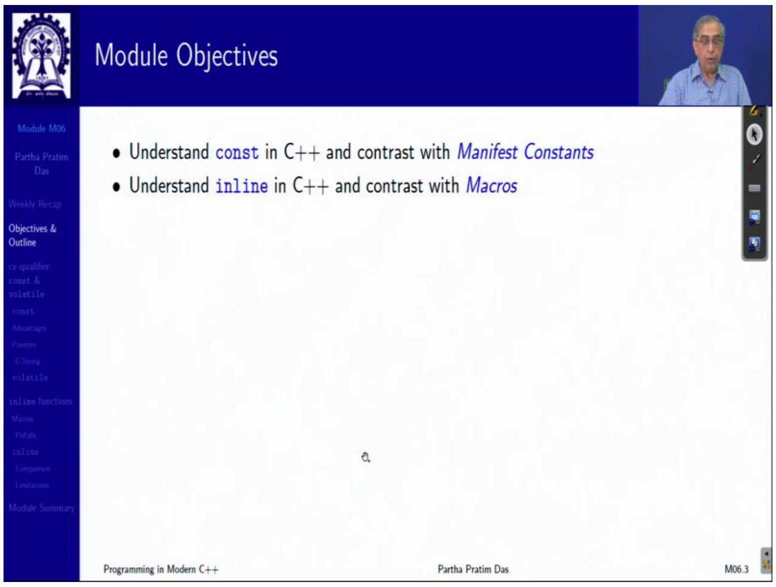
- Understood the importance and ease of C++ in programming
- KYC - Pre-requisites, Outline, Evaluation and Textbooks and References
- Understood some fundamental differences between C & C++:
  - IO, Variable declaration, and Loops
  - Arrays and Strings
  - Sorting and Searching
  - Stack and Common Containers in C++
  - Various Standard Library in C and in C++

Programming in Modern C++ Partha Pratim Das M06.2

In the last week, we have given an overview of C++ programming particularly contrasting it with the programming styles in C or the availability of standard library in C and C++ and we have shown how starting from very common simple programs involving IOs, variables, loops, to arrays, strings, sorting, searching, everything can become much easier compact and much less error prone to write in C++, where often you need to write much less code.

And you get a much better quality of code. And from this, what I would strongly suggest is all the examples that were discussed in the last week in the 4 modules first one being just introduction. In the 4 modules, I would honestly request you to try out those executing those builds them on your system using whatever GCC you are using. And really check out what you are getting and make small changes and see how things are changing. So, that slowly you start getting accustomed to the programming styles in C++.

(Refer Slide Time: 2:09)



Module Objectives

- Understand const in C++ and contrast with *Manifest Constants*
- Understand inline in C++ and contrast with *Macros*

Module M06

Partha Pratim Das

Weekly Recap

Objectives & Outline

cv-qualifier const & volatile

const

Advantages

Pointers

C-String

volatile

inline functions

Macros

Pitfalls

inline

Comparison

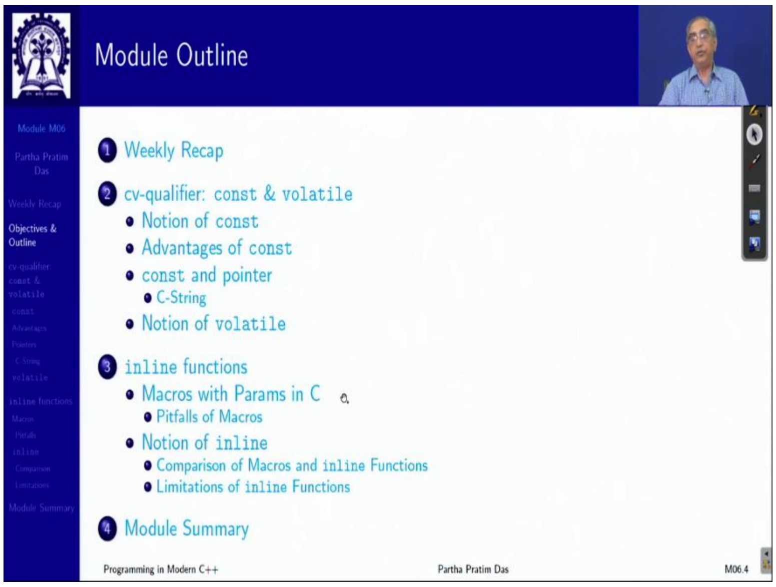
Limitations

Module Summary

Programming in Modern C++ Partha Pratim Das M06.3

Now, from the module today, we will we are actually get starting to get into C++ per se its features and primarily we will start focusing on certain features of C++ which actually make I should say it is a better C kind of a language that is these are these concepts were there in C but C++ have found better ways of expressing them or handling them, so that it gets easier to use as well as better in the quality that you get. So, today in this module, we are discussing about constants and inline functions, which are in contrast to 2 forms of macros that you had in C.

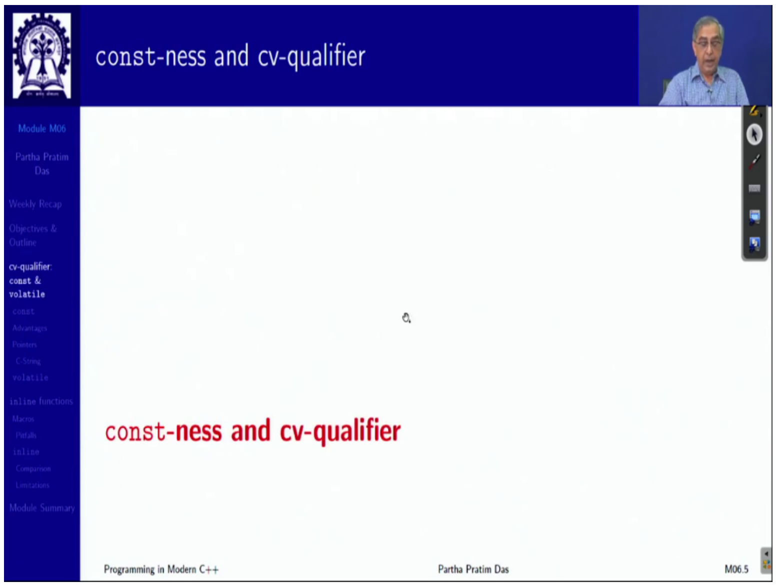
(Refer Slide Time: 2:56)



Module Outline

1. Weekly Recap
2. cv-qualifier: const & volatile
   - Notion of const
   - Advantages of const
   - const and pointer
     - C-String
   - Notion of volatile
3. inline functions
   - Macros with Params in C
     - Pitfalls of Macros
   - Notion of inline
     - Comparison of Macros and inline Functions
     - Limitations of inline Functions
4. Module Summary

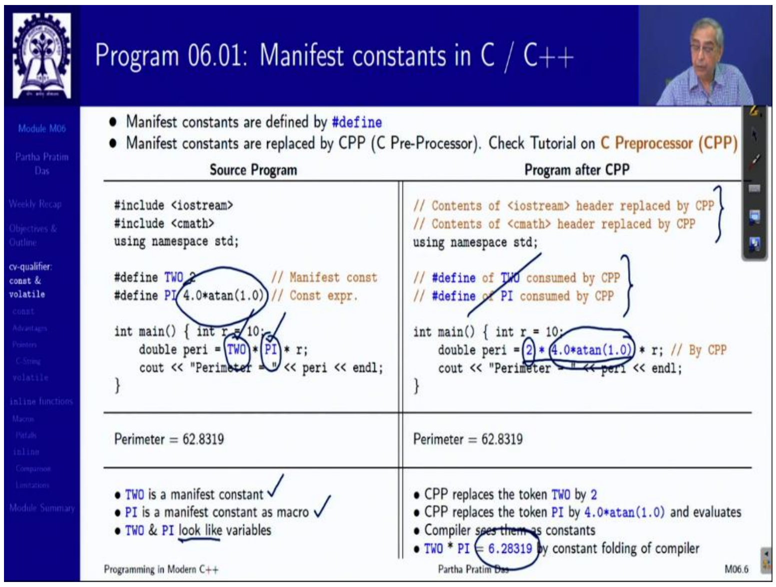
Programming in Modern C++ Partha Pratim Das M06.4



const-ness and cv-qualifier

const-ness and cv-qualifier

Programming in Modern C++ Partha Pratim Das M06.5



Program 06.01: Manifest constants in C / C++

- Manifest constants are defined by #define
- Manifest constants are replaced by CPP (C Pre-Processor). Check Tutorial on C Preprocessor (CPP)

Source Program

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

#define TWO 2                // Manifest const
#define PI 4.0*atan(1.0)     // Const expr.

int main() { int r = 10;
    double peri = TWO * PI * r;
    cout << "Perimeter = " << peri << endl;
}
```

```
Perimeter = 62.8319
```

- TWO is a manifest constant ✓
- PI is a manifest constant as macro ✓
- TWO & PI look like variables

Program after CPP

```
// Contents of <iostream> header replaced by CPP
// Contents of <cmath> header replaced by CPP
using namespace std;

// #define of TWO consumed by CPP
// #define of PI consumed by CPP

int main() { int r = 10;
    double peri = 2 * 4.0*atan(1.0) * r; // By CPP
    cout << "Perimeter = " << peri << endl;
}
```

```
Perimeter = 62.8319
```

- CPP replaces the token TWO by 2
- CPP replaces the token PI by 4.0*atan(1.0) and evaluates
- Compiler sees them as constants
- TWO * PI = 6.28319 by constant folding of compiler

Programming in Modern C++ Partha Pratim Das M06.6

This is outline as always, first let us talk about consonants and cv qualifiers. So, in C, C++, both we have manifest constants or some people call it macros without parameter. So, we do that using hash define. So, we hash define this is a C preprocessor directive and which takes the name and replaces its textual by the string that follows it till the next newline. So, this is called hash define, because you are defining a symbol and if you want to understand a lot more about the CP processor, there is a separate tutorial also you can go through that.

So, here what we are primarily doing is we are defining 2 symbols one is TWO which is defined as constant to another is PI, which is a constant expression. So, what does that mean? atan basically means tan inverse atan is the name of the function which is tan inverse. So, if you do atan 1.0 that is tan inverse 1.0.

So, we all know that tan PI by 4 is 1. So, tan inverse 1.0 is PI by 4. So, if you multiply this by 4 you get a value of PI. So, this is a convenient way to get the value of PI to a great precision. So, now what we are, we do is we use these 2 manifest constants in the expression here to find the perimeter of a circle of radius r. So, this is a manifest constant, and PI is a manifest constant as well as these are also called macros.

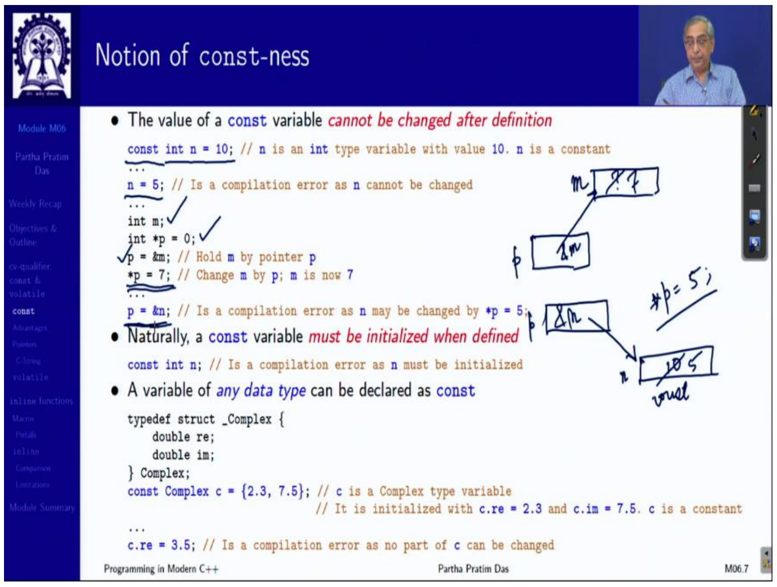
They look like variables, they look like variables this is important, but they are not actually variables. So, what happens when the CP processor works it takes every directive like the #include to #define and several others and takes the action according to the directives. So, #include includes the header files say here. So, if you see the output after your preprocessing you will not see #include<iostream> or #include<cmath> lines.

But you will see big code have come in which are replaced from the library files. Similarly, what it does for hash defines is it takes the definition of the symbol, fixed definition constant definition of the symbol and replaces it wherever the symbol has occurred. So, 2 was occurring here. So, this CPP the CP processor will replace 2 here. PI is occurring here. So, it will take this text string of PI and put it here textually.

And after it is done this then it goes on to the compilation. So, compilation starts after the C preprocessor has worked. So, the basic problem is that the compiler does never get to know that there was something called TWO or something called PI, because compiler has not seen that these are these are all gone.

Compiler sees only this, which is a constant expression, and possibly the compiler will evaluate that constant expression at the compile time because it can, and possibly what is left with is this kind of a numbered 6.28319, which is the approximate value of PI. So, that is where the basic problem of manifest constants lies in the fact that they are they look like variables, but they do not behave like variables compilers never get to see them.

(Refer Slide Time: 7:16)



Notion of const-ness

- The value of a const variable *cannot be changed after definition*

```
const int n = 10; // n is an int type variable with value 10. n is a constant
...
n = 5; // Is a compilation error as n cannot be changed
...
int m;
int *p = 0;
p = &m; // Hold m by pointer p
*p = 7; // Change m by p; m is now 7
...
p = &n; // Is a compilation error as n may be changed by *p = 5;
```

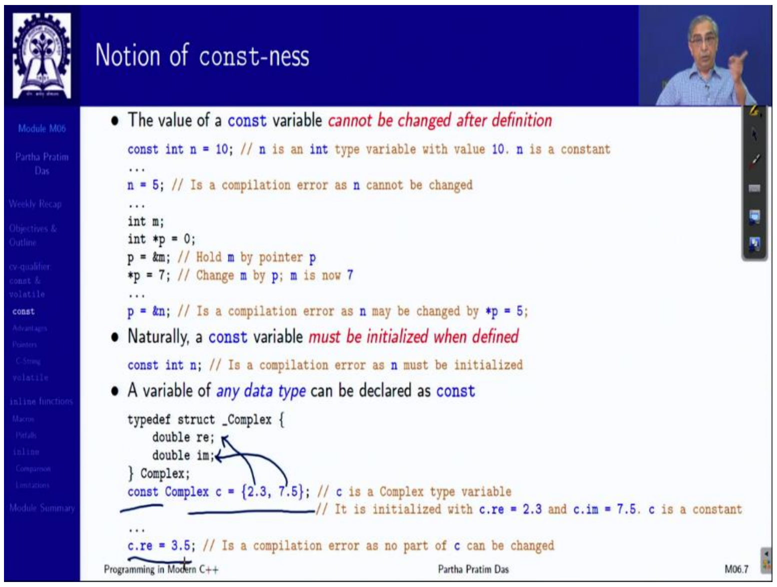
- Naturally, a const variable *must be initialized when defined*

```
const int n; // Is a compilation error as n must be initialized
```

- A variable of *any data type* can be declared as const

```
typedef struct _Complex {
    double re;
    double im;
} Complex;
const Complex c = {2.3, 7.5}; // c is a Complex type variable
                              // It is initialized with c.re = 2.3 and c.im = 7.5. c is a constant
...
c.re = 3.5; // Is a compilation error as no part of c can be changed
```

Programming in Modern C++ Partha Pratim Das M06.7



Notion of const-ness

- The value of a const variable *cannot be changed after definition*

```
const int n = 10; // n is an int type variable with value 10. n is a constant
...
n = 5; // Is a compilation error as n cannot be changed
...
int m;
int *p = 0;
p = &m; // Hold m by pointer p
*p = 7; // Change m by p; m is now 7
...
p = &n; // Is a compilation error as n may be changed by *p = 5;
```

- Naturally, a const variable *must be initialized when defined*

```
const int n; // Is a compilation error as n must be initialized
```

- A variable of *any data type* can be declared as const

```
typedef struct _Complex {
    double re;
    double im;
} Complex;
const Complex c = {2.3, 7.5}; // c is a Complex type variable
                              // It is initialized with c.re = 2.3 and c.im = 7.5. c is a constant
...
c.re = 3.5; // Is a compilation error as no part of c can be changed
```

Programming in Modern C++ Partha Pratim Das M06.7

So, that brings us to the notion of consciousness, the manifest constants, what are their purpose is to define a constant. And what is the purpose of a variable is to take a keep a value which can keep on changing. Now, we can combine these 2 in form of what is known as a constant variable, which is, is a variable in every respect, but, after it has been created, you cannot change its value.

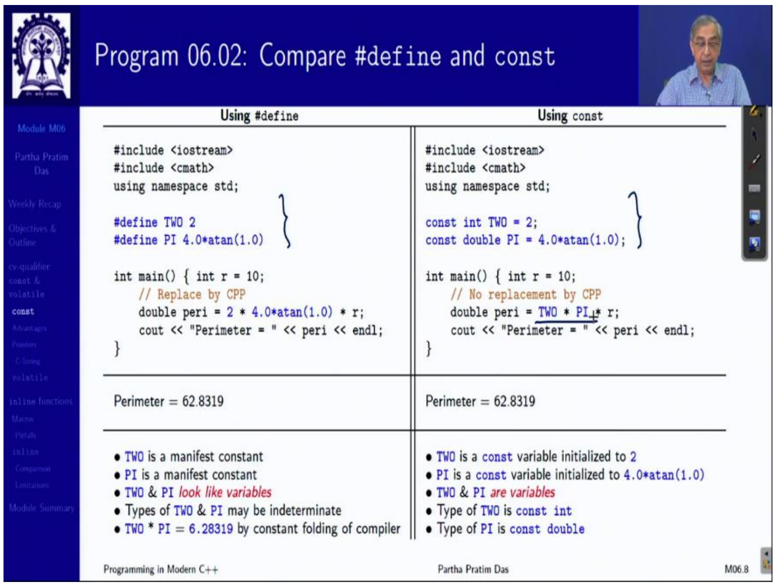
After it is created, after you have defined it, at the time of defining it, you give it a value, and then the value cannot change. If you think hash define is exactly doing that. So, we do that, using this notation, int n initialized with tan is a normal variable. Put the keyword const before that, which means in will not be able to change. So, now, if you want to make an assignment to n that whenever there is a compilation error, because compiler knows that value of n cannot change, it is a constant.

So, in general we have, we write this kind of code integer m is an int, then star p is a pointer to an int, which is initialized to 0. So, I can put the address of m in p. And using p in indirect way. I can do *p assigned 7, which will actually change m. So, it is like this, I have m here which has something, God knows what? Then I have p here, I have put &m here, so p points here. And I do *p assign 7 so this becomes 7.

So, I am not even talked about it, but I can change the value of m this is a common program. Can I do this? Can I have p and make it point to n which is ten const. Can I do this? If I could do this, then I could have done say *p assigns 5 and that will change this to 5 then the basic property that n is constant will get destroyed, so I cannot do this. So, I say that p, why p is a pointer to an integer &n address of n is actually a pointer to a constant integer. That is a different concept. So, being a pointer to an integer and being appointed to a constant integer are different.

So, like we will see a little more of that soon. So naturally, a constant variable must be initialized when defined, because otherwise you cannot change them. And variable of any data type can be a constant. So, I have given an example of the complex number variable. So, here, you can see that I am saying this is const and initialized it, so which respectively initializes the 2 components, but then I if I want to change any component separately, I will get a compilation error, because C is a constant, no component can be changed, that is the basic idea of const-ness.

(Refer Slide Time: 11:12)



Program 06.02: Compare #define and const

Using #define

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

#define TWO 2
#define PI 4.0*atan(1.0)

int main() { int r = 10;
    // Replace by CPP
    double peri = 2 * 4.0*atan(1.0) * r;
    cout << "Perimeter = " << peri << endl;
}
```

```
Perimeter = 62.8319
```

- TWO is a manifest constant
- PI is a manifest constant
- TWO & PI *look like variables*
- Types of TWO & PI may be indeterminate
- TWO * PI = 6.28319 by constant folding of compiler

Using const

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

const int TWO = 2;
const double PI = 4.0*atan(1.0);

int main() { int r = 10;
    // No replacement by CPP
    double peri = TWO * PI * r;
    cout << "Perimeter = " << peri << endl;
}
```

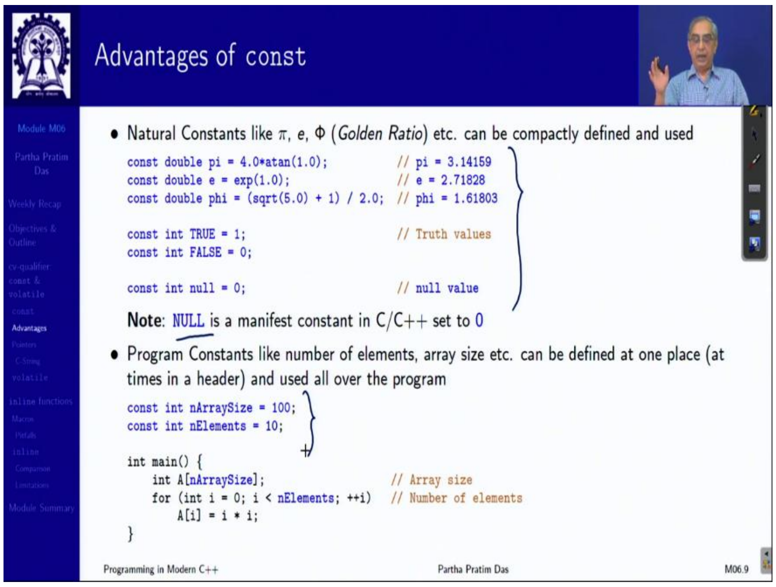
```
Perimeter = 62.8319
```

- TWO is a const variable initialized to 2
- PI is a const variable initialized to 4.0*atan(1.0)
- TWO & PI *are variables*
- Type of TWO is const int
- Type of PI is const double

Programming in Modern C++ Partha Pratim Das M06.8

So, now let us try to use this, this is this was my manifest constant or macro style of definitions. Now, I have it in terms of const. So, now the C preprocessor does not replace, TWO and PI, because they are no more #defined, they are no more manifest constants, they are just variables. So, the compiler gets to see them, compiler gets to know their type. And the compiler can take a lot of decisions based on what their type and the value is. So, that is the basic advantage of having const in the system.

(Refer Slide Time: 11:59)



Advantages of const

- Natural Constants like π, e, Φ (*Golden Ratio*) etc. can be compactly defined and used

```
const double pi = 4.0*atan(1.0);           // pi = 3.14159
const double e = exp(1.0);                 // e = 2.71828
const double phi = (sqrt(5.0) + 1) / 2.0;  // phi = 1.61803

const int TRUE = 1;                        // Truth values
const int FALSE = 0;

const int null = 0;                        // null value
```

Note: NULL is a manifest constant in C/C++ set to 0

- Program Constants like number of elements, array size etc. can be defined at one place (at times in a header) and used all over the program

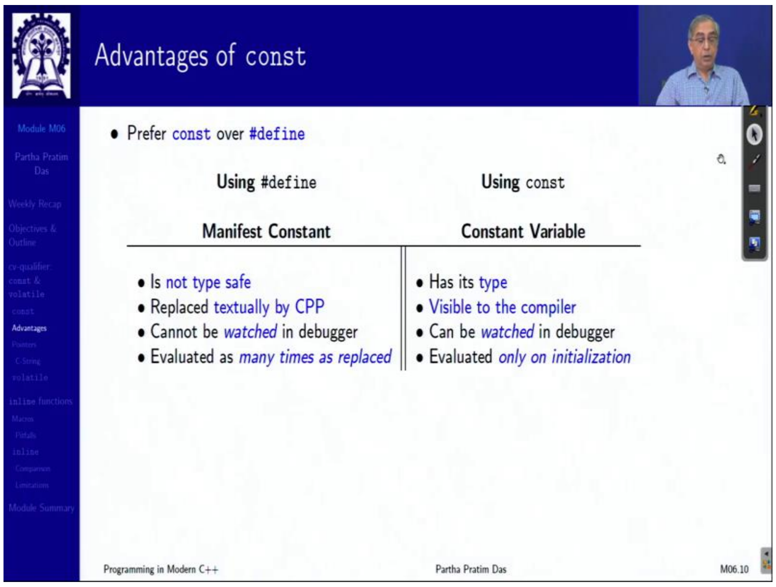
```
const int nArraySize = 100;
const int nElements = 10;

int main() {
    int A[nArraySize];                     // Array size
    for (int i = 0; i < nElements; ++i)    // Number of elements
        A[i] = i * i;
}
```

Programming in Modern C++ Partha Pratim Das M06.9

So, basic advantages, one set of constant values are basically naturally constant numbers like pi, e, golden ratio, and so on so forth. Whereas remember that what you use as in you, NULL capital in C or C++ is actually not a const value, it is actually manifest constant. And then the other value of the constant is to define program constants, there is several things like maximum size of an array number of elements, number of people, if there are these are constant values, and we you can always put them in the top or in some header together and define them only once and make sure that all through the program, they cannot be changed even inadvertently, because we have made them const. So, it makes the programming much safer, much easier to do.

(Refer Slide Time: 12:55)



Advantages of const

Module M06

Partha Pratim Das

Weekly Recap

Objectives & Outline

cv-qualifier: const & volatile

const

Advantages

Pointers

C-String

volatile

inline functions

Macros

Pitfalls

inline

Comparison

Limitations

Module Summary

- Prefer const over #define

Using #define	Using const
Manifest Constant	Constant Variable
• Is not type safe	• Has its type
• Replaced textually by CPP	• Visible to the compiler
• Cannot be *watched* in debugger	• Can be *watched* in debugger
• Evaluated as *many times as replaced*	• Evaluated *only on initialization*

Programming in Modern C++ Partha Pratim Das M06.10

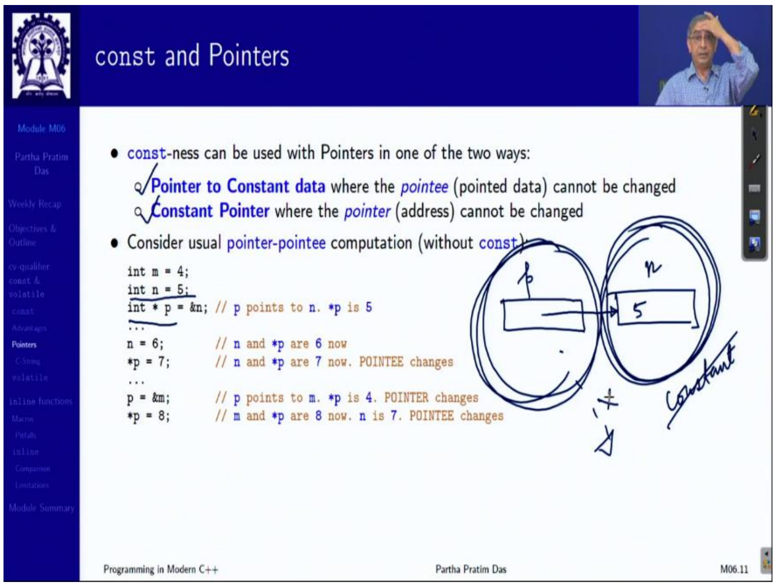
So, we would always prefer const over #define. So, these are some of the reasons that manifest constant does not have a type. Whereas constant variable has a type. So, it is type safe. Manifest constant is textually replaced by CPP. So, you do not get to actually see if you have a number of manifest constants and expressions involving them, then you do not get to see as to what is the result of this replacement.

Compiler, what the compiler sees, you do not get to see and what you see is what the compiler does not see. But if you use a constant variable, both of you see the same thing. Since manifest constants are replaced by the preprocessor, they cannot be watched in the debugger, you cannot see what is the value of n in the debugger, if it is manifest constant, whereas if it is const int, you can see, see it in the debugger.

And also, the manifest constant like the PI expression, we saw, they are replaced wherever they are replaced, they are evaluated. Whereas if you have a constant variable, then naturally its evaluation happened only at the definition time. Everywhere else, we just use that value.

So, it is also efficient. So, in every possible way, manifest constant or it no no, stop using manifest constant just use constant variables.

(Refer Slide Time: 14:24)



const and Pointers

- const-ness can be used with Pointers in one of the two ways:
  - **Pointer to Constant data** where the *pointee* (pointed data) cannot be changed
  - **Constant Pointer** where the *pointer* (address) cannot be changed
- Consider usual pointer-pointee computation (without const):

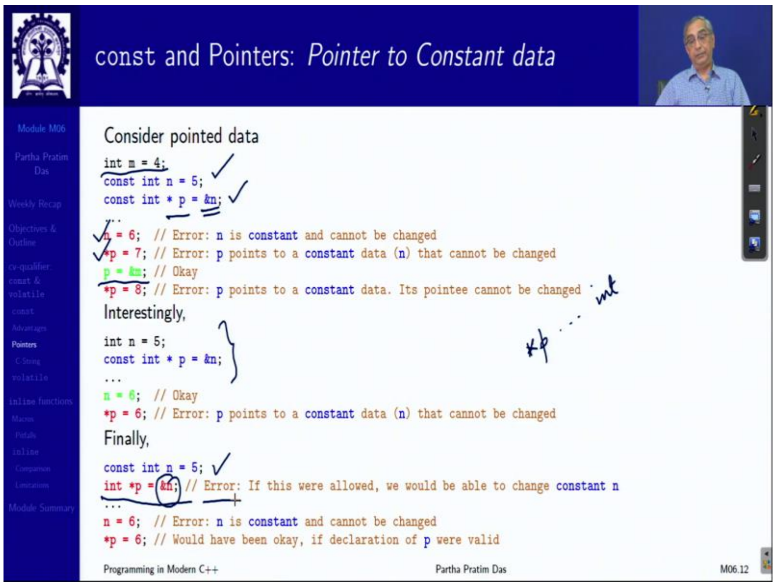
```
int m = 4;
int n = 5;
int * p = &n; // p points to n. *p is 5
...
n = 6;         // n and *p are 6 now
*p = 7;        // n and *p are 7 now. POINTEE changes
...
p = &m;        // p points to m. *p is 4. POINTER changes
*p = 8;        // m and *p are 8 now. n is 7. POINTEE changes
```

Programming in Modern C++ Partha Pratim Das M06.11

Now, let us quickly look at constants and their interplay with the pointer. See, how consciousness apply to the pointer scenario. So, suppose I have a like, I have here I have n here, I have p here, n is a has a value 5 and I have p as a pointer to n. So, this is a structure. Now, the question is if I want to talk about constants, what is constant? Is this constant that is I cannot change the value it is pointing to or is this const that I cannot change this point that is I cannot make it point to something else.

So, there are 2 notions of constants, one is pointed to constant data, that is if p points to something which cannot be changed, then it is a point at which constant data this is constant. Whereas, if p is a constant pointer, then it cannot be changed to point to anything else cannot be done it becomes a constant pointer, 2 types of consciousness. And actually, you can have both at the same time as well.

(Refer Slide Time: 15:59)



const and Pointers: *Pointer to Constant data*

Consider pointed data

```
int m = 4;
const int n = 5;
const int * p = &n;
...
n = 6;  // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
p = &m; // Okay
*p = 8; // Error: p points to a constant data. Its pointee cannot be changed
```

Interestingly,

```
int n = 5;
const int * p = &n;
...
n = 6;  // Okay
*p = 6; // Error: p points to a constant data (n) that cannot be changed
```

Finally,

```
const int n = 5;
int *p = &n; // Error: If this were allowed, we would be able to change constant n
...
n = 6;  // Error: n is constant and cannot be changed
*p = 6; // Would have been okay, if declaration of p were valid
```

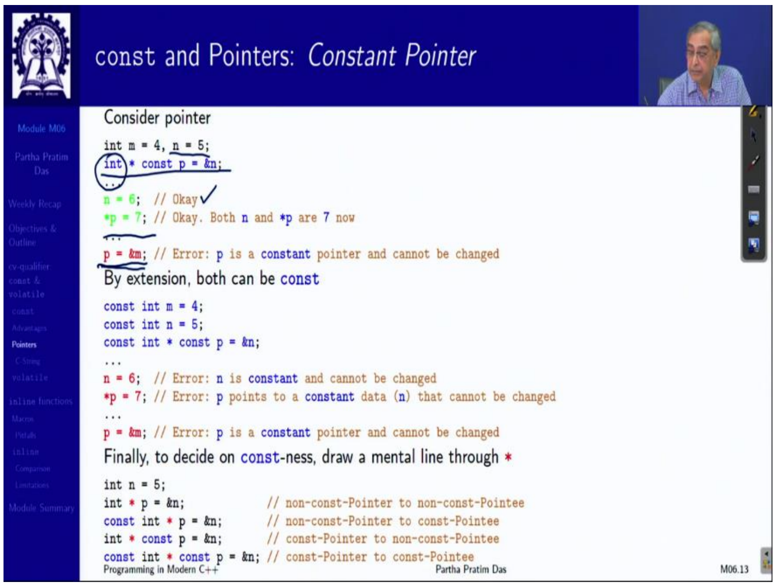
Programming in Modern C++ Partha Pratim Das M06.12

So, here are the examples. So, you see a pointer to constant data, so, you have a constant variable here and you have a *p which points to a constant int. So, you can use this. Now, naturally you cannot assign to n because it is constant you cannot assign to *p, because, if p is pointing to a constant, then *p is constant. So, the fact that he will use p to change n is not possible, you get this safe.

Whereas, p can point to some other m, which is not a constant, and you can change m accordingly. So, we can have this kind of situations as well. Now, we can also say we have a constant here. And suppose I want to use that address to initialize int *p that also is an error. Why? Because int *p tells me that *p is int it is not constant.

So, if I am allowed to put the address of const int into p, then when I do start p and change it, I will not be able to know that I am changing something which I am not supposed to change. So, this itself is an error. So, that is how the logic of that is just the way to look at the logic of const-ness.

(Refer Slide Time: 17:48)



const and Pointers: *Constant Pointer*

Consider pointer

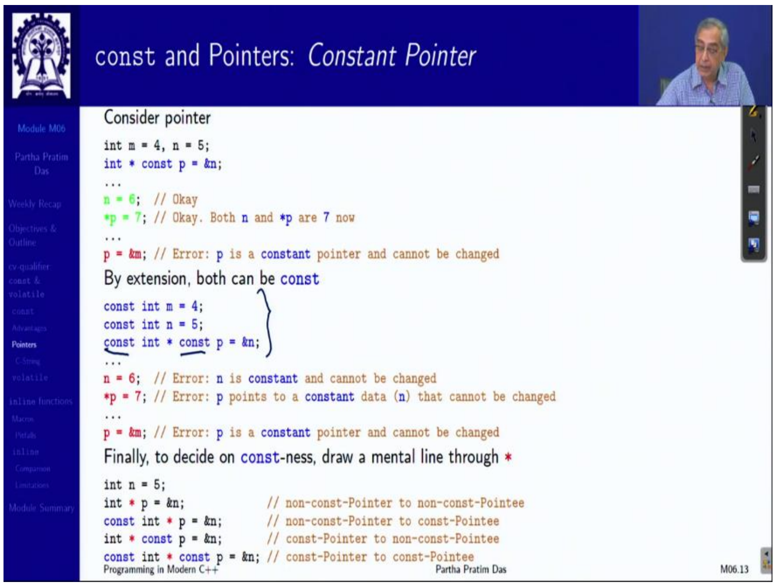
```
int m = 4, n = 5;
int * const p = &n;
...
n = 6;  // Okay ✓
*p = 7; // Okay. Both n and *p are 7 now
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

By extension, both can be const

```
const int m = 4;
const int n = 5;
const int * const p = &n;
...
n = 6;  // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Finally, to decide on const-ness, draw a mental line through *

```
int n = 5;
int * p = &n;             // non-const-Pointer to non-const-Pointee
const int * p = &n;       // non-const-Pointer to const-Pointee
int * const p = &n;       // const-Pointer to non-const-Pointee
const int * const p = &n; // const-Pointer to const-Pointee
```



const and Pointers: *Constant Pointer*

Consider pointer

```
int m = 4, n = 5;
int * const p = &n;
...
n = 6;  // Okay
*p = 7; // Okay. Both n and *p are 7 now
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

By extension, both can be const

```
const int m = 4;
const int n = 5;
const int * const p = &n;
...
n = 6;  // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Finally, to decide on const-ness, draw a mental line through *

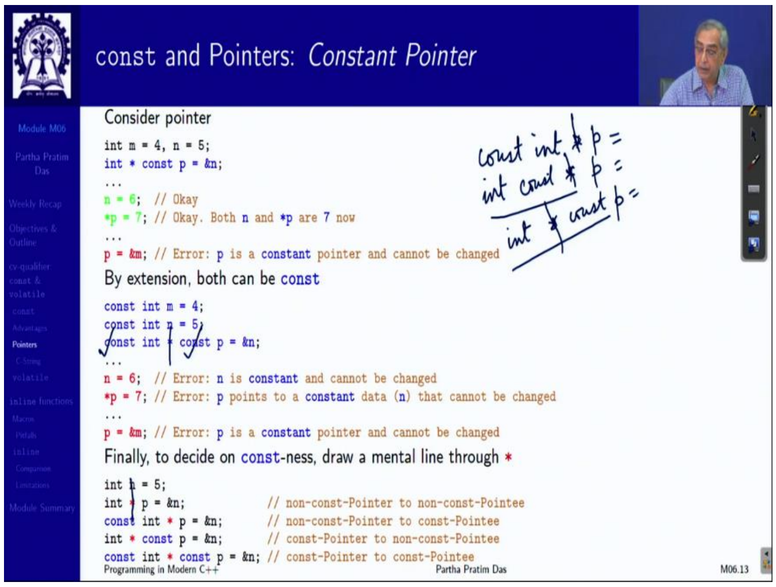
```
int n = 5;
int * p = &n;             // non-const-Pointer to non-const-Pointee
const int * p = &n;       // non-const-Pointer to const-Pointee
int * const p = &n;       // const-Pointer to non-const-Pointee
const int * const p = &n; // const-Pointer to const-Pointee
```

Programming in Modern C++ Partha Pratim Das M06.13

So, here, you can see that I am talking about another type of const-ness. That is I am saying int star const p which is a constant pointer to a non-constant data. So, as n is not a constant, I can put a value. I can do *p, *p is int, *p is int. So, it is changeable, I can assign 7 to it. So, by which n will change to 7 that is okay. But I cannot do this. I cannot make p point to a different variable m because the pointer itself is constant.

So, earlier we saw the pointed data point is constant here we see that the pointer is constant. So, it depends on which side of the star you write the constant. So, if we look in here, then we can certainly have constant, both sides, which says that it is a constant pointed to a constant data that is using p neither I can change n because it is a constant is pointing to a constant data. Nor can I assign any other address to p because it is a constant pointer. So, that is a basic type of const-ness with pointer. Now, you may from the syntax you may get confused at times. So, the for example, I would just tell you how to get confused.

(Refer Slide Time: 19:52)



const and Pointers: *Constant Pointer*

Consider pointer

```
int m = 4, n = 5;
int * const p = &n;
...
n = 6;  // Okay
*p = 7; // Okay. Both n and *p are 7 now
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

By extension, both can be const

```
const int m = 4;
const int n = 5;
const int * const p = &n;
...
n = 6;  // Error: n is constant and cannot be changed
*p = 7; // Error: p points to a constant data (n) that cannot be changed
...
p = &m; // Error: p is a constant pointer and cannot be changed
```

Finally, to decide on const-ness, draw a mental line through *

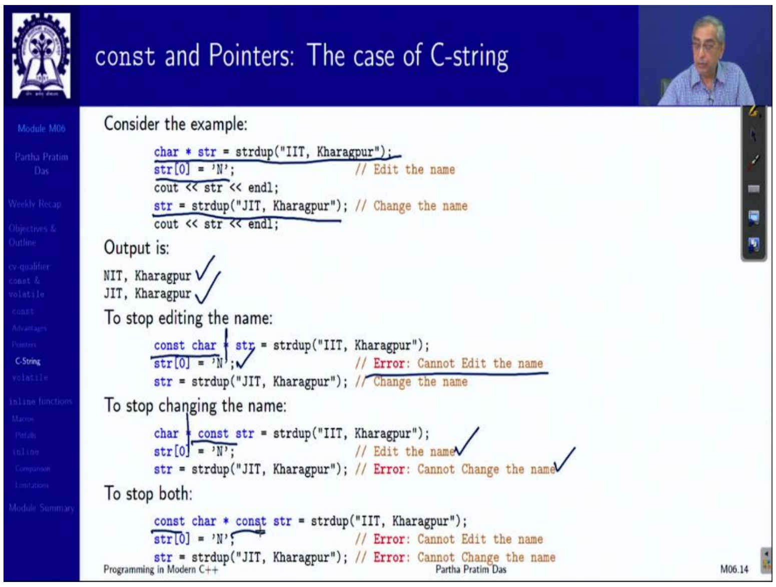
```
int n = 5;
int * p = &n;               // non-const-Pointer to non-const-Pointee
const int * p = &n;         // non-const-Pointer to const-Pointee
int * const p = &n;         // const-Pointer to non-const-Pointee
const int * const p = &n;   // const-Pointer to const-Pointee
```

Programming in Modern C++ Partha Pratim Das M06.13

For example, say I am writing const int *p that is alright. This is a pointer to a constant data I am writing int const * p this also is valid and it also means a pointer to a constant data. I write int * const p, now, it is not pointed to a constant it is a constant pointer to non-constant int confusing between these two?

There is a very simple rule whenever you have to write or when you see the code, just draw a vertical line through the star, whichever side the constants that side is constant. So, if you draw it here, it is on the point int side the point int constant, you write here it is also on the point side the point is called. If you write it, here it is on the pointed side. So, pointed is constant. You write here it is on both sides. So, both are constant. That is a simple way to resolve. Otherwise, it gets quite confusing at times.

(Refer Slide Time: 21:17)



const and Pointers: The case of C-string

Consider the example:

```
char * str = strdup("IIT, Kharagpur");
str[0] = 'N';                         // Edit the name
cout << str << endl;
str = strdup("JIT, Kharagpur"); // Change the name
cout << str << endl;
```

Output is:

```
NIT, Kharagpur
JIT, Kharagpur
```

To stop editing the name:

```
const char * str = strdup("IIT, Kharagpur");
str[0] = 'N';                         // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Change the name
```

To stop changing the name:

```
char * const str = strdup("IIT, Kharagpur");
str[0] = 'N';                         // Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

To stop both:

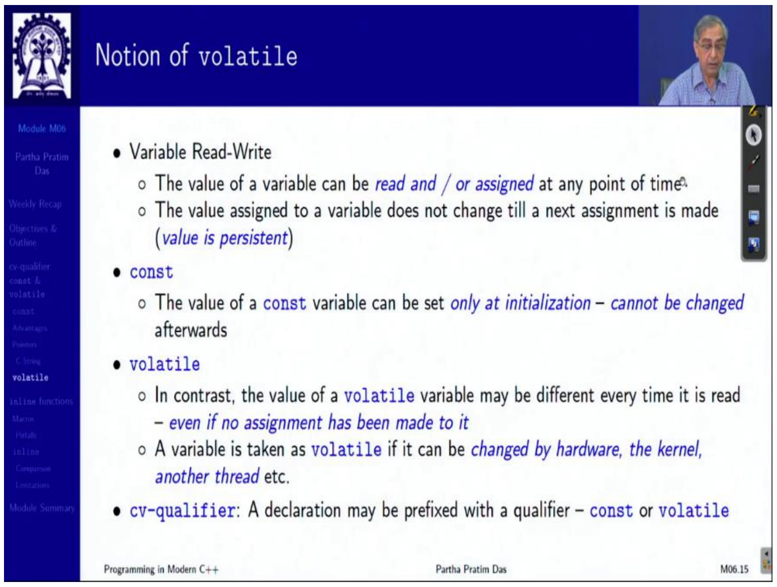
```
const char * const str = strdup("IIT, Kharagpur");
str[0] = 'N';                         // Error: Cannot Edit the name
str = strdup("JIT, Kharagpur"); // Error: Cannot Change the name
```

Programming in Modern C++ Partha Pratim Das M06.14

So, here is an example of using const with the C string, you can see that I can have a string created and I can access an element and change it. I can copy another string into this name that is I can change an element or I can change the name altogether. So, NIT Kharagpur, JIT Kharagpur you have.

Now, if I put a const here, that is if I make the elements constant studies here, so pointy elements are constant. So, if I make those constant, then naturally I cannot change them I cannot edit this becomes a compilation error. If I put it on this site that is if I put it on the pointer site, then I can change the element change said it the name but I cannot change the name. And if I put it on both, I cannot change either. So, it depends on what is what is that I want and accordingly I can do these changes.

(Refer Slide Time: 22:44)



Notion of volatile

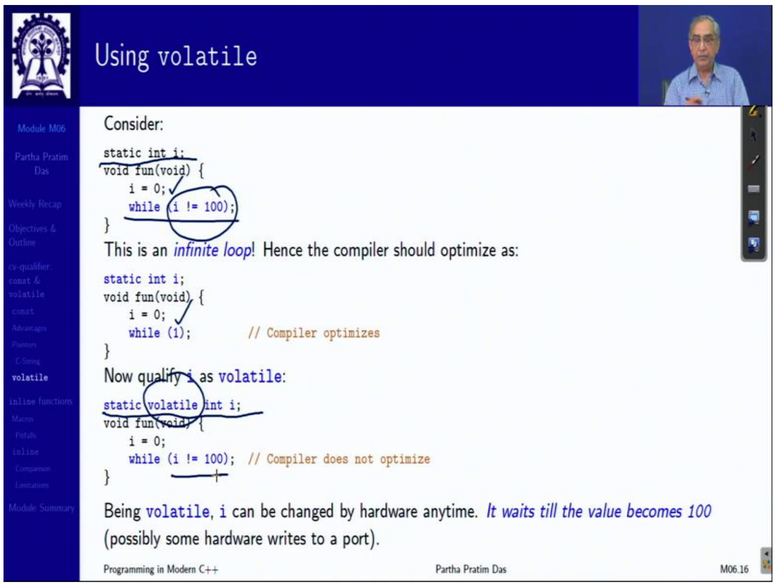
- Variable Read-Write
  - The value of a variable can be *read and / or assigned* at any point of time
  - The value assigned to a variable does not change till a next assignment is made (*value is persistent*)
- const
  - The value of a const variable can be set *only at initialization* – *cannot be 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

Programming in Modern C++ Partha Pratim Das M06.15

So, there is another concept related to this it is called cv qualifier, c is for constant, v is for volatile. So, we have talked about constant where initialize that cannot be changed. Volatile is a very peculiar concept when you say that a volatile a variable is volatile if its value can change even without an assignment. Normally what we know that you have to do a right for the value to change. But if I have a variable whose value can change by itself automatically, then it is called a volatile variable. Volatile variables can happen because of several things.

For example, I may have modeled a port or your port as a variable. When the data is written to that port, naturally, then the variable value has changed. A different data is written to the port to value a variable has changed and I do not know what value the hardware is going to write to that port. So, without me knowing the value of the variable can change. This can happen because of carnal writing something this can happen because of some other thread writing a variable.

(Refer Slide Time: 23:51)



Using volatile

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).

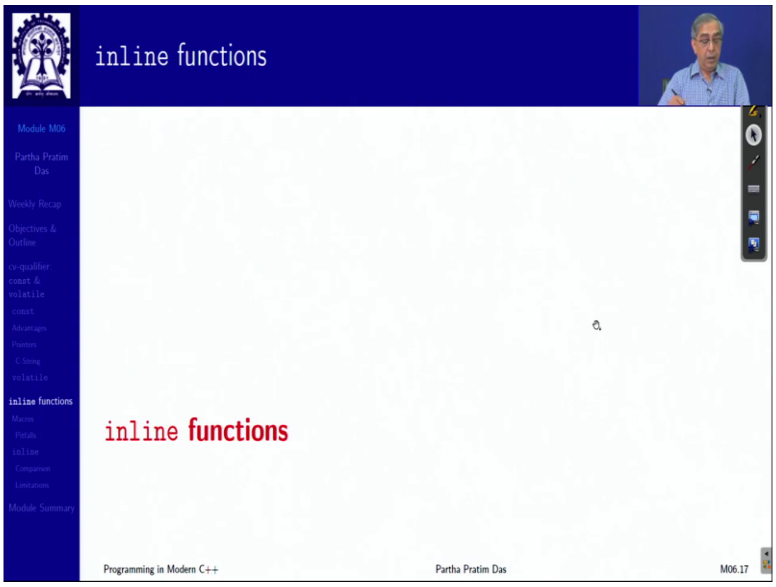
Programming in Modern C++ Partha Pratim Das M06.16

So, such variables are typically can be marked as a volatile, because it does help in a in a significant way for the compiler. Let us see here, i's assigned 0 and I have this loop. Now, you obviously can know that since no changes to i have being made, this loop is going to be an infinite, I can i is not equal to 100 and will never become 100, this is an infinite group.

So, what the compiler will do? It will optimize and say that this can never be this is always true. So, it will make it 0, this optimization. Now, suppose you change that instead of static in time, you just make it static volatile in time. So, you are saying that i can change by itself. Now, the compiler will not be able to optimize because it does not know if somebody can at some point of time can change the value of i, and make it 100.

If it does, then the loop will exit. So, being volatile I can be changed by the hardware or by other threat at any point of time, so it helps it is not a, it is not a very frequent use. But this is the other kind of qualified besides the const, which has, the other extreme of the meaning is what is useful in terms of the qualified variable values.

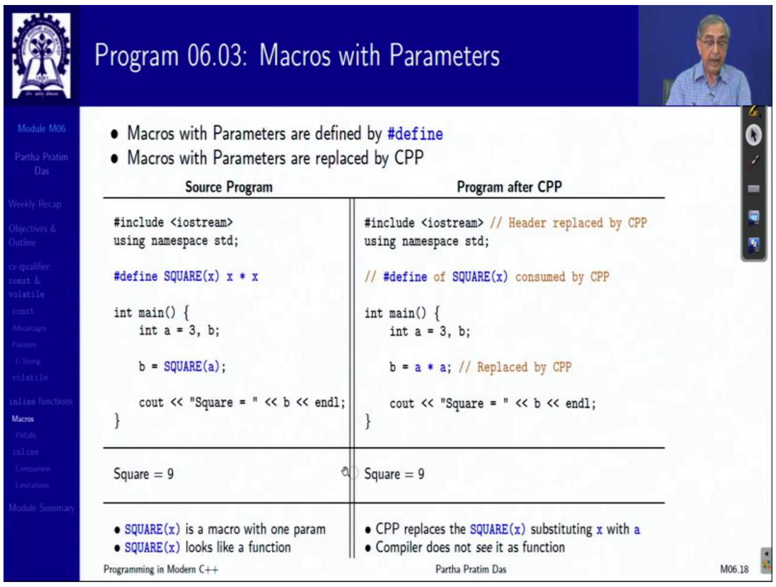
(Refer Slide Time: 25:21)



inline functions

inline functions

Programming in Modern C++ Partha Pratim Das M06.17



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
see code below	see code below

Source Program:

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

#define SQUARE(x) x * x

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

    b = SQUARE(a);

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

Program after CPP:

```
#include <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;
}
```

```
Square = 9
```

```
Square = 9
```

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

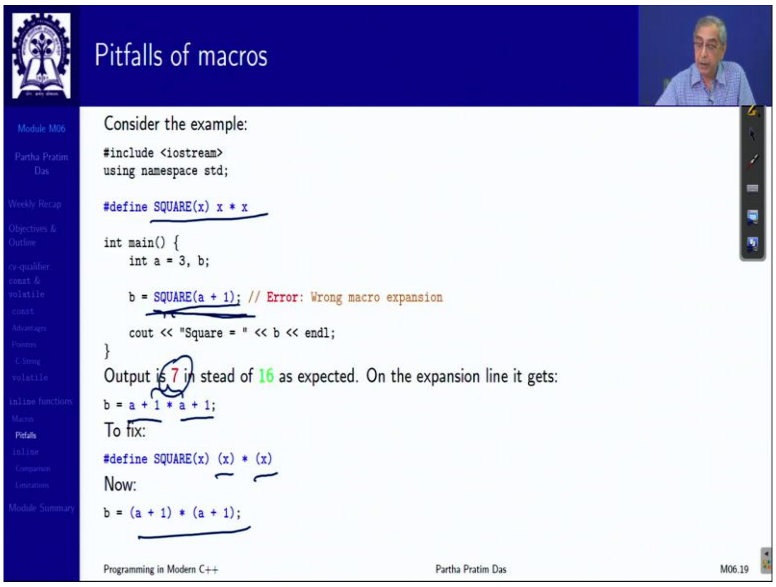
- CPP replaces the SQUARE(x) substituting x with a
- Compiler does not *see* it as function

Programming in Modern C++ Partha Pratim Das M06.18

Next, we will talk briefly about inline functions. Inline functions are like we have macros. So, so far, we talked about manifest constants, which are macros without parameters. But these are macros with parameters. So, this is like, again, their #define, so they have a parameter.

And then expressions, when I invoke I say, square a, they are textually replaced, and you actually get in preprocessor expansion, you will get this replaced by a * a, which is a very simple concept, and very useful for having compact computations done, it we I could have done a function also, but function has a lot more of overhead to write this multiplication, beyond this multiplications, which is just here in place, so it gets a lot more efficient to me.

(Refer Slide Time: 26:18)



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); // Error: Wrong macro expansion

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

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:

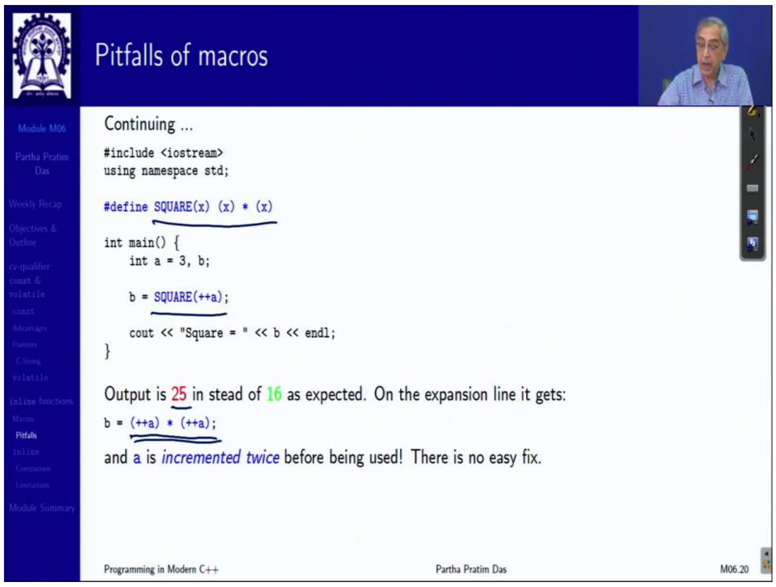
```
b = (a + 1) * (a + 1);
```

Programming in Modern C++ Partha Pratim Das M06.19

Now, there are several pitfalls and I am sure many of you would know about these that what if I define the macro like this and call it by a plus 1. Now, I would expect that if a is 3 then a plus 1 is 4. So, mentally, I am thinking that this is a function. So, I expect a square I want 16. But if you do this, you will get a result 7 why, because this is textually replaced by the C preprocessor.

So, what it will replace is a plus 1-star a plus 1. So, now this becomes the priority operation. So, it becomes a plus a plus 1, which is 3 plus 3 plus 1 is 6 plus 1 is 7, is what you get. It can you can get rid of this by putting parentheses around them, then it will be like this. And this is what you must have read in different books also that by doing this, you will make the macro safe. So, let us try to be the devil a little bit more.

(Refer Slide Time: 27:28)



Pitfalls of macros

Continuing ...

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

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

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

    b = SQUARE(++a);

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

Output is 25 in stead of 16 as expected. On the expansion line it gets:

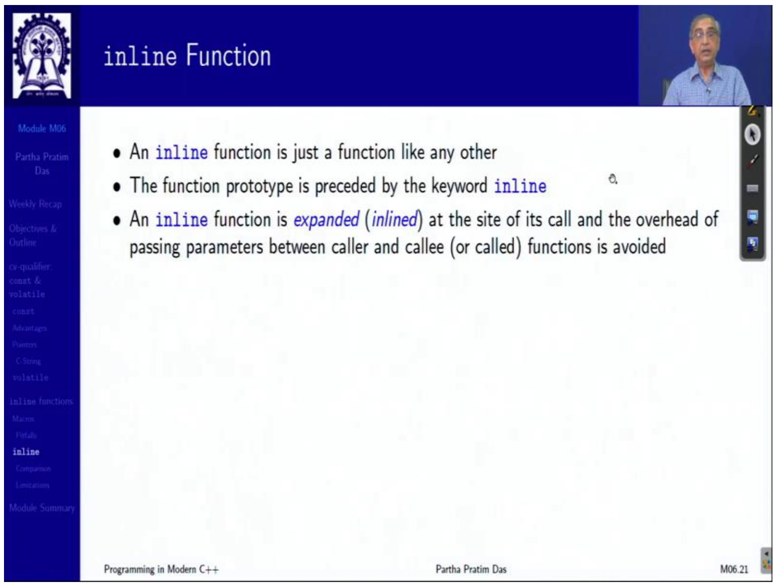
```
b = (++a) * (++a);
```

and a is *incremented twice* before being used! There is no easy fix.

Programming in Modern C++ Partha Pratim Das M06.20

I have the same. Now, I have the corrected macro. And I invoke it by plus, plus way, what I expect 3, plus plus a is 4, so I will get 16. But try this, you will get 25. Because you get this as the replacement code. So, instead of 1 plus plus, there are 2 plus pluses, 2 pre-increments. So, 3 gets 4, 4 gets 5, and then 5 gets multiplied by 5 gets 25. And it is not easy to fix by writing a macro in this way. So, macros are good, but they are an evil. So, what a C++ does?

(Refer Slide Time: 28:13)



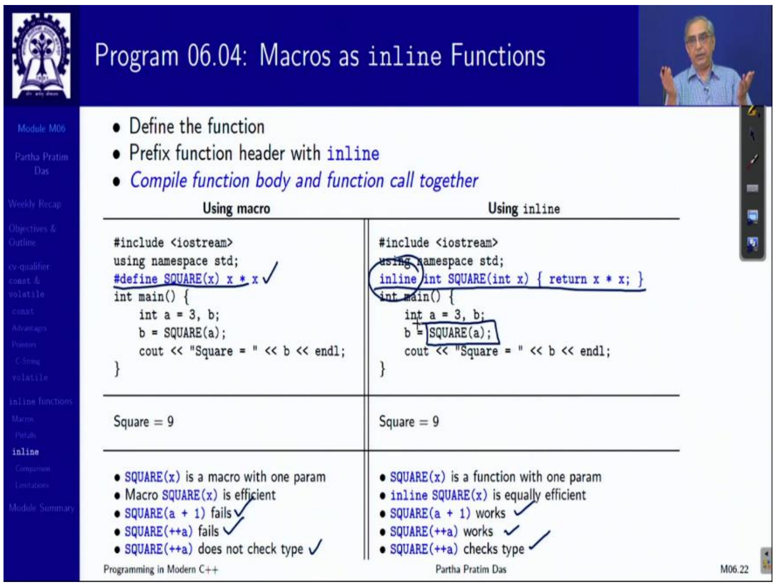
inline Function

- An inline function is just a function like any other
- 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

Programming in Modern C++ Partha Pratim Das M06.21

It defines the concept of an inline function, it is extremely simple, it is just like any other function just to write inline the keyword inline in front of it.

(Refer Slide Time: 28:23)



Program 06.04: Macros as inline Functions

- Define the function
- Prefix function header with `inline`
- *Compile function body and function call together*

Using macro

```
#include <iostream>
using namespace std;
#define SQUARE(x) x * x
int main() {
    int a = 3, b;
    b = SQUARE(a);
    cout << "Square = " << b << endl;
}
```

Square = 9

- `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 ✓

Using inline

```
#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;
}
```

Square = 9

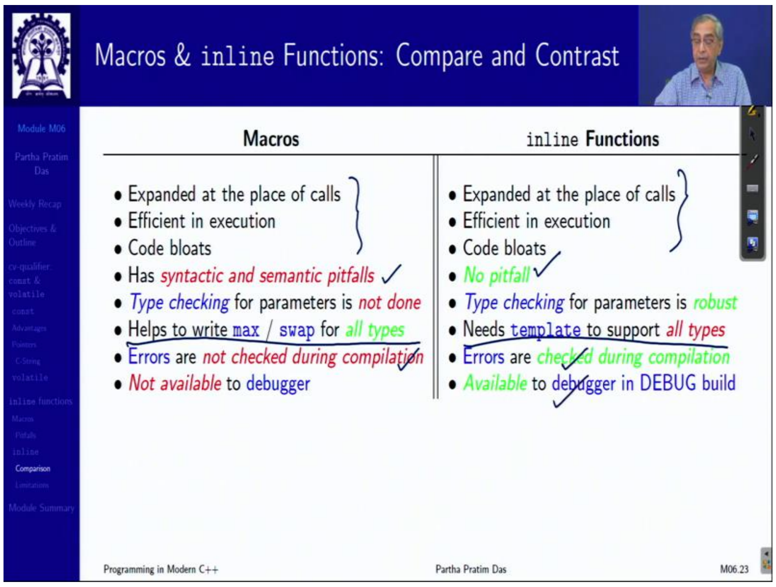
- `SQUARE(x)` is a function with one param
- `inline SQUARE(x)` is equally efficient
- `SQUARE(a + 1)` works ✓
- `SQUARE(++a)` works ✓
- `SQUARE(++a)` checks type ✓

Programming in Modern C++ Partha Pratim Das M06.22

So, instead of writing this, you just write this which is a function. So, it since is a function, the system will first evaluate the evaluate the actual parameter and then make a call to it. So, if I if we do this, then here with this, this will fail, this will fail. There is no type check, but here everything will work fine, because it is a function.

And because it is inline, it actually is not making the function call, the compiler does a trick so that the function is embedded right here. But with all the type checks and with all function protocols, like evaluate your actual parameter and then pass it to formal parameter, not just do a textual replacement and so on. So, the semantics of function remains but the efficiency of macro obtained it is a win-win situation. So, that is the basic idea of the inline function.

(Refer Slide Time: 29:32)



Macros & inline Functions: Compare and Contrast

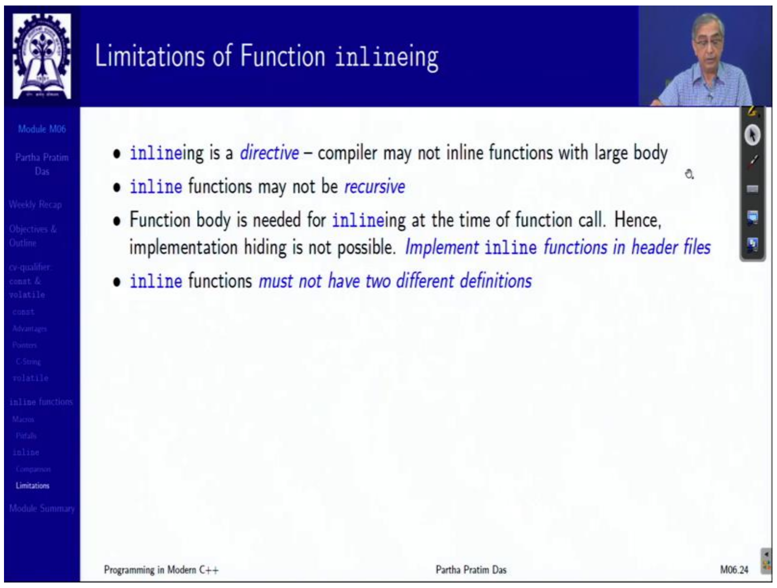
Macros	`inline` Functions
Expanded at the place of calls	Expanded at the place of calls
Efficient in execution	Efficient in execution
Code bloats	Code bloats
Has *syntactic and semantic pitfalls* ✓	*No pitfall* ✓
Type checking for parameters is *not done*	*Type checking* for parameters is *robust*
Helps to write `max` / `swap` for *all types*	Needs `template` to support *all types*
Errors are *not checked during compilation* ✓	Errors are *checked during compilation* ✓
Not available to debugger	*Available* to debugger in DEBUG build ✓

Programming in Modern C++ Partha Pratim Das M06.23

So, you should stop using macros and only use inline functions. Here is a simple comparison of that. In the first 3 key points are same between the 2, the code bloats and all that. But macro has significant syntactic and semantic pitfall which in line does not have type checking is available in inline and it helps to write max swap for all types, which obviously cannot be done by inline function.

And we will see a different solution for that, like you can, and one way you write a swap function using a macro, because you need to, we need to work for any type of data. But we have already seen and we can, we can handle those situations using that template and meta programming. Otherwise, everything else in terms of inline functions are a plus. And we would always use inline functions only instead of macros.

(Refer Slide Time: 30:36)



Limitations of Function inlineing

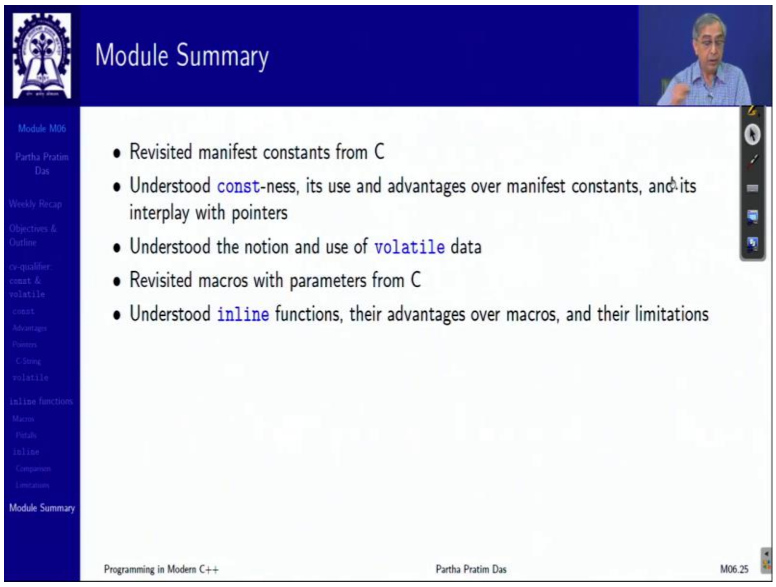
- inlineing is a *directive* – compiler may not inline functions with large body
- inline functions may not be *recursive*
- Function body is needed for inlineing at the time of function call. Hence, implementation hiding is not possible. *Implement inline functions in header files*
- inline functions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

Programming in Modern C++ Partha Pratim Das M06.24

Now, of course, inline also is you should keep in mind that it is a directive. So, just because you write inline, it may or may not be inlined. Because the compiler makes a judgment as to whether it is more efficient to inline or it is more efficient to not inline, outline the function so to say. So, it is a compiler directive compiler makes a judgment on that.

And always implement the inline functions on the headers. So, that you do not have 2 different definitions of the inline functions. Do not implement inline functions in the source file which is otherwise because the compiler needs to know the code it when the function is called just knowing the signature will not work.

(Refer Slide Time: 31:27)



Module Summary

- 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

Module M06

Partha Pratim Das

Weekly Recap

Objectives & Outline

cv-qualifier

const & volatile

const

Advantages

Pointers

C-String

volatile

inline functions

Macros

Pitfalls

inline

Comparison

Limitations

Module Summary

Programming in Modern C++ Partha Pratim Das M06.25

So, the to summarize, we have looked at the 2 very basic extensions of C into C++ that is const-ness and inline function which can do a with macros all kinds of macros without or with parameters. Thank you very much for your attention, and too we will meet in the next module.