


Programming in Modern C++
Professor. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 54
C++11 and beyond: Class Features

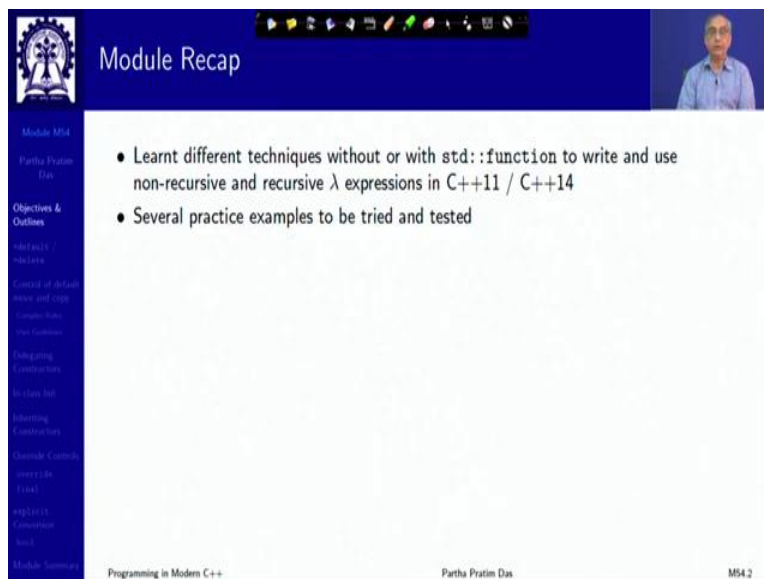
(Refer Slide Time: 00:32)



The slide features a dark blue header with the IIT Kharagpur logo on the left and navigation icons on the right. A vertical sidebar on the left contains a list of module topics. The main content area has a dark blue box at the top with the title 'Programming in Modern C++' and subtitle 'Module M54: C++11 and beyond: Class Features'. Below this, the presenter's name 'Partha Pratim Das' and affiliation 'Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur' are listed, along with the 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 'M54 1'.

Welcome to Programming in Modern C++. We are in Week 11 and we are going to discuss M54, Module 54.

(Refer Slide Time: 00:36)



The slide features a dark blue header with the IIT Kharagpur logo on the left, the title 'Module Recap' in the center, and a small video inset of the presenter on the right. The vertical sidebar on the left is the same as in the previous slide. The main content area contains two bullet points: 'Learnt different techniques without or with `std::function` to write and use non-recursive and recursive λ expressions in C++11 / C++14' and 'Several practice examples to be tried and tested'. The footer includes 'Programming in Modern C++', 'Partha Pratim Das', and 'M54 2'.

In the last module we have talked about different techniques using or without using `std::function` to write and use non-recursive as well as recursive lambda expressions in C++11 and 14, we have learnt about generic lambdas, we have learnt about generic capture and we have left with the several practice examples that you should try and test out.

(Refer Slide Time: 01:05)

Module Objectives

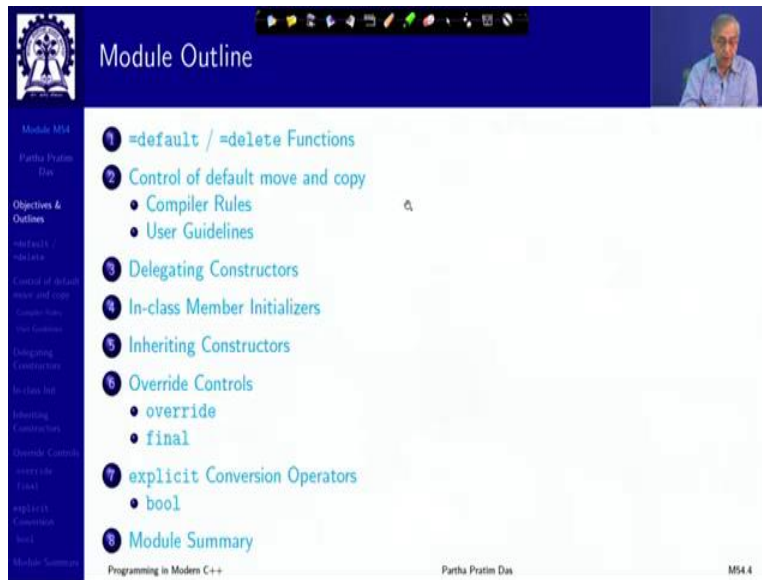
- Introducing class features in C++11:
 - =default and =delete
 - Control of default move and copy
 - Delegating constructors
 - In-class member initializers
 - Inherited constructors
 - Override controls: override & final
 - Explicit conversion operators
- These features enhance OOP, generic programming, readability, type-safety, and performance in C++11

With that, actually, the general features of C++11 we have now completed discussing and in the current module we will discuss a set of features which are related to the class. Class we learnt about, different things about the class, data members, member functions, constructor, destructor, all of that, inheritance all those.

So, the whole on those whole bunch of things that apply to classes, these features are extensions on those, modifications on those and unlike say rvalue semantics, rvalue reference and move semantics or like lambda which are big single feature of enhancement in C++11 these features are small features which kind of are standalone, but helps to enhance the object orientation in several ways. It specifically takes better generic programming. It improves readability.

Some features are just given to improve the readability. Some features improved the type safety, some improve the performance and mix of those. So, you will not see a very coherent discussion on examples of all of these features being used in the same place, but once you get comfortable with them in the subsequent module, you will see different of these being used at different places.

(Refer Slide Time: 02:45)



The slide is titled "Module Outline" and features a navigation menu on the left side. The menu items are: Module M4, Partha Pratim Das, Objectives & Outlines, =default / =delete, Control of default move and copy, Compiler Rules, User Guidelines, Delegating Constructors, In-class Member Initializers, Inheriting Constructors, Override Controls, explicit Conversion Operators, and Module Summary. The main content area contains a numbered list of topics: 1. =default / =delete Functions, 2. Control of default move and copy (with sub-points: Compiler Rules, User Guidelines), 3. Delegating Constructors, 4. In-class Member Initializers, 5. Inheriting Constructors, 6. Override Controls (with sub-points: override, final), 7. explicit Conversion Operators (with sub-point: bool), and 8. Module Summary. The footer includes "Programming in Modern C++", "Partha Pratim Das", and "MS4.4".

Module Outline

- 1. =default / =delete Functions
- 2. Control of default move and copy
 - Compiler Rules
 - User Guidelines
- 3. Delegating Constructors
- 4. In-class Member Initializers
- 5. Inheriting Constructors
- 6. Override Controls
 - `override`
 - `final`
- 7. explicit Conversion Operators
 - `bool`
- 8. Module Summary

Programming in Modern C++ Partha Pratim Das MS4.4



The slide is titled "default / delete Functions" and features a navigation menu on the left side. The menu items are: Module M4, Partha Pratim Das, Objectives & Outlines, =default / =delete, Control of default move and copy, Compiler Rules, User Guidelines, Delegating Constructors, In-class Member Initializers, Inheriting Constructors, Override Controls, explicit Conversion Operators, and Module Summary. The main content area contains a "Sources:" section with a list of references: =default and =delete, isocpp.org; C++ Class and Preventing Object Copy, ariya.io, 2015; C++ Core Guidelines, github.com; C.20: If you can avoid defining default operations, do; C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all; C.22: Make default operations consistent; and An Overview of the New C++ (C++11/14), Scott Meyers Training Courses. Below the sources, the text "default / delete Functions" is displayed in a large, bold, red font. The footer includes "Programming in Modern C++", "Partha Pratim Das", and "MS4.5".

default / delete Functions

Sources:

- =default and =delete, isocpp.org
- C++ Class and Preventing Object Copy, ariya.io, 2015
- C++ Core Guidelines, github.com
 - C.20: If you can avoid defining default operations, do
 - C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all
 - C.22: Make default operations consistent
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

default / delete Functions

Programming in Modern C++ Partha Pratim Das MS4.5

=default and =delete

- The idiom of *prohibiting copying* (for C++03 recall **Module 25**) can now be expressed directly:


```
class X { // ...
    X& operator=(const X&) = delete; // Disallow copying
    X(const X&) = delete;
};
```
- Conversely, we can also say *explicitly* that we want *default copy behavior*:


```
class Y { // ...
    Y& operator=(const Y&) = default; // default copy semantics
    Y(const Y&) = default;
};
```
- Explicitly writing out the default by hand is good for *readability*, but it has two *drawbacks*:
 - it sometimes generates *less efficient code* than the compiler-generated default would, and
 - it prevents types from *being considered PODs*
- The `=default` mechanism can be used for any function that has a default
- The `=delete` mechanism can be used for any function like to eliminate an undesired conversion:


```
struct Z { // ...
    Z(long long); // can initialize with a long long
    Z(long) = delete; // but not anything smaller
};
```


=default and =delete

- The idiom of *prohibiting copying* (for C++03 recall **Module 25**) can now be expressed directly:



```
class X { // ...
    X& operator=(const X&) = delete; // Disallow copying
    X(const X&) = delete;
};
```
- Conversely, we can also say *explicitly* that we want *default copy behavior*:


```
class Y { // ...
    Y& operator=(const Y&) = default; // default copy semantics
    Y(const Y&) = default;
};
```
- Explicitly writing out the default by hand is good for *readability*, but it has two *drawbacks*:
 - it sometimes generates *less efficient code* than the compiler-generated default would, and
 - it prevents types from *being considered PODs*
- The `=default` mechanism can be used for any function that has a default
- The `=delete` mechanism can be used for any function like to eliminate an undesired conversion:


```
struct Z { // ...
    Z(long long); // can initialize with a long long
    Z(long) = delete; // but not anything smaller
};
```



=default and =delete



Module 104

Partha Pratim Das

Algorithms & Solvers

=default / =delete

Control of default
move and copy
constructors

Designing Constructors

Access list

Initializing Constructors

Overriding Constructors

Final

explicit Construction


Final

- The idiom of *prohibiting copying* (for C++03 recall **Module 25**) can now be expressed directly:



```
class X { // ...
    X& operator=(const X&) = delete; // Disallow copying
    X(const X&) = delete;
};
```
- Conversely, we can also say *explicitly* that we want *default copy behavior*:


```
class Y { // ...
    Y& operator=(const Y&) = default; // default copy semantics
    Y(const Y&) = default;
};
```
- Explicitly writing out the default by hand is good for *readability*, but it has two *drawbacks*:
 - it sometimes generates *less efficient code* than the compiler-generated default would, and
 - it prevents types from *being considered PODs*
- The `=default` mechanism can be used for any function that has a default
- The `=delete` mechanism can be used for any function like to eliminate an undesired conversion:


```
struct Z { // ...
    Z(long long); // can initialize with a long long
    Z(long) = delete; // but not anything smaller
};
```



=default and =delete



Module 104

Partha Pratim Das

Algorithms & Solvers

=default / =delete

Control of default
move and copy
constructors

Designing Constructors

Access list

Initializing Constructors

Overriding Constructors

Final

explicit Construction

Final

- The idiom of *prohibiting copying* (for C++03 recall **Module 25**) can now be expressed directly:


```
class X { // ...
    X& operator=(const X&) = delete; // Disallow copying
    X(const X&) = delete;
};
```
- Conversely, we can also say *explicitly* that we want *default copy behavior*:


```
class Y { // ...
    Y& operator=(const Y&) = default; // default copy semantics
    Y(const Y&) = default;
};
```
- Explicitly writing out the default by hand is good for *readability*, but it has two *drawbacks*:
 - it sometimes generates *less efficient code* than the compiler-generated default would, and
 - it prevents types from *being considered PODs*
- The `=default` mechanism can be used for any function that has a default
- The `=delete` mechanism can be used for any function like to eliminate an undesired conversion:


```
struct Z { // ...
    Z(long long); // can initialize with a long long
    Z(long) = delete; // but not anything smaller
};
```

=default and =delete

- The idiom of *prohibiting copying* (for C++03 recall **Module 25**) can now be expressed directly:


```
class X { // ...
    X& operator=(const X&) = delete; // Disallow copying
    X(const X&) = delete;
};
```
- Conversely, we can also say *explicitly* that we want *default copy behavior*:


```
class Y { // ...
    Y& operator=(const Y&) = default; // default copy semantics
    Y(const Y&) = default;
};
```
- Explicitly writing out the default by hand is good for *readability*, but it has two *drawbacks*:
 - it sometimes generates *less efficient code* than the compiler-generated default would, and
 - it prevents types from *being considered PODs*
- The `=default` mechanism can be used for any function that has a default
- The `=delete` mechanism can be used for any function like to eliminate an undesired conversion:


```
struct Z { // ...
    Z(long long); // can initialize with a long long
    Z(long) = delete; // but not anything smaller
};
```

=default and =delete

- The idiom of *prohibiting copying* (for C++03 recall **Module 25**) can now be expressed directly:


```
class X { // ...
    X& operator=(const X&) = delete; // Disallow copying
    X(const X&) = delete;
};
```
- Conversely, we can also say *explicitly* that we want *default copy behavior*:


```
class Y { // ...
    Y& operator=(const Y&) = default; // default copy semantics
    Y(const Y&) = default;
};
```
- Explicitly writing out the default by hand is good for *readability*, but it has two *drawbacks*:
 - it sometimes generates *less efficient code* than the compiler-generated default would, and
 - it prevents types from *being considered PODs*
- The `=default` mechanism can be used for any function that has a default
- The `=delete` mechanism can be used for any function like to eliminate an undesired conversion:


```
struct Z { // ...
    Z(long long); // can initialize with a long long
    Z(long) = delete; // but not anything smaller
};
```

So, here is a module outline and let us get started with the first which is default and deleted functions. In C++03, in Module 25, if you recall, we had discussed a problem that if I want to create a class where copy is not allowed, copy will not be allowed, we saw the solution was to put the copy constructor and copy assignment operator as private or to have a base class something like uncopyable, non-copyable where these are private and then you privately inherit from those and so on you should recap on those slides before you proceed further here.

In C++11 delete feature equal to delete function feature all that you can do is you can simply write the signature of the function and put equal to delete, which means this function will be known to the compiler, but the function will not be provided by the compiler. This function

cannot be used as simple. delete says this function is prohibited to be used. This function is a member of the class. This function will be used to resolve overloading and all that. It is, as if it exists in every possible way, but it is not, it cannot be used.

Similar or kind of not similar, but maybe the other end of the spectrum is you can say that, explicitly say that some function is default. So, what we know that if copies are done for the objects of a class and no copy constructor is provided, the compiler provides one. Now, copy constructor can be provided with constant reference, with non-constant reference or other kinds of things.

But, what you want to say here is you are saying that this is the signature. The copy constructor must be their point number one. This is the signature and the compiler should provide a default copy semantics. It is not necessary that if you provide copy constructor as default, you will have to make copy assignment also as default, but whatever you make default the compiler will provide that, and everybody.

So, the advantage here is it improves the readability also, it improves the resolvability also, because for example, you might wanted that you want a default copy constructor which does not use a constant reference, because of some reason whatever. So, if you want to do that, you can provide it here and say default. The compiler will still provide it, but the signature would be as you have provided.

So, any, actually this default mechanism or delete mechanism is not limited to just constructors or copy operations, this default mechanism it can be used for any function that has a default. It can be used for move as well. And delete is possible for any function to eliminate something that you do not want. For example, say you have a structure Z, class Z and you have a constructor for long long.

Now, naturally, if you pass a long value to it, this object will get constructed. But suppose you would specifically want that you will not allow long values, you necessarily want a long long value, then what you can do, you can simply say that, Z(long) is equal to delete, which says that the constructor taking long as a parameter is a valid member function of this class, but you are not allowed to use it. So, this is the basic idea of the default and delete.

(Refer Slide Time: 07:01)

default Member Functions

- The *special* member functions are implicitly generated if used:
 - Default constructor
 - ▷ Only if no user-declared constructors
 - Destructor
 - Copy operations (copy constructor, copy operator=)
 - ▷ Only if move operations not user-declared
 - Move operations (move constructor, move operator=)
 - ▷ Only if copy operations and destructor not user-declared
- Generated versions are:
 - Public
 - Inline
 - Non-explicit
- defaulted member functions have:
 - *User-specified declarations with the usual compiler-generated implementations*

So, we know all the different functions which are available for default, default constructor, destructor, copy constructor, move, copy operations, move operations, and there are restrictions for them. But when you get the default from the compiler, you get them in this kind of form, that the default version is always public, it is always inline and it is always non-explicit. By using the default option now you can change those as well as we will see.

(Refer Slide Time: 07:35)

default Member Functions

- Typical use: *unsuppress* implicitly-generated functions:

```
class Widget { public:  
    Widget(const Widget&); // copy ctor prevents implicitly declared  
                          // default ctor & move ops  
    Widget() = default;    // declare default ctor, use default impl.  
    Widget(Widget&&) noexcept = default; // declare move ctor, use default impl.  
    ...  
};
```
- Or change *normal accessibility, explicitness, virtualness*:

```
class Widget { public:  
    virtual Widget() = default; // declare as virtual  
    explicit Widget(const Widget&) = default; // declare as explicit  
private:  
    Widget& operator=(Widget&&) noexcept = default; // declare as private  
    ...  
};
```


The screenshot shows a presentation slide with a blue header containing the title "default Member Functions" and a small video feed of a speaker. The slide content includes two bullet points and corresponding C++ code snippets. The first bullet point discusses the typical use of `default` to suppress implicitly-generated functions. The second bullet point discusses changing the normal accessibility, explicitness, or virtualness of default member functions. The code snippets show class definitions for `Widget` with various default member functions and their annotations.

default Member Functions

- Typical use: *suppress* implicitly-generated functions:


```
class Widget { public:
    Widget(const Widget&); // copy ctor prevents implicitly declared
                        // default ctor & move ops
    Widget() = default;   // declare default ctor, use default impl.
    Widget(Widget&&) noexcept = default; // declare move ctor, use default impl.
    ...
};
```
- Or change *normal accessibility*, *explicitness*, *virtualness*:

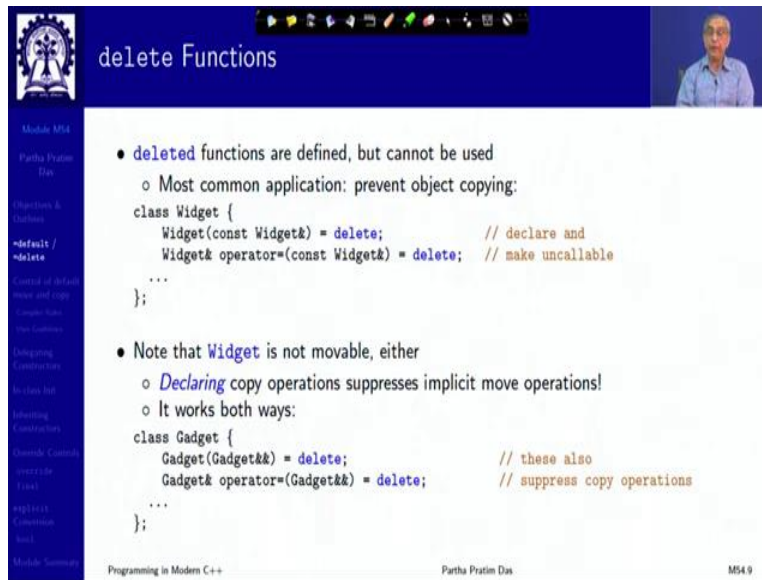

```
class Widget { public:
    virtual ~Widget() = default; // declare as virtual
    explicit Widget(const Widget&) = default; // declare as explicit
private:
    Widget& operator=(Widget&&) noexcept = default; // declare as private
    ...
};
```

So, here are examples of default member functions. We actually are saying this is the copy constructor that, if we provide that, then the default constructor will not be provided, move operations will not be provided and then you say that there is a declare, this is your default constructor which should use a default implementation of the, provided by the compiler. Similarly, this is a move constructor, which for which default should be given. But most importantly, what you can do, you can change the behavior of the default.

For example, as I said, the default is public. What we are doing here you are declaring a move assignment operator in private and calling it default. So, the compiler will still provide the default but it will not make it public, it will make it private. Similarly, here compiler gives constructors, copy constructors, which are all non explicit, but we are saying that explicit this is default.

So, the compiler will still provide that, but it will be an explicit constructor, which means that implicit conversions cannot be done with it. So, this is similarly you have made the destructor virtual and so on. So, these are the advantages of default.

(Refer Slide Time: 08:58)



delete Functions

- deleted functions are defined, but cannot be used
 - Most common application: prevent object copying:

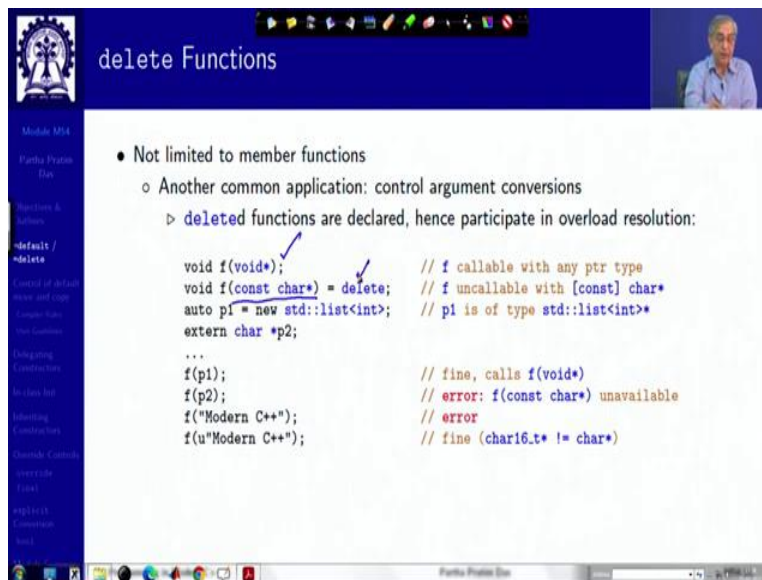
```
class Widget {
    Widget(const Widget&) = delete;           // declare and
    Widget& operator=(const Widget&) = delete; // make uncallable
    ...
};
```
- Note that `Widget` is not movable, either
 - Declaring copy operations suppresses implicit move operations!
 - It works both ways:

```
class Gadget {
    Gadget(Gadget&&) = delete;           // these also
    Gadget& operator=(Gadget&&) = delete; // suppress copy operations
    ...
};
```

Programming in Modern C++ Partha Pratim Das MS4.9

And in delete you can similarly do all this delete for preventing copying or suppressing implicit move operations and so on.

(Refer Slide Time: 09:14)



delete Functions

- Not limited to member functions
 - Another common application: control argument conversions
 - deleted functions are declared, hence participate in overload resolution:

```
void f(void*); // f callable with any ptr type
void f(const char*) = delete; // f uncallable with [const] char*
auto p1 = new std::list<int>; // p1 is of type std::list<int>*
extern char *p2;
...
f(p1); // fine, calls f(void*)
f(p2); // error: f(const char*) unavailable
f("Modern C++"); // error
f(u"Modern C++"); // fine (char16_t* != char*)
```

Partha Pratim Das

delete Functions

- Not limited to member functions
 - Another common application: control argument conversions
 - ▷ deleted functions are declared, hence participate in overload resolution:

```

void f(void*); // f callable with any ptr type
void f(const char*) = delete; // f uncalleable with [const] char*
auto p1 = new std::list<int>; // p1 is of type std::list<int>*
extern char *p2;
...
f(p1); // fine, calls f(void*)
f(p2); // error: f(const char*) unavailable
f("Modern C++"); // error
f(u"Modern C++"); // fine (char16_t* != char*)
  
```

delete Functions

- Not limited to member functions
 - Another common application: control argument conversions
 - ▷ deleted functions are declared, hence participate in overload resolution:

```

void f(void*); // f callable with any ptr type
void f(const char*) = delete; // f uncalleable with [const] char*
auto p1 = new std::list<int>; // p1 is of type std::list<int>*
extern char *p2;
...
f(p1); // fine, calls f(void*)
f(p2); // error: f(const char*) unavailable
f("Modern C++"); // error
f(u"Modern C++"); // fine (char16_t* != char*)
  
```

You can see these examples. So, here is an example of delete. So, here I have this void*, f() takes void* and it takes f() takes const char* which I have deleted. Now, with this, it does not mean that this function does not exist. It says that the function is defined, but deleted, which means that you cannot use it, but it will play a role in overload resolution.

So, there are actually two overloaded functions here. And I have here two variables p1 and p2 of two different pointer types. You call this by p1. It binds with f(), first overload, so it is fine. But if you call it by p2, then it tries to bind with the second one. And since it tries to bind with second one and f(const char*) is unavailable, this becomes an error.

Similarly, if you pass it as a `const char*` pointer directly, it tries to bind here and will be an error. But if you give it like this, this `u` here says that it is a unsigned, it is a 16 Unicode character string, 16 bit character string. So, it is, so this is not same as `char*`. So, this will not bind here, rather this will bind here and it will be allowed to use this. So, this is, these are the different ways that the delete can play a significant role in this.

(Refer Slide Time: 11:06)

The slide is titled "Control of default move and copy". It features a navigation sidebar on the left with items like "Module M14", "Partha Pratim Das", "Objectives & Outcomes", "Introduction", "Control of default move and copy", "Designing Constructors", "In-Class Init", "Identifying Constructors", "Destructor Control", "RAII", "RAII-II", "Conclusion", and "Module Summary". The main content area lists sources for the topic:

- Control of default move and copy, isocpp.org
- The rule of three/five/zero, cpreference.com
- C++ Core Guidelines, Eds. Bjarne Stroustrup and Herb Sutter, 2022
 - C.20: If you can avoid defining default operations, do
 - C.21: If you define or `=delete` any copy, move, or destructor function, define or `=delete` them all
 - C.22: Make default operations consistent
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

The title "Control of default move and copy" is repeated in red text at the bottom of the slide. The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M54.11".

So, having seen this delete and default feature, let us see how do we really control the default, move and copy that the compiler provides us with.

(Refer Slide Time: 11:20)

The slide is titled "Control of default move and copy". It features the same navigation sidebar as the previous slide. The main content area explains the control of default functions:

- We learnt in **Module 51** that *Move is an Optimization of Copy*. This is specifically true for *classes having resources* (like pointer / vector) that needs to be moved or copied
- This *needs Move and Copy operations* to be appropriately defined
- We also *need a destructor* in such cases for the release of the resources (like pointer)
- Further, the *compiler provide these functions as default* (if not provided and / or deleted by the user) so that the users do not need to write them for every class
- So there can be one of the following options for each of the five functions in a class:
 - [Un-declared] Do not mention the function in the class - *implicitly default*
 - [=default] Mention the function as `=default` - *explicitly default*
 - [Declared] Declare the function but not define it - *prohibit use*
 - [=deleted] Mention the function as `=deleted` - *prohibit use*
 - [Defined] Provide a user-defined implementation of the function - *proper use*
- For [1] & [2] *compiler provides default implementation* and for [5] *user provides the same*
- In total, we have $5 \times 5 = 25$ *scenarios* but only a few of them are *semantically consistent*
- So for a proper semantics, we *need to control move / copy / destruction*:
 - Compiler follows a set of rules
 - User needs to follow a set of guidelines

The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M54.11".

Control of default move and copy

- We learnt in **Module 51** that *Move is an Optimization of Copy*. This is specifically true for *classes having resources* (like pointer / vector) that needs to be moved or copied
- This *needs Move and Copy operations* to be appropriately defined
- We also *need a destructor* in such cases for the release of the resources (like pointer)
- Further, the *compiler provide these functions as default* (if not provided and / or deleted by the user) so that the users do not need to write them for every class
- So there can be one of the following options for each of the *five functions in a class*:
 - [1] [Un-declared] Do not mention the function in the class - *implicitly default*
 - [2] [=default] Mention the function as =default - *explicitly default*
 - [3] [Declared] Declare the function but not define it - *prohibit use*
 - [4] [=deleted] Mention the function as =deleted - *prohibit use*
 - [5] [Defined] Provide a user-defined implementation of the function - *proper use*
- For [1] & [2] *compiler provides default implementation* and for [5] *user provides the same*
- In total, we have $5 \times 5 = 25$ *scenarios* but only a few of them are *semantically consistent*
- So for a proper semantics, we *need to control move / copy / destruction*:
 - **Compiler follows a set of rules**
 - **User needs to follow a set of guidelines**

Control of default move and copy

- We learnt in **Module 51** that *Move is an Optimization of Copy*. This is specifically true for *classes having resources* (like pointer / vector) that needs to be moved or copied
- This *needs Move and Copy operations* to be appropriately defined
- We also *need a destructor* in such cases for the release of the resources (like pointer)
- Further, the *compiler provide these functions as default* (if not provided and / or deleted by the user) so that the users do not need to write them for every class
- So there can be one of the following options for each of the *five functions in a class*:
 - [1] [Un-declared] Do not mention the function in the class - *implicitly default* ✓
 - [2] [=default] Mention the function as =default - *explicitly default* ✓
 - [3] [Declared] Declare the function but not define it - *prohibit use* ✓
 - [4] [=deleted] Mention the function as =deleted - *prohibit use* ✓
 - [5] [Defined] Provide a user-defined implementation of the function - *proper use* ✓
- For [1] & [2] *compiler provides default implementation* and for [5] *user provides the same*
- In total, we have $5 \times 5 = 25$ *scenarios* but only a few of them are *semantically consistent*
- So for a proper semantics, we *need to control move / copy / destruction*:
 - **Compiler follows a set of rules**
 - **User needs to follow a set of guidelines**

Recall in Module 51, we have had a discussion on move is an optimization of copy, which means that it needs the move and copy operations and the destructor because move and copy are for resources. So, if there are resources destruction is incorrect. So, two move operations construction and assignment, two copy operations construction and assignment, and the delete operation these five have to be defined in a consistent semantics. Therefore, the compiler has to decide what it should provide and you have to decide how you should ask for them.

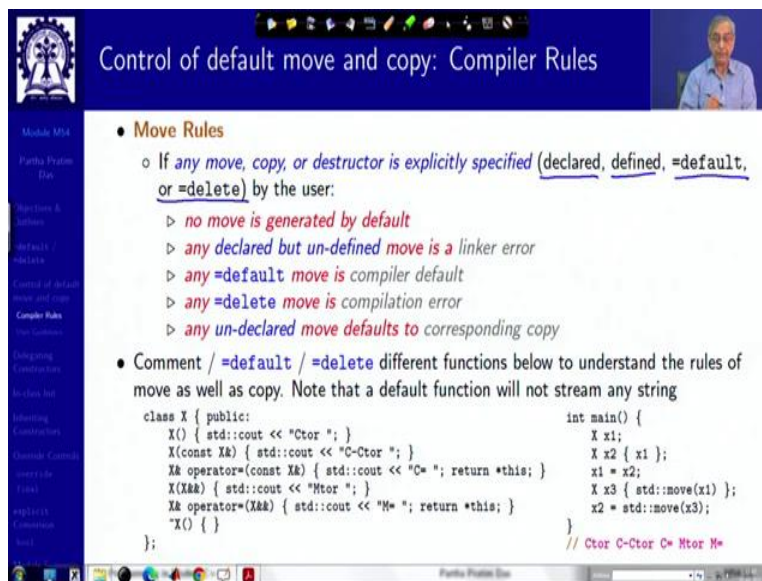
So, then let us consider that for each one of these five functions that I mentioned two move operations, two copy operations and the destructor, and the destruction, there are five possibilities each. One is nothing is said about a function. It is undeclared. One is the function

signature is given and defaulted. One is it is declared that you have written this but not defaulted, just ended with semicolon. One is you have given the signature and said deleted. And one is you have given the signature as well as you have defined it.

So, naturally, if you undeclared, then it is implicitly default. If it is equal to default, then it is explicitly default if signature is given and that the compiler has to provide. If it is declared but not defined then the use is prohibited, but you will have a linker error, because it is defined, declared, but not defined. If it is deleted, then you will not have a linker error because you will not be able to combine it. So, the prohibition appeals have two different semantics.

And if it is defined, then it is a proper use. So, this brings us to 5 times the 5, 25 possibilities, 25 possible semantic scenarios could be there. Each one of the five functions, for each one of the five functions I can do any one of the five options. But these 25 scenarios are not meaningful because they are not consistent semantically. So, to keep them consistent semantically the compilers have to follow a set of rules and as a user, as a programmer, you have to follow a set of guidelines.

(Refer Slide Time: 14:00)



The slide is titled "Control of default move and copy: Compiler Rules". It contains a list of "Move Rules" and a code example. The rules are:

- **Move Rules**
 - If *any move, copy, or destructor is explicitly specified* (declared, defined, =default, or =delete) by the user:
 - ▷ *no move is generated by default*
 - ▷ *any declared but un-defined move is a linker error*
 - ▷ *any =default move is compiler default*
 - ▷ *any =delete move is compilation error*
 - ▷ *any un-declared move defaults to corresponding copy*
 - Comment / =default / =delete different functions below to understand the rules of move as well as copy. Note that a default function will not stream any string

```
class X { public:
    X() { std::cout << "Ctor "; }
    X(const X&) { std::cout << "C-Ctor "; }
    X& operator=(const X&) { std::cout << "C= "; return *this; }
    X(X&&) { std::cout << "Mtor "; }
    X& operator=(X&&) { std::cout << "M= "; return *this; }
    ~X() { }
};

int main() {
    X x1;
    X x2 { x1 };
    x1 = x2;
    X x3 { std::move(x1) };
    x2 = std::move(x3);
} // Ctor C-Ctor C= Mtor M=
```

So, the rules compiler follow are into two groups one is called the move rules. Move rules say that if any move copy or destructor is explicitly specified, then no move is generated by default. If any one of these is explicitly defined, specified, then nothing is, so explicitly specified means that it is either declared or it is defined or it is defaulted or it is deleted any one of these. So, no

move is. If it is declared but undefined, it is a linker error. If it is defaulted, then it is a move error compiler default. If it is deleted, there is a compilation error. If it is undeclared, then it defaults to the corresponding copy operation. If you have not said anything about the move, it will take it to the corresponding move copy operation. Here is an example which you can study for yourself.

(Refer Slide Time: 15:01)

Control of default move and copy: Compiler Rules

- **Copy Rules**
 - o If *any copy or destructor is explicitly specified* (declared, defined, =default, or =delete) by the user:
 - ▷ *any undeclared copy operations are generated by default*
 - ▷ this is deprecated in C++11
 - o If *any move is explicitly specified* (declared, defined, =default, or =delete):
 - ▷ *no copy is generated by default*
 - o **Bad problem due to default copy in C++03 persists in C++11 onward**

```

class X { public: // copyable class
    ~X() { delete p; } // implicit invariant: p owns *p
    //... // no copy ops declared
private: int *p;
};

int main() {
    X x1;
    X x2(x1);
} // double delete! of p: run-time error: munmap_chunk(): invalid pointer in gcc
  
```

Control of default move and copy: Compiler Rules

- **Copy Rules**
 - o If *any copy or destructor is explicitly specified* (declared, defined, =default, or =delete) by the user:
 - ▷ *any undeclared copy operations are generated by default*
 - ▷ this is deprecated in C++11
 - o If *any move is explicitly specified* (declared, defined, =default, or =delete):
 - ▷ *no copy is generated by default*
 - o **Bad problem due to default copy in C++03 persists in C++11 onward**

```

class X { public: // copyable class
    ~X() { delete p; } // implicit invariant: p owns *p
    //... // no copy ops declared
private: int *p;
};

int main() {
    X x1;
    X x2(x1);
} // double delete! of p: run-time error: munmap_chunk(): invalid pointer in gcc
  
```

Copy rules are somewhat different, because copy rules existed from default. So, in copy rules, it is selective that out of the two copy operations if you provide one and do not provide the other but use it, whatever you have not provided will still be provided by the compiler. Move is not

like that. You provide any of the move operation, copy operation, destruction, all other will not be provided default for. So, any undeclared copy operation is generated by default, so that you can see the copy rules for the, default copy rules for the compiler is different.

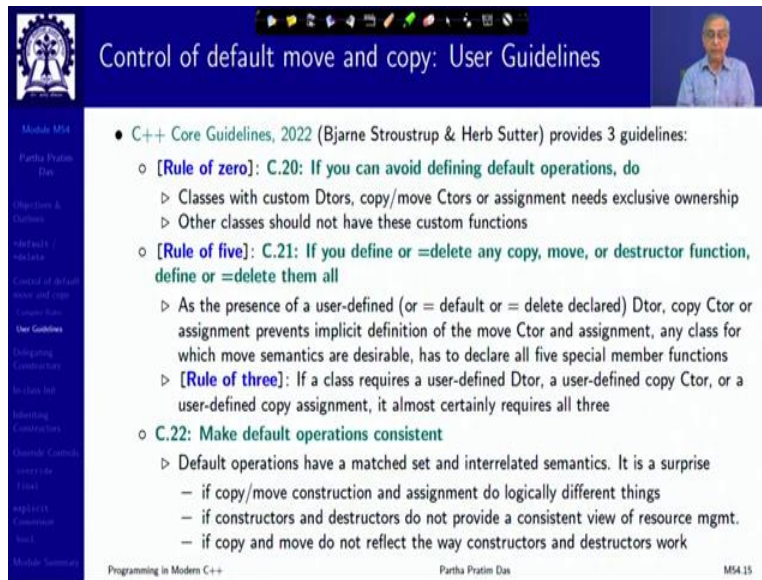
If any move is explicitly specified, then no copy is generated by default. So, usually, I expected the rules to be symmetric to be similar but that is not the case, because the default copy rules in C++03 is a legacy which has to exist as C++11 goes forward. So, yes, there could be some, because of this default copy you could have some very, very nasty problems which are very easy to create that you have a class X which has a pointer, which you have dynamically created something, and therefore, your destructor has a delete. It is very logical.

Now, you have not provided a constructor, you have not provided a copy constructor, you have not said anything, which is a C++03 scenario so what will happen. If I write X x1 a default construction will happen. And then say you have done operations to set this something to the speed it is pointing somewhere, then you will do x2 by a copy of x1. So, the default copy constructor will be invoked. And default copy constructor will do a shallow copy. It will just copy the pointer, not the pointed object. So, two pointers from x1 and from x2 will point to the same object.

Now, what will happen? As you come to the end of the scope, both of these are automatic objects. So, according to lifetime rules on both of them the destructor will be invoked. So, first you will have the destruction on x2. So, that will delete p. That object is 1. Pointed object is gone. But then, you will have the destruction for x1. So, it will again try to delete on the same point that because the pointer has been copied and you will get a double deletion error.

For example, if you try on that online gdb compiler this is the error that you will get. So, these are problems which exist in C++03 and C++11 has not been able to address this. But so far as the mix of move and copy is concerned, it is tried to do a decent job.

(Refer Slide Time: 18:05)



Control of default move and copy: User Guidelines

- C++ Core Guidelines, 2022 (Bjarne Stroustrup & Herb Sutter) provides 3 guidelines:
 - **[Rule of zero]: C.20: If you can avoid defining default operations, do**
 - ▷ Classes with custom Dtors, copy/move Ctors or assignment needs exclusive ownership
 - ▷ Other classes should not have these custom functions
 - **[Rule of five]: C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all**
 - ▷ As the presence of a user-defined (or = default or = delete declared) Dtor, copy Ctor or assignment prevents implicit definition of the move Ctor and assignment, any class for which move semantics are desirable, has to declare all five special member functions
 - ▷ **[Rule of three]:** If a class requires a user-defined Dtor, a user-defined copy Ctor, or a user-defined copy assignment, it almost certainly requires all three
 - **C.22: Make default operations consistent**
 - ▷ Default operations have a matched set and interrelated semantics. It is a surprise
 - if copy/move construction and assignment do logically different things
 - if constructors and destructors do not provide a consistent view of resource mgmt.
 - if copy and move do not reflect the way constructors and destructors work

Programming in Modern C++ Partha Pratim Das MS4.15

Now, given all this, it also is, if there are set of guidelines that have been provided as to how you use this default at functions. So, three guidelines; one is called the Rule of zero that try to avoid defining default operations if you can. That is the best thing. You take responsibility of everything or the Rule of five which says that if you define any in terms of declaration, delete, any copy, move or destructor function then do that for all.

So, Rule of zero and Rule of five are the basic thumb rules for using and there are some more details which you can see. These are drafted by none other than Bjarne Stroustrup and Herb Sutter. So, you have, you should follow that to make sure that you have a consistent default move and copy semantics. Out of the 25 possibilities only a few will be meaningful and you should stick to them, otherwise you will have the kind of surprise that I just showed you.

(Refer Slide Time: 19:15)

Delegating Constructors

Module M54
Partha Pratin Das
Objectives & Outcomes
Default Constructors
Control of default constructors and copy constructors
Delegating Constructors
In-class list
Inheriting Constructors
Destructor Constructors
explicit Constructors
Module Summary

Delegating Constructors

Sources:

- Delegating constructors, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

Programming in Modern C++ Partha Pratin Das M54.16

Moving on, we will take a number of small features. Delegating constructor is something very interesting.

(Refer Slide Time: 19:23)

Delegating Constructors

Multiple constructors with code copies

```
class Base { public:  
    explicit Base(int);  
    // ...  
};  
class Widget: public Base { public: // 4 Ctors  
    Widget();  
    explicit Widget(double fl);  
    explicit Widget(int sz);  
    Widget(const Widget& w);  
private:  
    static int calc(); // calculate Base value  
    static constexpr double defaultFlex = 1.5;  
    const int size;  
    long double flex;  
};  
Widget::Widget() // #1  
    : Base(calc()), size(0), flex(defaultFlex) {  
    regObj(this);  
}  
Widget::Widget(double fl) // #2  
    : Base(calc()), size(0), flex(fl) {  
    regObj(this);  
}  
Widget::Widget(int sz) // #3  
    : Base(calc()), size(sz), flex(defaultFlex) {  
    regObj(this);  
}  
Widget::Widget(const Widget& w) : Base(w), // #4  
    size(w.size), flex(w.flex) { regObj(this); }
```

Refactored for better reuse with delegated constructors

```
class Widget: public Base { public: // delegation happens when one Ctor calls another - code refactored  
    Widget(): Widget(defaultFlex) {} // #1: calls #2  
    explicit Widget(double fl): Widget(0, fl) {} // #2: calls #5  
    explicit Widget(int sz): Widget(sz, defaultFlex) {} // #3: calls #5  
    Widget(const Widget& w) : Base(w), size(w.size), flex(w.flex) { regObj(this); } // #4: same  
private: Widget(int sz, double fl): Base(calc()), size(sz), flex(fl) { regObj(this); } // #5: new  
    // ... same as above  
};
```



Delegating Constructors



- Module 004
- Partha Prasad Das
- Sections & Videos
- Default Methods
- Control of default access and scope
- Control Flow
- Delegating Constructors
- Access Modifiers
- Identifying Constructors
- Overload Control
- Overriding
- Final
- Explicit Constructors
- Final

```

Multiple constructors with code copies

class Base { public:
    explicit Base(int);
    // ...
};

class Widget: public Base { public: // 4 Ctors
    Widget();
    explicit Widget(double fl);
    explicit Widget(int sz);
    Widget(const Widget& w);
private:
    static int calc(); // calculate Base value
    static constexpr double defaultFlex = 1.5;
    const int size;
    long double flex;
};

Widget::Widget() // #1
    Base(calc()), size(0), flex(defaultFlex) {
    regObj(this);
}

Widget::Widget(double fl) // #2
    Base(calc()), size(0), flex(fl) {
    regObj(this);
}

Widget::Widget(int sz) // #3
    Base(calc()), size(sz), flex(defaultFlex) {
    regObj(this);
}

Widget::Widget(const Widget& w): Base(w) // #4
    size(w.size), flex(w.flex) { regObj(this); }

Refactored for better reuse with delegated constructors

class Widget: public Base { public: // delegation happens when one Ctor calls another - code refactored
    Widget(): Widget(defaultFlex) {} // #1: calls #2
    explicit Widget(double fl): Widget(0, fl) {} // #2: calls #5
    explicit Widget(int sz): Widget(sz, defaultFlex) {} // #3: calls #5
    Widget(const Widget& w): Base(w), size(w.size), flex(w.flex) { regObj(this); } // #4: same
private: Widget(int sz, double fl): Base(calc()), size(sz), flex(fl) { regObj(this); } // #5: new
    // ... same as above
};

```



Delegating Constructors



- Module 004
- Partha Prasad Das
- Sections & Videos
- Default Methods
- Control of default access and scope
- Control Flow
- Delegating Constructors
- Access Modifiers
- Identifying Constructors
- Overload Control
- Overriding
- Final
- Explicit Constructors
- Final

```

Multiple constructors with code copies

class Base { public:
    explicit Base(int);
    // ...
};

class Widget: public Base { public: // 4 Ctors
    Widget();
    explicit Widget(double fl);
    explicit Widget(int sz);
    Widget(const Widget& w);
private:
    static int calc(); // calculate Base value
    static constexpr double defaultFlex = 1.5;
    const int size;
    long double flex;
};

Widget::Widget() // #1
    Base(calc()), size(0), flex(defaultFlex) {
    regObj(this);
}

Widget::Widget(double fl) // #2
    Base(calc()), size(0), flex(fl) {
    regObj(this);
}

Widget::Widget(int sz) // #3
    Base(calc()), size(sz), flex(defaultFlex) {
    regObj(this);
}

Widget::Widget(const Widget& w): Base(w) // #4
    size(w.size), flex(w.flex) { regObj(this); }

Refactored for better reuse with delegated constructors

class Widget: public Base { public: // delegation happens when one Ctor calls another - code refactored
    Widget(): Widget(defaultFlex) {} // #1: calls #2
    explicit Widget(double fl): Widget(0, fl) {} // #2: calls #5
    explicit Widget(int sz): Widget(sz, defaultFlex) {} // #3: calls #5
    Widget(const Widget& w): Base(w), size(w.size), flex(w.flex) { regObj(this); } // #4: same
private: Widget(int sz, double fl): Base(calc()), size(sz), flex(fl) { regObj(this); } // #5: new
    // ... same as above
};

```

Delegating Constructors




Module 014

Patha Prasad Das

Navigation & Tables

Default Methods

Control of default methods and super class access

Delegating Constructors

Access list

Identifying Constructors

Overriding Constructors

Explicit Constructors

Self

Multiple constructors with code copies

```

class Base { public:
    explicit Base(int);
    // ...
};
class Widget: public Base { public: // 4 Ctors
    Widget();
    explicit Widget(double fl);
    explicit Widget(int sz);
    Widget(const Widget& w);
private:
    static int calc(); // calculate Base value
    static constexpr double defaultFlex = 1.5;
    const int size;
    long double flex;
};



Widget::Widget() // #1
    Base(calc()), size(0), flex(defaultFlex) {
    regObj(this);
}
Widget::Widget(double fl) // #2
    Base(calc()), size(0), flex(fl) {
    regObj(this);
}
Widget::Widget(int sz) // #3
    Base(calc()), size(sz), flex(defaultFlex) {
    regObj(this);
}
Widget::Widget(const Widget& w): Base(w) // #4
    size(w.size), flex(w.flex) { regObj(this); }
    
```

Refactored for better reuse with delegated constructors

```

class Widget: public Base { public: // delegation happens when one Ctor calls another - code refactored
    Widget(): Widget(defaultFlex) {} // #1: calls #2
    explicit Widget(double fl): Widget(0, fl) {} // #2: calls #5
    explicit Widget(int sz): Widget(sz, defaultFlex) {} // #3: calls #5
    Widget(const Widget& w): Base(w), size(w.size), flex(w.flex) { regObj(this); } // #4: same
private: Widget(int sz, double fl): Base(calc()), size(sz), flex(fl) { regObj(this); } // #5: new
    // ... same as above
};
    
```

Delegating Constructors

Module 014

Patha Prasad Das

Navigation & Tables

Default Methods

Control of default methods and super class access

Delegating Constructors

Access list

Identifying Constructors

Overriding Constructors

Explicit Constructors

Self

Multiple constructors with code copies

```

class Base { public:
    explicit Base(int);
    // ...
};
class Widget: public Base { public: // 4 Ctors
    Widget();
    explicit Widget(double fl);
    explicit Widget(int sz);
    Widget(const Widget& w);
private:
    static int calc(); // calculate Base value
    static constexpr double defaultFlex = 1.5;
    const int size;
    long double flex;
};




Widget::Widget() // #1
    Base(calc()), size(0), flex(defaultFlex) {
    regObj(this);
}
Widget::Widget(double fl) // #2
    Base(calc()), size(0), flex(fl) {
    regObj(this);
}
Widget::Widget(int sz) // #3
    Base(calc()), size(sz), flex(defaultFlex) {
    regObj(this);
}
Widget::Widget(const Widget& w): Base(w) // #4
    size(w.size), flex(w.flex) { regObj(this); }
    
```

Refactored for better reuse with delegated constructors

```

class Widget: public Base { public: // delegation happens when one Ctor calls another - code refactored
    Widget(): Widget(defaultFlex) {} // #1: calls #2
    explicit Widget(double fl): Widget(0, fl) {} // #2: calls #5
    explicit Widget(int sz): Widget(sz, defaultFlex) {} // #3: calls #5
    Widget(const Widget& w): Base(w), size(w.size), flex(w.flex) { regObj(this); } // #4: same
private: Widget(int sz, double fl): Base(calc()), size(sz), flex(fl) { regObj(this); } // #5: new
    // ... same as above
};
    
```

Delegating Constructors

Module M14

Partha Pratim Das

Structure & Syntax

Default namespace

Control of default namespace and scope

using-directives

Delegating Constructors

Namespace Aliases

Identifying Constructors

Overload Control

overriding

friend

explicit Constructors

final

Multiple constructors with code copies

```

class Base { public:
    explicit Base(int);
    // ...
};

class Widget: public Base { public: // 4 Ctors
    Widget();
    explicit Widget(double fl);
    explicit Widget(int sz);
    Widget(const Widget& w);
private:
    static int calc(); // calculate Base value
    static constexpr double defaultFlex = 1.5;
    const int size;
    long double flex;
};

Widget::Widget() // #1
    Base(calc()), size(0), flex(defaultFlex) {
    regObj(this);
}

Widget::Widget(double fl) // #2
    Base(calc()), size(0), flex(fl) {
    regObj(this);
}

Widget::Widget(int sz) // #3
    Base(calc()), size(sz), flex(defaultFlex) {
    regObj(this);
}

Widget::Widget(const Widget& w): Base(w), // #4
    size(w.size), flex(w.flex) { regObj(this);
};

```




Refactored for better reuse with delegated constructors

```

class Widget: public Base { public: // delegation happens when one Ctor calls another - code refactored
    Widget(): Widget(defaultFlex) {} // #1: calls #2
    explicit Widget(double fl): Widget(0, fl) {} // #2: calls #5
    explicit Widget(int sz): Widget(sz, defaultFlex) {} // #3: calls #5
    Widget(const Widget& w): Base(w), size(w.size), flex(w.flex) { regObj(this); } // #4: same
private:
    Widget(int sz, double fl): Base(calc()), size(sz), flex(fl) { regObj(this); } // #5: new
    // ... same as above
};

```

Delegating Constructors

Module M14

Partha Pratim Das

Structure & Syntax

Default namespace

Control of default namespace and scope

using-directives

Delegating Constructors

Namespace Aliases

Identifying Constructors

Overload Control

overriding

friend

explicit Constructors

final

Multiple constructors with code copies

```

class Base { public:
    explicit Base(int);
    // ...
};

class Widget: public Base { public: // 4 Ctors
    Widget();
    explicit Widget(double fl);
    explicit Widget(int sz);
    Widget(const Widget& w);
private:
    static int calc(); // calculate Base value
    static constexpr double defaultFlex = 1.5;
    const int size;
    long double flex;
};

Widget::Widget() // #1
    Base(calc()), size(0), flex(defaultFlex) {
    regObj(this);
}

Widget::Widget(double fl) // #2
    Base(calc()), size(0), flex(fl) {
    regObj(this);
}

Widget::Widget(int sz) // #3
    Base(calc()), size(sz), flex(defaultFlex) {
    regObj(this);
}

Widget::Widget(const Widget& w): Base(w), // #4
    size(w.size), flex(w.flex) { regObj(this); } // #5
};

```

Refactored for better reuse with delegated constructors

```

class Widget: public Base { public: // delegation happens when one Ctor calls another - code refactored
    Widget(): Widget(defaultFlex) {} // #1: calls #2
    explicit Widget(double fl): Widget(0, fl) {} // #2: calls #5
    explicit Widget(int sz): Widget(sz, defaultFlex) {} // #3: calls #5
    Widget(const Widget& w): Base(w), size(w.size), flex(w.flex) { regObj(this); } // #4: same
private:
    Widget(int sz, double fl): Base(calc()), size(sz), flex(fl) { regObj(this); } // #5: new
    // ... same as above
};

```

This is something, if you have a class with multiple constructors, it is often that there is code copies. So, there is a Widget class and there are four constructors. A default one a copy one and two parameterized one and these four are written here. And the red codes you can see are basically repeated code. The whole idea is to show that multiple constructors often share the same initializing code. So, for that in C++03 what we do we write different unit functions and try to call the init function because of that and cumbersome handover techniques.

Here, in C++11 you have a very nice feature, from one constructor you can call another constructor. So, by that you can make things much simpler. For example, what we notice is that here there is a string object is something which looks to be pretty repetitive, constructing the

base class with certain call to certain functions is repetitive. So, you define a constructor which as it is not amongst these four, a constructor with int and double. There is none such in this list. And to make sure that this constructor is not directly used, we make it private, where we do this base(calc()), setting the size, setting the size that is int, setting the floating value and these things.

Then what we do for using say this explicit constructor, we set a default value to int and let that. See this carefully. This is the constructor and it is calling this constructor. This is the constructor it is calling this construct, a feature which we have not seen before, because it did not exist. So, what it is doing is one constructor is delegating its responsibility to another. So, here for the default constructor, it does something interesting it delegates to a constructor which has a default double value calls 2. 2 intern delegates it to the private constructor here. So, double delegation. So, delegate T can also delegate further.

And you can see that with that in contrast to all that you had here, you have a small code, very readable, very manageable, and it cannot, for example, in writing this someone could miss out this, but here you cannot do that. But there may be exceptions to that as well. For example, the copy constructor did not fit into that model, because it does something different. So, we do not touch it. We not doing any delegation in that at all.

(Refer Slide Time: 23:02)

Delegating Constructors

- Delegation is independent of constructor characteristics
 - Delegator and delegatee may each be `inline`, `explicit`, `public` / `protected` / `private`, etc.
 - Delegatees can themselves delegate
 - Delegators' code bodies execute when delegatees return:

```
class Data { int i;
public:
    Data(): Data(0) { cout << "Data()" << endl; } // #1: calls delegated #2
    Data(int i): 1(i) { cout << "Data(int)" << endl; } // #2
};

int main() {
    Data d;
}

// Data(int)
// Data()
```

Programming in Modern C++ Partha Pratim Das MS4.18

So, this is the notion of delegating constructors and delegating, delegator and delegatee may each be in line, explicit, public, protected, private, all of those. And if you delegate naturally the delegated construction is done first and then the delegator will be done.

(Refer Slide Time: 23:24)

In-class Member Initializers

Module M14
Partha Pratin Das

Objectives & Outcomes
In-class Init

In-class Member Initializers

Sources:

- In-class member initializers, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

Programming in Modern C++ Partha Pratin Das M54.19

Second feature very important in terms of ease of use is in-class member initializer.

(Refer Slide Time: 23:34)

In-class Member Initializers

Module M14
Partha Pratin Das

Objectives & Outcomes
In-class Init

- In C++03, only static const members of integral types can be initialized in-class only with a constant expression so that the initialization can be done at compile-time:

```
int var = 7;
class X {
    static const int m1 = 7; // okay
    const int m2 = 7; // error: not static
    static int m3 = 7; // error: not const
    static const int m4 = var; // error: initializer not constant expression
    static const string m5 = "odd"; // error: not integral type
    // ...
};
```

- In C++11 a non-static data member may be initialized where it is declared in its class. A constructor can then use the initializer when run-time initialization is needed:

```
class A { public: // C++03
    int a;
    A() : a(7) { }
};

class A { public: // C++11
    int a = 7;
};
```

Partha Pratin Das

In-class Member Initializers

- In C++03, *only static const members of integral types* can be initialized in-class *only with a constant expression* so that the initialization can be done at *compile-time*:

```

int var = 7;
class X {
    static const int m1 = 7; // okay
    const int m2 = 7; // error: not static
    static int m3 = 7; // error: not const
    static const int m4 = var; // error: initializer not constant expression
    static const string m5 = "odd"; // error: not integral type
    // ...
};

```

- In C++11 a non-static data member may be initialized where it is declared in its class. A constructor can then use the initializer when *run-time* initialization is needed:

<pre> class A { public: // C++03 int a; A() : a(7) { } }; </pre>	<pre> class A { public: // C++11 int a = 7; }; </pre>
--------------------------------------------------------------------------	-----------------------------------------------------------

In-class Member Initializers

- In C++03, *only static const members of integral types* can be initialized in-class *only with a constant expression* so that the initialization can be done at *compile-time*:

```

int var = 7;
class X {
    static const int m1 = 7; // okay
    const int m2 = 7; // error: not static
    static int m3 = 7; // error: not const
    static const int m4 = var; // error: initializer not constant expression
    static const string m5 = "odd"; // error: not integral type
    // ...
};

```

- In C++11 a non-static data member may be initialized where it is declared in its class. A constructor can then use the initializer when *run-time* initialization is needed:

<pre> class A { public: // C++03 int a; A() : a(7) { } }; </pre>	<pre> class A { public: // C++11 int a = 7; }; </pre>
--------------------------------------------------------------------------	-----------------------------------------------------------

In C++03 we have, we had static members, non-static members. Static members could be initialized within the class provided it is a constant member, it is of integral type and you are using a constant expression to initialize it, otherwise you have to write the static member there and outside the class, you have to again write and put the initialization define, declare that variable and define its initializing value which often you have questioned really as to how useful it is or how convenient it is.

So, if you look at C++03 this is what is permitted. All of these are errors, because this is not static, this is not a constant, this is not using a constant expression and this is not an integral type. In C++11 all these have been relaxed. And further actually non-static members can also be

initialized in the class. So, in C++03 if you have a variable non-static data member a and in the construct, you will put this initialization. Here you can just, with int a you can say initialize at 7.

(Refer Slide Time: 25:11)

In-class Member Initializers

- This is useful for classes with multiple constructors. Often, all constructors use a common initializer for a member:

```

class A { public: int a, b; // C++03
A(): a(7), b(5),
    h_algo("MD5"), s("Ctor run") { }
A(int a_val): a(a_val), b(5),
    h_algo("MD5"), s("Ctor run") { }
A(D d): a(7), b(g(d)),
    h_algo("MD5"), s("Ctor run") { }
private: HashingFunction h_algo; // Hash algo
std::string s; // Tracer
};

class A { public: int a, b; // C++11
A(): a(7), b(5)
{ }
A(int a_val): a(a_val), b(5)
{ }
A(D d): a(7), b(g(d))
{ }
private: HashingFunction h_algo("MD5"); // Hash algo
std::string s("Ctor run"); // Tracer
};
    
```

- If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it "overrides" the default). So we can simplify further:

```

class A { public: int a = 7, b = 5; // default initializers
A() { }
A(int a_val): a(a_val) { } // Ctor initialization overrides
A(D d): b(g(d)) { } // Ctor initialization overrides
private: HashingFunction h_algo("MD5"); // Hash algo: Crypto. hash to be applied to all A instances
std::string s("Ctor run"); // Tracer: String indicating state in object lifecycle
};
    
```

In-class Member Initializers

- This is useful for classes with multiple constructors. Often, all constructors use a common initializer for a member:

```

class A { public: int a, b; // C++03
A(): a(7), b(5),
    h_algo("MD5"), s("Ctor run") { }
A(int a_val): a(a_val), b(5),
    h_algo("MD5"), s("Ctor run") { }
A(D d): a(7), b(g(d)),
    h_algo("MD5"), s("Ctor run") { }
private: HashingFunction h_algo; // Hash algo
std::string s; // Tracer
};

class A { public: int a, b; // C++11
A(): a(7), b(5)
{ }
A(int a_val): a(a_val), b(5)
{ }
A(D d): a(7), b(g(d))
{ }
private: HashingFunction h_algo("MD5"); // Hash algo
std::string s("Ctor run"); // Tracer
};
    
```

- If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it "overrides" the default). So we can simplify further:

```

class A { public: int a = 7, b = 5; // default initializers
A() { }
A(int a_val): a(a_val) { } // Ctor initialization overrides
A(D d): b(g(d)) { } // Ctor initialization overrides
private: HashingFunction h_algo("MD5"); // Hash algo: Crypto. hash to be applied to all A instances
std::string s("Ctor run"); // Tracer: String indicating state in object lifecycle
};
    
```


In-class Member Initializers

- This is useful for classes with multiple constructors. Often, all constructors use a common initializer for a member:

```

class A { public: int a, b; // C++03
A(): a(7), b(5),
    h_algo("MD5"), s("Ctor run") { }
A(int a_val): a(a_val), b(5),
    h_algo("MD5"), s("Ctor run") { }
A(D d): a(7), b(g(d)),
    h_algo("MD5"), s("Ctor run") { }
private: HashingFunction h_algo; // Hash algo
std::string s; // Tracer
};

class A { public: int a, b; // C++11
A(): a(7), b(5)
    { }
A(int a_val): a(a_val), b(5)
    { }
A(D d): a(7), b(g(d))
    { }
private: HashingFunction h_algo{"MD5"}; // Hash algo
std::string s{"Ctor run"}; // Tracer
};

```

- If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it "overrides" the default). So we can simplify further:

```

class A { public: int a = 7, b = 5; // default initializers
A() { }
A(int a_val): a(a_val) { } // Ctor initialization overrides
A(D d): b(g(d)) { } // Ctor initialization overrides
private: HashingFunction h_algo{"MD5"}; // Hash algo: Crypto. hash to be applied to all A instances
std::string s{"Ctor run"}; // Tracer: String indicating state in object lifecycle
};

```

In-class Member Initializers

- This is useful for classes with multiple constructors. Often, all constructors use a common initializer for a member:

```

class A { public: int a, b; // C++03
A(): a(7), b(5),
    h_algo("MD5"), s("Ctor run") { }
A(int a_val): a(a_val), b(5),
    h_algo("MD5"), s("Ctor run") { }
A(D d): a(7), b(g(d)),
    h_algo("MD5"), s("Ctor run") { }
private: HashingFunction h_algo; // Hash algo
std::string s; // Tracer
};

class A { public: int a, b; // C++11
A(): a(7), b(5)
    { }
A(int a_val): a(a_val), b(5)
    { }
A(D d): a(7), b(g(d))
    { }
private: HashingFunction h_algo{"MD5"}; // Hash algo
std::string s{"Ctor run"}; // Tracer
};

```

- If a member is initialized by both an in-class initializer and a constructor, only the constructor's initialization is done (it "overrides" the default). So we can simplify further:

```

class A { public: int a = 7, b = 5; // default initializers
A() { }
A(int a_val): a(a_val) { } // Ctor initialization overrides
A(D d): b(g(d)) { } // Ctor initialization overrides
private: HashingFunction h_algo{"MD5"}; // Hash algo: Crypto. hash to be applied to all A instances
std::string s{"Ctor run"}; // Tracer: String indicating state in object lifecycle
};

```

So, this is a great advantage because you may have multiple constructors, as we have just seen in the delegating constructor, which initialize you can have multiple constructors, as you have just seen in the delegating constructor case, that initialize the same data member with the same value. So, here is an example of a class where each of these three constructors initialized with the same value. It is difficult to write it repeatedly without making mistake and a lot of code messes.

Instead, what you can now do is you can write it simply along with the data member declaration that `h_algo` has an initial value which is within quotes MD5 and so on. So, what will happen in the constructor, once you write this in the constructor, if you are not initializing that variable or

that data member, that data member will get the initialization from this in-class member initializer.

Now, here, so this, these were taken care of. So, all that I am left with is a and b. And then again I see that a is 7, b is 5 in two cases. But there are cases where they are different. These are cases where they are different. Not, everywhere they are same unlike h_algo. So, what I do, I put a default initialization of 7 and 5, because they are more frequently occurring. So, with that, I would not need to specify anything here. But while I define this constructor I would not need to specify this. But I will still need to specify what is the value of a.

So, now, there are two initializations available one which is in-class member initial value, which is 7, and one is a_val which the constructor is saying. The rule is whatever the constructor says will prevail. And with that, you can see this whole mess of code that you have here now becomes so simple, so readable, so maintainable and so on. So, in-member initialization is a great feature to use.

(Refer Slide Time: 27:46)

Inheriting Constructors

Module M54
Partha Pratim Das
Objectives & Outline
Inheritance
Control of default member and static
Control flow
Debugging Constructors
Inheriting Constructors
Overload Control
Overload Control
Application
Module Summary

Inheriting Constructors

Sources:

- Inherited constructors, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

Programming in Modern C++ Partha Pratim Das M54.22

The next is about inheriting constructors.

(Refer Slide Time: 27:53)

Inheriting Constructors

A member of a base class is not in the same scope as a member of a derived class. We can "lift" a set of overloaded functions from a base class into a derived class.

```
struct B { void f(double); };
struct D : B {
    void f(int);
};
B b; b.f(4.5); // fine
// surprise: calls f(int) with argument 4
D d; d.f(4.5);
```

```
struct B { void f(double); };
struct D : B {
    using B::f; // bring all f()'s from B into scope
    void f(int); // add a new f()
};
B b; b.f(4.5); // fine
// fine: calls D::f(double) which is B::f(double)
D d; d.f(4.5);
```

- Stroustrup has said that "Little more than a historical accident prevents using this to work for a constructor as well as for an ordinary member function"
- C++11 provides that facility to lift a base class constructor into the derived class
- We present an illustrative example for various scenarios

You know about inheriting member functions and you know that a base, a derived class will inherit the member functions of the base class. But the moment you declare a member functions in the derived class by the same name, the base class member function will get delete. So, here you have a f() here double and the moment you define it in the derived class you have f() of int then f() of double is also hidden.

We know that by using B::f, we can make the member function of the base available here as well that is inheriting explicitly from the base without being hidden. So, now, D has two f() functions. But this entire thing applies only to ordinary member functions not to the constructor.

(Refer Slide Time: 28:57)

Inheriting Constructors

```
#include <iostream>
#include <string>

class B { public: // Base class
    B() { std::cout << "B::B() "; }
    B(int) { std::cout << "B::B(int) "; }
    void f(int) { std::cout << "B::f(int) "; }
};

class D : public B { public: // Derived class
    using B::f; // lift B::f into D's scope -- works in C++03 and C++11
    void f(string) { std::cout << "D::f(string) "; } // provide a new overload f
    void f(int) { std::cout << "D::f(int) "; } // prefer this override f to B::f(int)
    using B::B; // lift B::B into D's scope -- new in C++11 -- Inheriting Constructors
    // causes implicit declaration of D::D() (or D::D(int)), which, if used, calls B::B() (or B::B(int))
    D(const string&) { std::cout << "D::D(string) "; } // provide a new overloaded constructor
    D(int) : B(0) { std::cout << "D::D(int) "; } // prefer this overloaded constructor to B::B(int)
};

int main() {
    B b(5); std::cout << std::endl; // B::B(int)
    D d; std::cout << std::endl; // B::B() // okay due to ctor inheritance if D::D() is undeclared
    D d1(2); std::cout << std::endl; // B::B(int) D::D(int)
    D d2("pppd"); std::cout << std::endl; // B::B() D::D(string)
    b.f(3); std::cout << std::endl; // B::f(int)
    d1.f(1); std::cout << std::endl; // D::f(int)
    d2.f("cd"); std::cout << std::endl; // D::f(string)
}
```

Inheriting Constructors

```
#include <iostream>
#include <string>

class B { public: // Base class
    B() { std::cout << "B::B() "; }
    B(int) { std::cout << "B::B(int) "; }
    void f(int) { std::cout << "B::f(int) "; }
};

class D : public B { public: // Derived class
    using B::f; // lift B::f into D's scope -- works in C++03 and C++11
    void f(string) { std::cout << "D::f(string) "; } // provide a new overload f
    void f(int) { std::cout << "D::f(int) "; } // prefer this override f to B::f(int)
    using B::B; // lift B::B into D's scope -- new in C++11 -- Inheriting Constructors
    // causes implicit declaration of D::D() (or D::D(int)), which, if used, calls B::B() (or B::B(int))
    D(const string&) { std::cout << "D::D(string) "; } // provide a new overloaded constructor
    D(int) : B(0) { std::cout << "D::D(int) "; } // prefer this overloaded constructor to B::B(int)
};

int main() {
    B b(5); std::cout << std::endl; // B::B(int)
    D d; std::cout << std::endl; // B::B() // okay due to ctor inheritance if D::D() is undeclared
    D d1(2); std::cout << std::endl; // B::B(int) D::D(int)
    D d2("pppd"); std::cout << std::endl; // B::B() D::D(string)
    b.f(3); std::cout << std::endl; // B::f(int)
    d1.f(1); std::cout << std::endl; // D::f(int)
    d2.f("cd"); std::cout << std::endl; // D::f(string)
}
```

```

#include <iostream>
#include <string>

class B { public: // Base class
    B() { std::cout << "B::B() "; }
    B(int) { std::cout << "B::B(int) "; }
    void f(int) { std::cout << "B::f(int) "; }
};

class D : public B { public: // Derived class
    using B::f; // lift B::f into D's scope -- works in C++03 and C++11
    void f(string) { std::cout << "D::f(string) "; } // provide a new overload f
    void f(int) { std::cout << "D::f(int) "; } // prefer this override f to B::f(int)
    using B::B; // lift B::B into D's scope -- new in C++11 -- Inheriting Constructors
    // causes implicit declaration of D::D() (or D::D(int)), which, if used, calls B::B() (or B::B(int))
    D(const string&) { std::cout << "D::D(string) "; } // provide a new overloaded constructor
    D(int): B(0) { std::cout << "D::D(int) "; } // prefer this overloaded constructor to B::B(int)
};

int main() {
    B b(5); std::cout << std::endl; // B::B(int)
    D d; std::cout << std::endl; // B::B() // okay due to ctor inheritance if D::D() is undeclared
    D di(2); std::cout << std::endl; // B::B(int) D::D(int)
    D d2("ppd"); std::cout << std::endl; // B::B() D::D(string)
    b.f(3); std::cout << std::endl; // B::f(int)
    d1.f(1); std::cout << std::endl; // D::f(int)
    d2.f("ed"); std::cout << std::endl; // D::f(string)
}

```

What C++11 has done, it has simply allowed that to be also be present. So, if this has two constructors and if I have not given any constructor here, then if I do say `D d` that is I want to do a default construction what will happen. It will do the construction by the default constructor of the base. This is fine. Now, suppose I have provided, I have not provided this, but I provided this, so what will it do? It will hide the default constructor of D. It will hide the default constructor of D and we will say that there is no constructor and because there is only one constructor given. So, this problem was not directly solvable in C++03.

So, now what you can do is you can say using `B::f`, so which means that these two that is a default constructor of D and a default construct, and a parameterized constructor of D with int is available at this point, then it depends on what you, how you override it, how you overload it, and that will take the effect. So, that is the inheriting of constructors.

(Refer Slide Time: 30:37)

The slide is titled "Inheriting Constructors" and shows two examples of C++ code and their compiler outputs. The code defines a base class B and a derived class D. The first example shows D inheriting from B with various constructors and a 'using' directive. The second example shows D inheriting from B with various constructors and a 'using namespace' directive. The outputs show compilation errors and successful compilations for different combinations of these directives.

Example 1: Using namespace B

```

class B: B(); B:B(int); B:f(int);
class D: using B::f; D::f(string); D::f(int); using B::B; D::D(const string&); D::D(int);

```

Calls	// using B::B; // D(const string&); // D(int): B(0);	// using B::B; D(const string&); // D(int): B(0);	<u>using B::B;</u> D(const string&); // D(int): B(0);	using B::B; D(const string&); D(int): B(0);
B b(5); D d; D d1(2); D d2("ppd");	okay okay error: D::D(int) error: D::D(const char[4]) B::B(int) hidden	okay error: D::D() error: D::D(int) okay B::B(int) hidden	B::B(int) B::B() B::B(int) B::B() D::D(string) D exposes B::B's	B::B(int) B::B() B::B(int) D::D(int) B::B() D::D(string) Overloads
Calls	// using B::f; // void f(string); // void f(int);	// using B::f; void f(string); // void f(int);	using B::f; void f(string); // void f(int);	using B::f; void f(string); void f(int);
b.f(3); d1.f(1); d2.f("cd");	okay: B::f(int) okay: B::f(int) error: D::f(const char*) D inherits B::f(int)	okay: B::f(int) error: D::f(int) okay: D::f(string) B::f(int) hidden	B::f(int) B::f(int) D::f(string) D exposes B::f(int)	B::f(int) D::f(int) D::f(string) Overload + Override

Example 2: Using B::B

```

class B: B(); B:B(int); B:f(int);
class D: using B::f; D::f(string); D::f(int); using B::B; D::D(const string&); D::D(int);

```

Calls	// using B::B; // D(const string&); // D(int): B(0);	// using B::B; D(const string&); // D(int): B(0);	<u>using B::B;</u> D(const string&); // D(int): B(0);	using B::B; D(const string&); D(int): B(0);
B b(5); D d; D d1(2); D d2("ppd");	okay okay error: D::D(int) error: D::D(const char[4]) B::B(int) hidden	okay error: D::D() error: D::D(int) okay B::B(int) hidden	B::B(int) B::B() <u>B::B(int)</u> B::B() D::D(string) D exposes B::B's	B::B(int) B::B() B::B(int) D::D(int) B::B() D::D(string) Overloads
Calls	// using B::f; // void f(string); // void f(int);	// using B::f; void f(string); // void f(int);	using B::f; void f(string); // void f(int);	using B::f; void f(string); void f(int);
b.f(3); d1.f(1); d2.f("cd");	okay: B::f(int) okay: B::f(int) error: D::f(const char*) D inherits B::f(int)	okay: B::f(int) error: D::f(int) okay: D::f(string) B::f(int) hidden	B::f(int) B::f(int) D::f(string) D exposes B::f(int)	B::f(int) D::f(int) D::f(string) Overload + Override

Here I have made, again, that principle of single slide take back. Here I have made a single slide where I show that with or without using and with different combinations of which function is available or which constructor is made available, what will be the basic effect try to go through each one entry for this and get comfortable with the inheriting constructors.

For example, if you are using this, that is if you inherit, you have defined a constructor with string, but you are trying to do a default construction. Then it will not give you a compilation error like C++03 without this is a compilation error, because there is no default constructor, but here the default constructor of B will be used.

But if you have provided similarly for this say for D, here it is a parameterized construction. So, look at this, a parameterized construction. So, if you have provided something some constructor then you will not be able to compile this because there is no constructor. If you inherit from B, then the constructor, parameterized constructor of B taking int will be used. If you overload that, override that in a way by providing a constructor in D of taking a parameter int and then use the base class constructor you can see that that constructor will be used. So, this is what you gain by providing int, the inheriting the construction, which is what was not available in C++03.

(Refer Slide Time: 32:49)

The slide is titled "Inheriting Constructors: Member Initialization in Derived". It contains the following content:

- Inheriting constructors into classes with data members risky. Consider a base class B:

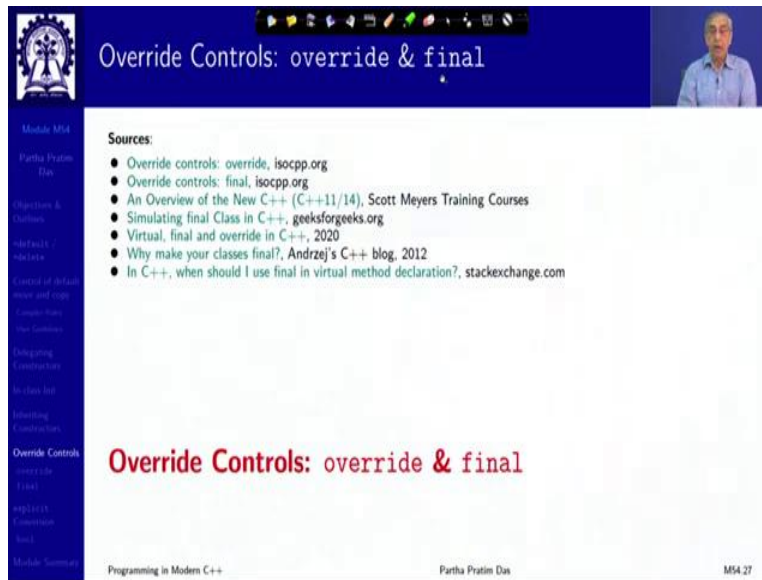

```
class B { public:
    explicit B(int);
};
```
- Derive a class D with data members:

<p style="text-align: center;">Default Initialization</p> <pre>class D: public B { public: using B::B; // Inherits B::B(int) private: std::u16string name; int x, y; }; D d(10); // compiles, but // d.name is default-initialized, and // d.x and d.y are uninitialized</pre>	<p style="text-align: center;">In-class Initialization</p> <pre>class D: public B { public: using B::B; // Inherits B::B(int) private: std::u16string name = "Uninitialized"; int x = 0, y = 0; }; D d(10); // d.name == "Uninitialized", // d.x == d.y == 0</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
- Use in-class member initialization when inheriting constructor/s

At the bottom of the slide, it says "Programming in Modern C++" on the left, "Partha Pratim Das" in the center, and "MS4.26" on the right.

So, inherited constructors are something which is very, very important, but you have to be careful if it has, if the derived class has data members, then you may be in for surprise because your inherited constructor obviously will not construct the data members of derived class because inherited constructor is of the base class. So, you have to use proper in-class member initialization to make sure that your derived class data members are properly initialized.

(Refer Slide Time: 33:24)



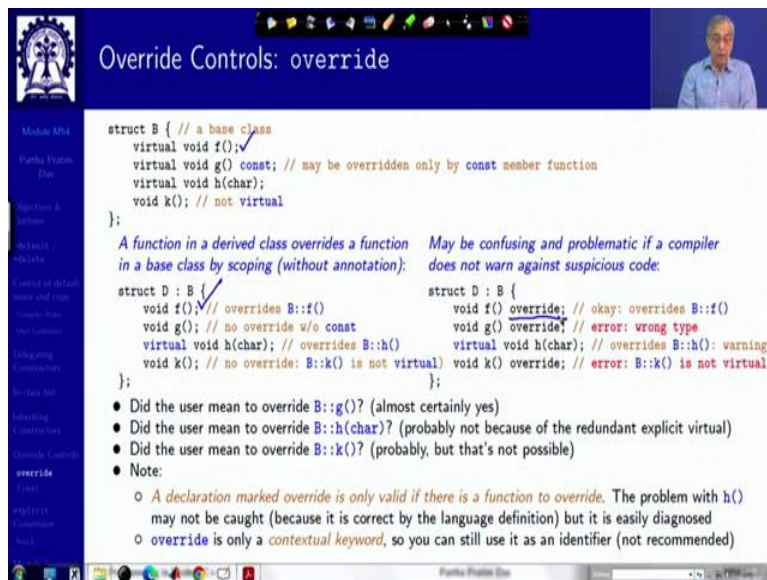
Override Controls: override & final

Sources

- Override controls: override, isocpp.org
- Override controls: final, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Simulating final Class in C++, geeksforgeeks.org
- Virtual, final and override in C++, 2020
- Why make your classes final?, Andrzej's C++ blog, 2012
- In C++, when should I use final in virtual method declaration?, stackexchange.com

Override Controls: override & final

Programming in Modern C++ Partha Pratim Das M54.27



Override Controls: override

```
struct B { // a base class
    virtual void f();
    virtual void g() const; // may be overridden only by const member function
    virtual void h(char);
    void k(); // not virtual
};
```

A function in a derived class overrides a function in a base class by scoping (without annotation): *May be confusing and problematic if a compiler does not warn against suspicious code.*

```
struct D : B {
    void f(); // overrides B::f()
    void g(); // no override w/o const
    virtual void h(char); // overrides B::h()
    void k(); // no override: B::k() is not virtual
};
```

```
struct D : B {
    void f() override; // okay: overrides B::f()
    void g() override; // error: wrong type
    virtual void h(char); // overrides B::h(): warning?
    void k() override; // error: B::k() is not virtual
};
```

- Did the user mean to override B::g()? (almost certainly yes)
- Did the user mean to override B::h(char)? (probably not because of the redundant explicit virtual)
- Did the user mean to override B::k()? (probably, but that's not possible)
- Note:
 - A declaration marked `override` is only valid if there is a function to override. The problem with `h()` may not be caught (because it is correct by the language definition) but it is easily diagnosed
 - `override` is only a *contextual keyword*, so you can still use it as an identifier (not recommended)

Partha Pratim Das

Override Controls: override

```

struct B { // a base class
    virtual void f();
    virtual void g() const; // may be overridden only by const member function
    virtual void h(char);
    void k(); // not virtual
};

struct D : B {
    void f(); // overrides B::f()
    void g(); // no override w/o const
    virtual void h(char); // overrides B::h()
    void k(); // no override: B::k() is not virtual
};

struct D : B {
    void f() override; // okay: overrides B::f()
    void g() override; // error: wrong type
    virtual void h(char); // overrides B::h(): warning?
    void k() override; // error: B::k() is not virtual
};

```

A function in a derived class overrides a function in a base class by scoping (without annotation): *May be confusing and problematic if a compiler does not warn against suspicious code.*

- Did the user mean to override B::g()? (almost certainly yes)
- Did the user mean to override B::h(char)? (probably not because of the redundant explicit virtual)
- Did the user mean to override B::k()? (probably, but that's not possible)
- Note:
 - A declaration marked `override` is only valid if there is a function to override. The problem with `h()` may not be caught (because it is correct by the language definition) but it is easily diagnosed
 - `override` is only a contextual keyword, so you can still use it as an identifier (not recommended)

Overrides are also given some more controls. This is something which does not add anything specific but it is more for clarity. For example, as you inherit functions for override like you have a function `f()` in the base class which is virtual and if you write it again here, you override, here in the override feature what you say is you explicitly say that you override. It does not do anything else. It does not give you any other functionality, but it just makes it easier to understand.

For example, here just the difference in meaning you can see that here you have a function `g()`, here you have written this. Now, in the `g()` virtual so first you will tend to think that this override, because it is `g()` function is there, but it is actually not because what you inherit is not `g()` but `g()` which is constant. But what you are writing here is a `g()` which is non-cont. Therefore, there is a overload. This is a different, this is without the `const`.

So, if you write `override`, now the compiler will be able to help you on the small slip. The compiler will be able to tell you look this is not a override, because `g()` is not constant, whereas your parent class member `g()` is a `const` function. So, it will refuse to combine because of the wrong type. So, these are the kind of advantages you can get by using `override`. But `override` as such is, this is a keyword which is new concept being added that is the contextual keyword in the sense that you can still, unlike other keywords, you can still keep on using `override` as a variable which is not advisable to do that. But only when it is used at this place, it is, it behaves like a keywords.

(Refer Slide Time: 35:32)

Override Controls: final

- Sometimes, a programmer wants to prevent a **virtual** function from being overridden. This can be achieved by adding the specifier **final**. For example:

```
struct B {  
    virtual void f() const final; // do not override  
    virtual void g();  
};  
struct D : B {  
    void f() const; // error: D::f attempts to override final B::f  
    void g(); // okay  
};
```
- **Why should we use final in C++?**
 - If it is performance (inlining) we want or we simply never want to override, it is typically better not to define a function to be **virtual**
 - This is in contrast to Java where all functions are **virtual** and **final** provides better performance
- It should be used sparingly with care because in a way it contradicts the polymorphic design and in C++ there are other ways to circumvent the required issues in a hierarchy
- **Note:**
 - The **final** keyword applies to member function, but unlike **override**, it also applies to types:

```
class X final { /* ... */ };
```

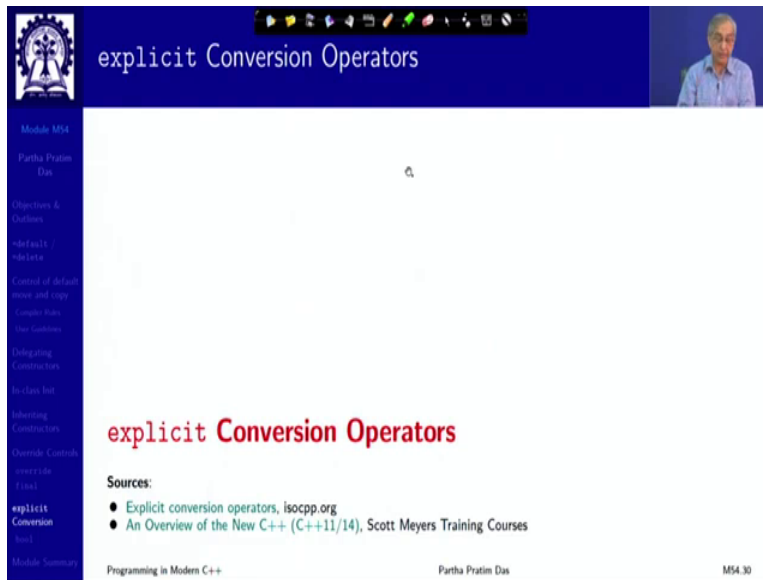
This prevent the type X to be inherited from
 - **final** is only a **contextual keyword**, so you can still use it as an identifier (not recommended)

The other override control that has been added is feature called final which is similar to what Java has but very different from what Java has. So, if you have a virtual function then you can say it is final. If you do that, then any specialization of that class will not be able to override this particular virtual function. Similarly, a class can be said to be final. You can say that a class is final then you will not be able to derive from that class.

Now, it is still a lot of debate as to whether this feature has any specific value in C++, because in Java final has a different requirement, because in Java all functions are virtual and therefore there is a overhead of calling those functions. In C++ first of all the overhead of calling a virtual function is extremely minimal. But more importantly C++ does have non-virtual functions.

So, in java you need to use final for those reasons. In C++11 or in C++ do you really need that. The debate is still going on. And after a lot of study of the recent material also I failed to produce here a meaningful example of where final really adds value in terms of programming or semantics, so we know that it is there, but use it only if you are convinced.

(Refer Slide Time: 37:06)



explicit Conversion Operators

Module M54
Partha Pratim Das

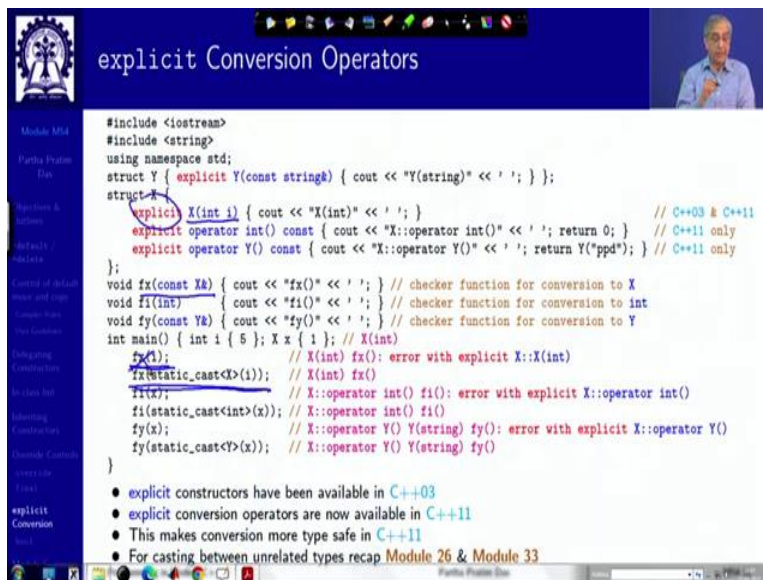
Objectives & Outcomes
Defaults & Aliases
Control of default access and scope
Control flow
Delegating Constructors
In-class Init
Inheriting Constructors
Overriding Constructors
Explicit Conversion

explicit Conversion Operators

Sources:

- Explicit conversion operators, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses

Programming in Modern C++ Partha Pratim Das M54.30



explicit Conversion Operators

Module M54
Partha Pratim Das

```
#include <iostream>
#include <string>
using namespace std;
struct Y { explicit Y(const string&) { cout << "Y(string)" << ' '; } };
struct X {
    explicit X(int i) { cout << "X(int)" << ' '; } // C++03 & C++11
    explicit operator int() const { cout << "X::operator int()" << ' '; return 0; } // C++11 only
    explicit operator Y() const { cout << "X::operator Y()" << ' '; return Y("ppd"); } // C++11 only
};
void fx(const X&) { cout << "fx()" << ' '; } // checker function for conversion to X
void fi(int) { cout << "fi()" << ' '; } // checker function for conversion to int
void fy(const Y&) { cout << "fy()" << ' '; } // checker function for conversion to Y
int main() { int i { 5 }; X x { 1 }; // X(int)
    fx(1); // X(int) fx(): error with explicit X::X(int)
    fi(static_cast<int>(1)); // X(int) fi()
    fi(x); // X::operator int() fi(): error with explicit X::operator int()
    fy(static_cast<int>(x)); // X::operator int() fi()
    fy(x); // X::operator Y() Y(string) fy(): error with explicit X::operator Y()
    fy(static_cast<Y>(x)); // X::operator Y() Y(string) fy()
}
```

- explicit constructors have been available in C++03
- explicit conversion operators are now available in C++11
- This makes conversion more type safe in C++11
- For casting between unrelated types recap [Module 26](#) & [Module 33](#)

Partha Pratim Das

The screenshot shows a presentation slide with the title "explicit Conversion Operators". On the left is a navigation menu with items like "Module 33", "Partha Prasad Das", "Functions & Polymorph", "Methods & Overload", "Control of Access", "User-Defined Literals", "Designing Constructors", "Arithmetic Operators", "Identifying Constructors", "User-Defined Literals", "Explicit Conversion", and "List". The main content area contains C++ code and a list of bullet points.

```

#include <iostream>
#include <string>
using namespace std;
struct Y { explicit Y(const string&) { cout << "Y(string)" << ' '; } };
struct X {
    explicit X(int i) { cout << "X(int)" << ' '; } // C++03 & C++11
    explicit operator int() const { cout << "X::operator int()" << ' '; return 0; } // C++11 only
    explicit operator Y() const { cout << "X::operator Y()" << ' '; return Y("ppd"); } // C++11 only
};
void fx(const X&) { cout << "fx()" << ' '; } // checker function for conversion to X
void fi(int) { cout << "fi()" << ' '; } // checker function for conversion to int
void fy(const Y&) { cout << "fy()" << ' '; } // checker function for conversion to Y
int main() { int i { 5 }; X x { 1 }; // X(int)
    fx(i); // X(int) fx(): error with explicit X::X(int)
    fx(static_cast<X>(1)); // X(int) fx()
    X(x); // X::operator int() fi(): error with explicit X::operator int()
    Y(static_cast<int>(x)); // X::operator int() fi()
    fy(x); // X::operator Y() Y(string) fy(): error with explicit X::operator Y()
    fy(static_cast<Y>(x)); // X::operator Y() Y(string) fy()
}

```

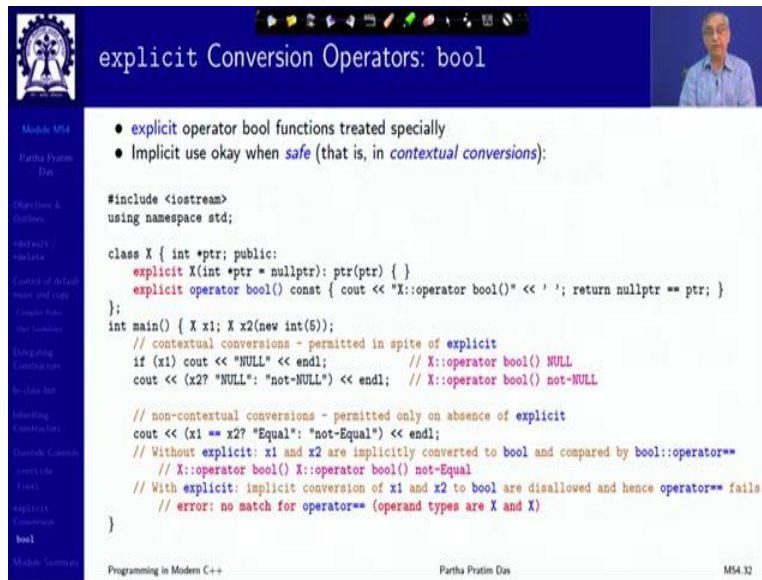
- explicit constructors have been available in C++03
- explicit conversion operators are now available in C++11
- This makes conversion more type safe in C++11
- For casting between unrelated types recap [Module 26 & Module 33](#)

Finally, in terms of type casting, we have seen a lot of that in Module 26 and Module 33, in terms of type casting we have seen that type can be cast by having a constructor or having a type cast operator. In terms of the constructor we can make the constructor explicit to say that implicit conversion is not allowed. But in terms of the type cast operator no such feature was there which is convert by type cast operator, it will be implicitly, explicitly always it will be permitted.

So, this is a basic difference between, if you have a fx() we expect the reference to a constant reference to an X object and X as a constructor like this, this is the implicit instantiation this is the explicit instantiation. If this is explicit then this will not compile. But the similar thing you cannot do in terms of the cast operator. So, if you expect an int and pass an X either way then in C++03 both of them will always compile. You cannot control that it has to be explicit.

So, what you get in C++11 is you get to use the explicit keyword if you want in terms of the cast operator so that you can invalidate this use. You can say that only casting will have to be only explicit. So, that is a semantics that it supports, very simple in that way to use and a nice addition.

(Refer Slide Time: 38:47)



explicit Conversion Operators: bool

- explicit operator bool functions treated specially
- Implicit use okay when *safe* (that is, in *contextual conversions*):

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

class X { int *ptr; public:
    explicit X(int *ptr = nullptr): ptr(ptr) { }
    explicit operator bool() const { cout << "X::operator bool()" << ' '; return nullptr == ptr; }
};

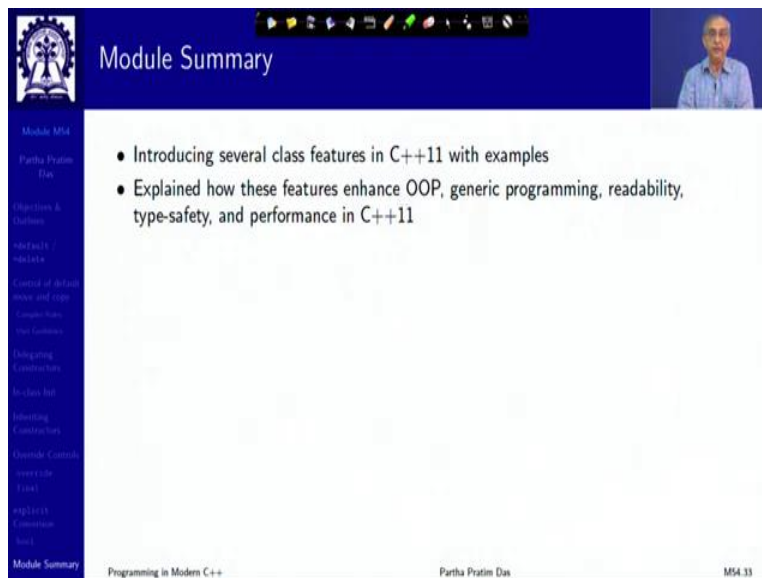
int main() { X x1; X x2(new int(5));
    // contextual conversions - permitted in spite of explicit
    if (x1) cout << "NULL" << endl; // X::operator bool() NULL
    cout << (x2? "NULL": "not-NULL") << endl; // X::operator bool() not-NULL

    // non-contextual conversions - permitted only on absence of explicit
    cout << (x1 == x2? "Equal": "not-Equal") << endl;
    // Without explicit: x1 and x2 are implicitly converted to bool and compared by bool::operator==
    // X::operator bool() X::operator bool() not-Equal
    // With explicit: implicit conversion of x1 and x2 to bool are disallowed and hence operator== fails
    // error: no match for operator== (operand types are X and X)
}
```

Programming in Modern C++ Partha Pratim Das MS4.32

And here are some examples of how to use that particularly in the context of bool where it behaves with a different use.

(Refer Slide Time: 38:57)



Module Summary

- Introducing several class features in C++11 with examples
- Explained how these features enhance OOP, generic programming, readability, type-safety, and performance in C++11

Programming in Modern C++ Partha Pratim Das MS4.33

So, in this module we have discussed several class features of C++11 with examples that enhance the object oriented generic programming features, readability, type safety and performance of the language. Thank you very much for the attention and we will meet in the next module.