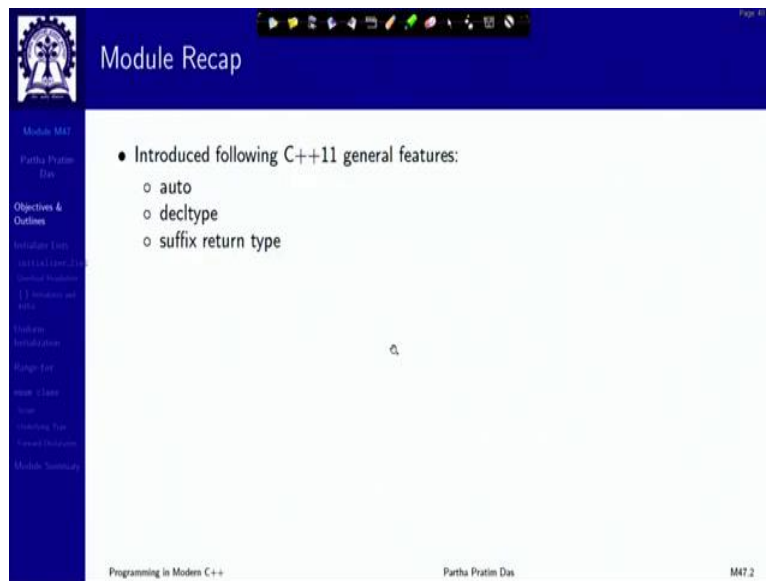


Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 47
C++ 11 and beyond: General Features: Part 2

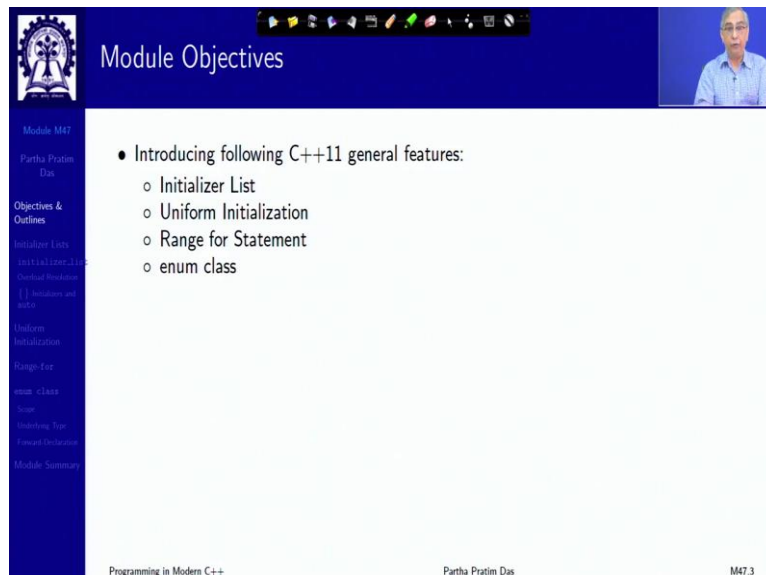
Welcome to Programming in Modern C++. We are in week 10 and we are going to discuss module 47.

(Refer Slide Time: 0:35)



The slide is titled "Module Recap" and is part of a presentation. It features a blue header with the IIT Kharagpur logo on the left and a navigation bar at the top. The main content area is white and contains a bulleted list of C++11 general features. A vertical sidebar on the left lists various topics, with "Objectives & Outlines" highlighted. The footer includes the course name, the professor's name, and the slide number.

- Introduced following C++11 general features:
 - auto
 - decltype
 - suffix return type

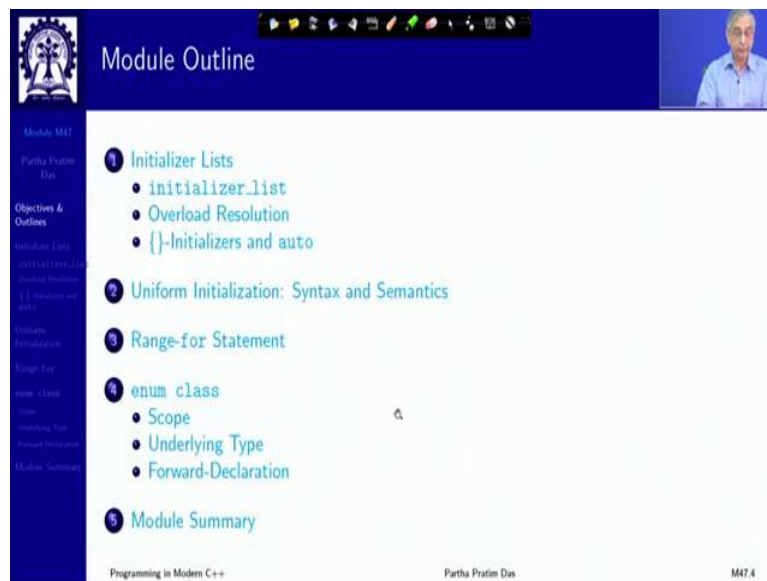


The slide is titled "Module Objectives" and is part of a presentation. It features a blue header with the IIT Kharagpur logo on the left and a navigation bar at the top. The main content area is white and contains a bulleted list of C++11 general features. A small video inset of the professor is visible in the top right corner. A vertical sidebar on the left lists various topics, with "Objectives & Outlines" highlighted. The footer includes the course name, the professor's name, and the slide number.

- Introducing following C++11 general features:
 - Initializer List
 - Uniform Initialization
 - Range for Statement
 - enum class

In the last module we have introduced the overall features of C++11 and talked specifically about auto, decltype and suffix return type. We will continue on that and introduce these four general features in this module.

(Refer Slide Time: 0:53)



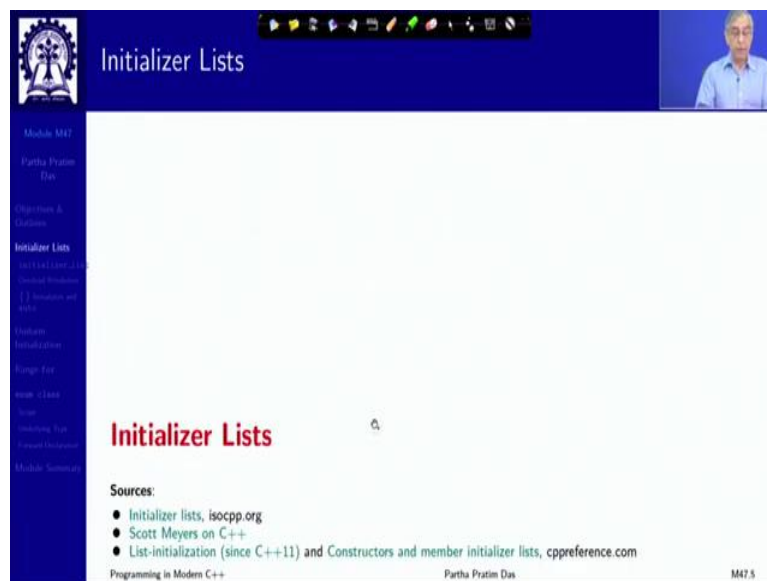
The slide titled "Module Outline" shows a table of contents for Module M47. The left sidebar lists the following items: Module M47, Partha Pratim Das, Objectives & Outlines, Initializer Lists, Uniform Initialization, Range-for, enum class, and Module Summary. The main content area lists the following topics:

- 1 Initializer Lists
 - initializer_list
 - Overload Resolution
 - {}-Initializers and auto
- 2 Uniform Initialization: Syntax and Semantics
- 3 Range-for Statement
- 4 enum class
 - Scope
 - Underlying Type
 - Forward-Declaration
- 5 Module Summary

At the bottom, it says "Programming in Modern C++", "Partha Pratim Das", and "M47.4".

Naturally this outline will be available on your left all the time.

(Refer Slide Time: 0:57)



The slide titled "Initializer Lists" shows the same left sidebar as the previous slide. The main content area has the title "Initializer Lists" in red. Below it, under the heading "Sources:", there is a list of references:

- Initializer lists, isocpp.org
- Scott Meyers on C++
- List-initialization (since C++11) and Constructors and member initializer lists, cpreference.com

At the bottom, it says "Programming in Modern C++", "Partha Pratim Das", and "M47.5".

So, we start with `initializer_list`. The whole idea is how do you initialize a particular variable in C++11.

(Refer Slide Time: 1:07)

Initializer Lists

- Consider:

```
vector<double> v = { 1, 2, 3.456, 99.99 }; // list of doubles
list<pair<string,string>> languages = { // list of pairs of strings
    {"Nyggaard", "Simula"}, {"Richards", "BCPL"}, {"Ritchie", "C"}
};
map<vector<string>,vector<int>> years = { // list of vector<string>s and vector<int>s
    {"Maurice", "Vincent", "Wilkes"}, {1913, 1945, 1951, 1967, 2000} },
    {"Martin", "Ritchards"}, {1982, 2003, 2007} },
    {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004}
};
```
- Initializer lists are not just for arrays. The mechanism for accepting a {}-list (*braced list*) is a *function* (often a *constructor*) accepting an argument of type `std::initializer_list<T>`:

```
void f(initializer_list<int>);
f({1,2});
f({23, 345, 4567, 56789});
f({}); // the empty list
f{1,2}; // error: function call () missing
years.insert({"Bjarne", "Stroustrup"}, {1950, 1975, 1985});
```

Initializer Lists

- Consider:

```
vector<double> v = { 1, 2, 3.456, 99.99 }; // list of doubles
list<pair<string,string>> languages = { // list of pairs of strings
    {"Nyggaard", "Simula"}, {"Richards", "BCPL"}, {"Ritchie", "C"}
};
map<vector<string>,vector<int>> years = { // list of vector<string>s and vector<int>s
    {"Maurice", "Vincent", "Wilkes"}, {1913, 1945, 1951, 1967, 2000} },
    {"Martin", "Ritchards"}, {1982, 2003, 2007} },
    {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004}
};
```
- Initializer lists are not just for arrays. The mechanism for accepting a {}-list (*braced list*) is a *function* (often a *constructor*) accepting an argument of type `std::initializer_list<T>`:

Initializer Lists

- Consider:


```
vector<double> v = { 1, 2, 3.456, 99.99 }; // list of doubles
list<pair<string,string>> languages = { // list of pairs of strings
    {"Nygaard", "Simula"}, {"Richards", "BCPL"}, {"Ritchie", "C"}
};
map<vector<string>,vector<int>> years = { // list of vector<string>s and vector<int>s
    {"Maurice", "Vincent", "Wilkes"}, {1913, 1945, 1951, 1967, 2000} },
    {"Martin", "Ritchards"}, {1982, 2003, 2007} },
    {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
};
```
- Initializer lists are not just for arrays. The mechanism for accepting a {}-list (*braced list*) is a *function* (often a *constructor*) accepting an argument of type `std::initializer_list<T>`:


```
void f(initializer_list<int>);
f({1,2});
f({23, 345, 4567, 56789});
f({}); // the empty list
f{1,2}; // error: function call () missing
years.insert({{"Bjarne", "Stroustrup"}, {1950, 1975, 1985}});
```

So, let us look at some of the initialization styles that we already know. We can initialize a vector by putting the values like this, we can initialize a list of pairs of strings, so this each one gives you a kind of a literal which is a pair of strings and then you have a list of them, so this is a list. You can do it for a map, which is just to declutter, do it for a map which has a key which is a vector of string and value which is a vector of int.

So, what you have, in each case is the vector of string, vector of int paired together, vector of string, vector of int paired together. So, these are some of the ways that you initialize and you can see that in every case that, I mean, for these kind of vectors and arrays you have a nice curly brace or braced initializer syntax. But initializers list is not just for is, it kind of tries to generalize and it is often actually a function, it is often actually, a constructor hidden behind it which accepts arguments of this type.

So, this is provided by a new component in C++11 standard library that gives you an `initializer_list`. So, we will see what that `initializer_list` is, so with that you can just define a function with `initializer_list` of type `int`, which means that you will have braced initialization of integer values or you can directly then call this function like this or like this or like this that is an empty list, but you cannot call it without the parentheses. This is missing. You can use it say in insert, we did insert earlier and, for example, for this vector, map I can do an insert of this form, so that is what the `initializer_list` is about.

(Refer Slide Time: 3:37)

Initializer Lists

- The initializer list can be of *arbitrary length*, but must be *homogeneous* (all elements must be of the *template argument type, T, or convertible to T*). An *initializer-list constructor* may be implemented as:

```
template<class E> class vector { public:  
    vector(std::initializer_list<E> s) { // initializer-list constructor  
        reserve(s.size()); // get the right amount of space  
        uninitialized_copy(s.begin(), s.end(), elem); // init. elements (in elem[0:s.size()])  
        sz = s.size(); // set vector size  
    }  
};
```
- The distinction between *direct* and *copy initialization* is maintained for {}-initialization, but is *less relevant*. For `std::vector` with an explicit ctor from `int` and an `initializer_list` ctor:

```
vector<double> v1(7); // okay: v1 has 7 elements  
v1 = 9; // error: no conversion from int to vector  
vector<double> v2 = 9; // error: no conversion from int to vector  
void f(const vector<double>&);  
f(9); // error: no conversion from int to vector  
vector<double> v1{7}; // okay: v1 has 1 element (with its value 7.0)  
v1 = {9}; // okay v1 now has 1 element (with its value 9.0)  
vector<double> v2 = {9}; // okay: v2 has 1 element (with its value 9.0)  
f({9}); // okay: f is called with the list { 9 }  
vector<vector<double>> vs = {  
    vector<double>(10), // okay: explicit construction (10 elements)  
    vector<double>(10), // okay explicit construction (1 element with the value 10.0)  
    10 // error: vector's constructor is explicit  
};
```

Initializer Lists

- The initializer list can be of *arbitrary length*, but must be *homogeneous* (all elements must be of the *template argument type, T, or convertible to T*). An *initializer-list constructor* may be implemented as:

```
template<class E> class vector { public:  
    vector(std::initializer_list<E> s) { // initializer-list constructor  
        reserve(s.size()); // get the right amount of space  
        uninitialized_copy(s.begin(), s.end(), elem); // init. elements (in elem[0:s.size()])  
        sz = s.size(); // set vector size  
    }  
};
```
- The distinction between *direct* and *copy initialization* is maintained for {}-initialization, but is *less relevant*. For `std::vector` with an explicit ctor from `int` and an `initializer_list` ctor:

```
vector<double> v1(7); // okay: v1 has 7 elements  
v1 = 9; // error: no conversion from int to vector  
vector<double> v2 = 9; // error: no conversion from int to vector  
void f(const vector<double>&);  
f(9); // error: no conversion from int to vector  
vector<double> v1{7}; // okay: v1 has 1 element (with its value 7.0)  
v1 = {9}; // okay v1 now has 1 element (with its value 9.0)  
vector<double> v2 = {9}; // okay: v2 has 1 element (with its value 9.0)  
f({9}); // okay: f is called with the list { 9 }  
vector<vector<double>> vs = {  
    vector<double>(10), // okay: explicit construction (10 elements)  
    vector<double>(10), // okay explicit construction (1 element with the value 10.0)  
    10 // error: vector's constructor is explicit  
};
```

Initializer Lists

- The initializer list can be of *arbitrary length*, but must be *homogeneous* (all elements must be of the *template argument type, T, or convertible to T*). An *initializer-list constructor* may be implemented as:


```
template<class E> class vector { public:
    vector(std::initializer_list<E> s) {           // initializer-list constructor
        reserve(s.size());                       // get the right amount of space
        uninitialized_copy(s.begin(), s.end(), elem); // init. elements (in elem[0:s.size()))
        sz = s.size();                           // set vector size
    }
};
```
- The distinction between *direct* and *copy initialization* is maintained for `{}`-initialization, but is *less relevant*. For `std::vector` with an explicit ctor from `int` and an `initializer_list` ctor:


```
vector<double> v1(7);           // okay: v1 has 7 elements
v1 = 9;                        // error: no conversion from int to vector
vector<double> v2 = 9;         // error: no conversion from int to vector
void f(const vector<double>&k);
f(9);                          // error: no conversion from int to vector
vector<double> v1{7};          // okay: v1 has 1 element (with its value 7.0)
v1 = {9};                      // okay v1 now has 1 element (with its value 9.0)
vector<double> v2 = {9};       // okay: v2 has 1 element (with its value 9.0)
f({9});                        // okay: f is called with the list { 9 }
vector<vector<double>> vs = {
    vector<double>(10),         // okay: explicit construction (10 elements)
    vector<double>{10},        // okay explicit construction (1 element with the value 10.0)
    10                          // error: vector's constructor is explicit
};
```

Initializer Lists

- The initializer list can be of *arbitrary length*, but must be *homogeneous* (all elements must be of the *template argument type, T, or convertible to T*). An *initializer-list constructor* may be implemented as:


```
template<class E> class vector { public:
    vector(std::initializer_list<E> s) {           // initializer-list constructor
        reserve(s.size());                       // get the right amount of space
        uninitialized_copy(s.begin(), s.end(), elem); // init. elements (in elem[0:s.size()))
        sz = s.size();                           // set vector size
    }
};
```
- The distinction between *direct* and *copy initialization* is maintained for `{}`-initialization, but is *less relevant*. For `std::vector` with an explicit ctor from `int` and an `initializer_list` ctor:


```
vector<double> v1(7);           // okay: v1 has 7 elements
v1 = 9;                        // error: no conversion from int to vector
vector<double> v2 = 9;         // error: no conversion from int to vector
void f(const vector<double>&k);
f(9);                          // error: no conversion from int to vector
vector<double> v1{7};          // okay: v1 has 1 element (with its value 7.0)
v1 = {9};                      // okay v1 now has 1 element (with its value 9.0)
vector<double> v2 = {9};       // okay: v2 has 1 element (with its value 9.0)
f({9});                        // okay: f is called with the list { 9 }
vector<vector<double>> vs = {
    vector<double>(10),         // okay: explicit construction (10 elements)
    vector<double>{10},        // okay explicit construction (1 element with the value 10.0)
    10                          // error: vector's constructor is explicit
};
```

So, an `initializer_list` can be of arbitrary length 0, 1, anything, but the key point is it has to be homogeneous, which means that each element you use in the `initializer_list` must be of the same type, `initializer_list` has a type `t`, so it must be of the same type, `t` or something which is convertible to `t`. So, with that I can write a constructor of `vector` which takes an `initializer_list` of underlying type `E`. So, it is a vector of `E`.

So, that list is `s`, so I, internally I reserve that much of space for, that is required for `s` and then I do a `uninitialized_copy` is an algorithm available, so I do a copy of from iterating from `s.begin` to `s.end` into `elem` which is the element array set the size. So, this is a typical way an `initializer_list` can be used. Now, the distinction between direct and copy initialization is there for braced initialization, but it is less relevant I should say.

So, if I say `v1` within parentheses `7`, then `v1` has 7 elements, I cannot use this because there is no conversion from `int` to `vector`. I cannot use this again for the same reason there is no conversion, if I have this function I cannot use this because again there is no conversion from `int` to `vector`, but I can use this. What it does? I am doing an `initializer_list` of length one having the value `7`, so this will invoke the `initializer_list` constructor and create a `vector` having the value `7`, I have given the type, underlying type as `double`, so this will implicitly convert that to `7.0`, so that is the use of the `initializer_list` in construction.

So, similarly, this will give a `vector` with value, with one element value `9.0`, similarly here, this will call `f()`, this `f()` with the list containing `9` and now if I want to see the subtle difference, suppose I am trying to define a `vector` of `vector` of `double`. So, I have a `vector` of `double` and I have a `vector` of that. So, every element, the initializing element for `vs` has to be a `vector` of `double` because that is element type.

So, I define `vector<double>` within parentheses `10`. What does that mean? That means an explicit construction of 10 elements, whereas if I use curly brace, it means an `initializer_list`, so it means a list containing one element `10`, so it is a explicit construction of a `vector` having one element with the value `10.0`, `10.0` because its `double`.

If I try to use `10`, I will get an error, because there is no implicit construction available, you can see that. We have said that the constructor is explicit, which means that unless you specify the constructor will not be invoked, here it was specified so it was got invoked. So, that is the property of the `initializer_list`.

(Refer Slide Time: 7:48)

Initializer Lists

- The function can access the `initializer_list` as an *immutable sequence*. For example:


```
void f(initializer_list<int> args) {
    for (auto p = args.begin(); p != args.end(); ++p) cout << *p << "\n";
}
```
- `std::initializer_list` stores initializer values in an array and offers these member functions:
 - `size` // # of elements in the array
 - `begin` // ptr to first array element
 - `end` // ptr to one-beyond-last array element
- A constructor that takes a single argument of type `std::initializer_list` is called an **initializer-list constructor**
- The STL containers, `string`, and `regex` have initializer-list constructors, assignment, etc. An initializer-list can be used as a range, for example, in a `range for` statement (TBD later).
- The initializer lists are part of the scheme for *uniform and general initialization*. They also prevent *narrowing*
- Usually initializing using `{}` is preferred instead of `()` unless:
 - The code is shared with a C++98 compiler or
 - There is a need to use `()` to call a non-`initializer_list` overloaded constructor (rare)

Programming in Modern C++ Partha Pratim Das M47.8

So, it is an immutable sequence that means that you cannot change it, everything is a constant there and it gives you three functions to know the size, the number of elements in the `initializer_list` array and a `begin()` and `end()` to do iteration for that. Now, the constructor of the kind we just saw where there is only one parameter which is a `initializer_list` type is called an `initializer_list` constructor.

Like we had default constructor, we had copy constructor, now we have a new kind of constructor called `initializer_list` constructor. So, several STL components now have `initializer_list` constructor as well. So they are a way to have uniform initialization across various different kinds of objects.

(Refer Slide Time: 8:48)

Initializer Lists: `std::initializer_list`

- `std::initializer_list` looks something like: [Initializer Lists in C++ - `std::initializer_list`]

```
typedef unsigned int size_t; // #include <bits/C++config.h>
namespace std {
    template<class _E> class initializer_list { // initializer list
    public:
        typedef _E value_type;
        typedef const _E& reference;
        typedef const _E& const_reference;
        typedef size_t size_type;
        typedef const _E* iterator;
        typedef const _E* const_iterator;
    private:
        iterator _M_array;
        size_type _M_len;
        // The compiler can call a private constructor
        // constexpr defines compile-time constant expressions - TBD later
        constexpr initializer_list(const_iterator __a, size_type __l): _M_array(__a), _M_len(__l) {}
    public:
        constexpr initializer_list() noexcept: _M_array(0), _M_len(0) {}
        constexpr size_type size() const noexcept { return _M_len; } // Number of elements
        constexpr const_iterator begin() const noexcept { return _M_array; } // First element
        constexpr const_iterator end() const noexcept { return begin()+size(); } // One past last element
    };
};
```

Partha Pratim Das


```
• std::initializer_list looks something like: [Initializer Lists in C++ - std::initializer_list]
typedef unsigned int size_t; // #include <bits/C++config.h>
namespace std {
    template<class _E> class initializer_list { // initializer list
    public:
        typedef _E value_type;
        typedef const _E& reference;
        typedef const _E& const_reference;
        typedef size_t size_type;
        typedef const _E* iterator;
        typedef const _E* const_iterator;
    private:
        iterator _M_array;
        size_type _M_len;
        // The compiler can call a private constructor
        // constexpr defines compile-time constant expressions - TBD later
        constexpr initializer_list(const_iterator __a, size_type __l): _M_array(__a), _M_len(__l) {}
    public:
        constexpr initializer_list() noexcept: _M_array(0), _M_len(0) {}
        constexpr size_type size() const noexcept { return _M_len; }
        constexpr const_iterator begin() const noexcept { return _M_array; }
        constexpr const_iterator end() const noexcept { return begin()+size(); }
    };
};
```

Now, if you just wonder as to how does the `initializer_list` look like in the `std`, the standard library namespace, so here is how it goes, this is underlying type and this is the class, these are the different types, so you can say, see that I said that `initializer_list` will actually keep it as an array, it will have to remember the initialization values, so it will keep it as an array. So, it is kind of a container.

So, you can see all different types that we had seen in the container are also defined for the `initializer_list`, beautiful uniformity as you see. And it has certainly private members to store the values and the size and it has a constructor which takes the iteration over `__a` and the size `__l` to actually do the construction, the constructor is made private because we do not expect to call it explicitly from the user code.

The compiler will call it and compiler can always call the private member and this actually is a constant expression. We will talk about constant expression later on, what it in general means is, a constant expression is one that can be evaluated at the compilation type and will not change after that, it becomes immutable after that. So, with that what you have in the public is a constructor, a default constructor which just constructs a null list.

You have (a size operator) a `size()` member function, a `begin` iterator and an `end` iterator, all of them are constant expression because it is, initialization is with constant so everything is computable at the compile time as a constant expression. So, this is what your basic `initializer_list` internally looks like. So, knowing that really helps.

(Refer Slide Time: 10:51)

```
#include <iostream>
#include <string>
#include <vector>
#include <initializer_list>

template <typename T> // T is the type of initializer_list elements
class MyClass { _std::vector<T> elems; /* vector to keep initialized values*/ public:
    // Default constructor
    MyClass(): elems({-1}) { std::cout << "Default Ctor: "; ShowElements(); }
    // Parameterized constructor
    MyClass(int b): elems({b}) { std::cout << "Parameterized Ctor: "; ShowElements(); }
    // Constructor using std::initializer_list
    MyClass(std::initializer_list<T> init_list): elems({init_list}) { // Using parenthesis
        // We can directly iterate on init_list as we do over elems
        std::cout << "Initializer List Ctor: "; ShowElements();
    }
    // Mixed Constructor
    MyClass(int i, std::initializer_list<T> init_list): elems({init_list}) { // Without using parenthesis
        std::cout << "Mixed Ctor: " << i << ", "; ShowElements();
    }
};

void ShowElements() /* Display the elements of elems */ { std::cout << "{ ";
    for (auto it = elems.begin(); it != elems.end(); ++it) std::cout << *it << ' ';
    std::cout << "}\n";
}
```

So, with that let us take an example with different variants. So, I am trying to define MyClass with a vector of T elements, my class is templated by T, I have a default constructor, so if I have a default constructor then I just initialize it with some arbitrary default value list of -1. I have a parameterized construction, which takes b, the list containing, I have an initializer_list construction which takes an initializer_list and copies it to elem.

I can have a mixed constructor also, which takes an integer and an initialization list and copies it, you can see that using this brace is, using this parenthesis is optional here. So, the four types of different constructors I have defined and I have given a ShowElements() function to really iterate over what I have initialized.

(Refer Slide Time: 12:07)

```
// template <typename T> class MyClass { std::vector<T> elems; public:
// MyClass(); // Default constructor
// MyClass(int b); // Parametrized constructor
// MyClass(std::initializer_list<T> init_list); // Constructor using std::initializer_list
// MyClass(int i, std::initializer_list<T> init_list); // Mixed Constructor
// void ShowElements();
// };
int main() {
MyClass<int> my_obj; // my_obj{} */ // Default Ctor: { -1 } ✓
MyClass<int> my_obj_i = MyClass<int>(500); // my_obj_i(500) */ // Parametrized Ctor: { 500 } ✓
MyClass<int> my_obj_il = MyClass<int>{500}; // my_obj_il{500} */ // Initializer List Ctor: { 500 }
// initializer_list objects: std::initializer_list<int>
auto init_list = { 1, 2, 3, 4, 5 };
// May use init_list for { init_list }
MyClass<int> my_obj_il_int = { init_list }; // Initializer List Ctor: { 1 2 3 4 5 }
// initializer_list object
std::initializer_list<std::string> il = { "Hello", "from", "PPD" };
// May use il for { il }
MyClass<std::string> my_obj_il_string = { il }; // Initializer List Ctor: { Hello from PPD }
MyClass<std::string> my_obj_m = { 5, { "Thank", "You" } }; // Mixed Ctor: 5, { Thank You }
}
```

```
// template <typename T> class MyClass { std::vector<T> elems; public:
// MyClass(); // Default constructor
// MyClass(int b); // Parametrized constructor
// MyClass(std::initializer_list<T> init_list); // Constructor using std::initializer_list
// MyClass(int i, std::initializer_list<T> init_list); // Mixed Constructor
// void ShowElements();
// };
int main() {
MyClass<int> my_obj; // my_obj{} */ // Default Ctor: { -1 }
MyClass<int> my_obj_i = MyClass<int>(500); // my_obj_i(500) */ // Parametrized Ctor: { 500 }
MyClass<int> my_obj_il = MyClass<int>{500}; // my_obj_il{500} */ // Initializer List Ctor: { 500 }
// initializer_list objects: std::initializer_list<int>
auto init_list = { 1, 2, 3, 4, 5 };
// May use init_list for { init_list }
MyClass<int> my_obj_il_int = { init_list }; // Initializer List Ctor: { 1 2 3 4 5 }
// initializer_list object
std::initializer_list<std::string> il = { "Hello", "from", "PPD" };
// May use il for { il }
MyClass<std::string> my_obj_il_string = { il }; // Initializer List Ctor: { Hello from PPD }
MyClass<std::string> my_obj_m = { 5, { "Thank", "You" } }; // Mixed Ctor: 5, { Thank You }
}
```

So, now if I try to use these constructors, this different construction I have repeated again here for your quick reference, so if I just do my_obj, it must construct things by default, so it default constructs and it has a list containing -1 that we did. If I do it with my int within parentheses 500, it picks the parameterized constructor and puts 500 as a list.

If I do class my int with braced initialization, so I have an initializer_list containing 500 that is b, that will call certainly my initializer_list constructor, it calls the initializer_list construction. So, you can see that you know both of these give me finally the same object or same, the object containing the same value, but use different constructors. You can also do it by separately creating the initializer_list.

So, what auto deduces (is it deduces) this type from the actual initializer_list and then you can use that to initialize the my object, the initializer_list constructor will be used. Similarly, you can do it for a string, again the initializer_list constructor is used, you can do it for a pair of integers and an initializer_list of string, so you will get mixed constructor called. So, this is how the constructors will map to the respective type and will be called.

(Refer Slide Time: 13:59)

Initializer Lists: Overload Resolution

- Constructor with `std::initializer_list` parameter prefers `{}`-delimited arguments

```

class Widget { public:
    Widget(std::initializer_list<double> values); // #1 ✓
    Widget(double value, double uncertainty); // #2 ✓
    ...
};
double d1, d2;
...
Widget w1 { d1, d2 }; // calls #1 ✓
Widget w2(d1, d2); // calls #2 ✓
    
```

- Choose carefully between `{}` and `()` when initializing objects!

```

template <class T, class Allocator = allocator<T> > // from the C++11 standard
class vector { public: ...
    vector(size_type n, const T& value, const Allocator& = Allocator());
    vector(initializer_list<T>, const Allocator& = Allocator());
    ...
};
std::vector<int> v1(10, 5); // v1.size() == 10, all values == 5
std::vector<int> v2{10, 5}; // v2.size() == 2, values == {10, 5}
    
```

Initializer Lists: Overload Resolution

- Constructor with `std::initializer_list` parameter prefers `{}`-delimited arguments

```

class Widget { public:
    Widget(std::initializer_list<double> values); // #1
    Widget(double value, double uncertainty); // #2
    ...
};
double d1, d2;
...
Widget w1 { d1, d2 }; // calls #1
Widget w2(d1, d2); // calls #2
    
```

- Choose carefully between `{}` and `()` when initializing objects!

```

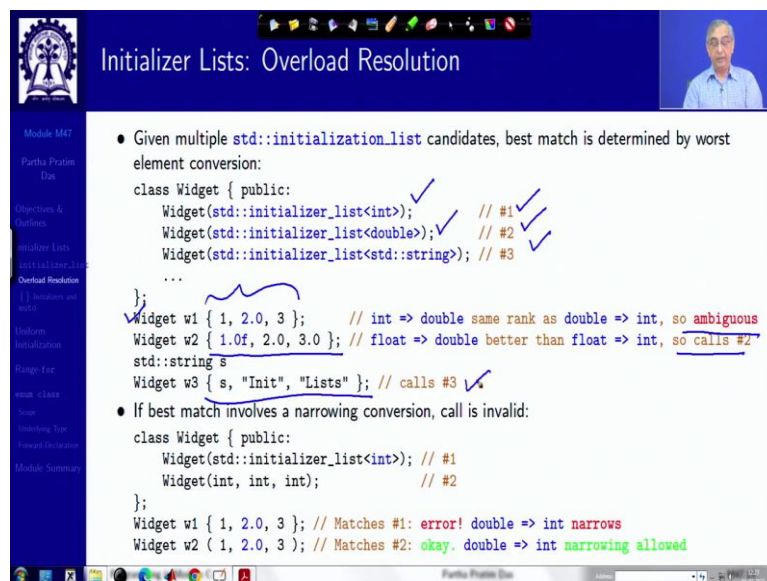
template <class T, class Allocator = allocator<T> > // from the C++11 standard
class vector { public: ...
    vector(size_type n, const T& value, const Allocator& = Allocator());
    vector(initializer_list<T>, const Allocator& = Allocator());
    ...
};
std::vector<int> v1(10, 5); // v1.size() == 10, all values == 5
std::vector<int> v2{10, 5}; // v2.size() == 2, values == {10, 5}
    
```

Now, naturally, since you have multiple constructors there will be overload issues. What do you, overload, so you must have understood this now, that if I have curly braces, then the initializer_list constructor if it is there will be preferred, so here I have one which is initializer_list constructor and one which is simple parameterize constructor. So, for the same d1, d2 if I define w1 as with curly braces, it will call 1, but with parentheses it will call 2.

Similarly, if I look at the vector class in the standard it has one parameterized constructor which takes the size and the default value, it says that construct a vector of size n each should be filled up with value and other takes just the initializer constructor, so if you call v1, if you construct v1 with 10 and 5 given in parentheses, this will take likeness of the first constructor. And therefore, what you will get?

10 will be considered as n, 5 will be considered as value, so you will get a vector of 10 elements each initialized with 5, whereas if you use curly braces, then your initializer_list constructor will be preferred. So, you will get a, consider this as a list containing 10 and 5, so your vector of int will now have two elements, size is 2, one is first, one is 10, second one is 05. So, there are, in terms of overload there are subtle differences, in terms of using parentheses and using curly brace, so be careful about that.

(Refer Slide Time: 15:58)



The slide is titled "Initializer Lists: Overload Resolution" and features a small video inset of a speaker in the top right corner. The main content consists of two bullet points and code snippets illustrating how C++ resolves constructor overloads based on the quality of conversion.

- Given multiple `std::initializer_list` candidates, best match is determined by worst element conversion:

```
class Widget { public:  
    Widget(std::initializer_list<int>); // #1 ✓  
    Widget(std::initializer_list<double>); // #2 ✓  
    Widget(std::initializer_list<std::string>); // #3 ✓  
    ...  
};  
Widget w1 { 1, 2.0, 3 }; // int => double same rank as double => int, so ambiguous ✓  
Widget w2 { 1.0f, 2.0, 3.0 }; // float => double better than float => int, so calls #2 ✓  
std::string s  
Widget w3 { s, "Init", "Lists" }; // calls #3 ✓
```
- If best match involves a narrowing conversion, call is invalid:

```
class Widget { public:  
    Widget(std::initializer_list<int>); // #1  
    Widget(int, int, int); // #2  
};  
Widget w1 { 1, 2.0, 3 }; // Matches #1: error! double => int narrows  
Widget w2 { 1, 2.0, 3 }; // Matches #2: okay. double => int narrowing allowed
```

Initializer Lists: Overload Resolution

- Given multiple `std::initializer_list` candidates, best match is determined by worst element conversion:


```
class Widget { public:
    Widget(std::initializer_list<int>); // #1
    Widget(std::initializer_list<double>); // #2
    Widget(std::initializer_list<std::string>); // #3
    ...
};
Widget w1 { 1, 2.0, 3 }; // int => double same rank as double => int, so ambiguous
Widget w2 { 1.0f, 2.0, 3.0 }; // float => double better than float => int, so calls #2
std::string s
Widget w3 { s, "Init", "Lists" }; // calls #3
```
- If best match involves a narrowing conversion, call is invalid:


```
class Widget { public:
    Widget(std::initializer_list<int>); // #1 ✓
    Widget(int, int, int); // #2 ✓
};
Widget w1 { 1, 2.0, 3 }; // Matches #1: error! double => int narrows ✗
Widget w2 { 1, 2.0, 3 }; // Matches #2: okay, double => int narrowing allowed
```

Initializer Lists: Overload Resolution

- Given multiple `std::initializer_list` candidates, best match is determined by worst element conversion:


```
class Widget { public:
    Widget(std::initializer_list<int>); // #1
    Widget(std::initializer_list<double>); // #2
    Widget(std::initializer_list<std::string>); // #3
    ...
};
Widget w1 { 1, 2.0, 3 }; // int => double same rank as double => int, so ambiguous
Widget w2 { 1.0f, 2.0, 3.0 }; // float => double better than float => int, so calls #2
std::string s
Widget w3 { s, "Init", "Lists" }; // calls #3
```
- If best match involves a narrowing conversion, call is invalid:


```
class Widget { public:
    Widget(std::initializer_list<int>); // #1
    Widget(int, int, int); // #2 ✓
};
Widget w1 { 1, 2.0, 3 }; // Matches #1: error! double => int narrows
Widget w2 { 1, 2.0, 3 }; // Matches #2: okay, double => int narrowing allowed
```

This, some more on this, like these are three `initializer_list` constructors are overloaded with different element types. Now, what will happen if I try to call construct `w1` with this, it is all are `initializer_list`, so the choice is to be between them, so if I want to do `w1`, where the second element is `2.0`, it has two choices, one is to convert `2.0` to `2` and call the first one. Or it can convert `1` and `3`, to `1.0` and `3.0` and call `2`.

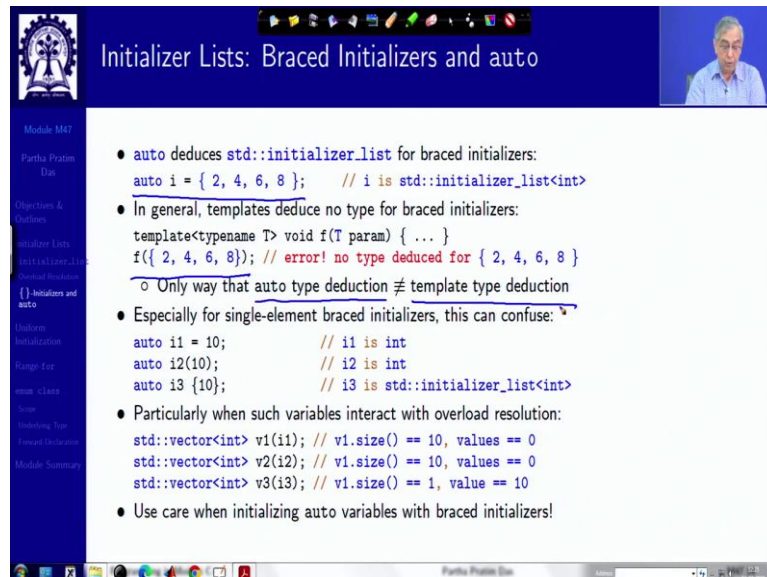
So, it is either `int` to `double` conversion or `double` to `int` conversion, both of these conversions of the same rank, remember the overload resolution strategy we had. So, this kind of a construction will be ambiguous. It will not compile. If you do this, so then also you have conversion issues because the first element is a `float` where the other two are of, type `2.0` and `3.0` are `double` that is C++ default type mechanism.

So, if we have to, so the only candidate is this one here where you need to do a float to double or if you have to do this then you have to do a float to int, double to int. Now, float to double has a better rank than float to int, so what will happen, it will call the constructor number 2, and if you just use strings, then it will call constructor number 3, obviously.

Now, let us say I do a overloading with some twist, I have an initializer_list and I have another constructor with 3 int parameterized, I am calling it with 1, 2.0 and 3. Now, what will this mean? This will mean that the only constructor to call is this, which means the double value will have to be converted to int, which is narrowing.

So, here we learnt a golden rule that initializer_list does not allow narrowing, does not allow narrowing implicitly, so this double to int will not be allowed and there will be an error, whereas if you use three values within curly brace, within parentheses, then you are actually by overload you are binding to the second constructor where the narrowing is allowed and that will compile and w2 will get constructed. So, this is the difference, you have to remember that narrowing is not allowed for braced initializer.

(Refer Slide Time: 19:10)



The image shows a presentation slide with a blue header and a white content area. The header contains the title 'Initializer Lists: Braced Initializers and auto' and a small video feed of a speaker. The content area lists several bullet points and code snippets. The first bullet point states that 'auto' deduces 'std::initializer_list' for braced initializers, followed by the code 'auto i = { 2, 4, 6, 8 }; // i is std::initializer_list<int>'. The second bullet point explains that templates do not deduce a type for braced initializers, showing a function 'f' and a call 'f({ 2, 4, 6, 8 });' with a red 'error!' message. The third bullet point notes that auto type deduction is not the same as template type deduction, especially for single-element braced initializers, and provides code for 'auto i1 = 10;', 'auto i2(10);', and 'auto i3 {10};'. The fourth bullet point discusses how these variables interact with overload resolution, showing vector initialization examples. The final bullet point is a warning to use care when initializing auto variables with braced initializers.

Module: M07
Partha Pratim Das
Objectives & Outcomes
Initializer Lists
Braced Initializers
{ }-Initializers and auto
Uniform Initialization
Range For
enum class
Scope
Modifiable Type
Forward Declaration
Module Summary

Initializer Lists: Braced Initializers and auto

- auto deduces `std::initializer_list` for braced initializers:
`auto i = { 2, 4, 6, 8 }; // i is std::initializer_list<int>`
- In general, templates deduce no type for braced initializers:
`template<typename T> void f(T param) { ... }
f({ 2, 4, 6, 8 }); // error! no type deduced for { 2, 4, 6, 8 }`
 - Only way that auto type deduction \neq template type deduction
- Especially for single-element braced initializers, this can confuse:
`auto i1 = 10; // i1 is int
auto i2(10); // i2 is int
auto i3 {10}; // i3 is std::initializer_list<int>`
- Particularly when such variables interact with overload resolution:
`std::vector<int> v1(i1); // v1.size() == 10, values == 0
std::vector<int> v2(i2); // v1.size() == 10, values == 0
std::vector<int> v3(i3); // v1.size() == 1, value == 10`
- Use care when initializing auto variables with braced initializers!

Initializer Lists: Braced Initializers and auto

- auto deduces `std::initializer_list` for braced initializers:


```
auto i = { 2, 4, 6, 8 }; // i is std::initializer_list<int>
```
- In general, templates deduce no type for braced initializers:


```
template<typename T> void f(T param) { ... }
f({ 2, 4, 6, 8 }); // error! no type deduced for { 2, 4, 6, 8 }
```

 - Only way that auto type deduction \neq template type deduction
- Especially for single-element braced initializers, this can confuse:


```
auto i1 = 10; // i1 is int
auto i2(10); // i2 is int
auto i3 {10}; // i3 is std::initializer_list<int>
```
- Particularly when such variables interact with overload resolution:


```
std::vector<int> v1(i1); // v1.size() == 10, values == 0
std::vector<int> v2(i2); // v1.size() == 10, values == 0
std::vector<int> v3(i3); // v1.size() == 1, value == 10
```
- Use care when initializing auto variables with braced initializers!

So, if you look at it with auto, this auto we have seen earlier also will deduce a `initializer_list` of `int`, you cannot directly use it in place of a template parameter. You would have expected to do that, but this is only one place where auto can go ahead with the type deduction, but template type deduction does not work.

So, auto can do more, template type detection will not be able to deduce, the `T` has a type, `std::initializer_list`, but of `int` but auto would be able to do that. So, with that there are more examples that you can use, you can use a single element with initialization, a parenthesized initialization, `initializer_list` and you can see what the effects would be. The first two will necessarily mean vectors of size 10 initialized with default value 0, and the last one will mean a vector of size one with value 10.

(Refer Slide Time: 20:25)

Uniform Initialization: Syntax and Semantics

- Module M47
- Partha Pratim Das
- Objectives & Outcomes
- Initialization Lists
- Default Member Functions
- Initializers and auto
- Uniform Initialization**
- Range for
- name class
- enum
- lifetime type
- Forward Declaration
- Module Summary

Sources:

- Uniform initialization syntax and semantics, isocpp.org
- Scott Meyers on C++
- Initialization and Constructors and member initializer lists, cppreference.com

Programming in Modern C++ Partha Pratim Das M47.15

Uniform Initialization Syntax

- C++03 offers multiple initialization forms
 - Initialization \neq assignment. For example, `const` objects can be initialized, not assigned
- Examples:

```
const int y(5);           // direct initialization syntax ✓
const int x = 5;         // copy initialization syntax ✓
int arr[] = { 5, 10, 15 }; // brace initialization ✓
struct Point1 { int x, y; };
const Point1 p1 = { 10, 20 }; // brace initialization ✓
class Point2 { public: Point2(int x, int y); };
const Point2 p2(10, 20); // function call syntax ✓
```
- Containers require another container:

```
int vals[] = { 10, 20, 30 };
const std::vector<int> cv(vals, vals+3); // init from another container
```
- Member and heap arrays are impossible:

```
class Widget {
public: Widget(): data(???) {}
private: const int data[5]; // not initializable
};
const float * pData = new const float[4]; // not initializable
```

The screenshot shows a presentation slide titled "Uniform Initialization Syntax". The slide content is as follows:

- C++03 offers multiple initialization forms
 - Initialization \neq assignment. For example, `const` objects can be initialized, not assigned
- Examples:


```
const int y(5);           // direct initialization syntax
const int x = 5;         // copy initialization syntax
int arr[] = { 5, 10, 15 }; // brace initialization
struct Point1 { int x, y; };
const Point1 p1 = { 10, 20 }; // brace initialization
class Point2 { public: Point2(int x, int y); };
const Point2 p2(10, 20); // function call syntax
```
- Containers require another container:


```
int vals[] = { 10, 20, 30 };
const std::vector<int> cv(vals, vals+3); // init from another container
```
- Member and heap arrays are impossible:


```
class Widget {
public: Widget(): data(???) {}
private: const int data[5]; // not initializable
};
const float * pData = new const float[4]; // not initializable
```

So, this was the `initializer_list`. The question is why are we trying to do that because what we want is we want to make initialization uniform, initialization syntax and semantics uniform. So, is it non-uniform? The answer is yes. If you look at C++03 there are multiple ways to do initialization. And remind you initialization is not same as assignment, like constant objects can be initialized, they cannot be assigned.

Initialization is something which happens when you are defining the variable, so these are the choices you have in C++03, 98. This is you can have a direct initialization syntax by parentheses, by using initialization symbol you can have copy initialization, you can have braced initialization for array, you can have braced initialization for a structure, you can have a function called syntax for calling the constructor.

You can also initialize one container from another. So, these are all different types and depending on the context we have got used to different types of syntax, somewhere it is braced, somewhere it is not braced, but that still lives out certain things which we cannot initialize. For example, if your object has a ..., you want to have a array of ..., array data of size 5 containing constant integers.

Now, there is no way to initialize this. There is no way to initialize arrays in this context where it is `const`. You cannot do this. Similarly, if you are trying to dynamically allocate an array you cannot initialize that, so we say that if I have to dynamically allocate a objects, array of the objects of a user defined type, then that type must provide default construction so that I do not need to do an initialization.

(Refer Slide Time: 22:52)

Uniform Initialization Syntax

- Brace initialization syntax now allowed *everywhere*:

```
const int val1 {5};  
const int val2 {5};  
int a[] { 1, 2, val1, val1+val2 };  
struct Point1 { int x, y };  
const Point1 p1 {10, 20};  
class Point2 { public: Point2(int x, int y); };  
const Point2 p2 {10, 20};  
const std::vector<int> cv { a[0], 20, val2 }; // calls Point2 ctor  
class Widget {  
public: Widget(): data {1, 2, a[3], 4, 5} {}  
private: const int data[5];  
};  
const float * pData = new const float[4] { 1.5, val1-val2, 3.5, 4.5 };
```
- Really, *everywhere*:

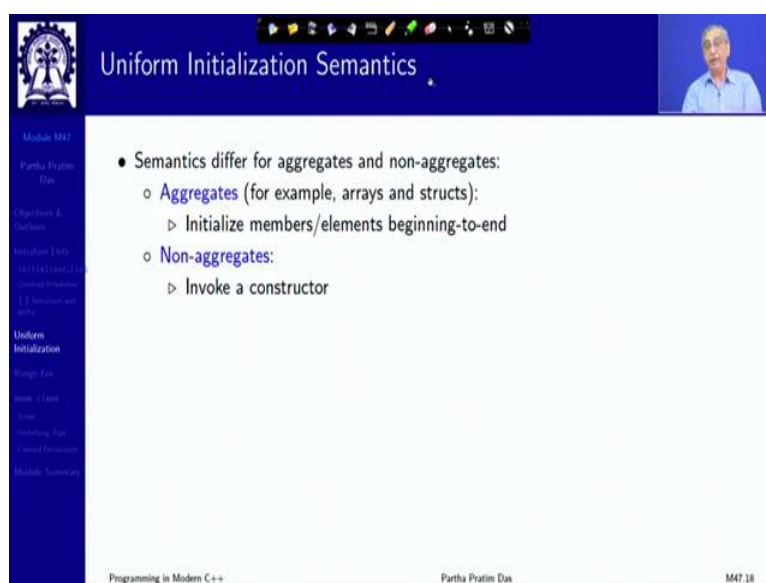
```
Point2 makePoint() { return { 0, 0 }; } // return expression; calls Point2 ctor  
void f(const std::vector<int>& v); // function declaration  
f({ val1, val2, 10, 20, 30 }); // function argument
```

Now, this is in terms of syntax, the braced initialization is now allowed everywhere, so you can use it simply like this or you can use it in the way you are doing in terms of the array. You can obviously, you have to make sure that every element of the initializer_list is homogeneous and is a constant expression, so this is a constant variable val1, constant variable, so I am using those in the initialization of the array.

I can use this as it is for a structure, I can use it for calling a constructor, I can use it to initialize vector, everywhere, braced initialization can be used, and the interesting thing is it can be used now even if I have a constant integer array or something like that, I can just do a braced initialization here. I can also do braced initialization with dynamic allocation.

So, it can be used really everywhere, for example, I can return a braced initialized value from a function, so this is in the context of say class Point2, this will call the constructor of the class, I can pass it to a function, so everywhere braced initialization can be used, so that makes things really uniform.

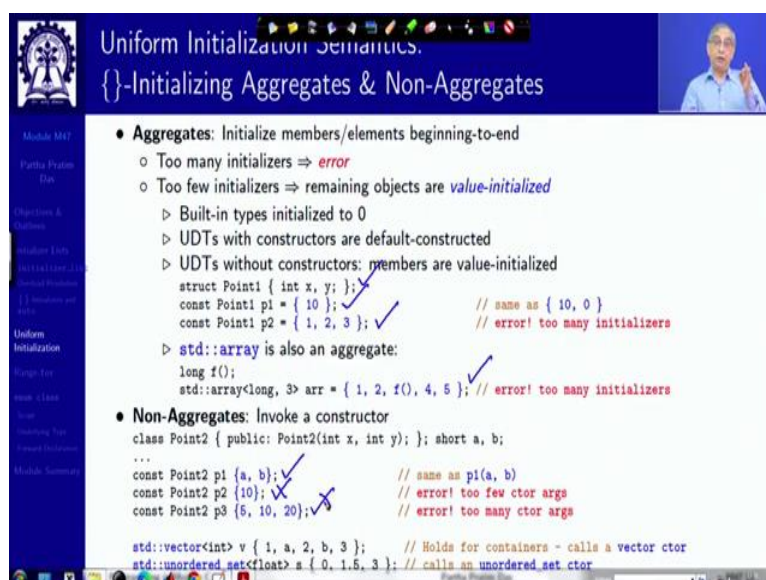
(Refer Slide Time: 24:27)



Slide 1: Uniform Initialization Semantics

- Semantics differ for aggregates and non-aggregates:
 - **Aggregates** (for example, arrays and structs):
 - ▷ Initialize members/elements beginning-to-end
 - **Non-aggregates**:
 - ▷ Invoke a constructor

Programming in Modern C++ Parthiv Pratim Das M47.18



Slide 2: {}-Initializing Aggregates & Non-Aggregates

- **Aggregates**: Initialize members/elements beginning-to-end
 - Too many initializers ⇒ **error**
 - Too few initializers ⇒ remaining objects are **value-initialized**
 - ▷ Built-in types initialized to 0
 - ▷ UDTs with constructors are default-constructed
 - ▷ UDTs without constructors: members are value-initialized
- **Non-Aggregates**: Invoke a constructor

```
struct Point1 { int x, y; };
const Point1 p1 = { 10 }; // same as { 10, 0 }
const Point1 p2 = { 1, 2, 3 }; // error! too many initializers

std::array<long, 3> arr = { 1, 2, f(), 4, 5 }; // error! too many initializers

class Point2 { public: Point2(int x, int y); }; short a, b;
...
const Point2 p1 { a, b }; // same as p1(a, b)
const Point2 p2 { 10 }; // error! too few ctor args
const Point2 p3 { 6, 10, 20 }; // error! too many ctor args

std::vector<int> v { 1, a, 2, b, 3 }; // Holds for containers - calls a vector ctor
std::unordered_set<float> s { 0, 1.5, 3 }; // calls an unordered_set ctor
```

Now, that was about the syntax, in terms of semantics what we have, we have some difference between aggregate types and non-aggregate type. If it is an aggregate type, like it is an array or that kind of a container something then you have to provide the initialization and those will be initialized member by member. If you have too many initializers, then you will have an error, if you have too few, then you will have default values.

And for built-in types that is 0. So, you can see that I have a structure x, y here, so if I initialize it with 10 the second value will be taken as 0, if I try to initialize with 1, 2, 3 it will give me an error, similar thing for an array, `std::array` also we have not done it yet. It is pretty much similar to array but it is a new container here.

But if it, if I have a non-aggregate type that is which is not an array, then I can have not an array or a structure so to say, then it will invoke the constructor, so in all these cases it will try to invoke the constructor. In this case it will go through fine and in this cases this will not work because the number of parameters do not match. So, this is a construction process, this is not just the initialization, therefore, for non-aggregates the number of parameters have to match which was not a requirement right here.

(Refer Slide Time: 26:17)

Uniform Initialization Semantics

- Brace-initialized variables may use =:

```
const int val1 = {5};  
const int val2 = {5};  
int a[] = { 1, 2, val1, val1+val2 };  
struct Point1 { ... };  
const Point1 p1 = {10, 20};  
class Point2 { ... };  
const Point2 p2 = {10, 20};  
const std::vector<int> cv = { a[0], 20, val2 };
```
- Other uses of brace initialization cannot:

```
class Widget {  
public: Widget(): data = {1, 2, a[3], 4, 5} {} // error!  
private: const int data[5];  
};  
const float *pData = new const float[4] = { 1.5, val1-val2, 3.5, 4.5 }; // error!  
Point2 makePoint() { return = { 0, 0 }; } // error!  
  
void f(const std::vector<int>& v);  
f( = { val1, val2, 10, 20, 30 }); // error!
```

So, there are several places where you can also use the equality symbol, the initialization symbol as in these cases. I will not go through each one of them with a minus syntax issues, but these are places where you cannot use the initialization symbol equal to, so these are error.

(Refer Slide Time: 26:47)

Uniform Initialization Semantics

- And `T var = expr` syntax cannot call **explicit** constructors:

```
class Widget {
public:
    explicit Widget(int);
    ...
};
Widget w1(10); // okay, direct init: explicit ctor callable
Widget w2{10}; // okay, direct init: explicit ctor callable
Widget w3 = 10; // error! copy init: explicit ctor not callable
Widget w4 = {10}; // error! copy init: explicit ctor not callable
```
- Develop the habit of using brace initialization without `=`
- Uniform initialization syntax a feature *addition*, not a replacement
 - Almost all initialization code valid in C++03 remains valid
 - ▷ Rarely a need to modify existing code

So, basically the core idea is as you get used to this, do not, I mean, learn not to use the initialization symbol anymore, just use the braced initializer or the parentheses if you have to do that, so this is small examples of that.

(Refer Slide Time: 27:08)

Uniform Initialization Semantics: {}-Initialization and Implicit Narrowing

- Sole exception: implicit narrowing
 - C++03 *allows it via brace initialization*, C++11 *does not*

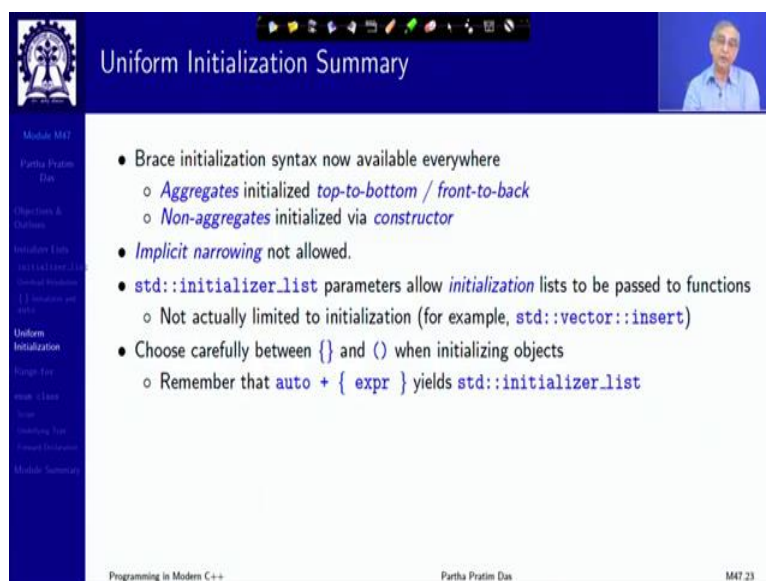
```
struct Point { int x, y; };
Point p1 = { 1, 2.5 }; // fine in C++03
// implicit double => int conversion
// error in C++11
Point p2 = { 1, static_cast<int>(2.5) }; // fine in both C++03 and C++11
```
- Direct constructor calls and brace initialization thus differ subtly:

```
class Widget {
public: Widget(unsigned u);
    ...
};
int i;
...
Widget w1(i); // okay, implicit int => unsigned
Widget w2{i}; // error! int => unsigned narrows
unsigned u;
Widget w3(u); // fine
Widget w4{u}; // also fine, same as w3's init.
```

Programming in Modern C++ Partha Pratim Das M47.22

So, the implicit narrowing, there is another example on implicit narrowing. I will leave it for your self-study.

(Refer Slide Time: 27:19)

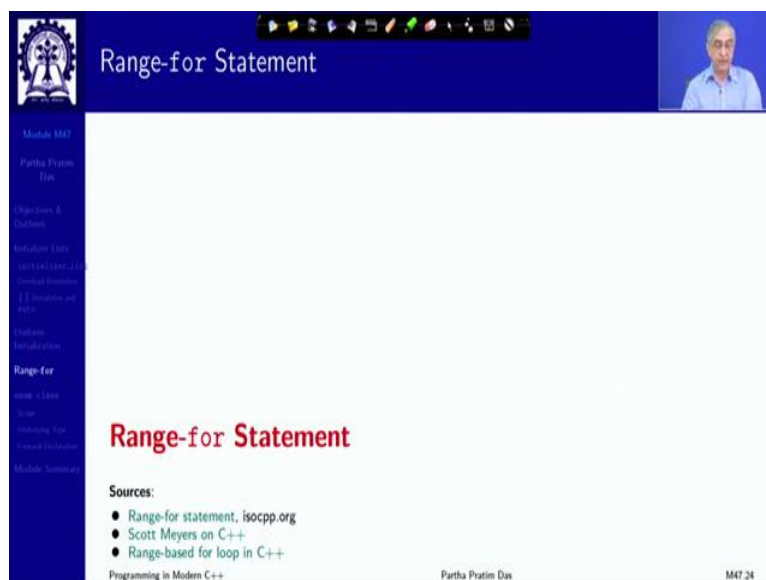


The slide is titled "Uniform Initialization Summary" and features a blue header with a logo on the left and a small video inset of the presenter on the right. The main content area is white with a list of bullet points. The footer contains the text "Programming in Modern C++", "Partha Pratim Das", and "M47.23".

- Brace initialization syntax now available everywhere
 - *Aggregates* initialized *top-to-bottom / front-to-back*
 - *Non-aggregates* initialized via *constructor*
- *Implicit narrowing* not allowed.
- `std::initializer_list` parameters allow *initialization* lists to be passed to functions
 - Not actually limited to initialization (for example, `std::vector::insert`)
- Choose carefully between `{}` and `()` when initializing objects
 - Remember that `auto + { expr }` yields `std::initializer_list`

In summary braced initialization is available everywhere, aggregates will initialize top to bottom, front to back, non-aggregates will initialize via constructor and implicit narrowing is not allowed and initializer_list parameters will allow the initialization list to be passed to functions.

(Refer Slide Time: 27:42)



The slide is titled "Range-for Statement" and features a blue header with a logo on the left and a small video inset of the presenter on the right. The main content area is white with the title "Range-for Statement" in red. Below the title is a "Sources:" section with a list of three bullet points. The footer contains the text "Programming in Modern C++", "Partha Pratim Das", and "M47.24".

Range-for Statement

Sources:

- Range-for statement, isocpp.org
- Scott Meyers on C++
- Range-based for loop in C++

Ways of traversing a vector: Recap (Module 44)

- Let us revisit ways for the traversal of a vector as a sample container:


```
vector<int> v;

// subscript style ✓
for(int i = 0; i < v.size(); ++i) { /* use v[i] */ } // native int for size
for(vector<T>::size_type i = 0; i < v.size(); ++i) { /* use v[i] */ } // correct size_type

// iterator style
for(vector<T>::iterator p = v.begin(); p != v.end(); ++p) { /* use *p */ } // verbose
for(vector<T>::value_type x : v) { /* use x read-only */ } // range for [C++11]
for(auto& x : v) { /* use x read-write */ } // range for [C++11]
```
- Comparing *subscript* and *iterator* styles:
 - The *subscript style* is used in essentially every language
 - The *subscript style* does not work for lists and non-linear data structures (in C++ and in most languages)
 - The *iterator style* is used in C (pointers only) and C++
 - The *iterator style* is used for standard library algorithms
 - While both styles work for vectors, *iterator style* is more generic – works for all sequences

Ways of traversing a vector: Recap (Module 44)

- Let us revisit ways for the traversal of a vector as a sample container:


```
vector<int> v;

// subscript style
for(int i = 0; i < v.size(); ++i) { /* use v[i] */ } // native int for size
for(vector<T>::size_type i = 0; i < v.size(); ++i) { /* use v[i] */ } // correct size_type

// iterator style
for(vector<T>::iterator p = v.begin(); p != v.end(); ++p) { /* use *p */ } // verbose ✓
for(vector<T>::value_type x : v) { /* use x read-only */ } // range for [C++11]
for(auto& x : v) { /* use x read-write */ } // range for [C++11]
```
- Comparing *subscript* and *iterator* styles:
 - The *subscript style* is used in essentially every language
 - The *subscript style* does not work for lists and non-linear data structures (in C++ and in most languages)
 - The *iterator style* is used in C (pointers only) and C++
 - The *iterator style* is used for standard library algorithms
 - While both styles work for vectors, *iterator style* is more generic – works for all sequences

Ways of traversing a vector: Recap (Module 44)

- Let us revisit ways for the traversal of a vector as a sample container:


```
vector<int> v;

// subscript style
for(int i = 0; i < v.size(); ++i) { /* use v[i] */ } // native int for size
for(vector<T>::size_type i = 0; i < v.size(); ++i) { /* use v[i] */ } // correct size_type

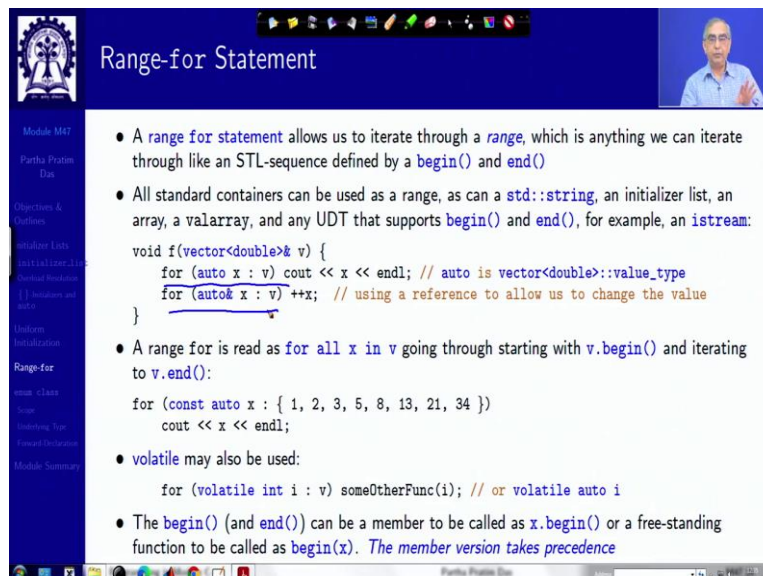
// iterator style
for(vector<T>::iterator p = v.begin(); p != v.end(); ++p) { /* use *p */ } // verbose
for(vector<T>::value_type x : v) { /* use x read-only */ } // range for [C++11]
for(auto& x : v) { /* use x read-write */ } // range for [C++11]
```
- Comparing *subscript* and *iterator* styles:
 - The *subscript style* is used in essentially every language
 - The *subscript style* does not work for lists and non-linear data structures (in C++ and in most languages)
 - The *iterator style* is used in C (pointers only) and C++
 - The *iterator style* is used for standard library algorithms
 - While both styles work for vectors, *iterator style* is more generic – works for all sequences

Now, with this let us look at some of the other related features, one is range-for. We have seen this in module 44, when we are discussing about vector. Range-for talks about how to iterate over an entire data structure. So, we saw different styles that we could use a subscript style for a vector, either using the native int for size or the actual size type of the vector or we can use the iterator style, which is the verbose actual iterated style.

But C++11 allows us to do short forms of this. What were you saying in this one? We are saying that the value type, take the value type as x and : v, it means, this means this, so it will allow you to go over this, only thing is this will be read only. You can even use a shorter form, you can just say auto& x or auto x.

If you do auto& x this will be a reference, so it will, that reference will allow you to also make changes to the elements of the vector. The subscript style is common but the iterator style has power because it can be used not only in vectors, but in any other container which has support for iterators.

(Refer Slide Time: 29:36)



The image shows a presentation slide titled "Range-for Statement" with a blue header and a white content area. A small video inset of a speaker is in the top right. The slide contains a list of bullet points and code examples explaining the range-for statement.

Range-for Statement

- A **range for** statement allows us to iterate through a *range*, which is anything we can iterate through like an STL-sequence defined by a `begin()` and `end()`
- All standard containers can be used as a range, as can a `std::string`, an initializer list, an array, a `valarray`, and any UDT that supports `begin()` and `end()`, for example, an `istream`:

```
void f(vector<double>& v) {  
    for (auto x : v) cout << x << endl; // auto is vector<double>::value_type  
    for (auto& x : v) ++x; // using a reference to allow us to change the value  
}
```
- A range for is read as **for all x in v** going through starting with `v.begin()` and iterating to `v.end()`:

```
for (const auto x : { 1, 2, 3, 5, 8, 13, 21, 34 })  
    cout << x << endl;
```
- **volatile** may also be used:

```
for (volatile int i : v) someOtherFunc(i); // or volatile auto i
```
- The `begin()` (and `end()`) can be a member to be called as `x.begin()` or a free-standing function to be called as `begin(x)`. *The member version takes precedence*

Range-for Statement

- A *range for* statement allows us to iterate through a *range*, which is anything we can iterate through like an STL-sequence defined by a `begin()` and `end()`
- All standard containers can be used as a range, as can a `std::string`, an initializer list, an array, a `valarray`, and any UDT that supports `begin()` and `end()`, for example, an `istream`:

```
void f(vector<double>& v) {
    for (auto x : v) cout << x << endl; // auto is vector<double>::value_type
    for (auto& x : v) ++x; // using a reference to allow us to change the value
}
```

- A range for is read as *for all x in v* going through starting with `v.begin()` and iterating to `v.end()`:

```
for (const auto x : { 1, 2, 3, 5, 8, 13, 21, 34 })
    cout << x << endl;
```

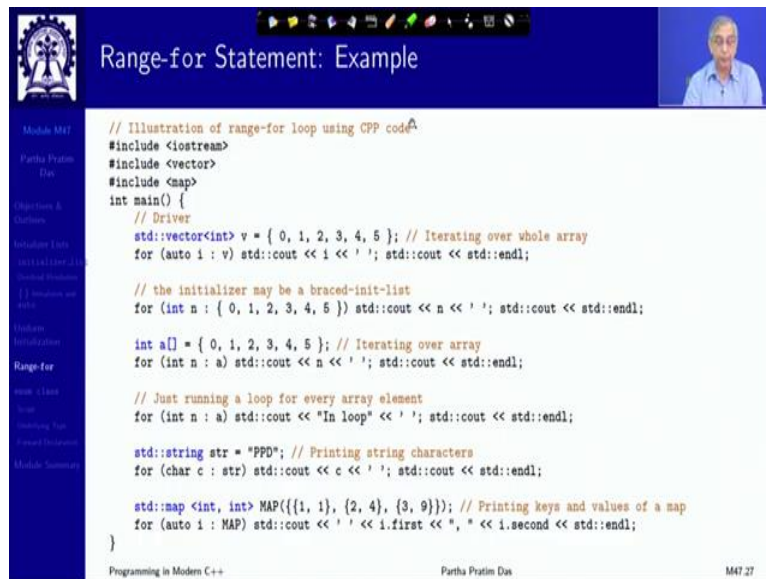
- `volatile` may also be used:


```
for (volatile int i : v) someOtherFunc(i); // or volatile auto i
```
- The `begin()` (and `end()`) can be a member to be called as `x.begin()` or a free-standing function to be called as `begin(x)`. *The member version takes precedence*

So, some, so this range statement basically allows you to do this. These are the, this is just that what you need to learn is you can use `auto x` : the container you want to traverse or `auto&` that and `auto&` that will allow you to write, `auto` will only be read. And this is possible, this range-for, this is possible provided the element on which you are trying to go over supports `begin` and `end` either as member function or as free-standing function.

Without that this will not be possible to perform this. It is a very convenient way to write very, very compact and uniform code and makes more code generic in that way, because now you are not even having to write what is the type of that iterator at all. You can, not `begin`, `end`, anything, of course, it works only when you have to iterate the entire data structure.

(Refer Slide Time: 30:45)



The slide is titled "Range-for Statement: Example" and features a video thumbnail of the presenter in the top right corner. The main content is a code block illustrating various uses of the range-for loop in C++:

```
// Illustration of range-for loop using CPP code
#include <iostream>
#include <vector>
#include <map>
int main() {
    // Driver
    std::vector<int> v = { 0, 1, 2, 3, 4, 5 }; // Iterating over whole array
    for (auto i : v) std::cout << i << ' '; std::cout << std::endl;

    // the initializer may be a braced-init-list
    for (int n : { 0, 1, 2, 3, 4, 5 }) std::cout << n << ' '; std::cout << std::endl;

    int a[] = { 0, 1, 2, 3, 4, 5 }; // Iterating over array
    for (int n : a) std::cout << n << ' '; std::cout << std::endl;

    // Just running a loop for every array element
    for (int n : a) std::cout << "In loop" << ' '; std::cout << std::endl;

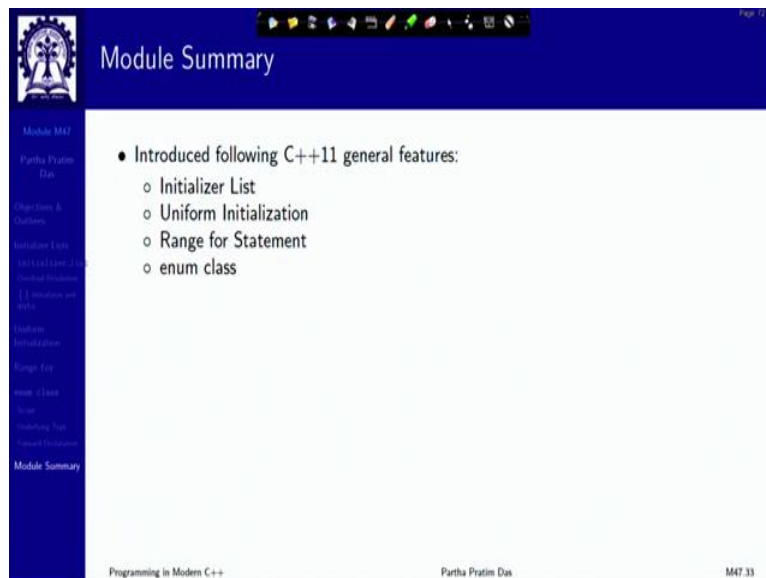
    std::string str = "PPD"; // Printing string characters
    for (char c : str) std::cout << c << ' '; std::cout << std::endl;

    std::map<int, int> MAP({{1, 1}, {2, 4}, {3, 9}}); // Printing keys and values of a map
    for (auto i : MAP) std::cout << ' ' << i.first << ", " << i.second << std::endl;
}
```

The slide footer includes "Programming in Modern C++", "Partha Pratim Das", and "M47.27".

And, so this is, there are, these are some examples for you to study at home and get conversant with the use of range and with that we will we will close on the discussion of this module.

(Refer Slide Time: 31:05)



The slide is titled "Module Summary" and lists the following C++11 general features:

- Introduced following C++11 general features:
 - Initializer List
 - Uniform Initialization
 - Range for Statement
 - enum class

The slide footer includes "Programming in Modern C++", "Partha Pratim Das", and "M47.33".

In this we have introduced three C++ general features, `initializer_list`, uniform initialization and range-for statement. Thank you very much for your attention and will meet in the next module.