



## Module M54

Partha Pratim  
Das

Objectives &  
Outlines

`=default` /  
`=delete`

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

Module Summary

# Programming in Modern C++

Module M54: C++11 and beyond: Class Features

Partha Pratim Das

Department of Computer Science and Engineering  
Indian Institute of Technology, Kharagpur

*ppd@cse.iitkgp.ac.in*

*All url's in this module have been accessed in September, 2021 and found to be functional*



# Module Recap

## Module M54

Partha Pratim  
Das

### Objectives & Outlines

`=default` /  
`=delete`

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

Module Summary

- Learnt different techniques without or with `std::function` to write and use non-recursive and recursive  $\lambda$  expressions in C++11 / C++14
- Several practice examples to be tried and tested



# Module Objectives

## Module M54

Partha Pratim  
Das

### Objectives & Outlines

`=default` /  
`=delete`

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

Module Summary

- 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



# Module Outline

## Module M54

Partha Pratim  
Das

### Objectives & Outlines

`=default /`  
`=delete`

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

Module Summary

- 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



# default / delete Functions

## Module M54

Partha Pratim Das

Objectives & Outlines

`=default` / `=delete`

Control of default move and copy

Compiler Rules

User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls

`override`

`final`

`explicit` Conversion

`bool`

Module Summary

## Sources:

- `=default` and `=delete`, [isocpp.org](http://isocpp.org)
- C++ Class and Preventing Object Copy, [ariya.io](http://ariya.io), 2015
- C++ Core Guidelines, [github.com](http://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



# =default and =delete

## Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules  
User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- 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 Member Functions

## Module M54

Partha Pratim  
Das

Objectives &  
Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- 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*



# default Member Functions

## Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules  
User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- 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  
    ...  
};
```





# delete Functions

## Module M54

Partha Pratim Das

Objectives & Outlines

=default /  
=delete

Control of default move and copy

Compiler Rules  
User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- **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  
    ...  
};
```



# delete Functions

Module M54

Partha Pratim  
Das

Objectives &  
Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- 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*)
```



# Control of default move and copy

## Module M54

Partha Pratim Das

Objectives & Outlines

`=default /`  
`=delete`

Control of default move and copy

Compiler Rules  
User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

Module Summary

## Sources:

- [Control of default move and copy](http://isocpp.org), isocpp.org
- [The rule of three/five/zero](http://cppreference.com), cppreference.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

## Control of default move and copy



# Control of default move and copy

## Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules

User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls

override final

explicit Conversion

bool

Module Summary

- 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: Compiler Rules

## Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules

User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls

override final

explicit Conversion

bool

Module Summary

## • 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=
```



# Control of default move and copy: Compiler Rules

## Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules

User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls

override final

explicit Conversion

bool

Module Summary

## ● Copy Rules

- 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
- If *any move is explicitly specified* (declared, defined, =default, or =delete):
  - ▷ *no copy is generated by default*
- **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: User Guidelines

## Module M54

Partha Pratim  
Das

Objectives &  
Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- **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



# Delegating Constructors

## Module M54

Partha Pratim  
Das

Objectives &  
Outlines

`=default` /  
`=delete`

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

Module Summary

NPTEL

## Delegating Constructors

### Sources:

- [Delegating constructors](http://isocpp.org), isocpp.org
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses





# Delegating Constructors

Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules  
User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

## 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 M54

Partha Pratim Das

Objectives & Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- 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): i(i) { cout << "Data(int)" << endl; } // #2  
};  
  
int main() {  
    Data d;  
}  
  
// Data(int)  
// Data()
```



# In-class Member Initializers

## Module M54

Partha Pratim  
Das

Objectives &  
Outlines

`=default` /  
`=delete`

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

Module Summary

## In-class Member Initializers

### Sources:

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



# In-class Member Initializers

Module M54

Partha Pratim Das

Objectives & Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- 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;
};
```



# In-class Member Initializers

Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules  
User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- 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
};
```



# Inheriting Constructors

## Module M54

Partha Pratim  
Das

Objectives &  
Outlines

`=default` /  
`=delete`

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

Module Summary

NPTEL

## Inheriting Constructors

### Sources:

- [Inherited constructors](https://ericniebler.com/2014/05/27/default-constructors/), [isocpp.org](https://ericniebler.com/2014/05/27/default-constructors/)
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses



# Inheriting Constructors

## Module M54

Partha Pratim Das

Objectives & Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules

User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

*A member of a base class is not in the same scope as a member of 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);
```

*We can “lift” a set of overloaded functions from a base class into a derived class:*

```
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



# Inheriting Constructors

Module M54

Partha Pratim Das

Objectives & Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

```
#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("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("cd");  std::cout << std::endl; // D::f(string)
}
```

Programming in Modern C++





# Inheriting Constructors

Module M54

Partha Pratim Das

Objectives & Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

```
class B: 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);	using B::B; 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



# Inheriting Constructors: Member Initialization in Derived Class

Module M54

Partha Pratim  
Das

Objectives &  
Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override

final

explicit  
Conversion

bool

Module Summary

- 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:

## Default Initialization

```
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
```

## In-class Initialization

```
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
```

- Use in-class member initialization when inheriting constructor/s



# Override Controls: `override` & `final`

## Module M54

Partha Pratim Das

Objectives & Outlines

`=default` / `=delete`

Control of default move and copy

Compiler Rules  
User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

Module Summary

## 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`



# Override Controls: override

Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules

User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls

override final

explicit Conversion

bool

Module Summary

```
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)



# Override Controls: final

Module M54

Partha Pratim  
Das

Objectives &  
Outlines

=default /  
=delete

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
override  
final

explicit  
Conversion  
bool

Module Summary

- 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)



# explicit Conversion Operators

## Module M54

Partha Pratim  
Das

Objectives &  
Outlines

`=default` /  
`=delete`

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
`override`  
`final`

**explicit**  
Conversion  
`bool`

Module Summary

## explicit Conversion Operators

### Sources:

- [Explicit conversion operators](http://isocpp.org), isocpp.org
- [explicit specifier](#), cppreference
- [An Overview of the New C++ \(C++11/14\)](#), Scott Meyers Training Courses



# explicit Conversion Operators

Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules  
User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls  
override  
final

explicit Conversion  
bool

Module Summary

```

#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>(i)); // X(int) fx()
    fi(x); // X::operator int() fi(): error with explicit X::operator int()
    fi(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**



# explicit Behavior Example

# Post-Recording

Module M54

Partha Pratim Das

Objectives & Outlines

=default / =delete

Control of default move and copy

Compiler Rules  
User Guidelines

Delegating Constructors

In-class Init.

Inheriting Constructors

Override Controls  
override  
final

explicit Conversion  
bool

Module Summary

```

struct A { // converting ctor & operator
    A(int) { }
    A(int, int) { } // C++11
    operator bool() const
    { return true; }
};

```

```

int main() {
    A a1 = 1;          // OK: copy-initialization selects A::A(int)
    A a2(2);           // OK: direct-initialization selects A::A(int)
    A a3 {4, 5};       // OK: direct-list-initialization selects A::A(int, int)
    A a4 = {4, 5};     // OK: copy-list-initialization selects A::A(int, int)
    A a5 = (A)1;       // OK: explicit cast performs static_cast
    if (a1);           // OK: A::operator bool()
    bool na1 = a1;     // OK: copy-initialization selects A::operator bool()
    bool na2 = static_cast<bool>(a1); // OK: static_cast performs direct-initialization
}

```

```

// B b1 = 1;          // error: copy-initialization does not consider B::B(int)
B b2(2);             // OK: direct-initialization selects B::B(int)
B b3 {4, 5};         // OK: direct-list-initialization selects B::B(int, int)
// B b4 = {4, 5};     // error: copy-list-initialization does not consider B::B(int,int)
B b5 = (B)1;         // OK: explicit cast performs static_cast
if (b2);             // OK: B::operator bool()
// bool nb1 = b2;     // error: copy-initialization does not consider B::operator bool()
bool nb2 = static_cast<bool>(b2); // OK: static_cast performs direct-initialization
}

```

Programming in Modern C++

Partha Pratim Das

M54.32





# 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)
}
```



# Module Summary

## Module M54

Partha Pratim Das

Objectives & Outlines

`=default` /  
`=delete`

Control of default  
move and copy

Compiler Rules  
User Guidelines

Delegating  
Constructors

In-class Init.

Inheriting  
Constructors

Override Controls  
`override`  
`final`

`explicit`  
Conversion  
`bool`

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