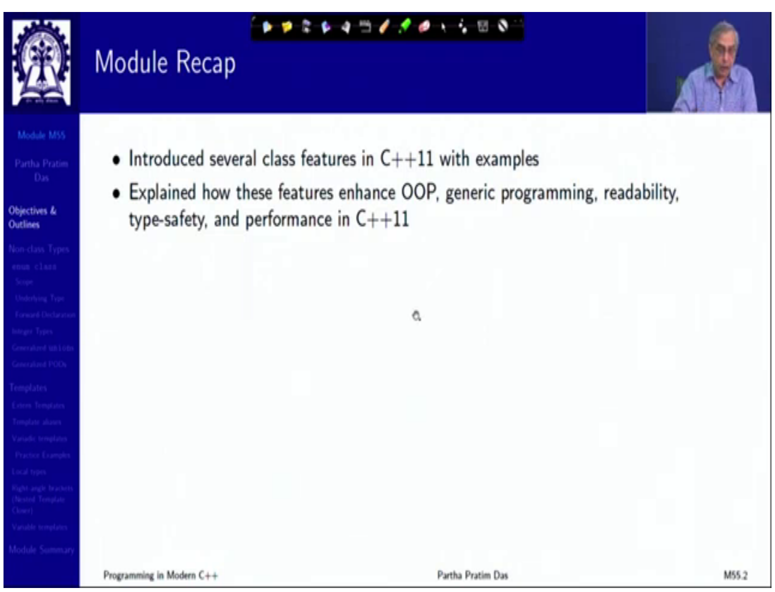
Programming in Modern C++
Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 55
C++11 and beyond: Non-class Types and Template Features

(Refer Slide Time: 00:32)

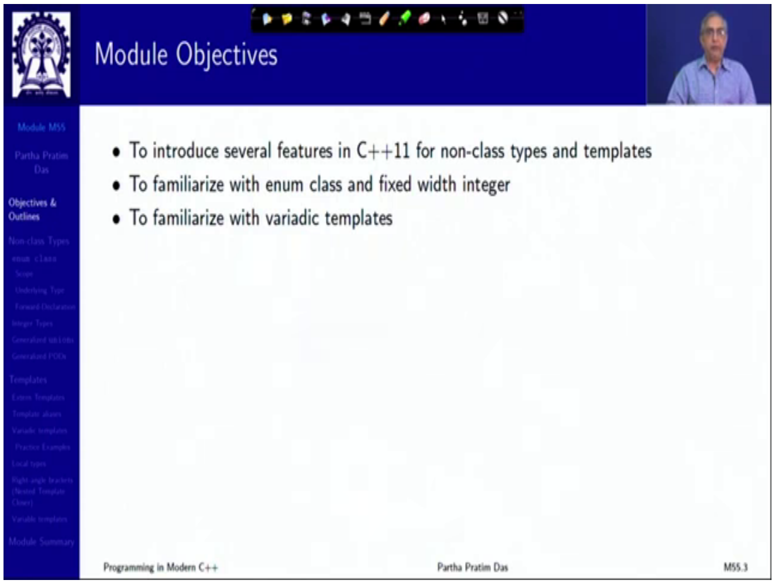
Welcome to Programming in Modern C++. We are in Week 11 and we are going to discuss Module 55.

(Refer Slide Time: 00:37)



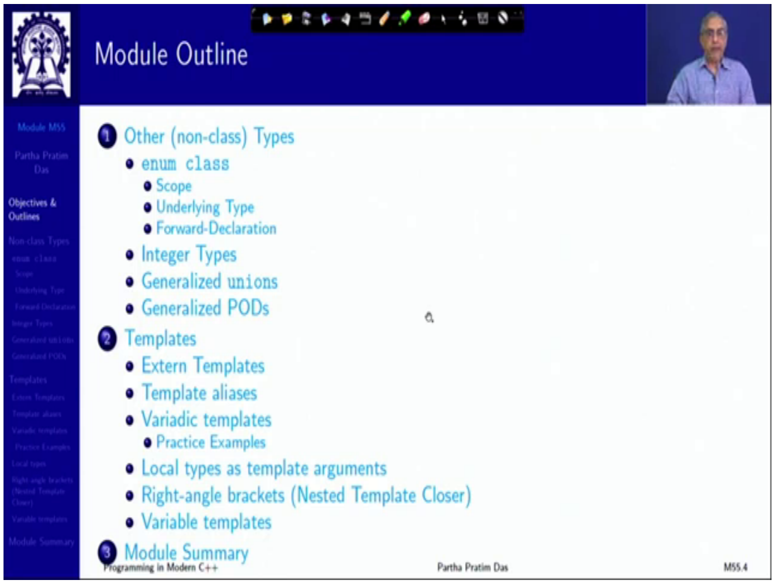
In the last module, we concluded discussions on several features of C++ which relate to classes and explained how these features enhance the object orientation, generic programming, readability, type safety and performance in C++11.

(Refer Slide Time: 00:59)



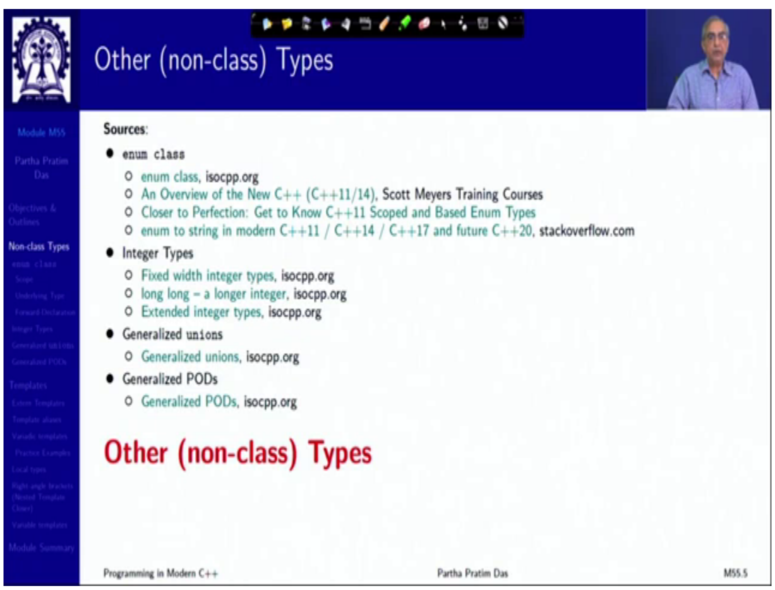
In the current module, we in a way continue that discussion for several features of C++11 that apply to non-class types, we typically call them other types, and to templates and the objective remains the same is to enhance the OOP, generic programming, readability, performance, type safety and so on. In specific, we familiarize with very important features like enum class fixed width integer, and most importantly, the variadic templates.

(Refer Slide Time: 01:31)



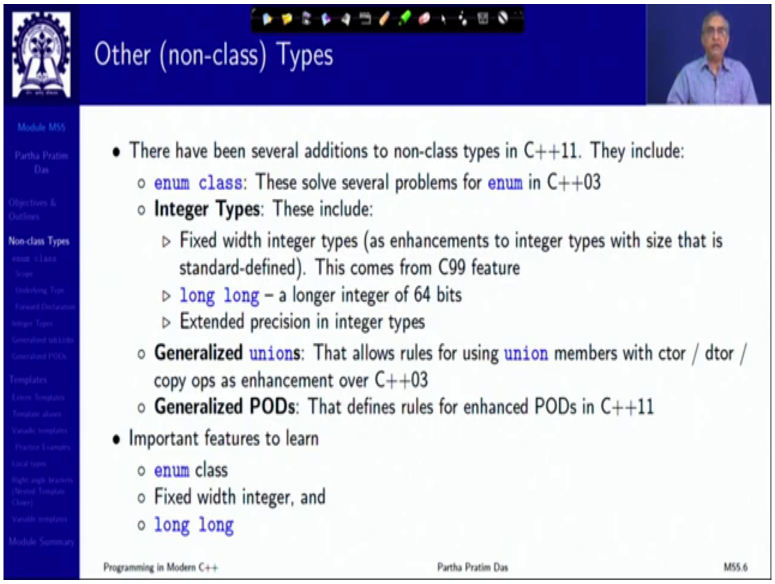
So, here is the outline which as always will be available on the left panel.

(Refer Slide Time: 01:38)



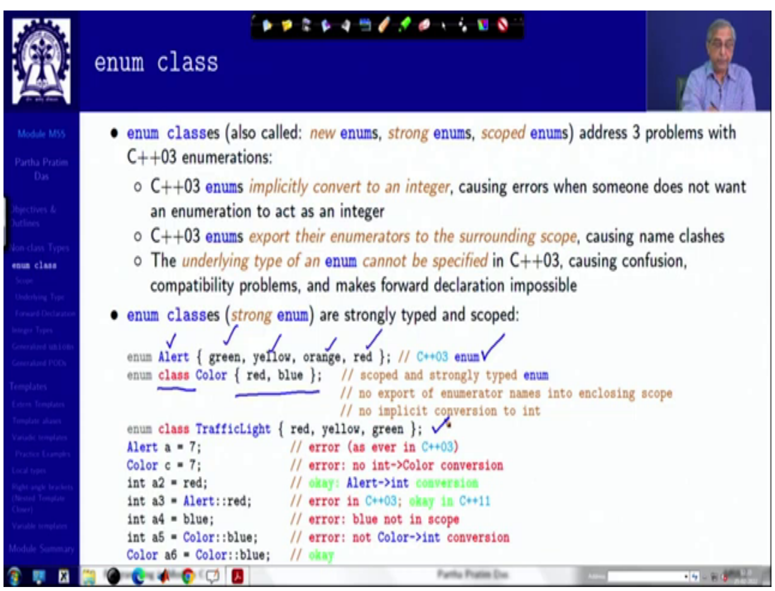
So, first, we will discuss about non-types or non-class types so to say, I mean, other types or non-class types.

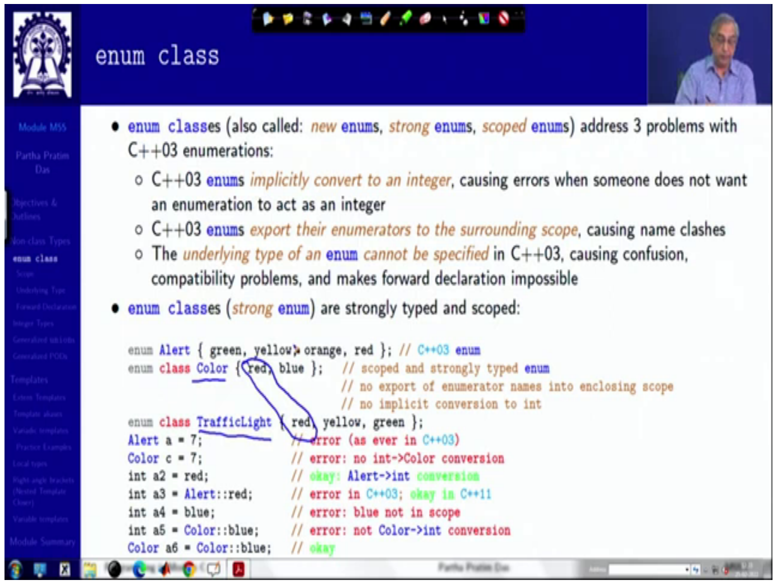
(Refer Slide Time: 01:52)

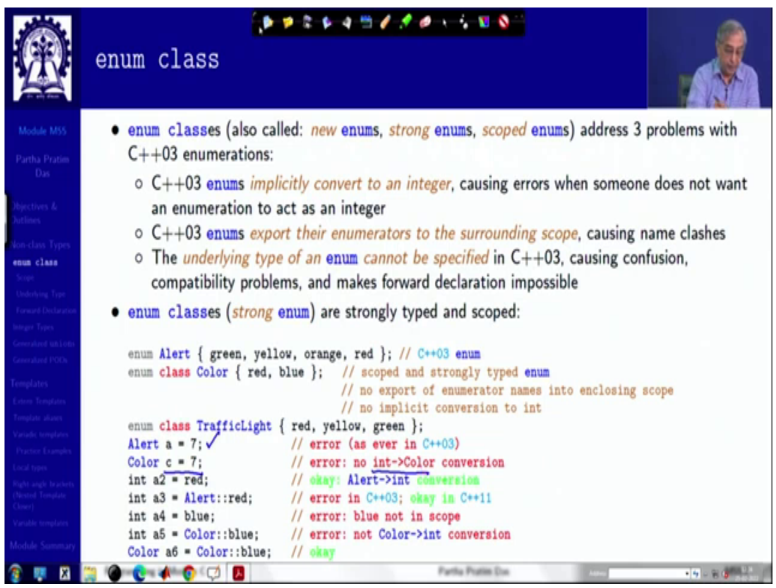


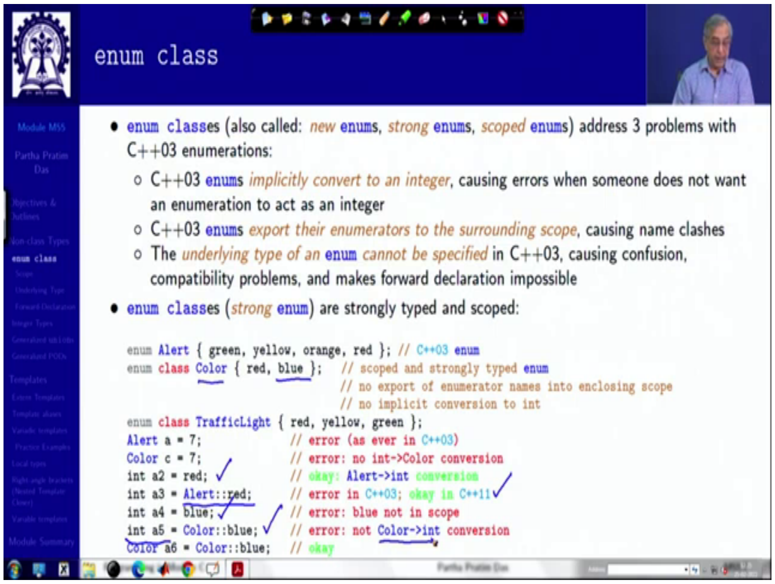
So, there are quite a few of them. So, the additions, the primary addition is enum class which solves three significant problems that enum had in C++03. Then we have integer types to discuss particularly the fixed width integer. Then we will say how unions have been generalized in C++11 and also the plain old data types have been somewhat generalized in C++11. And in particular, in between the sea of features that we will introduce here particularly focus on learning enum class fixed width integer and long long, very well.

(Refer Slide Time: 02:32)




enum class
Module M55
Partha Pratim Das
enum classes (also called: new enums, strong enums, scoped enums) address 3 problems with C++03 enumerations:
C++03 enums implicitly convert to an integer, causing errors when someone does not want an enumeration to act as an integer
C++03 enums export their enumerators to the surrounding scope, causing name clashes
The underlying type of an enum cannot be specified in C++03, causing confusion, compatibility problems, and makes forward declaration impossible
enum classes (strong enum) are strongly typed and scoped:
enum Alert { green, yellow, orange, red }; // C++03 enum
enum class Color { red, blue }; // scoped and strongly typed enum
// no export of enumerator names into enclosing scope
// no implicit conversion to int
enum class TrafficLight { red, yellow, green };
Alert a = 7; // error (as ever in C++03)
Color c = 7; // error: no int->Color conversion
int a2 = red; // okay: Alert->int conversion
int a3 = Alert::red; // error in C++03; okay in C++11
int a4 = blue; // error: blue not in scope
int a5 = Color::blue; // error: not Color->int conversion
Color a6 = Color::blue; // okay


enum class
Module M55
Partha Pratim Das
enum classes (also called: new enums, strong enums, scoped enums) address 3 problems with C++03 enumerations:
C++03 enums implicitly convert to an integer, causing errors when someone does not want an enumeration to act as an integer
C++03 enums export their enumerators to the surrounding scope, causing name clashes
The underlying type of an enum cannot be specified in C++03, causing confusion, compatibility problems, and makes forward declaration impossible
enum classes (strong enum) are strongly typed and scoped:
enum Alert { green, yellow, orange, red }; // C++03 enum
enum class Color { red, blue }; // scoped and strongly typed enum
// no export of enumerator names into enclosing scope
// no implicit conversion to int
enum class TrafficLight { red, yellow, green };
Alert a = 7; // error (as ever in C++03)
Color c = 7; // error: no int->Color conversion
int a2 = red; // okay: Alert->int conversion
int a3 = Alert::red; // error in C++03; okay in C++11
int a4 = blue; // error: blue not in scope
int a5 = Color::blue; // error: not Color->int conversion
Color a6 = Color::blue; // okay



So, first, talk about enum class. enum is available in C++03. Actually, it was available in C as well. So, by enum we can define a set of tags which in C and C++03 take equivalent integer values or we can provide specific tag values as well. Now, the problem is that it deals, I mean, it encounters is that it is implicitly convertible to integer even in C++03. So, even when you do not want such conversion to happen, you cannot, you have no way to stop them.

Also, the enum in C++03 export the enumerators to the surrounding scope that is it kind of makes them global. So if you have an enumerated tag having certain name, then you cannot have another enum or another kind of global variable which has that name. So that is a shortcoming. And finally there is no underlying type specifiable in C++03 enum. It is always integer. It is always implemented in terms of integer.

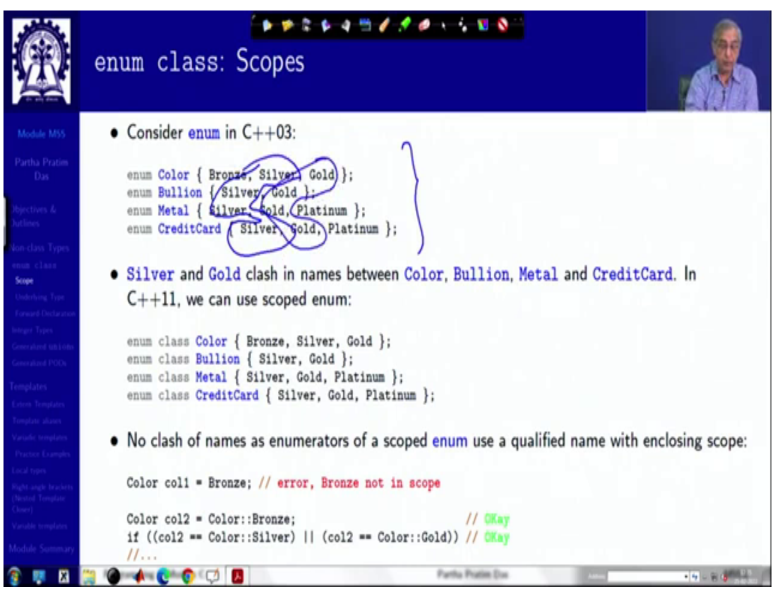
So, in C++11 this is enhanced to, enums are enhanced to enum classes. The good old enum is also available, but we will prefer to use enum classes, which is also called a new enum, strong enum, scoped enum, and so on. And they are strongly typed and scoped. So, let us start with examples. So, here is a C++03 enum. As you can see, I am defining an enum having different enumerator, four different enumerators. And I am defining a enum class in C++11.

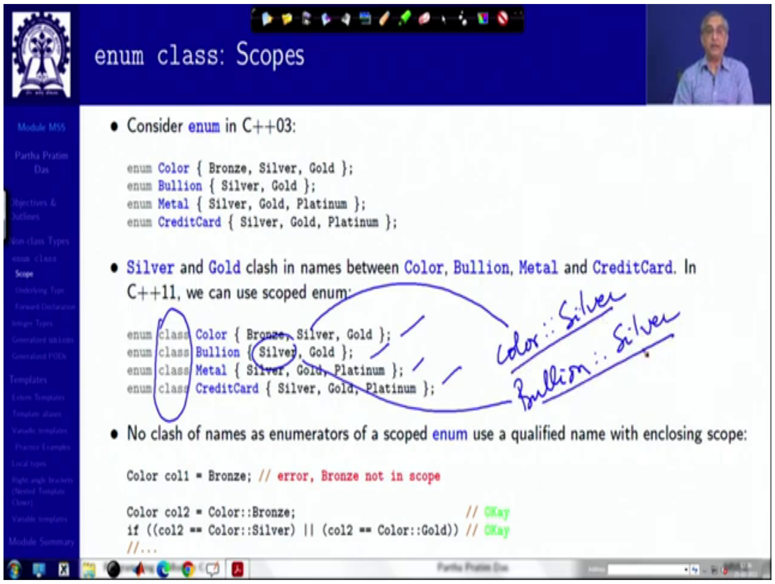
So, what is new is after the keyword enum and before the name of the enumeration you write this keyword class and rest of it is written in the same way. Now, the moment you write it as a class, it kind of imposes a namespace on to the set of enumerated values. So there is another enum class here and you can see that if you see the enum class color and enum class traffic light, you will see that I am using common enumeration of red which in C++03 would not have been possible.

Now, let us see what happens if you do try to define a, declare a with initialization 7. This is error in C++03 because you have only four enumerations. In color if you want to do the same thing you will also have an error, because if you want to do this first of all the first error that you get is because 7 is an integer and color is now a defined class, an enum class, so there is no conversion available between them. Now, old ones like assigning red to a2 is possible because a lot can be converted to int as you know in C++03.

Now, if you take Alert::red specifically then it is an error in C++03 because in C++03 the names were global. Now, we are writing it in a scoped manner. So, this is okay only in C++11. If I try to set blue, it is obviously an error because it is not in the scope. But if I say Color::blue, then it is fine. But I cannot define a variable of type int and initialized with Color::blue, because Color::blue has a different type than int and there is no conversion. I can only have a variable of type color to be able to do this. I can only have this. So, that is the basic thing.

(Refer Slide Time: 06:49)

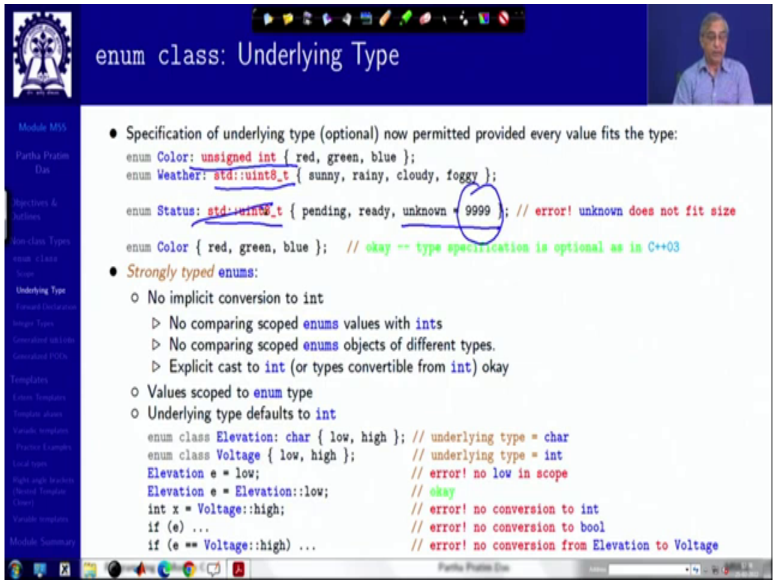


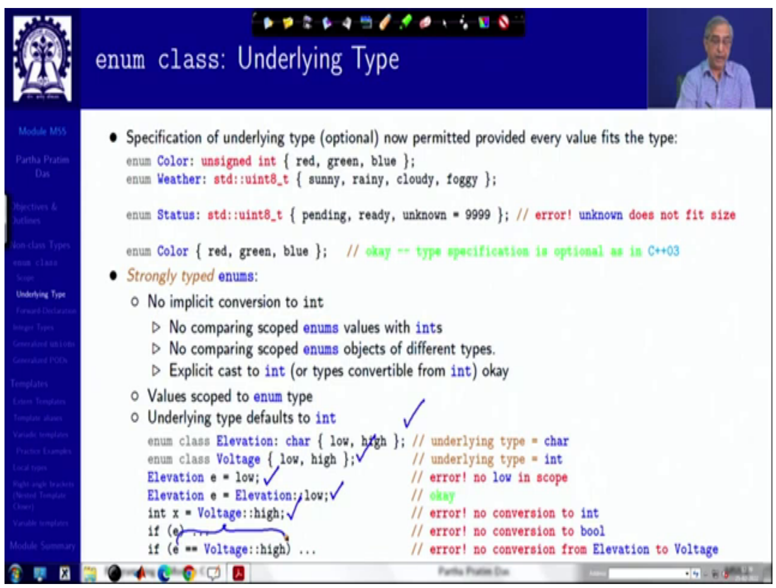


So, let us look at these three aspects one by one. Let us talk about scope. So, here I have defined four enums in C++03 and they will have severe problems because Silver is a, is common between them, Gold is common between them and so on. But if you look at semantically then these are not synthetic examples. So, in terms of color, you will certainly consider bronze, silver, gold as color in terms of bullion that is the coins, currency coins. You will certainly have silver and gold in terms of metal. You will talk about silver, gold, platinum as metal.

In terms of credit card you will talk about silver, gold, platinum. In terms of memberships, you will talk about that. So, it is the same kind of tag name is applicable in multiple contexts, which is not expressible in C++03. In C++11, by using this enum class you can easily do that. These names are all now scoped to within the particular enum class. So, Silver here is actually Color::Silver, and Silver here is Bullion::Silver. So the names do not clash anymore. So with this scoping, we can enhance the expressive ability of enums to a significant extent.

(Refer Slide Time: 08:24)



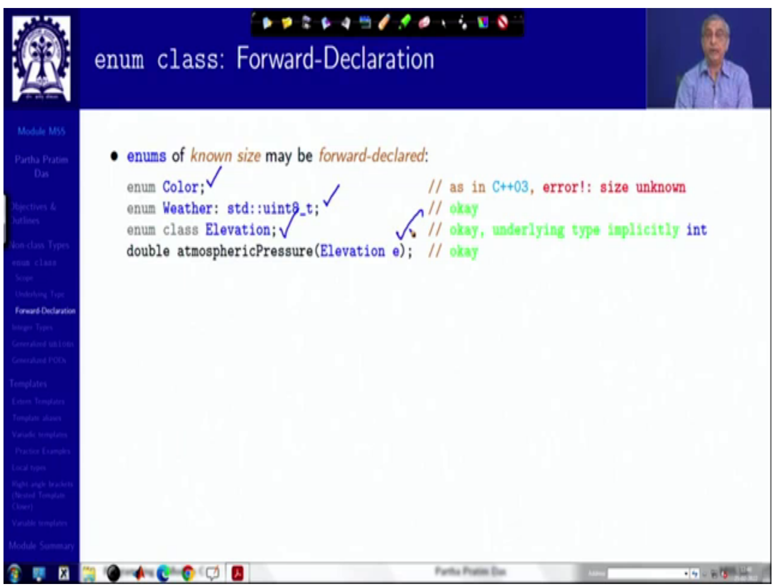


If we look at the underlying type, then we know that in C++03, the underlying type was always int. It is enum is converted to int, but now you can specify what is the underlying type. So what you do is the syntax for that is after the enum name and before the list starts, you put a colon and then write the underlying type name. So, this is kind of similar to the way you write derived classes you can say so this is the underlying type. And when you do that, the enum enumerator definitions follow that underline.

For example, if I, for status if I have written std::uint8_t which means it is an unsigned integer of 8 bits is that I will use as my underlying type then this becomes an error. But if you had not written this, if you had not specified this, then the underlying type by default is int and therefore 9999 is a valid value in int and this will be accepted. So this gives kind of strong typing to enums and you cannot compare scoped enum values with ints, you cannot compare scope enum objects of different types, you cannot, I mean, it is not, int is not, cannot be explicitly cast into enum or otherwise, you have to, implicitly cast in that, you have to do that explicitly and so on.

So, here are a couple of examples of, here is an Elevation and a Voltage, both of them could be low and high. And for example, if I try to define e with low I will get an error because it is not scoped. I have to write it whether, I have to specifically say that it is Elevation::low, similarly Voltage::high. I cannot compare e with Voltage::high because e is of type Elevation and Voltage is of, Voltage::high is of type Voltage. So, you can see that basic difference that is kind of the notion that we have, strong typing in terms of classes come in here with the convenience of having the tags of the enums.

(Refer Slide Time: 10:51)

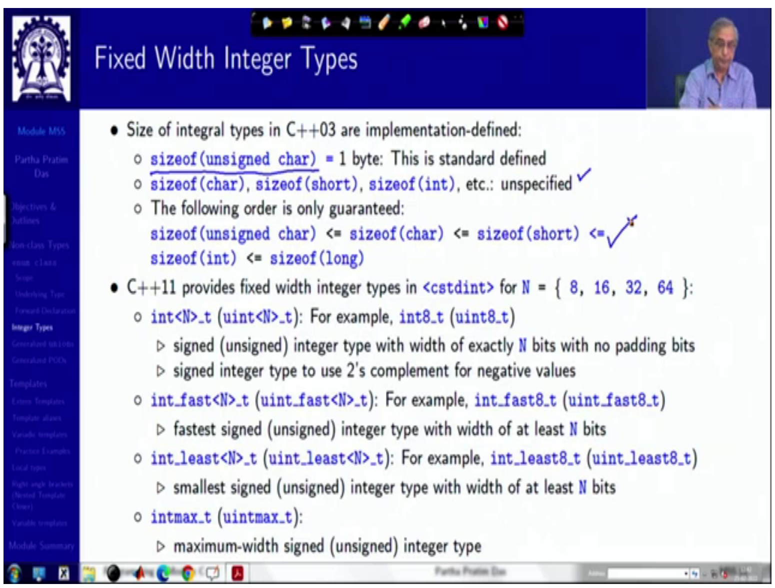


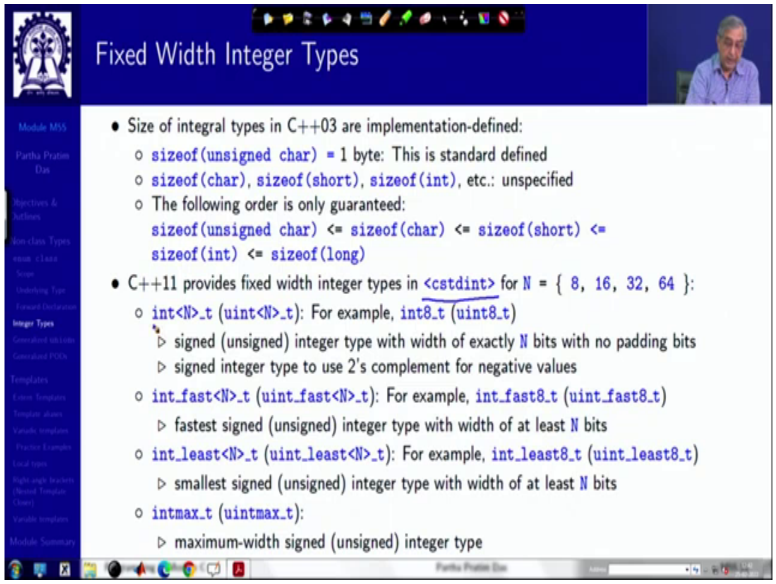
Finally, with these, another big advantage that you get, you can now forward-declared enums. So, if you try to just say that, well, I want to say that I have an enumeration, but I am not, right now, I am not sure about the enumerators and I do not want to list them out, you cannot do that in C++03. So, if you just write enum this then it will give you an error saying that the size is unknown, because unless you give the list the compiler does not know the size.

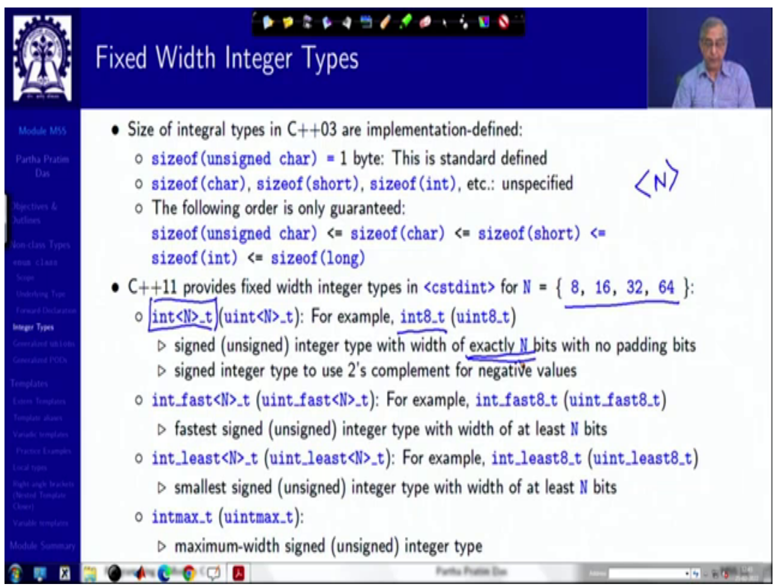
But if you write it like this saying that the underlying type is such and such, then the compiler immediately understands that you are talking about, not talking about enum in C++03, but you are talking about an enum class even though you have not written the keyword class, but from the syntax of the underlying type the compiler figures out that you are talking about an enum class and it will allow you to specify this without the enumerators being specified.

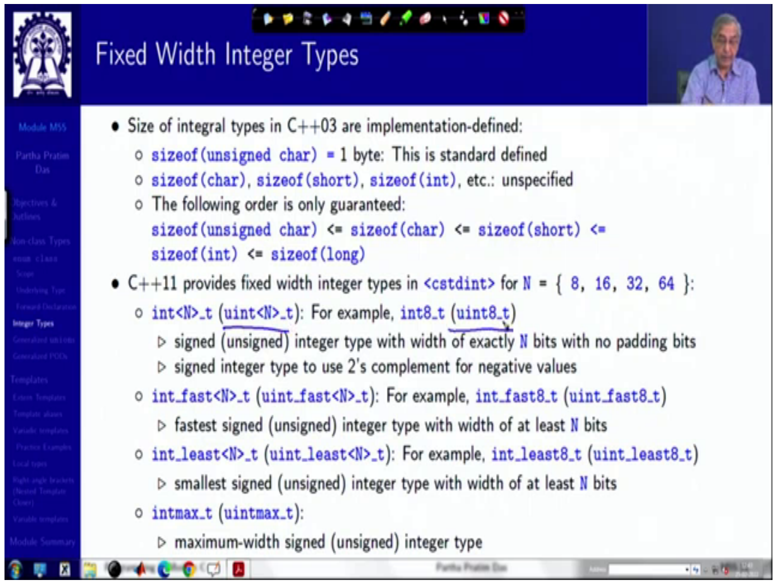
Similar thing can be done here and then it can be used in terms of passing to, passing as function parameters and so on. Of course, before the actual execution, you will have to specify what the enumerations are and the, only then the code can be generated. So, this gives enum class really a much stronger and much well tight behavior in C++03, C++11 compared to what you had in C++03 and you must start using them extensively.

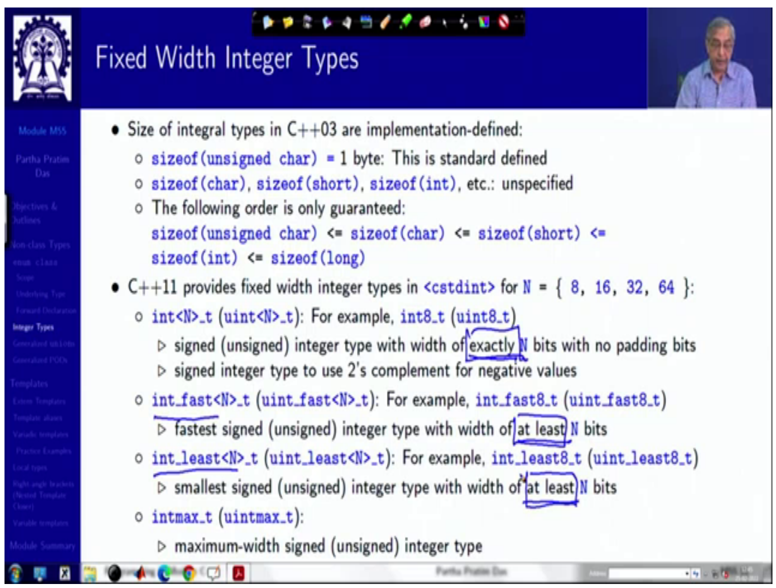
(Refer Slide Time: 12:30)




Fixed Width Integer Types
Module M55
Partha Pratim Das
Size of integral types in C++03 are implementation-defined:
sizeof(unsigned char) = 1 byte: This is standard defined
sizeof(char), sizeof(short), sizeof(int), etc.: unspecified
The following order is only guaranteed:
sizeof(unsigned char) <= sizeof(char) <= sizeof(short) <=
sizeof(int) <= sizeof(long)
C++11 provides fixed width integer types in <cstdint> for N = { 8, 16, 32, 64 }:
int<N>_t (uint<N>_t): For example, int8_t (uint8_t)
signed (unsigned) integer type with width of exactly N bits with no padding bits
signed integer type to use 2's complement for negative values
int_fast<N>_t (uint_fast<N>_t): For example, int_fast8_t (uint_fast8_t)
fastest signed (unsigned) integer type with width of at least N bits
int_least<N>_t (uint_least<N>_t): For example, int_least8_t (uint_least8_t)
smallest signed (unsigned) integer type with width of at least N bits
intmax_t (uintmax_t):
maximum-width signed (unsigned) integer type


Fixed Width Integer Types
Module M55
Partha Pratim Das
Size of integral types in C++03 are implementation-defined:
sizeof(unsigned char) = 1 byte: This is standard defined
sizeof(char), sizeof(short), sizeof(int), etc.: unspecified
<N>
The following order is only guaranteed:
sizeof(unsigned char) <= sizeof(char) <= sizeof(short) <=
sizeof(int) <= sizeof(long)
C++11 provides fixed width integer types in <cstdint> for N = { 8, 16, 32, 64 }:
int<N>_t (uint<N>_t): For example, int8_t (uint8_t)
signed (unsigned) integer type with width of exactly N bits with no padding bits
signed integer type to use 2's complement for negative values
int_fast<N>_t (uint_fast<N>_t): For example, int_fast8_t (uint_fast8_t)
fastest signed (unsigned) integer type with width of at least N bits
int_least<N>_t (uint_least<N>_t): For example, int_least8_t (uint_least8_t)
smallest signed (unsigned) integer type with width of at least N bits
intmax_t (uintmax_t):
maximum-width signed (unsigned) integer type





The next feature is about the types, integer types. Now, as you know that we have different integral types in C++03, there is unsigned char, there is char, there is short, there is int, there is long and so on so forth. Now, if I asked you what is the size of say int or what is the size of char or what is the size of long, you will say that it depends on the implementation. The standard has not defined what the size will be.

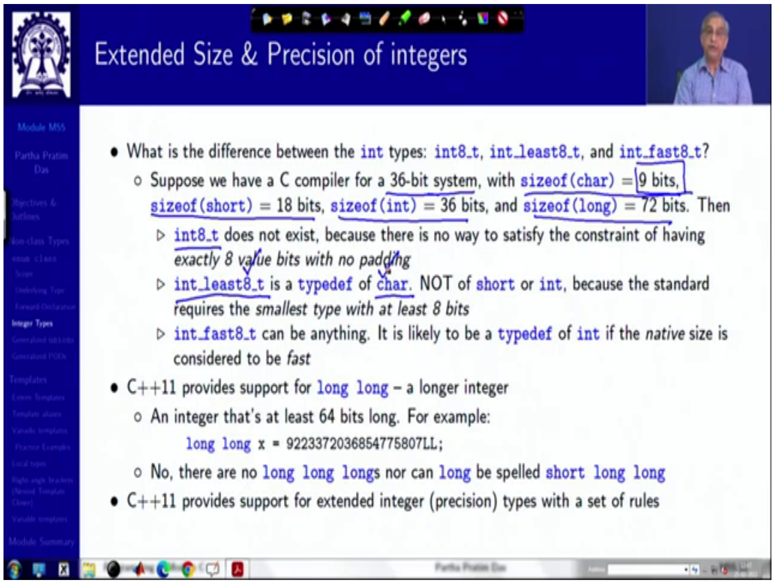
So, it depends on the compiler and the actual implementation which defines the size of every type in C++03 with the exception of unsigned char which has been defined to be 1 byte that is 8 bits in the standard, otherwise this size is an unspecified. Only thing that the standard specify is there is an ordering between their size that is unsigned char cannot be larger than the size of char that cannot be larger than the size of short and so on so forth.

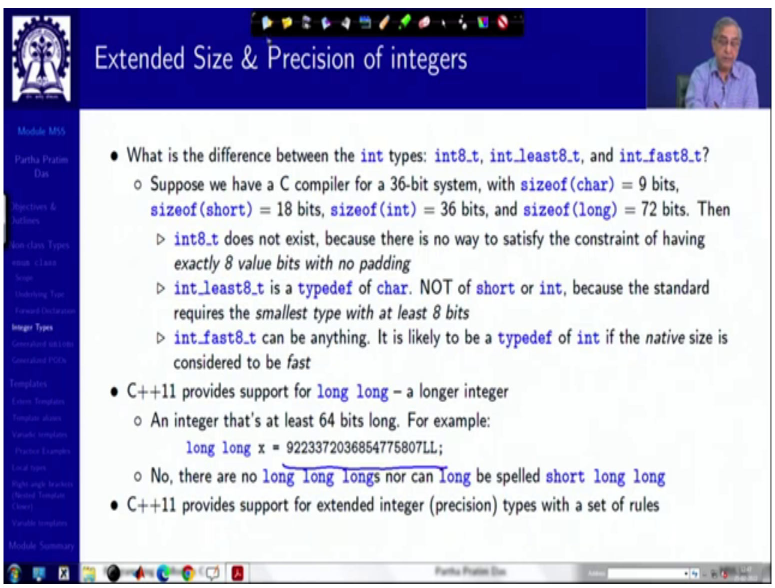
In contrast, in C++11 you have a number of types which are either directly implemented or given us typedefs, that is type alias in this particular component called cstdint. So, you can see that it is actually being borrowed from the C standard library because this also is a part of the C99 feature. So, you can, you have things like say int<N>_t, let us just look at one and the rest will become. So, here by this what I mean, I mean one of these numbers. That is it could be 8, it could be 16, it could be 32, it could be 64. So, for example, if N is 8, then the name of that type in the cstdint is int8_t. So, this tells you that this is an integer type which is signed and must be of 8 bits specifically so fixed one.

So, the implementer has to give you kind of, I mean, a way to do that. So it has to be exactly or this type will not be available. If it is, then if it is not if available then you will get a compilation error. So, you know that you cannot do this on the particular implementation you are using. But if it is available, if it compiles, then you are guaranteed that you will have an 8 bit signed integer. Similar thing you have for unsigned integer as well and the signed integer represented in n bits with no padding bits and sign it is represented also as 2’s complement for negative values.

You have two other such one is called int_fast. In fast is implementer’s judgment on what is the fastest signed type of size N that is available. So, the implementer will type def it appropriately so, to a particular type. It could be, for example, int_fast8_t could be typedef to char, because it is one but and could be the fastest. Similarly, there is an int_least it say what is the smallest integer type. I mean, it maps to it typedefs 2 the smallest integer type having at least N bits. Please note that between int<N> and int_fast<N> or int_least<N> the difference is int<N> needs exactly N number of bits, whereas these two need at least N number of bits. So, even bigger size will also work. So, this is the, these are the fixed type.

(Refer Slide Time: 16:39)



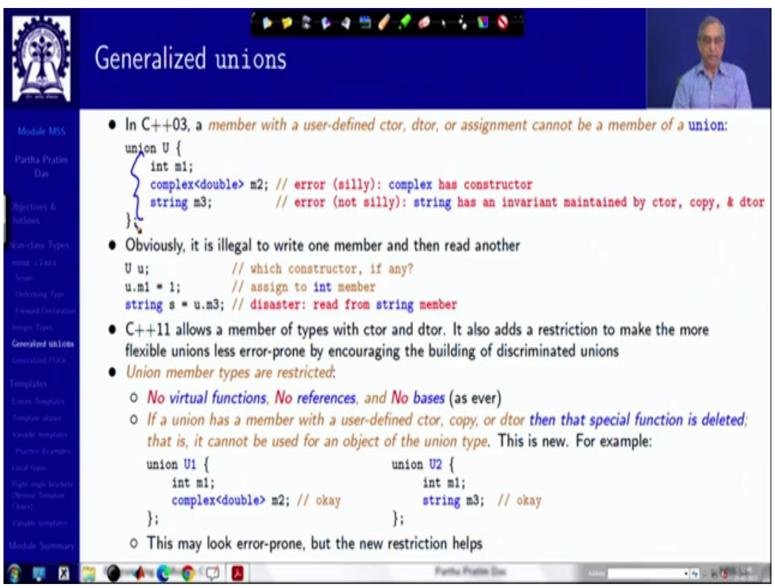


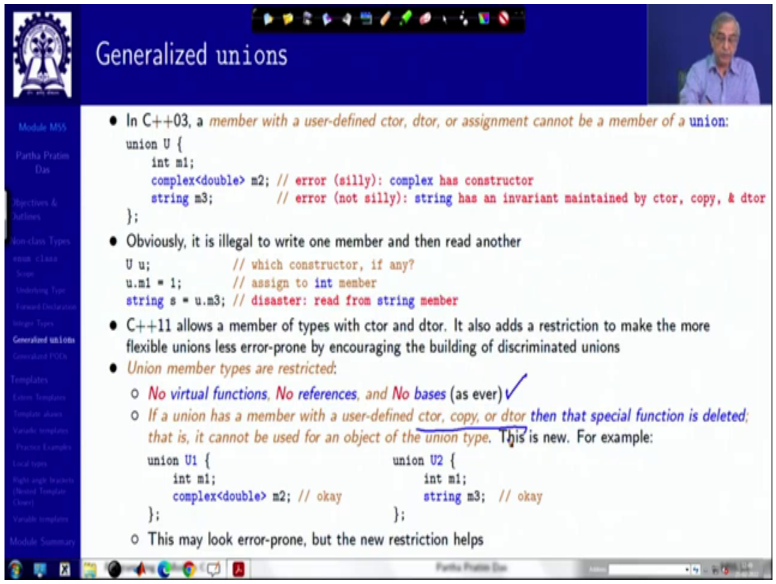
So, what is the difference between them, what is the difference between int_t, int_least_t, int_fast8_t and so on. So, suppose you have a compiler some hypothetical machine which is a 36 bit system, do not get shocked with that, because your PCs and servers are not the only computing systems that exist in the world. There are several other embedded systems and so on which has arbitrary number of bits available for them. So, suppose you have a 36 bit system where the char is 9 bits, short is 18 bits, double of that, and int is 36 bits and long is 72 bits.

Now, in this if you try to see what is int8, it is you will see that it does not exist, because it has to give you an integer type with exactly 8 bits which does not exist. So, with this type the code will not compile. So which tells you clearly that 8 bits are not, 8 bit integers are not available, whereas int_least8_t will be a typedef of char because it is a smallest signed integer having at least 8 bits. So, char has 9 bit, so it is, it satisfies the at least part and between 9, 18, 36, 72 it is the smallest. So the list int_least8_t will typedef to char in the system. Whereas, int_fast8_t could be anything depends on what the implementer considers, it the most appropriate type for the speed is concerned.

You also have in C++11 a new type called a long long for integer which is meant for 64 bit integer. So here is an example of a long long variable being declared with a literal, but mind you that you cannot write long long long or you cannot write short long long those things do not make any sense besides that several extended precision, but the biggest take back for you from this should be that fixed bit integer types are possible now in C++11. And in terms of 64 bit computation, you have the long long as new integral time for 64 bits.

(Refer Slide Time: 19:13)

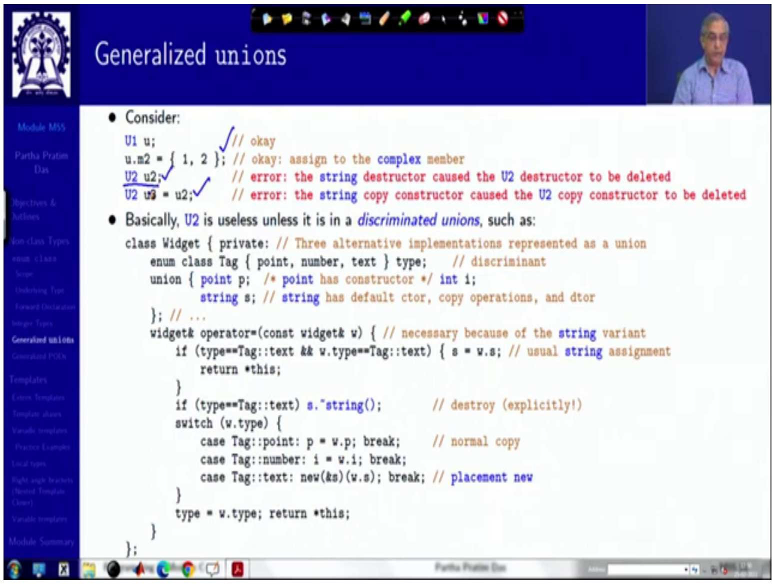


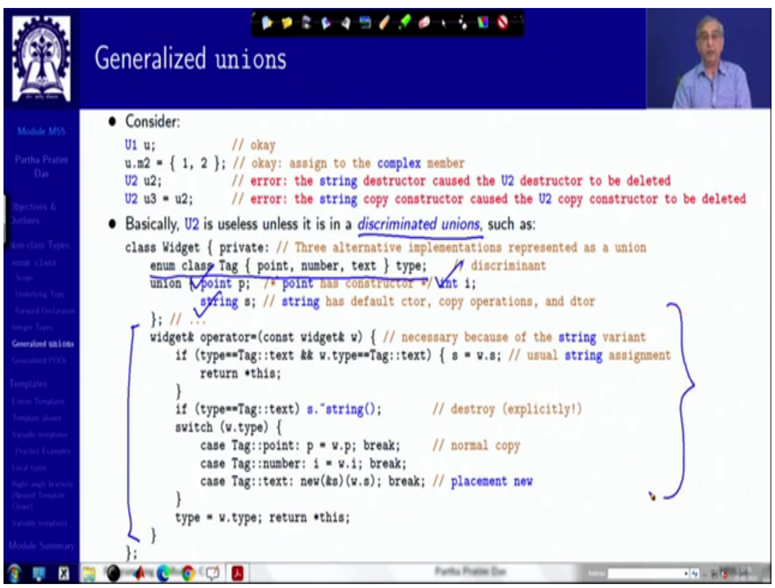


Moving on, union as you know is a collection of mixed types like in structure, but the fact that only one component can be available in a union at any point of time. So this naturally creates a problem. Suppose if I define this as a union, I do not know which of these components is being used. Therefore, to construct the object of this union, which constructor should I use, constructor of int, maybe which is the pod default constructor or the constructor of complex or the constructor of string. So, therefore, C++03 does not allow any component of a union to be a user defined type having constructor destructor assignment and so on. This is simply not allowed.

In C++11, this rule has been somewhat relaxed. So, it says that well you can use such member variables in the union provided these are rules are followed that is the member you are including does not have any virtual function or a reference or base class. And if it defines any of the constructor copy or destructor, then the corresponding function in the union gets automatically deleted, you cannot, because you cannot then delete. I mean, if you have defined a constructor, then you cannot have a constructor for the union. So, this kind of may look a little weird to you.

(Refer Slide Time: 20:59)

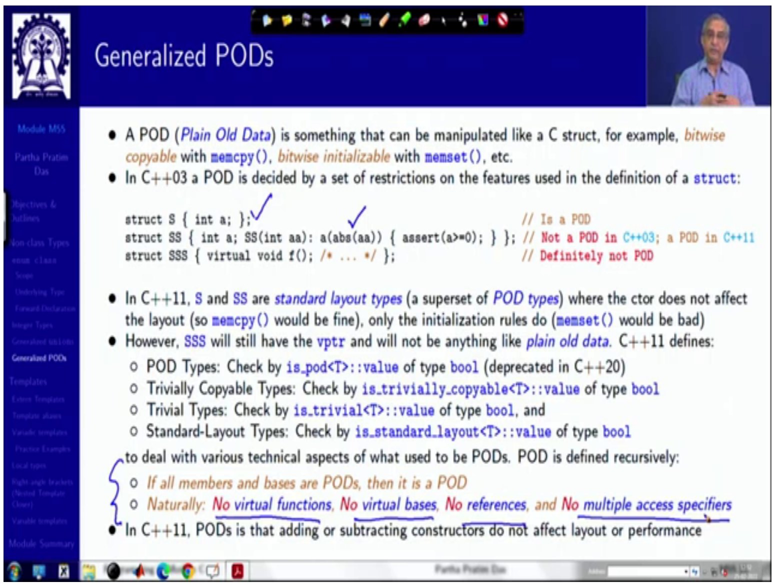




But if we see an example, you can easily make out. So, this U1 has int and complex. So, it is okay. In U2 you have int string. So, if you try to, this is an error because the moment you have U2, then the string destructor will cause that U2's destructor is deleted. So, there is no destructor so U2 cannot be defined. So, you will wonder as to why then have this. The basic point is if you are making a discriminated union, that is you are making an union where you use a enum class tag, we have seen this style before in C++03 also, and say three types point, string and int, three types of variables, then you will be able to still do that.

Only difference is you will have to now define your operators like say if you have to give assignment to this because assignment per se will not be given it is because string has an assignment operator, so you have to provide an assignment operator now, no default is to be provided, where you can specifically decide based on the components and decide on what needs to be done. Please go through this code. This should be pretty straightforward to understand. And you will know why this, how this generalized union work.

(Refer Slide Time: 22:28)

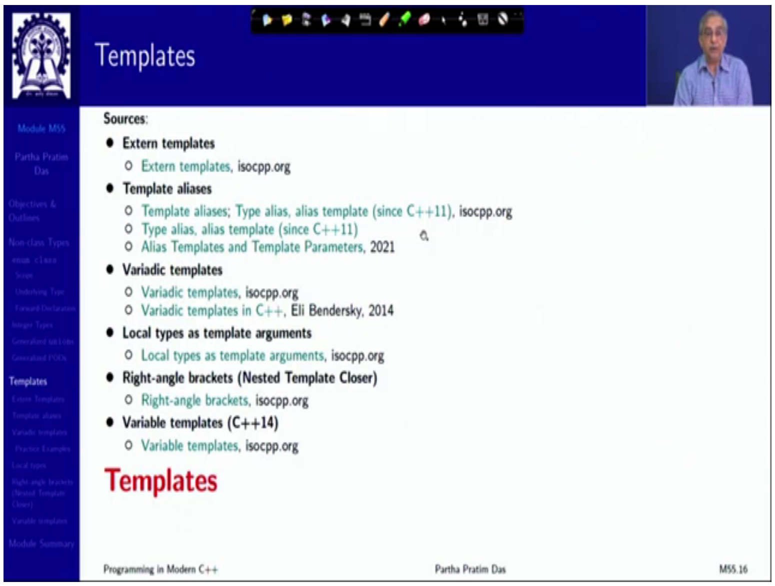


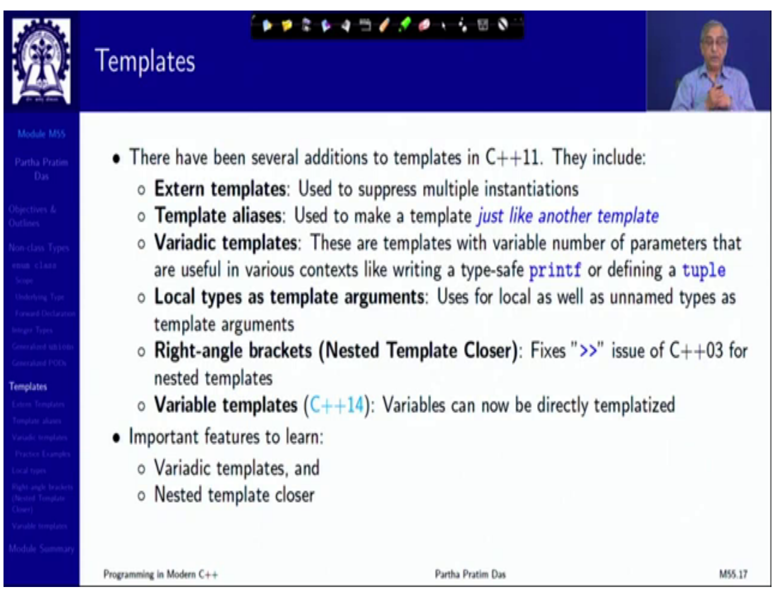
There is a some generalization of the plain old data types also which are bitwise copyable by mem copy or initially visible by memset. Like if I had a struct without any constructor. Now, that is a plain old data type because there is a bit pattern. In terms of C++03 also, these kinds of classes or structs are also considered to be a plain old data type, because it, what it is actually doing is basically assigning a pattern to the data member.

So, again, the plain old data type extends a lot with two basic rules. One is any structure is a plain old data type provided all members and bases are plain old data type, everything inside is plain old data type. So that I can bid copy everything, bit initialize everything. And naturally there is no virtual function, no virtual base, no reference, and no multiple access specifier. All access specification has to be the same.

So, the crux of the thing is that it becomes in C++11 a data type becomes a plain old data type if it is, it does not make a difference whether I have a constructor or I do not. In terms of the layout or in terms of the performance, if it does not make a difference, then it becomes a plain old data type. I am not going into the details of that just the notions are important, because in terms of actual programming I would not advise that you use these as plain old data types in general. These are meant more for the experienced library programmer and you should restrict but you should know that such kind of codes are expected.

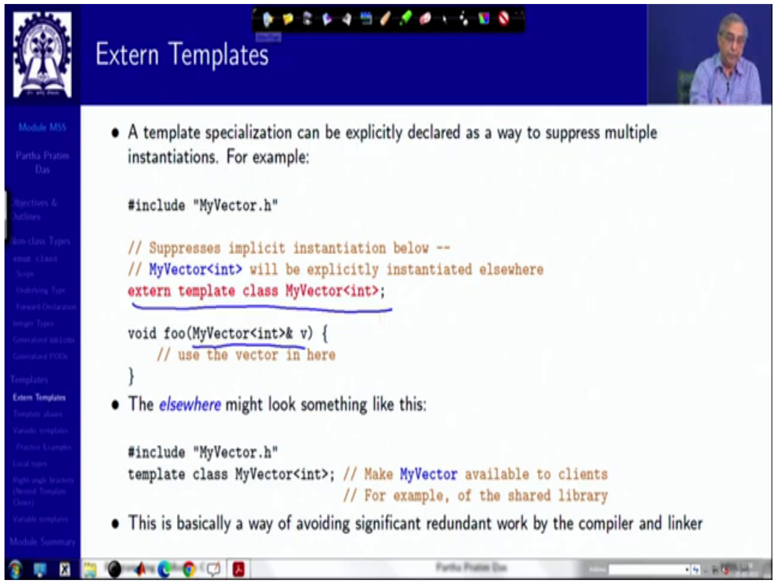
(Refer Slide Time: 24:27)





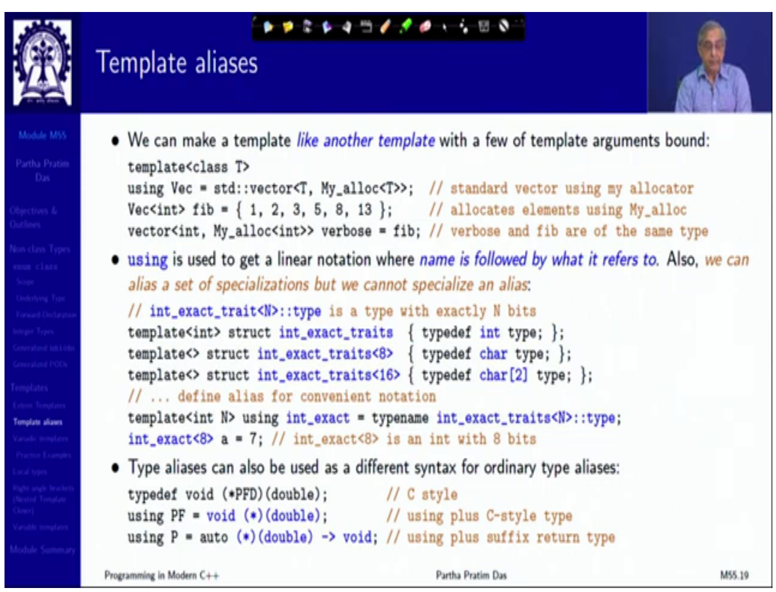
Now, let us move on to templates. So, there are varied extensions to templates. First, there is an external template, template alias, variadic template is the most important, local type arguments, right angle brackets and variable templates. Of these, these two are the most important ones.

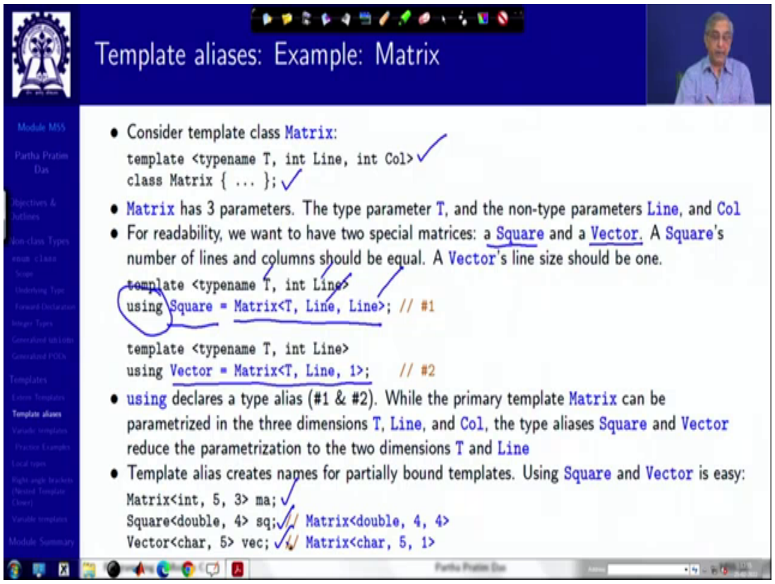
(Refer Slide Time: 24:48)



External template is simply, if you have a template after you instantiate it at multiple places, then naturally it is expanded at every instantiation space, whereas those extent instantiations for the same template type parameter will be identical. So, it allows, C++11 allows that you can instantiate, use the instantiation of a template, but tell the compiler that do not instantiate it here. It will be instantiated somewhere else. So, in some other file, it is instantiated and use. So, it is basically a performance issue for the compiler, not a great thing for really the programmer semantics.

(Refer Slide Time: 25:30)



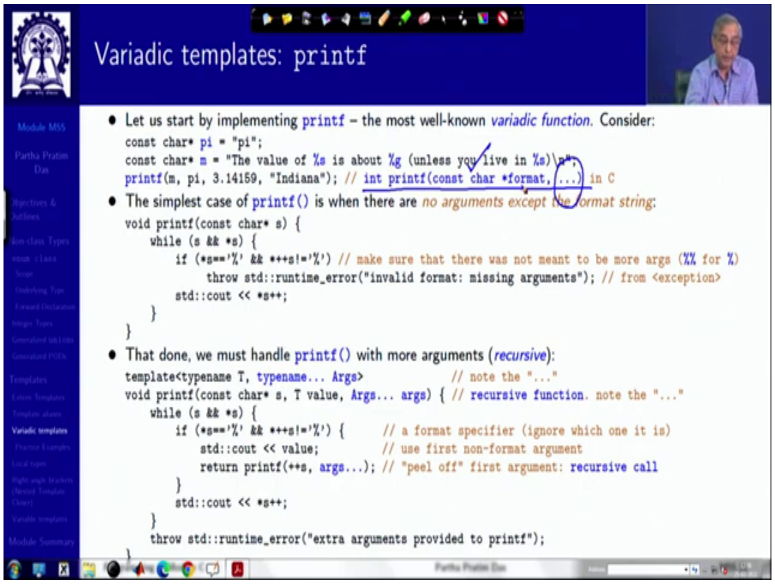


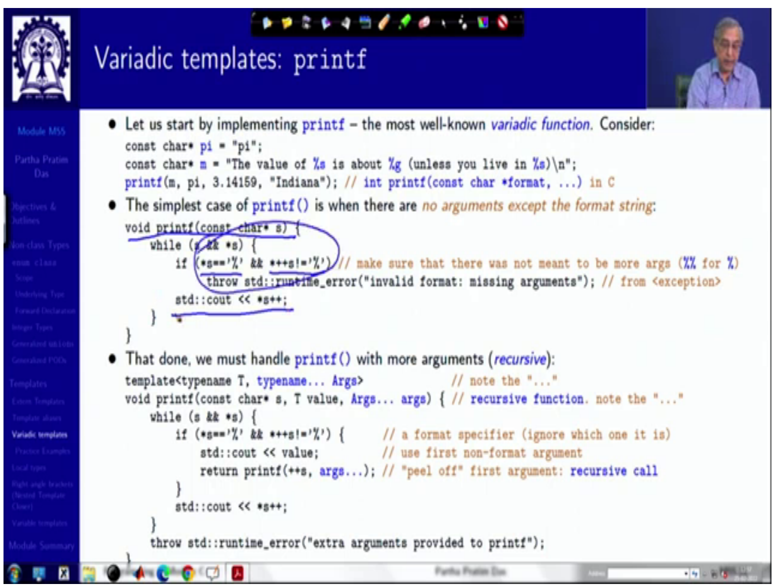
Template alias is nothing but using a different name for a template with one or more of its parameter types already specified. There is a detailed discussion here, but what I want to really make you to note is consider this example. Suppose you have defined a class Matrix as a template, naturally you will have three template parameters, one is the type of the element, and there are two non-template, non-type parameters like int, Line, number of lines and number of columns. But specifically, you also want to deal with squares and vectors.

What will happen in the square, the line and column must be same. What will happen in a vector, in the vector the number of columns should be 1. It is a liner one. So, this is what you can do using the template alias. For example, take that Matrix template. You are making both of them same. And defining a new template with T and Line and giving it in name Square, this keyword using, use of this keyword using here allows you to do that.

So, Square now becomes an alias for this Matrix template where you can just provide the type and the Line and it will use the Matrix template with the line and the column would be same as a Line. Similar thing you can do for a Vector as well. So, that makes naturally the expression a lot more readable, expressible and semantically clear.

(Refer Slide Time: 27:17)





Variadic templates: printf

- Let us start by implementing `printf` – the most well-known *variadic function*. Consider:

```
const char* pi = "pi";
const char* m = "The value of %s is about %g (unless you live in %s)\n";
printf(m, pi, 3.14159, "Indiana"); // int printf(const char *format, ...) in C
```

- The simplest case of `printf()` is when there are *no arguments except the format string*:

```
void printf(const char* s) {
    while (s && *s) {
        if (*s=='%' && *++s!='%') // make sure that there was not meant to be more args (%% for %)
            throw std::runtime_error("invalid format: missing arguments"); // from <exception>
        std::cout << *s++;
    }
}
```

- That done, we must handle `printf()` with more arguments (*recursive*):

```
template<typename T, typename... Args>             // note the "..."
void printf(const char* s, T value, Args... args) { // recursive function. note the "..."
    while (s && *s) {
        if (*s=='%' && *++s!='%') {      // a format specifier (ignore which one it is)
            std::cout << value;          // use first non-format argument
            return printf(++s, args...); // "peel off" first argument: recursive call
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}
```


Variadic templates: printf

- Let us start by implementing `printf` – the most well-known *variadic function*. Consider:

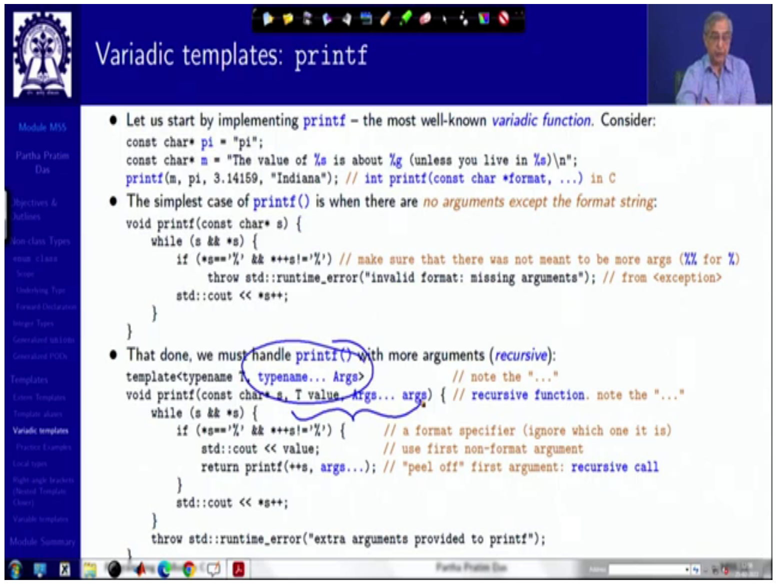
```
const char* pi = "pi";
const char* m = "The value of %s is about %g (unless you live in %s)\n";
printf(m, pi, 3.14159, "Indiana"); // int printf(const char *format, ...) in C
```

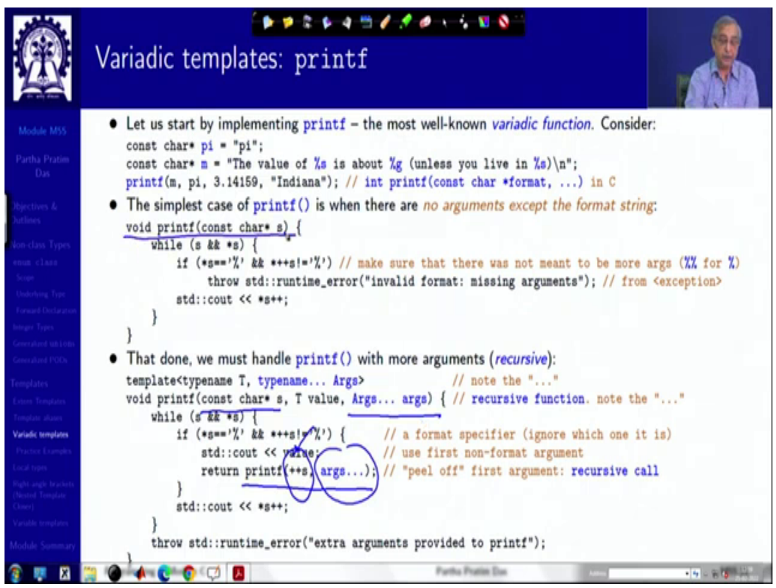
- The simplest case of `printf()` is when there are *no arguments except the format string*:

```
void printf(const char* s) {
    while (s && *s) {
        if (*s=='%' && *++s!='%') // make sure that there was not meant to be more args (%% for %)
            throw std::runtime_error("invalid format: missing arguments"); // from <exception>
        std::cout << *s++;
    }
}
```

- That done, we must handle `printf()` with more arguments (*recursive*):

```
template<typename T, typename... Args>             // note the "..."
void printf(const char* s, T value, Args... args) { // recursive function. note the "..."
    while (s && *s) {
        if (*s=='%' && *++s!='%') {      // a format specifier (ignore which one it is)
            std::cout << value;          // use first non-format argument
            return printf(++s, args...); // "peel off" first argument: recursive call
        }
        std::cout << *s++;
    }
    throw std::runtime_error("extra arguments provided to printf");
}
```





Now, what is a significant contribution of C++11 in terms of templates is what is known as variadic template. What is a variadic template? A variadic template is one where there is variable number of type parameters type non-type parameters. So, why is it important, because we have variadic functions. The most well-known variadic function is printf, where we know that we have a one parameter, first parameter is must, which is format string, and then we just say ellipses … to mean that there could be any number of parameters.

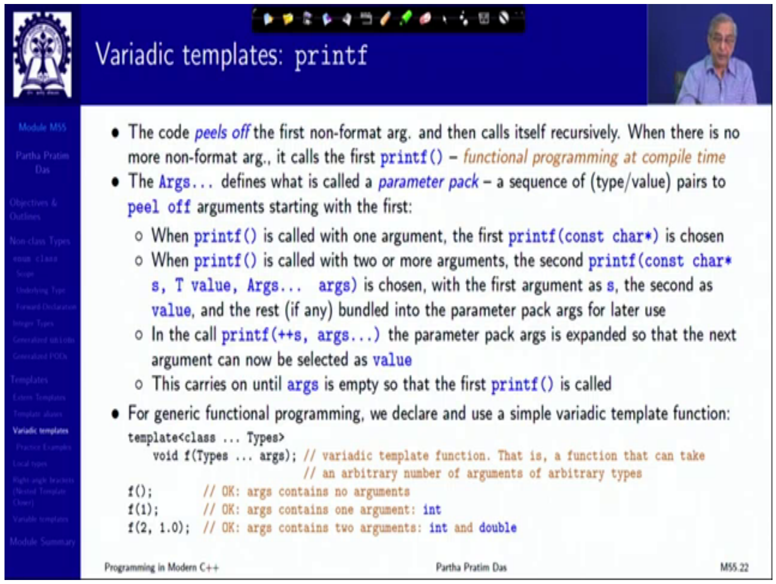
And that is real sour for the type checking, because in the code you do not get to see what the type is corresponding the format definition, like %d, %s with the actual parameter type is user's responsibility. So, printf is a really, really sour area. Using variadic templates, you can do this, get rid of this by specifying the printf in a very type safe manner. What you do first is you write a template just for the, write a template function just for the format string.

So, you have the format string s and you will have % or %%, if you have, if you do not have any of that then naturally, then it is not valid. Otherwise, you just print whatever text is given. What is there in the format string? There is %d, %s like that, there are certain character strings or %%. So here you take out those and anything in the character you print. So, that is how you get the format.

Now, you do what is the variadic template here. Note carefully that I said that there has to be a first type T and then there are … variadic number of template parameters. Similarly, in the function I have given that first parameter is the string, the format string, second is a value which is T of type T and then the rest of the parameters. So, what I do is I, taking this, so here I have, say, three parameters. Here what I am doing is I am taking out one the first of them and remaining two I keep as a separate pack. So, as if there is a pack of three parameters given I take out the first one and relieve the other two in the pack itself.

So, the one that I take out, I simply do an std::cout. I do not care about what is the %d, %s because in C++ we know that how to print, how to stream is known from the type. So, I use that feature. And then I simply recur and when I recur I use the format string again as the first parameter here and rest of whatever is remaining. And if I do that, then naturally it will keep on, you will call this function itself with one less parameter, then again it will call this function itself with one less template parameter till it has only the format string left in which case it will call this particular version. So, that is the typical way the recursion works.

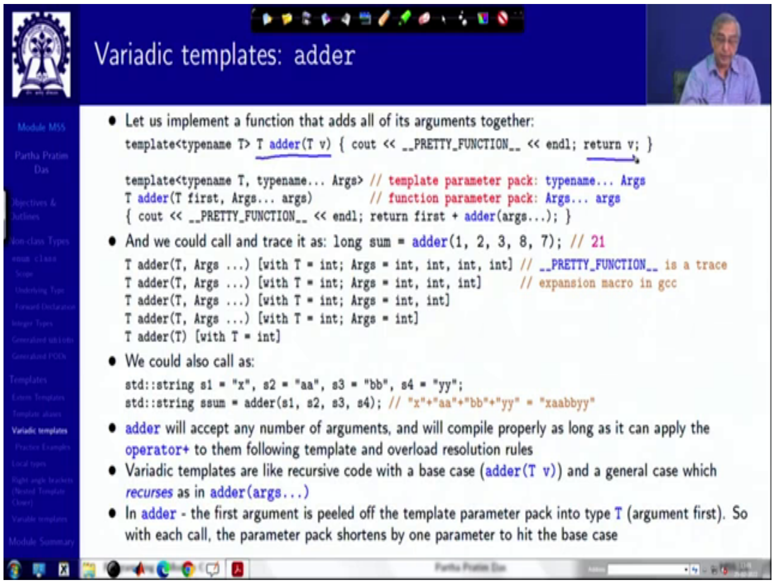
(Refer Slide Time: 31:09)

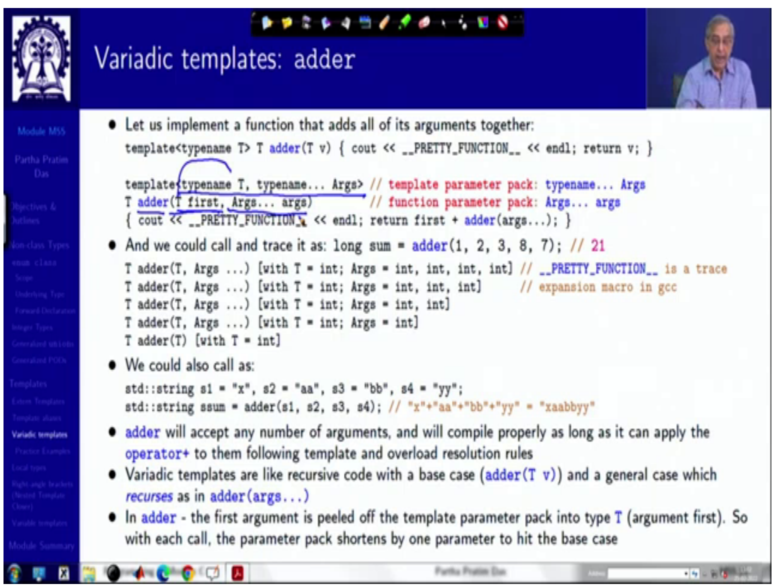


Though you may not strictly call it a recursion, this is called variadic templates. This is not strictly called a recursion, because in recursion you expect the same function to be called recursively. But here every time the function is changing, because every time the function has certain number of parameters, where a bunch of parameters are packed, and when it calls the next version, it actually has reduced one parameter from the pack. So, that is what the printf does. So, you can think of, I mean, if you think about lambdas in C++ that this basically is a kind of a functional programming at the compile time that we are doing.

So, it is, printf is just, so this is a very easy way, very short code. And if you actually look the printf code in C, it is a very, very huge one. It is a very short code which is a very type-safe way to print anything that you want to print in that way. In fact, it allows you to also print user defined types provided you have overloaded the output streaming operator appropriately.

(Refer Slide Time: 32:24)





Variadic templates: adder

- Let us implement a function that adds all of its arguments together:

```
template<typename T> T adder(T v) { cout << __PRETTY_FUNCTION__ << endl; return v; }

template<typename T, typename... Args> // template parameter pack: typename... Args
T adder(T first, Args... args)         // function parameter pack: Args... args
{ cout << __PRETTY_FUNCTION__ << endl; return first + adder(args...); }
```

- And we could call and trace it as: `long sum = adder(1, 2, 3, 8, 7); // 21`

```
T adder(T, Args ...) [with T = int; Args = int, int, int, int] // __PRETTY_FUNCTION__ is a trace
T adder(T, Args ...) [with T = int; Args = int, int, int]      // expansion macro in gcc
T adder(T, Args ...) [with T = int; Args = int, int]
T adder(T, Args ...) [with T = int; Args = int]
T adder(T) [with T = int]
```

- We could also call as:

```
std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
std::string ssum = adder(s1, s2, s3, s4); // "x"+"aa"+"bb"+"yy" = "xaabbyy"
```

- `adder` will accept any number of arguments, and will compile properly as long as it can apply the `operator+` to them following template and overload resolution rules
- Variadic templates are like recursive code with a base case (`adder(T v)`) and a general case which *recurses* as in `adder(args...)`
- In `adder` - the first argument is peeled off the template parameter pack into type `T` (argument `first`). So with each call, the parameter pack shortens by one parameter to hit the base case

Variadic templates: adder

- Let us implement a function that adds all of its arguments together:

```
template<typename T> T adder(T v) { cout << __PRETTY_FUNCTION__ << endl; return v; }

template<typename T, typename... Args> // template parameter pack: typename... Args
T adder(T first, Args... args)         // function parameter pack: Args... args
{ cout << __PRETTY_FUNCTION__ << endl; return first + adder(args...); }
```

- And we could call and trace it as: `long sum = adder(1, 2, 3, 8, 7); // 21`

```
T adder(T, Args ...) [with T = int; Args = int, int, int, int] // __PRETTY_FUNCTION__ is a trace
T adder(T, Args ...) [with T = int; Args = int, int, int]      // expansion macro in gcc
T adder(T, Args ...) [with T = int; Args = int, int]
T adder(T, Args ...) [with T = int; Args = int]
T adder(T) [with T = int]
```

- We could also call as:

```
std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
std::string ssum = adder(s1, s2, s3, s4); // "x"+"aa"+"bb"+"yy" = "xaabbyy"
```

- `adder` will accept any number of arguments, and will compile properly as long as it can apply the `operator+` to them following template and overload resolution rules
- Variadic templates are like recursive code with a base case (`adder(T v)`) and a general case which *recurses* as in `adder(args...)`
- In `adder` - the first argument is peeled off the template parameter pack into type `T` (argument `first`). So with each call, the parameter pack shortens by one parameter to hit the base case

Variadic templates: adder

- Let us implement a function that adds all of its arguments together:

```
template<typename T> T adder(T v) { cout << __PRETTY_FUNCTION__ << endl; return v; }

template<typename T, typename... Args> // template parameter pack: typename... Args
T adder(T first, Args... args)          // function parameter pack: Args... args
{ cout << __PRETTY_FUNCTION__ << endl; return first + adder(args...); }
```

- And we could call and trace it as: `long sum = adder(1, 2, 3, 8, 7); // 21`

```
T adder(T, Args ...) [with T = int; Args = int, int, int, int] // __PRETTY_FUNCTION__ is a trace
T adder(T, Args ...) [with T = int; Args = int, int, int]      // expansion macro in gcc
T adder(T, Args ...) [with T = int; Args = int, int]
T adder(T, Args ...) [with T = int; Args = int]
T adder(T) [with T = int]
```

- We could also call as:

```
std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
std::string ssum = adder(s1, s2, s3, s4); // "x"+"aa"+"bb"+"yy" = "xaabbyy"
```

- `adder` will accept any number of arguments, and will compile properly as long as it can apply the `operator+` to them following template and overload resolution rules
- Variadic templates are like recursive code with a base case (`adder(T v)`) and a general case which *recurses* as in `adder(args...)`
- In `adder` - the first argument is peeled off the template parameter pack into type `T` (argument first). So with each call, the parameter pack shortens by one parameter to hit the base case

Variadic templates: adder

- Let us implement a function that adds all of its arguments together:

```
template<typename T> T adder(T v) { cout << __PRETTY_FUNCTION__ << endl; return v; }

template<typename T, typename... Args> // template parameter pack: typename... Args
T adder(T first, Args... args)          // function parameter pack: Args... args
{ cout << __PRETTY_FUNCTION__ << endl; return first + adder(args...); }
```

- And we could call and trace it as: `long sum = adder(1, 2, 3, 8, 7); // 21`

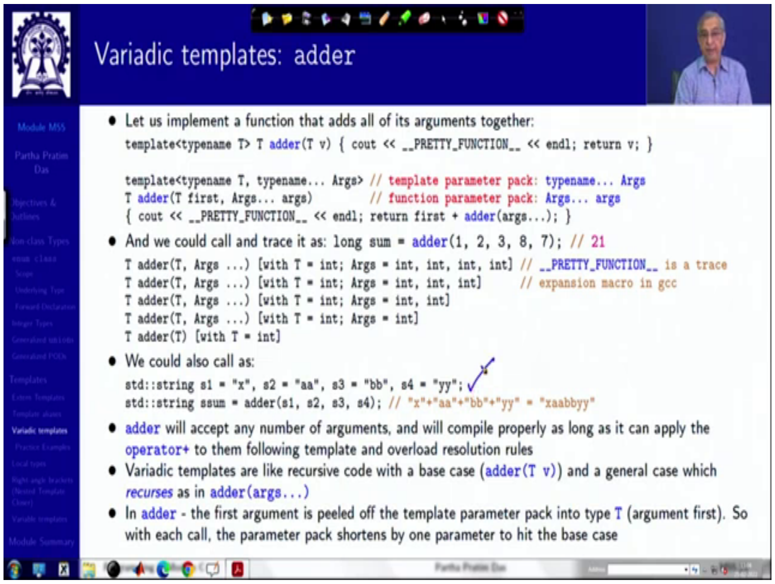
```
T adder(T, Args ...) [with T = int; Args = int, int, int, int] // __PRETTY_FUNCTION__ is a trace
T adder(T, Args ...) [with T = int; Args = int, int, int]      // expansion macro in gcc
T adder(T, Args ...) [with T = int; Args = int, int]
T adder(T, Args ...) [with T = int; Args = int]
T adder(T) [with T = int]
```

- We could also call as:

```
std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
std::string ssum = adder(s1, s2, s3, s4); // "x"+"aa"+"bb"+"yy" = "xaabbyy"
```

- `adder` will accept any number of arguments, and will compile properly as long as it can apply the `operator+` to them following template and overload resolution rules
- Variadic templates are like recursive code with a base case (`adder(T v)`) and a general case which *recurses* as in `adder(args...)`
- In `adder` - the first argument is peeled off the template parameter pack into type `T` (argument first). So with each call, the parameter pack shortens by one parameter to hit the base case

1+(2+(3+(8+(7))))



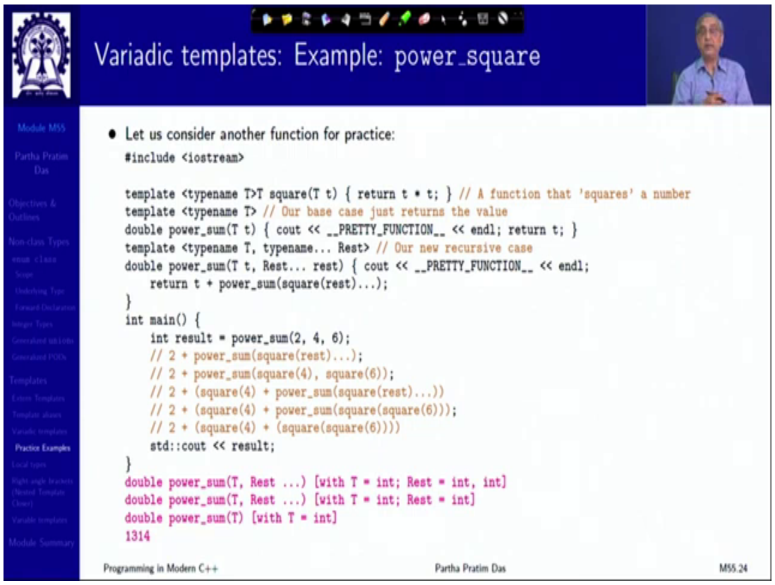
Now, let us think about doing this for, to build an adder. Suppose I want to build a adder function to add arbitrary number of values. So, I define an adder function first for one value, single value. So, I say, if it is a single value to add, it is that value itself, otherwise I use variadic template. What do I do? I say that I have two things. One is I have the type of the value being added and the other is I have a pack of parameters.

In the recursive or variadic char, I define adder with the first parameter from the pack. I call it first. It has to have the type T. And I leave the remaining function parameter within the pack. It will stay there. And then what do I do, I have the first value. So, what is the addition of the entire thing, the first value plus whatever is addition of the remaining pack which is a recursive call with the remaining function parameter pack.

So, if I try to do this on adder 1, 2, 3, 8, 7, it will give me 21 and this is how it actually expands. So, what I have used here for demonstration I have used a macro from gcc, online gdb compiler that we are using so that you can try out so that will tell you for every variadic expansion what is it be expanded on to. So, what happens is when you first try to instantiate call this template, so what will happen is first parameter is int, T is int so that goes there and the remaining 2, 3, 8, 7 is packed into four ints. So, you have one plus the remaining has to be added.

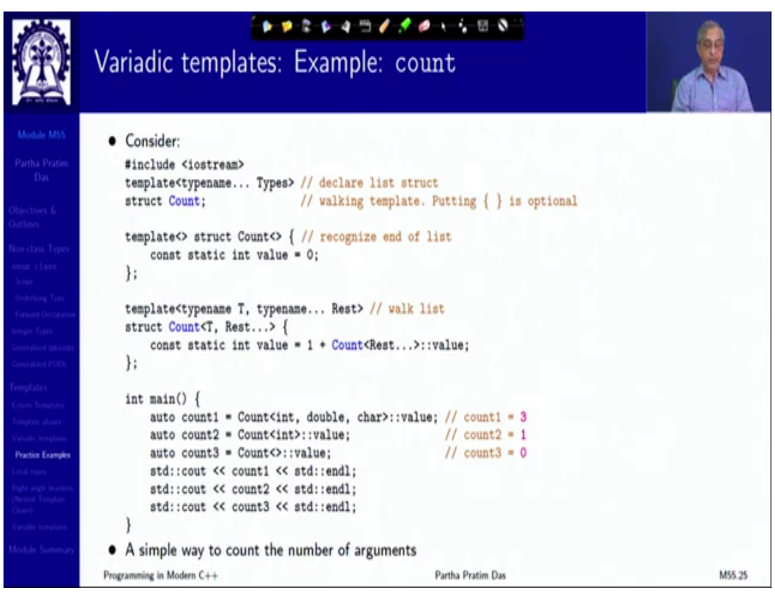
So, you go to the recurrence, as you go the recurrence, now you are expanding on 2, 3, 8, 7, so this was 1. So, this now becomes 2. The first 1 is gone. And what you are left with is just 2, 3, 8, 3 in the pack. So, you have 1 plus 2 plus 3 plus 8 plus goes on in this way. And when you hit this, you call T with int which is basically the exit function call. All expands, computed, a beautiful way to actually write very compact complied time, I mean, for any constant values is a compile time computed functional programming. It can be used for, for example, this adder could be used for any type which has operator class defined.

(Refer Slide Time: 35:53)



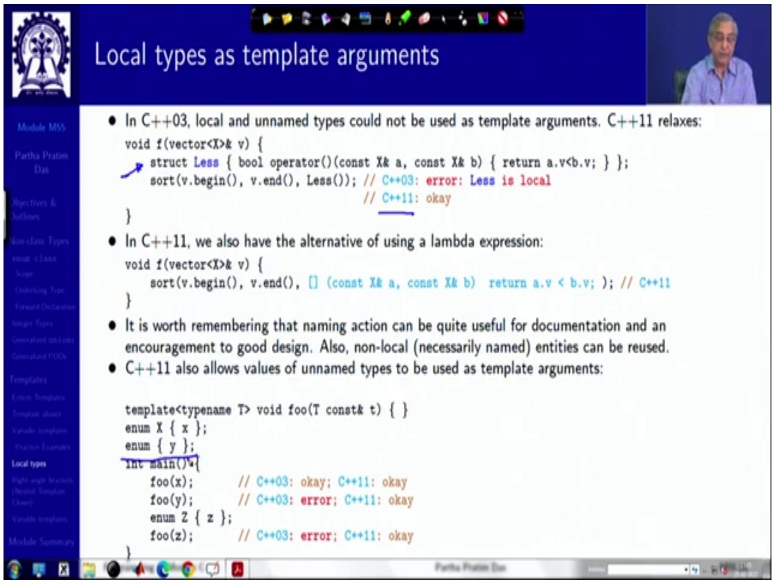
So, this variadic template is very important feature. So, therefore, I have given two practice examples, one is a peculiar way of squaring and adding that you should try out.

(Refer Slide Time: 36:04)



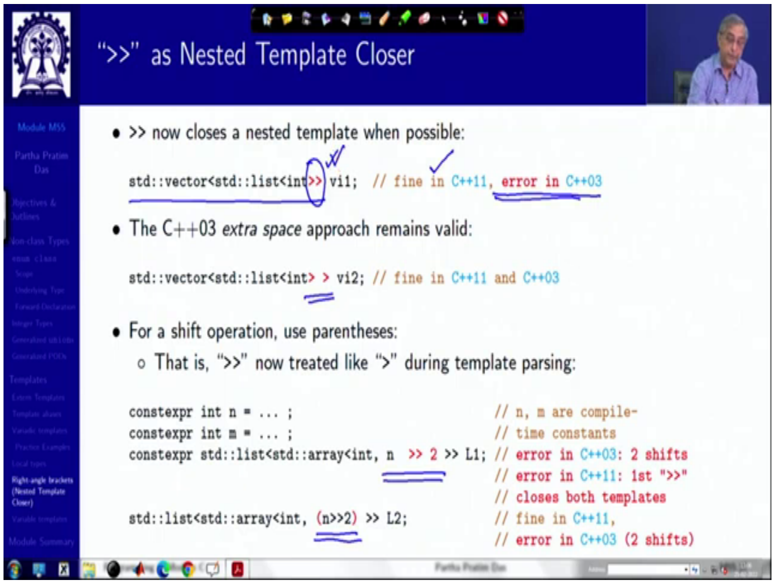
And another is a simple example to show that how can you write a variadic template to just count in an instantiation of the variadic template how many template parameters you have actually given. So, this will give me the count. Try these out at home.

(Refer Slide Time: 36:22)



Some of the less important features of templates use, include local types as template arguments. For example, in C++03 within a template function you cannot use a type that is local here. With C++11 you can use that. You can also use types which do not have a name, unnamed types. So, these are minor features which you have, which you may or may not use. There are other ways of doing that.

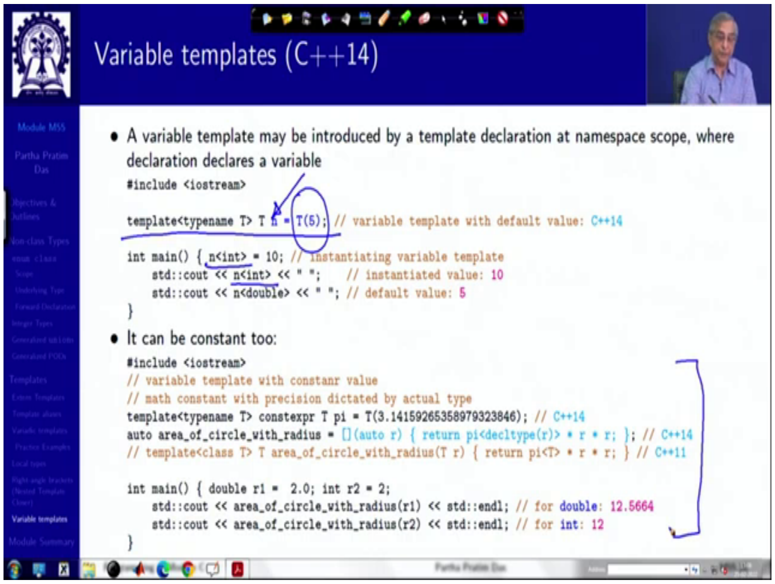
(Refer Slide Time: 36:53)



But this is important that in terms of syntax, C++03 had a major difficulty in terms of nested template. So, this is a nested template. So, what I am saying is std list int. So, I have a list of integer and then I have a vector of it. So, naturally, this will be the syntax. Now, the problem is this one is a write, as you write two consecutive, write bracket then it actually represents right shift in C++03. So, C++03 gives you an error and you have to write it, remember to write it them with a gap which is unnatural.

In C++11, this problem has been solved so you can write them as consecutive symbols. But this certainly means that when you want to write specifically about shifting along with the template expansion you will get into subsequent errors and you will have to guard them by putting proper parenthesis which is a very rare case, but this is a big advantage.

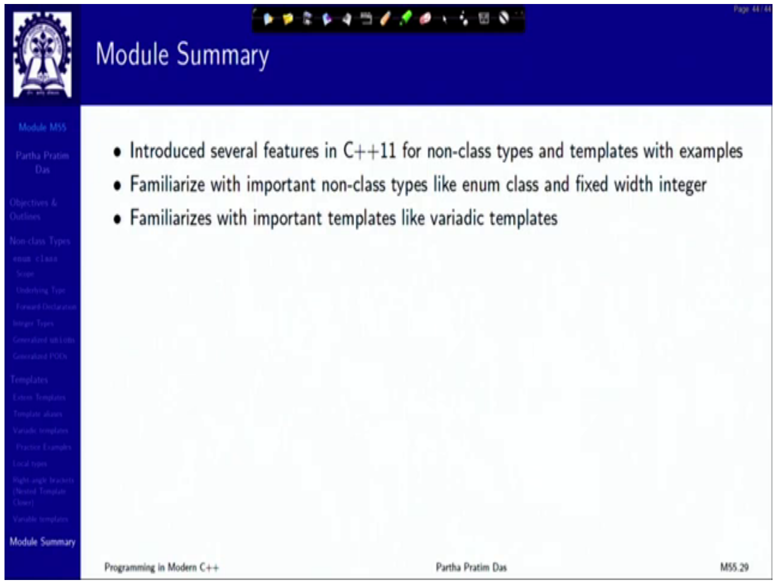
(Refer Slide Time: 38:13)



The last but not the least is an extension that is not in C++11 but in C++14 is that you have so far known templates are for classes, templates are for functions. So you do not have, you cannot have a variable as a template. So, if you want to have that, then you have to define a class, define it as static within that and so on so forth. But in C++11, you can simply have a variable as a template. So, here n is a simple variable. So, you say n is of type T. You can also specify a default value for that and use it simply in this way.

This is something which was earlier restricted only to classes and functions, but in C++14 you can have variables also which are templatize. There is a bigger example below here which illustrates same thing. It could be just a variable or it could be a constant variable and so on, but its instantiation will actually create that variable in your code.

(Refer Slide Time: 39:28)



So, that brings us to the end of this module where we have introduced several features of C++11 for non-class types and templates with examples and I remind you again that is very important that you learn the non-class type features like enum class and fixed width integer and among templates the variadic templates will be very, very important to work on. Thank you very much for your attention. See you in the next week.