

**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**

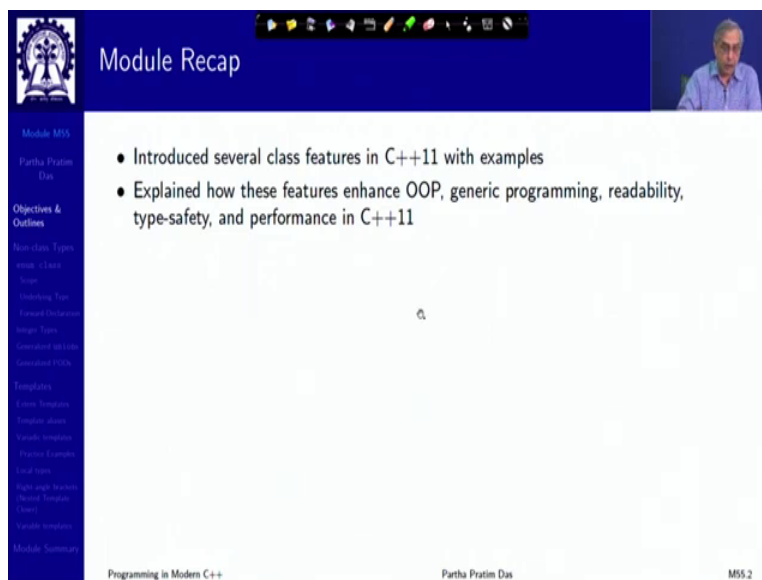
(Refer Slide Time: 00:32)



The slide features a blue header with the IIT Kharagpur logo on the left and navigation icons on the right. A central blue box contains the title "Programming in Modern C++" and the subtitle "Module M55: C++11 and beyond: Non-class Types and Template Features". Below this, the presenter's name "Partha Pratim Das" is listed, followed by his affiliation: "Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur" and his email "ppd@cse.iitkgp.ac.in". A note at the bottom states: "All url's in this module have been accessed in September, 2021 and found to be functional". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M55.1".

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

(Refer Slide Time: 00:37)



The slide is titled "Module Recap" and features a small video inset of the professor in the top right corner. The main content consists of two bullet points: "Introduced several class features in C++11 with examples" and "Explained how these features enhance OOP, generic programming, readability, type-safety, and performance in C++11". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M55.2".



(Refer Slide Time: 01:31)

The slide is titled "Module Outline" and is part of "Module M55: Programming in Modern C++". It features a navigation menu on the left and a main content area. The main content area lists the following topics:

- 1 Other (non-class) Types
  - enum class
    - Scope
    - Underlying Type
    - Forward-Declaration
  - Integer Types
  - Generalized unions
  - Generalized PODs
- 2 Templates
  - Extern Templates
  - Template aliases
  - Variadic templates
    - Practice Examples
  - Local types as template arguments
  - Right-angle brackets (Nested Template Closer)
  - Variable templates
- 3 Module Summary

The footer of the slide includes "Programming in Modern C++", "Partha Pratim Das", and "M55.4".

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

(Refer Slide Time: 01:38)

The slide is titled "Other (non-class) Types" and is part of "Module M55: Programming in Modern C++". It features a navigation menu on the left and a main content area. The main content area lists the following sources:

- Sources:
  - enum class
    - enum class, [isocpp.org](http://isocpp.org)
    - An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
    - Closer to Perfection: Get to Know C++11 Scoped and Based Enum Types
    - enum to string in modern C++11 / C++14 / C++17 and future C++20, [stackoverflow.com](http://stackoverflow.com)
  - Integer Types
    - Fixed width integer types, [isocpp.org](http://isocpp.org)
    - long long - a longer integer, [isocpp.org](http://isocpp.org)
    - Extended integer types, [isocpp.org](http://isocpp.org)
  - Generalized unions
    - Generalized unions, [isocpp.org](http://isocpp.org)
  - Generalized PODs
    - Generalized PODs, [isocpp.org](http://isocpp.org)

The title "Other (non-class) Types" is displayed in red text at the bottom of the main content area. The footer of the slide includes "Programming in Modern C++", "Partha Pratim Das", and "M55.5".

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)

Other (non-class) Types

- There have been several additions to non-class types in C++11. They include:
  - **enum class**: These solve several problems for **enum** in C++03
  - **Integer Types**: These include:
    - ▷ Fixed width integer types (as enhancements to integer types with size that is standard-defined). This comes from C99 feature
    - ▷ **long long** – a longer integer of 64 bits
    - ▷ Extended precision in integer types
  - **Generalized unions**: That allows rules for using **union** members with ctor / dtor / copy ops as enhancement over C++03
  - **Generalized PODs**: That defines rules for enhanced PODs in C++11
- Important features to learn
  - **enum class**
  - Fixed width integer, and
  - **long long**

Programming in Modern C++ Partha Pratim Das M5.6

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

- **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

- 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

- 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
```

The screenshot shows a presentation slide titled "enum class". It contains a list of bullet points and a code block. The bullet points discuss the problems with C++03 enumerations and the benefits of enum classes. The code block shows three examples: a standard C++03 enum, an enum class, and a class with an enum member. Comments and checkmarks indicate the behavior and errors of each.

```

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 an enum class in C++11.

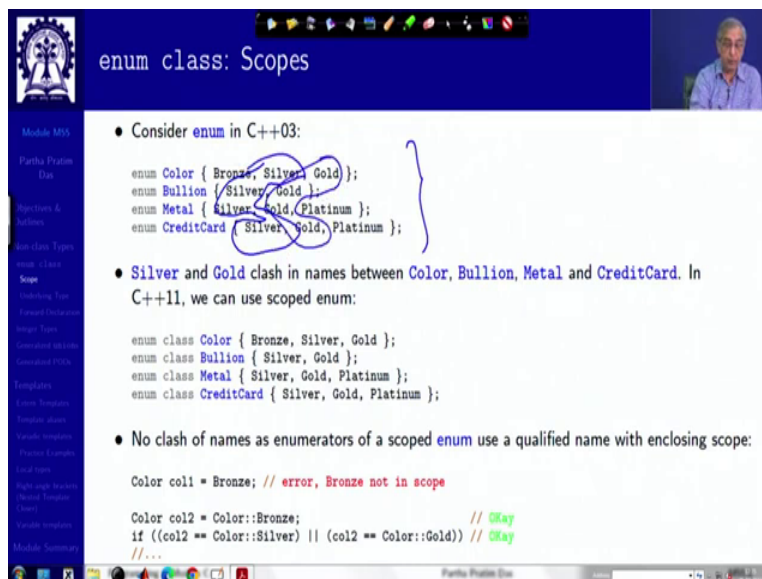
So, what is new is after the keyword enum and before the name of the enumeration you write this keyword class and rest of it is written in the same way. Now, the moment you write it as a class, it kind of imposes a namespace on to the set of enumerated values. So there is another enum class here and 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)



The slide is titled "enum class: Scopes" and features a video feed of a speaker in the top right corner. The main content includes:

- Consider `enum` in C++03:  

```
enum Color { Bronze, Silver, Gold };  
enum Bullion { Silver, Gold };  
enum Metal { Silver, Gold, Platinum };  
enum CreditCard { Silver, Gold, Platinum };
```
- `Silver` and `Gold` clash in names between `Color`, `Bullion`, `Metal` and `CreditCard`. In C++11, we can use scoped `enum`:  

```
enum class Color { Bronze, Silver, Gold };  
enum class Bullion { Silver, Gold };  
enum class Metal { Silver, Gold, Platinum };  
enum class CreditCard { Silver, Gold, Platinum };
```
- No clash of names as enumerators of a scoped `enum` use a qualified name with enclosing scope:  

```
Color col1 = Bronze; // error, Bronze not in scope  
  
Color col2 = Color::Bronze; // OKay  
if ((col2 == Color::Silver) || (col2 == Color::Gold)) // OKay  
//...
```

enum class: Scopes

- Consider `enum` in C++03:
 

```
enum Color { Bronze, Silver, Gold };
enum Bullion { Silver, Gold };
enum Metal { Silver, Gold, Platinum };
enum CreditCard { Silver, Gold, Platinum };
```
- Silver and Gold clash in names between `Color`, `Bullion`, `Metal` and `CreditCard`. In C++11, we can use scoped enum:
 

```
enum class Color { Bronze, Silver, Gold };
enum class Bullion { Silver, Gold };
enum class Metal { Silver, Gold, Platinum };
enum class CreditCard { Silver, Gold, Platinum };
```

*Color::Silver*  
*Bullion::Silver*
- No clash of names as enumerators of a scoped `enum` use a qualified name with enclosing scope:
 

```
Color col1 = Bronze; // error, Bronze not in scope
Color col2 = Color::Bronze; // OKay
if ((col2 == Color::Silver) || (col2 == Color::Gold)) // OKay
//...
```

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)

enum class: Underlying Type

- Specification of underlying type (optional) now permitted provided every value fits the type:  

```
enum Color: unsigned int { red, green, blue };  
enum Weather: std::uint8_t { sunny, rainy, cloudy, foggy };  
enum Status: std::uint8_t { pending, ready, unknown = 9999 }; // error! unknown does not fit size  
enum Color { red, green, blue }; // okay == type specification is optional as in C++03
```
- Strongly typed enums:
  - No implicit conversion to int
    - ▷ No comparing scoped `enums` values with `ints`
    - ▷ No comparing scoped `enums` objects of different types.
    - ▷ Explicit cast to `int` (or types convertible from `int`) okay
  - Values scoped to `enum` type
  - Underlying type defaults to `int`  

```
enum class Elevation: char { low, high }; // underlying type = char  
enum class Voltage { low, high }; // underlying type = int  
Elevation e = low; // error! no low in scope  
Elevation e = Elevation::low; // okay  
int x = Voltage::high; // error! no conversion to int  
if (e) ... // error! no conversion to bool  
if (e == Voltage::high) ... // error! no conversion from Elevation to Voltage
```

enum class: Underlying Type

- Specification of underlying type (optional) now permitted provided every value fits the type:  

```
enum Color: unsigned int { red, green, blue };  
enum Weather: std::uint8_t { sunny, rainy, cloudy, foggy };  
enum Status: std::uint8_t { pending, ready, unknown = 9999 }; // error! unknown does not fit size  
enum Color { red, green, blue }; // okay == type specification is optional as in C++03
```
- Strongly typed enums:
  - No implicit conversion to int
    - ▷ No comparing scoped `enums` values with `ints`
    - ▷ No comparing scoped `enums` objects of different types.
    - ▷ Explicit cast to `int` (or types convertible from `int`) okay
  - Values scoped to `enum` type
  - Underlying type defaults to `int`  

```
enum class Elevation: char { low, high }; // underlying type = char  
enum class Voltage { low, high }; // underlying type = int  
Elevation e = low; // error! no low in scope  
Elevation e = Elevation::low; // okay  
int x = Voltage::high; // error! no conversion to int  
if (e) ... // error! no conversion to bool  
if (e == Voltage::high) ... // error! no conversion from Elevation to Voltage
```

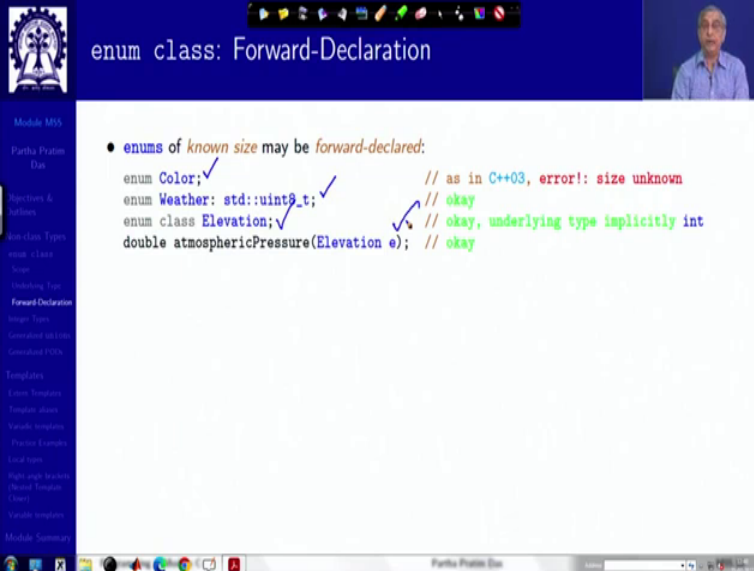
If we look at the underlying type, then we know that in C++03, the underlying type was always `int`. It is `enum` is converted to `int`, but now you can specify what is the underlying type. So what you do is the syntax for that is after the `enum` name and before the list starts, you put a colon and 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)



The screenshot shows a presentation slide with a blue header containing the title "enum class: Forward-Declaration" and a small video feed of the presenter. The main content area is white and contains a bulleted point: "enums of known size may be forward-declared:". Below this, there are four lines of C++ code with blue checkmarks and green annotations:

```
enum Color; ✓ // as in C++03, error!: size unknown
enum Weather: std::uint8_t; ✓ // okay
enum class Elevation; ✓ // okay, underlying type implicitly int
double atmosphericPressure(Elevation e); // okay
```

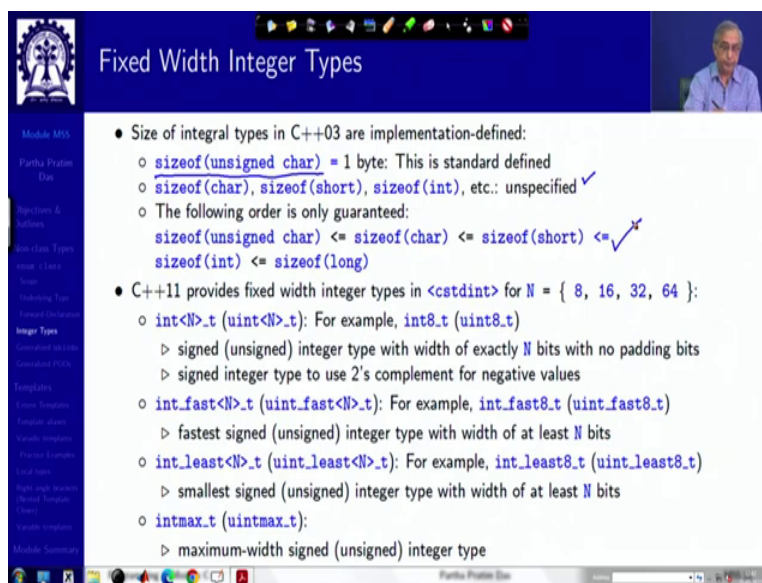
The slide also features a vertical navigation menu on the left side with various topics like "Module 005", "Partha Prasad Das", "Structures & Unions", "enum class", "Forward Declaration", "Templates", and "Module Summary".

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)



The image shows a presentation slide titled "Fixed Width Integer Types" with a blue header and a white content area. A small video inset of a speaker is visible in the top right corner. The slide content is as follows:

- 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 `<stdint>` 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**

- 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**

- 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

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)

The slide is titled "Extended Size & Precision of integers" and features a blue header with a logo on the left and a small video feed of a speaker on the right. The main content is a list of bullet points explaining the differences between integer types in a 36-bit system. The text is as follows:

- What is the difference between the `int` types: `int8_t`, `int_least8_t`, and `int_fast8_t`?
  - Suppose we have a C compiler for a 36-bit system, with `sizeof(char) = 9` bits, `sizeof(short) = 18` bits, `sizeof(int) = 36` bits, and `sizeof(long) = 72` bits. Then
    - ▷ `int8_t` does not exist, because there is no way to satisfy the constraint of having exactly 8 value bits with no padding
    - ▷ `int_least8_t` is a typedef of `char`. NOT of `short` or `int`, because the standard requires the smallest type with at least 8 bits
    - ▷ `int_fast8_t` can be anything. It is likely to be a typedef of `int` if the native size is considered to be fast
- C++11 provides support for `long long` – a longer integer
  - An integer that's at least 64 bits long. For example:

```
long long x = 9223372036854775807LL;
```
  - No, there are no `long long longs` nor can `long` be spelled `short long long`
- C++11 provides support for extended integer (precision) types with a set of rules

This slide is identical to the one above, containing the same text and structure. It explains the differences between integer types in a 36-bit system, specifically addressing `int8_t`, `int_least8_t`, `int_fast8_t`, and `long long`.

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)

**Generalized unions**

- In C++03, a *member with a user-defined ctor, dtor, or assignment cannot be a member of a union*:
 

```
union U {
    int m1;
    complex<double> m2; // error (silly): complex has constructor
    string m3;         // error (not silly): string has an invariant maintained by ctor, copy, & dtor
};
```
- Obviously, it is illegal to write one member and then read another
 

```
U u; // which constructor, if any?
u.m1 = 1; // assign to int member
string s = u.m3; // disaster: read from string member
```
- C++11 allows a member of types with ctor and dtor. It also adds a restriction to make the more flexible unions less error-prone by encouraging the building of discriminated unions
- **Union member types are restricted:**
  - o **No virtual functions, No references, and No bases** (as ever)
  - o **If a union has a member with a user-defined ctor, copy, or dtor then that special function is deleted; that is, it cannot be used for an object of the union type. This is new. For example:**

```
union U1 {                union U2 {
    int m1;                int m1;
    complex<double> m2; // okay    string m3; // okay
};                          };
```
  - o This may look error-prone, but the new restriction helps



**Generalized unions**

- In C++03, a member with a user-defined ctor, dtor, or assignment cannot be a member of a union:
 

```
union U {
    int m1;
    complex<double> m2; // error (silly): complex has constructor
    string m3;         // error (not silly): string has an invariant maintained by ctor, copy, & dtor
};
```
- Obviously, it is illegal to write one member and then read another
 

```
U u; // which constructor, if any?
u.m1 = 1; // assign to int member
string s = u.m3; // disaster: read from string member
```
- C++11 allows a member of types with ctor and dtor. It also adds a restriction to make the more flexible unions less error-prone by encouraging the building of discriminated unions
- Union member types are restricted:
  - No virtual functions, No references, and No bases (as ever) ✓
  - If a union has a member with a user-defined ctor, copy, or dtor then that special function is deleted; that is, it cannot be used for an object of the union type. This is new. For example:
 

```
union U1 {
    int m1;
    complex<double> m2; // okay
};

union U2 {
    int m1;
    string m3; // okay
};
```
  - This may look error-prone, but the new restriction helps

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)

**Generalized unions**

- Consider:  
`U1 u;` // okay  
`u.m2 = { 1, 2 };` // okay: assign to the complex member  
`U2 u2;` // error: the string destructor caused the U2 destructor to be deleted  
`U2 u3 = u2;` // error: the string copy constructor caused the U2 copy constructor to be deleted
- Basically, U2 is useless unless it is in a *discriminated unions*, such as:  

```
class Widget { private: // Three alternative implementations represented as a union
    enum class Tag { point, number, text } type; // discriminant
    union { point p; /* point has constructor */ int i;
            string s; // string has default ctor, copy operations, and dtor
    }; // ...
    widget& operator=(const widget& v) { // necessary because of the string variant
        if (type==Tag::text && v.type==Tag::text) { s = v.s; // usual string assignment
            return *this;
        }
        if (type==Tag::text) s."string(); // destroy (explicitly!)
        switch (v.type) {
            case Tag::point: p = v.p; break; // normal copy
            case Tag::number: i = v.i; break;
            case Tag::text: new(&s)(v.s); break; // placement new
        }
        type = v.type; return *this;
    }
};
```

**Generalized unions**

- Consider:  
`U1 u;` // okay  
`u.m2 = { 1, 2 };` // okay: assign to the complex member  
`U2 u2;` // error: the string destructor caused the U2 destructor to be deleted  
`U2 u3 = u2;` // error: the string copy constructor caused the U2 copy constructor to be deleted
- Basically, U2 is useless unless it is in a *discriminated unions*, such as:  

```
class Widget { private: // Three alternative implementations represented as a union
    enum class Tag { point, number, text } type; // discriminant
    union { point p; /* point has constructor */ int i;
            string s; // string has default ctor, copy operations, and dtor
    }; // ...
    widget& operator=(const widget& v) { // necessary because of the string variant
        if (type==Tag::text && v.type==Tag::text) { s = v.s; // usual string assignment
            return *this;
        }
        if (type==Tag::text) s."string(); // destroy (explicitly!)
        switch (v.type) {
            case Tag::point: p = v.p; break; // normal copy
            case Tag::number: i = v.i; break;
            case Tag::text: new(&s)(v.s); break; // placement new
        }
        type = v.type; return *this;
    }
};
```

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)

**Generalized PODs**

- A POD (*Plain Old Data*) is something that can be manipulated like a C struct, for example, *bitwise copyable* with `memcpy()`, *bitwise initializable* with `memset()`, etc.
- In C++03 a POD is decided by a set of restrictions on the features used in the definition of a `struct`:
 

```
struct S { int a; }; // Is a POD
struct SS { int a; SS(int aa): a(abs(aa)) { assert(a>=0); } }; // Not a POD in C++03; a POD in C++11
struct SSS { virtual void f(); /* ... */ }; // Definitely not POD
```
- In C++11, `S` and `SS` are *standard layout types* (a superset of *POD types*) where the ctor does not affect the layout (so `memcpy()` would be fine), only the initialization rules do (`memset()` would be bad)
- However, `SSS` will still have the `vp_ptr` and will not be anything like *plain old data*. C++11 defines:
  - POD Types: Check by `is_pod<T>::value` of type `bool` (deprecated in C++20)
  - Trivially Copyable Types: Check by `is_trivially_copyable<T>::value` of type `bool`
  - Trivial Types: Check by `is_trivial<T>::value` of type `bool`, and
  - Standard-Layout Types: Check by `is_standard_layout<T>::value` of type `bool`

to deal with various technical aspects of what used to be PODs. POD is defined recursively:

- *If all members and bases are PODs, then it is a POD*
- *Naturally: No virtual functions, No virtual bases, No references, and No multiple access specifiers*

- In C++11, PODs is that adding or subtracting constructors do not affect layout or performance

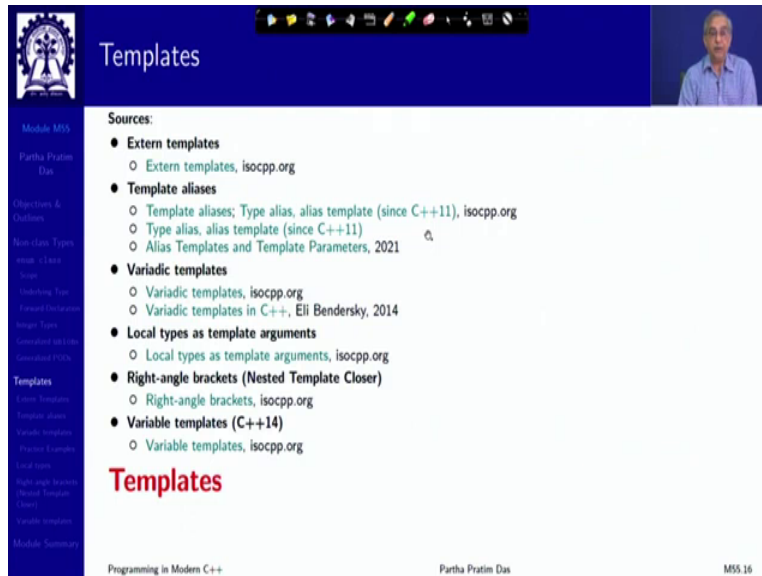
There is a some generalization of the plain old data types also which are bitwise copyable by mem copy or initially visible by memset. Like if I had a struct without any constructor. Now, that is a plain old data type because there is a bit pattern. In terms of C++03 also, these kinds of classes or structs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

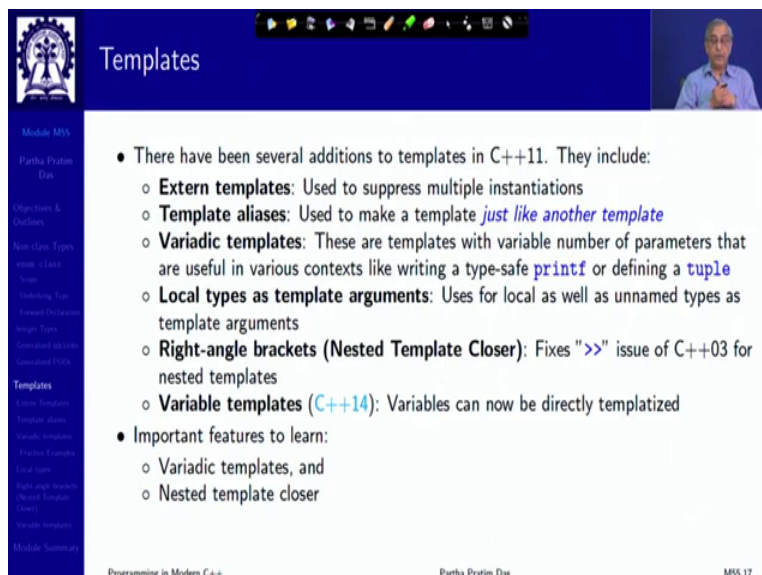
(Refer Slide Time: 24:27)



The slide is titled "Templates" and features a navigation sidebar on the left with categories like "Module M55", "Dispositions & Outlines", "Non-class Types", and "Templates". The main content area lists sources for various C++ template features:

- **Extern templates**
  - Extern templates, [isocpp.org](http://isocpp.org)
- **Template aliases**
  - Template aliases; Type alias, alias template (since C++11), [isocpp.org](http://isocpp.org)
  - Type alias, alias template (since C++11)
  - Alias Templates and Template Parameters, 2021
- **Variadic templates**
  - Variadic templates, [isocpp.org](http://isocpp.org)
  - Variadic templates in C++, Eli Bendersky, 2014
- **Local types as template arguments**
  - Local types as template arguments, [isocpp.org](http://isocpp.org)
- **Right-angle brackets (Nested Template Closer)**
  - Right-angle brackets, [isocpp.org](http://isocpp.org)
- **Variable templates (C++14)**
  - Variable templates, [isocpp.org](http://isocpp.org)

The word "Templates" is written in large red font at the bottom of the content area. The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M55.16".



The slide is titled "Templates" and features the same navigation sidebar as the previous slide. The main content area summarizes the additions to templates in C++11:

- There have been several additions to templates in C++11. They include:
  - **Extern templates**: Used to suppress multiple instantiations
  - **Template aliases**: Used to make a template *just like another template*
  - **Variadic templates**: These are templates with variable number of parameters that are useful in various contexts like writing a type-safe `printf` or defining a `tuple`
  - **Local types as template arguments**: Uses for local as well as unnamed types as template arguments
  - **Right-angle brackets (Nested Template Closer)**: Fixes ">>" issue of C++03 for nested templates
  - **Variable templates (C++14)**: Variables can now be directly templated
- Important features to learn:
  - Variadic templates, and
  - Nested template closer

The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M55.17".

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)

**Extern Templates**

- A template specialization can be explicitly declared as a way to suppress multiple instantiations. For example:

```
#include "MyVector.h"

// Suppresses implicit instantiation below --
// MyVector<int> will be explicitly instantiated elsewhere
extern template class MyVector<int>;

void foo(MyVector<int>& v) {
    // use the vector in here
}
```
- The *elsewhere* might look something like this:

```
#include "MyVector.h"
template class MyVector<int>; // Make MyVector available to clients
// For example, of the shared library
```
- This is basically a way of avoiding significant redundant work by the compiler and linker

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 aliases**

- We can make a template *like another template* with a few of template arguments bound:

```
template<class T>
using Vec = std::vector<T, My_alloc<T>>; // standard vector using my allocator
Vec<int> fib = { 1, 2, 3, 5, 8, 13 }; // allocates elements using My_alloc
vector<int, My_alloc<int>> verbose = fib; // verbose and fib are of the same type
```
- `using` is used to get a linear notation where *name is followed by what it refers to*. Also, we can *alias a set of specializations but we cannot specialize an alias*:

```
// int_exact_traits<N>::type is a type with exactly N bits
template<int> struct int_exact_traits { typedef int type; };
template<> struct int_exact_traits<8> { typedef char type; };
template<> struct int_exact_traits<16> { typedef char[2] type; };
// ... define alias for convenient notation
template<int N> using int_exact = typename int_exact_traits<N>::type;
int_exact<8> a = 7; // int_exact<8> is an int with 8 bits
```
- Type aliases can also be used as a different syntax for ordinary type aliases:

```
typedef void (*PFD)(double); // C style
using PF = void (*)(double); // using plus C-style type
using P = auto (*)(double) -> void; // using plus suffix return type
```

Programming in Modern C++ Partha Pratim Das M55.19

Template aliases: Example: Matrix

- Consider template class `Matrix`:
 

```
template <typename T, int Line, int Col>
class Matrix { ... };
```
- `Matrix` has 3 parameters. The type parameter `T`, and the non-type parameters `Line`, and `Col`
- For readability, we want to have two special matrices: a `Square` and a `Vector`. A `Square`'s number of lines and columns should be equal. A `Vector`'s line size should be one.

```
template <typename T, int Line>
using Square = Matrix<T, Line, Line>; // #1

template <typename T, int Line>
using Vector = Matrix<T, Line, 1>; // #2
```

- `using` declares a type alias (#1 & #2). While the primary template `Matrix` can be parametrized in the three dimensions `T`, `Line`, and `Col`, the type aliases `Square` and `Vector` reduce the parametrization to the two dimensions `T` and `Line`
- Template alias creates names for partially bound templates. Using `Square` and `Vector` is easy:
 

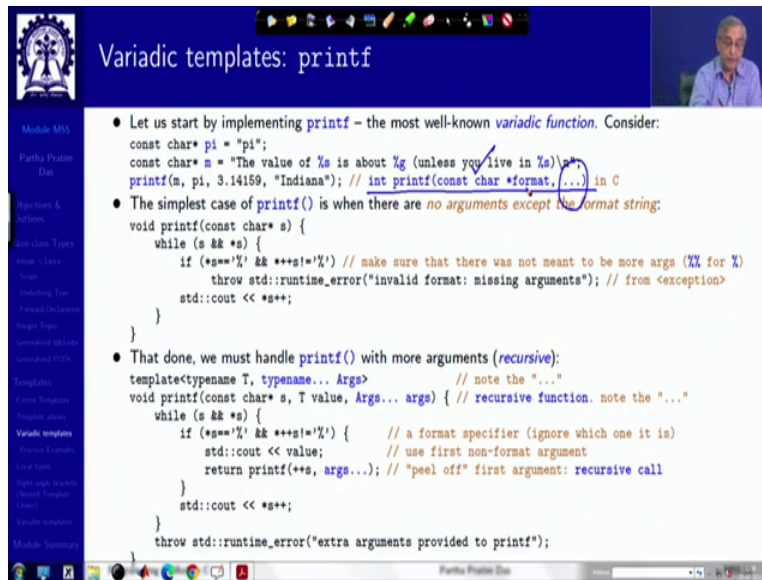
```
Matrix<int, 5, 3> ma;
Square<double, 4> sq; // Matrix<double, 4, 4>
Vector<char, 5> vec; // Matrix<char, 5, 1>
```

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)



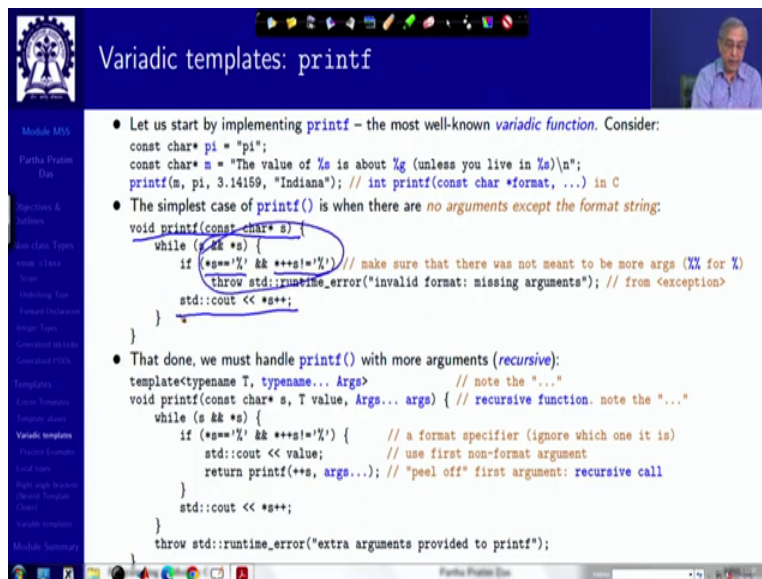
### Variadic templates: printf

- 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 (%X 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 (%X 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 (%X 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 (%X 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 (%X for %X)
            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 (%X for %X)
            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)

**Variadic templates: printf**

- The code *peels off* the first non-format arg. and then calls itself recursively. When there is no more non-format arg., it calls the first `printf()` – *functional programming at compile time*
- The `Args...` defines what is called a *parameter pack* – a sequence of (type/value) pairs to *peel off* arguments starting with the first:
  - When `printf()` is called with one argument, the first `printf(const char*)` is chosen
  - When `printf()` is called with two or more arguments, the second `printf(const char* s, T value, Args... args)` is chosen, with the first argument as *s*, the second as *value*, and the rest (if any) bundled into the parameter pack *args* for later use
  - In the call `printf(++s, args...)` the parameter pack *args* is expanded so that the next argument can now be selected as *value*
  - This carries on until *args* is empty so that the first `printf()` is called
- For generic functional programming, we declare and use a simple variadic template function:

```
template<class ... Types>
void f(Types ... args); // variadic template function. That is, a function that can take
                        // an arbitrary number of arguments of arbitrary types

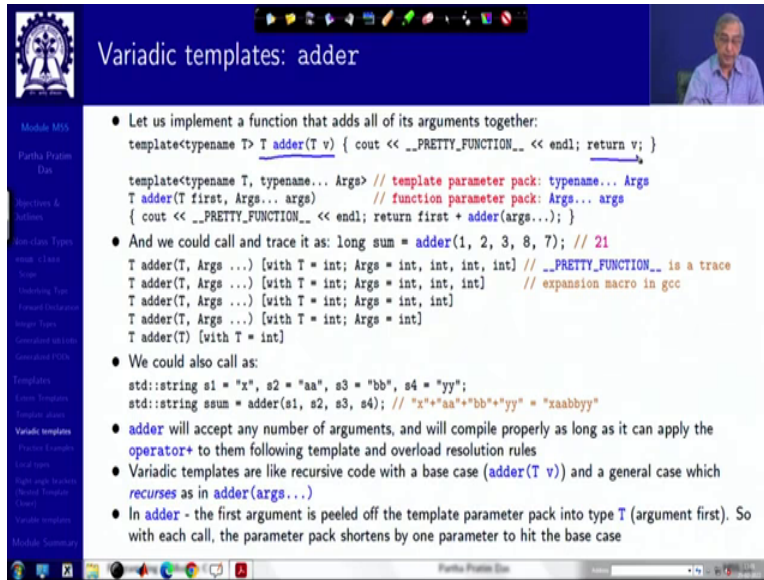
f(); // OK: args contains no arguments
f(1); // OK: args contains one argument: int
f(2, 1.0); // OK: args contains two arguments: int and double
```

Programming in Modern C++ Partha Pratim Das M55.22

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

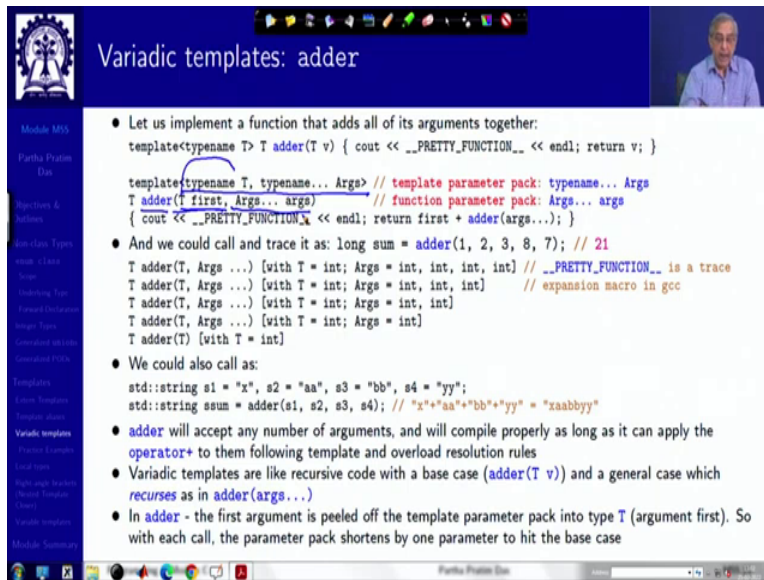
- Let us implement a function that adds all of its arguments together:

```
template<typename T> T adder(T v) { cout << __PRETTY_FUNCTION__ << endl; return v; }
```
- 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; }
```
- 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" = "xaabyyy"
```
- 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]
```

*Handwritten note:*  $1 + (2 + (3 + (8 + (7))))$
- 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" = "xaabyyy"
```
- 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" = "xaabby"
```
- `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

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 compiled 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)

**Variadic templates: Example: power\_square**

- Let us consider another function for practice:

```
#include <iostream>

template<typename T> square(T t) { return t * t; } // A function that 'squares' a number
template<typename T> // Our base case just returns the value
double power_sum(T t) { cout << __PRETTY_FUNCTION__ << endl; return t; }
template<typename T, typename... Rest> // Our new recursive case
double power_sum(T t, Rest... rest) { cout << __PRETTY_FUNCTION__ << endl;
    return t + power_sum(square(rest)...);
}

int main() {
    int result = power_sum(2, 4, 6);
    // 2 + power_sum(square(rest)...);
    // 2 + power_sum(square(4), square(6));
    // 2 + (square(4) + power_sum(square(rest)...));
    // 2 + (square(4) + power_sum(square(square(6))));
    // 2 + (square(4) + (square(square(6))))
    std::cout << result;
}

double power_sum(T, Rest ...) [with T = int; Rest = int, int]
double power_sum(T, Rest ...) [with T = int; Rest = int]
double power_sum(T) [with T = int]
1314
```

Programming in Modern C++ Partha Pratim Das M55.24

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)

**Variadic templates: Example: count**

- Consider:

```
#include <iostream>
template<typename... Types> // declare list struct
struct Count; // walking template. Putting { } is optional

template<> struct Count<> { // recognize end of list
    const static int value = 0;
};

template<typename T, typename... Rest> // walk list
struct Count<T, Rest...> {
    const static int value = 1 + Count<Rest...>::value;
};

int main() {
    auto count1 = Count<int, double, char>::value; // count1 = 3
    auto count2 = Count<int>::value; // count2 = 1
    auto count3 = Count<>::value; // count3 = 0
    std::cout << count1 << std::endl;
    std::cout << count2 << std::endl;
    std::cout << count3 << std::endl;
}

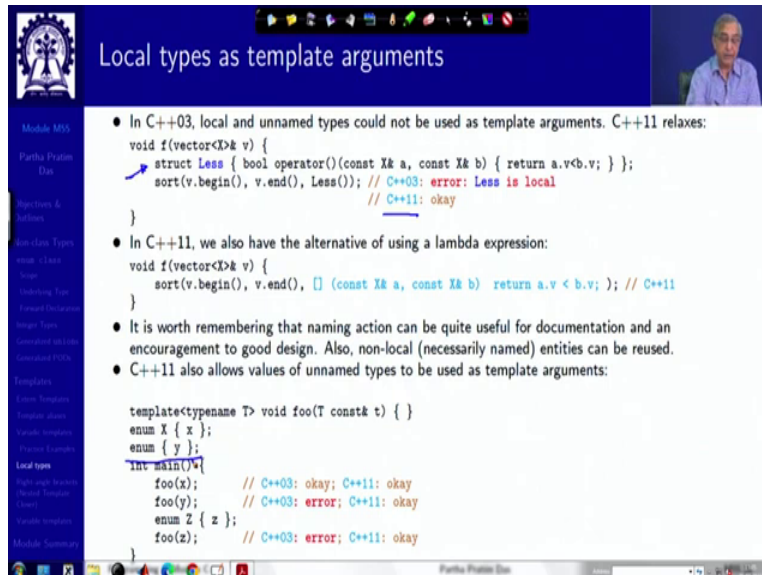
• A simple way to count the number of arguments
```

Programming in Modern C++ Partha Pratim Das M55.25



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)



The slide is titled "Local types as template arguments" and features a blue header with a logo on the left and a small video inset of a speaker on the right. The main content is a white box with a blue border containing text and code. The text explains the evolution of C++ regarding local types as template arguments. It shows two code snippets: one for C++03 that fails to compile due to a local type being used as a template argument, and another for C++11 that uses a lambda expression to avoid this. A third code snippet demonstrates the use of unnamed local types in a template function.

Local types as template arguments

- In C++03, local and unnamed types could not be used as template arguments. C++11 relaxes:

```
void f(vector<X&& v) {  
    struct Less { bool operator()(const X& a, const X& b) { return a.v<b.v; } };  
    sort(v.begin(), v.end(), Less()); // C++03: error: Less is local  
                                     // C++11: okay  
}
```

- In C++11, we also have the alternative of using a lambda expression:

```
void f(vector<X&& v) {  
    sort(v.begin(), v.end(), [] (const X& a, const X& b) return a.v < b.v; ); // C++11  
}
```

- It is worth remembering that naming action can be quite useful for documentation and an encouragement to good design. Also, non-local (necessarily named) entities can be reused.
- C++11 also allows values of unnamed types to be used as template arguments:

```
template<typename T> void foo(T const& t) {  
    enum X { x };  
    enum { y };  
    int main() {  
        foo(x); // C++03: okay; C++11: okay  
        foo(y); // C++03: error; C++11: okay  
        enum Z { z };  
        foo(z); // C++03: error; C++11: okay  
    }
```

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)

**">>" as Nested Template Closer**

- >> now closes a nested template when possible:  
`std::vector<std::list<int>>> vi1; // fine in C++11, error in C++03`
- The C++03 extra space approach remains valid:  
`std::vector<std::list<int> > vi2; // fine in C++11 and C++03`
- For a shift operation, use parentheses:
  - That is, ">>" now treated like ">" during template parsing:  
`constexpr int n = ... ; // n, m are compile-  
constexpr int m = ... ; // time constants  
constexpr std::list<std::array<int, n >> 2 >> L1; // error in C++03: 2 shifts  
// error in C++11: list ">>"  
// closes both templates  
std::list<std::array<int, (n>>2) >> L2; // fine in C++11,  
// error in C++03 (2 shifts)`

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 then 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)

Variable templates (C++14)

- A variable template may be introduced by a template declaration at namespace scope, where declaration declares a variable

```
#include <iostream>

template<typename T> T f = T(5); // variable template with default value: C++14

int main() { n<int> = 10; // instantiating variable template
std::cout << n<int> << " "; // instantiated value: 10
std::cout << n<double> << " "; // default value: 5
}
```

- It can be constant too:

```
#include <iostream>
// variable template with constant value
// math constant with precision dictated by actual type
template<typename T> constexpr T pi = T(3.14159265358979323846); // C++14
auto area_of_circle_with_radius = [](auto r) { return pi<decltype(r)> * r * r; }; // C++14
// template<class T> T area_of_circle_with_radius(T r) { return pi<T> * r * r; }; // C++11

int main() { double r1 = 2.0; int r2 = 2;
std::cout << area_of_circle_with_radius(r1) << std::endl; // for double: 12.5664
std::cout << area_of_circle_with_radius(r2) << std::endl; // for int: 12
}
```

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 templated. 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)

Module Summary

- Introduced several features in C++11 for non-class types and templates with examples
- Familiarize with important non-class types like enum class and fixed width integer
- Familiarizes with important templates like variadic templates

Module M55  
Partha Pratim Das  
Objectives & Outlines  
Non-Class Types  
enum class  
fixed width integer  
constexpr  
Templates  
Variadic Templates  
Module Summary

Programming in Modern C++ Partha Pratim Das M55.29

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.