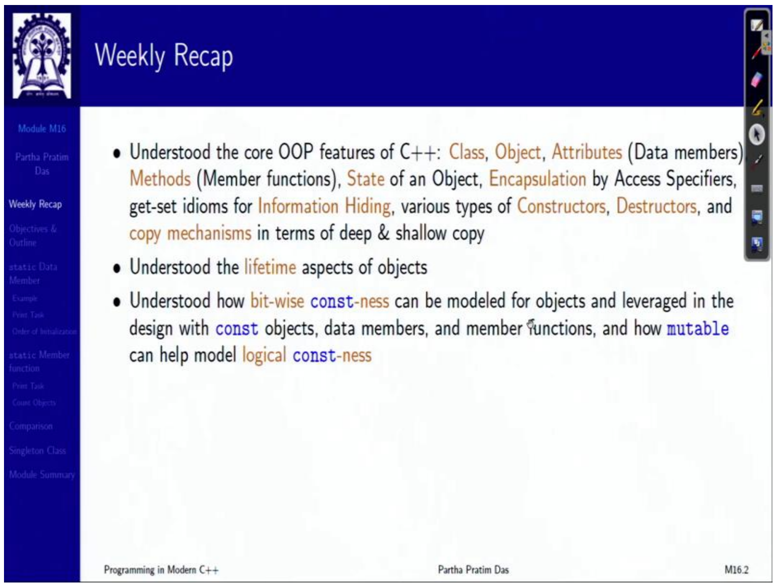
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Lecture 16
Static Members

Welcome to Programming in Modern C++. We are in week four and starting module 16.

(Refer slide time: 0:35)



Weekly Recap

- Understood the core OOP features of C++: Class, Object, Attributes (Data members), Methods (Member functions), State of an Object, Encapsulation by Access Specifiers, get-set idioms for Information Hiding, various types of Constructors, Destructors, and copy mechanisms in terms of deep & shallow copy
- Understood the lifetime aspects of objects
- Understood how bit-wise const-ness can be modeled for objects and leveraged in the design with const objects, data members, and member functions, and how mutable can help model logical const-ness

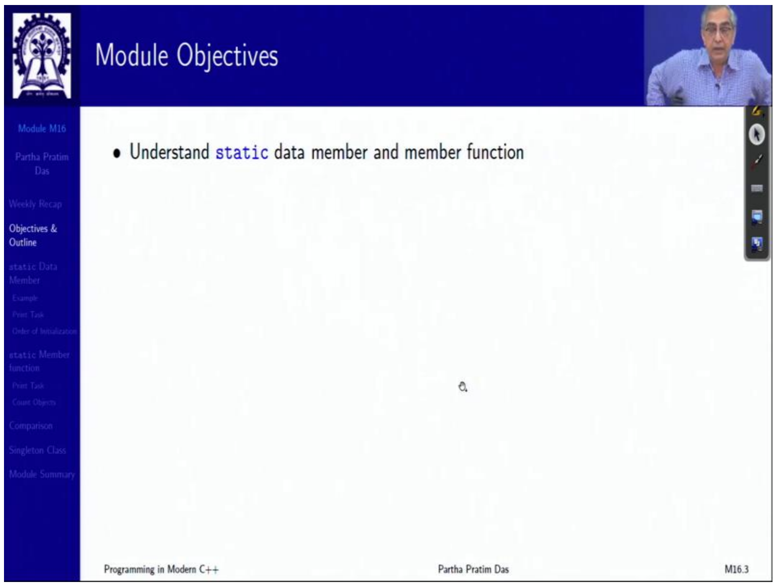
Programming in Modern C++ Partha Pratim Das M16.2

We have in the last week covered a major part of the core object-oriented programming features of C++ including what is a class, what is instance object, attributes, data members, methods or member functions, what is meant by the state of an object encapsulation by private and public access specifiers?

How to create a properly designed information hiding structure by using get-set idioms, various types of constructors, default, parameterized, free and so on, destructor and copy mechanism, copy constructor as well as copy assignment operator with deep and shallow copy options for pointer values. We also based on this we also understood the lifetime of different objects, which is very key to our design.

And finally, we took a look at const-ness at a bitwise level where we can have objects which are constant or data members can be constant, member functions can be constant and a combination of these can give you a good design in terms of the logical const-ness which is further supported by the mutable feature of C++. So, we have got a foundational basis on the object oriented programming aspects. Now, we will build further on this.

(Refer slide time: 2:07)



Module Objectives

- Understand static data member and member function

Module M16

Partha Pratim Das

Weekly Recap

Objectives & Outline

static Data Member

Example

Print Task

Order of Initialization

static Member function

Print Task

Count Objects

Comparison

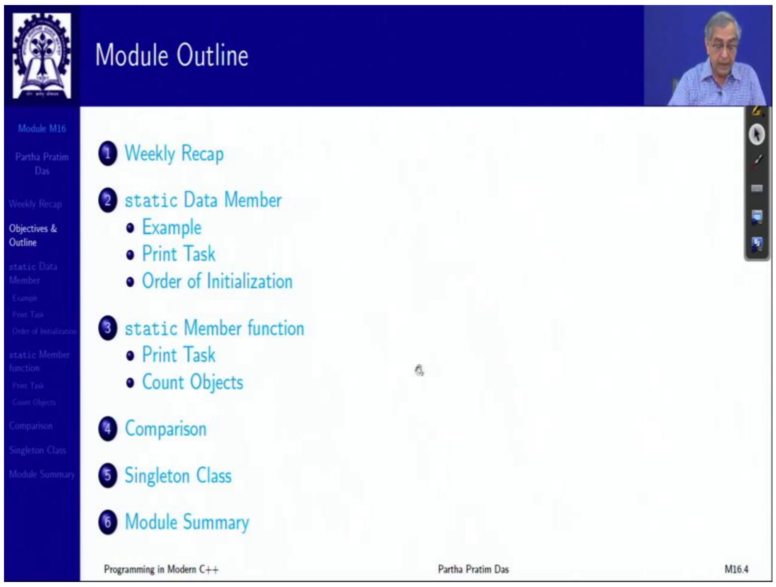
Singleton Class

Module Summary

Programming in Modern C++ Partha Pratim Das M16.3

What I am going to discuss in this module is, understanding of static data members and member functions.

(Refer slide time: 2:18)



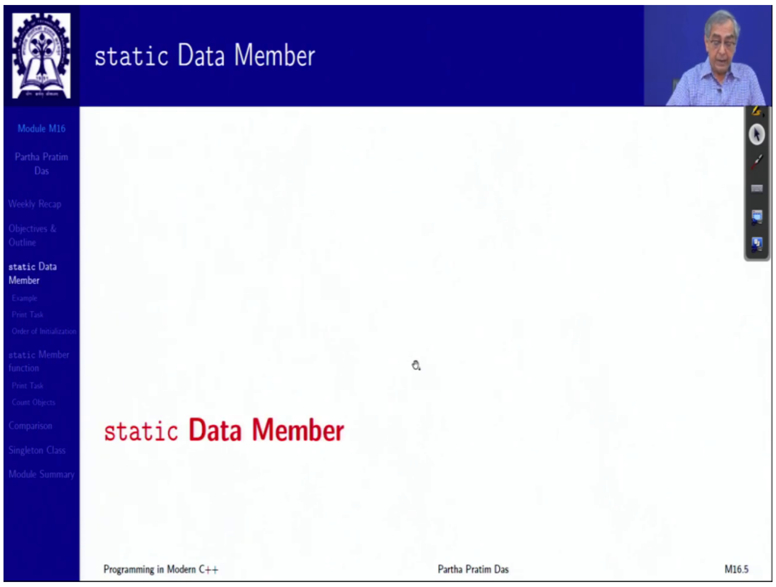
Module Outline

1. Weekly Recap
2. static Data Member
   - Example
   - Print Task
   - Order of Initialization
3. static Member function
   - Print Task
   - Count Objects
4. Comparison
5. Singleton Class
6. Module Summary

Programming in Modern C++ Partha Pratim Das M16.4

So, they will be extensions to the class design we have already talked off and they will cover some specific areas of problem solving. Here is outline as you can see.

(Refer slide time: 2:28)



static Data Member

Module M16

Partha Pratim Das

Weekly Recap

Objectives & Outline

static Data Member

Example

Print Task

Order of Initialization

static Member function

Print Task

Count Objects

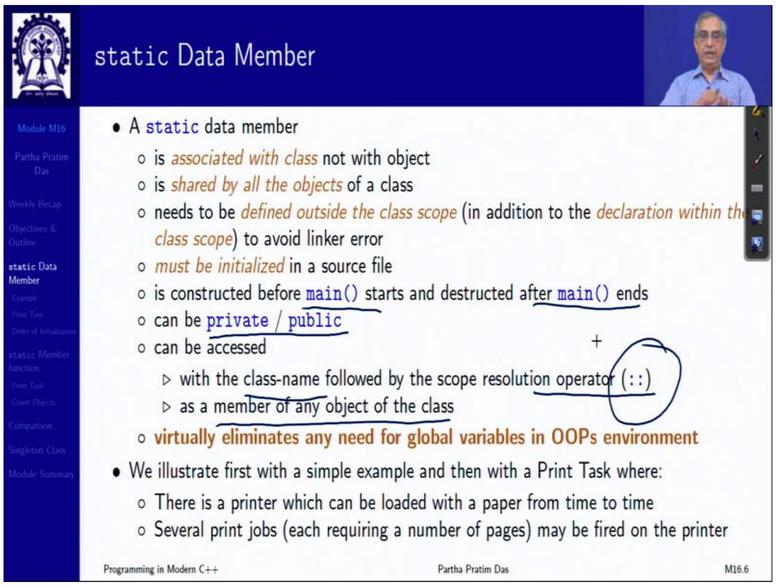
Comparison

Singleton Class

Module Summary

static Data Member

Programming in Modern C++ Partha Pratim Das M16.5



static Data Member

Module M16

Partha Pratim Das

Weekly Recap

Objectives & Outline

static Data Member

Example

Print Task

Order of Initialization

static Member function

Print Task

Count Objects

Comparison

Singleton Class

Module Summary

- A `static` data member
  - is *associated with class* not with object
  - is *shared by all the objects* of a class
  - needs to be *defined outside the class scope* (in addition to the *declaration within th class scope*) to avoid linker error
  - *must be initialized* in a source file
  - is constructed before `main()` starts and destructed after `main()` ends
  - can be `private` / `public`
  - can be accessed
    - ▷ with the class-name followed by the scope resolution operator (`::`)
    - ▷ as a member of any object of the class
  - **virtually eliminates any need for global variables in OOPs environment**
- We illustrate first with a simple example and then with a Print Task where:
  - There is a printer which can be loaded with a paper from time to time
  - Several print jobs (each requiring a number of pages) may be fired on the printer

Programming in Modern C++ Partha Pratim Das M16.6

And we will start with static data member. Static data member in contrast to the data members that we have seen, which henceforth, we will have to call them as non-static data members, a static data member is associated with a class not with an object that is that is the basic difference that is a class that I define can have multiple instances of objects created, all of them will share the same static data member, it is like a global value, but limited to that particular class only.

So, it will have to be it will be declared inside the class scope, but it will be defined with its type outside the class scope to avoid linked error. And it must be since it is declared defined outside the class scope, it will also have to be initialized in a source file. So, any static data member will be constructed before main starts and will be destructed after main ends. So, this is something which is which is very, very critical that it works, it encompasses the whole lifetime of main. So, all static data members of all, all classes will be initialized before main start.

So, now we can see that it is like the global variables which I said will be initialized before main and will be destructed after main. Similarly, the static data members will also be done in that way, which is a basic difference in the initialization of C++ compared to the initialization of C. A static data member like a non-static data member can be private or public.

Now, since it is a static data member, it is not dependent on the instance of the object. So, it is possible that I can access a static data member by just using the class name that is fully qualified class name class name::the data member name, the static data member name.

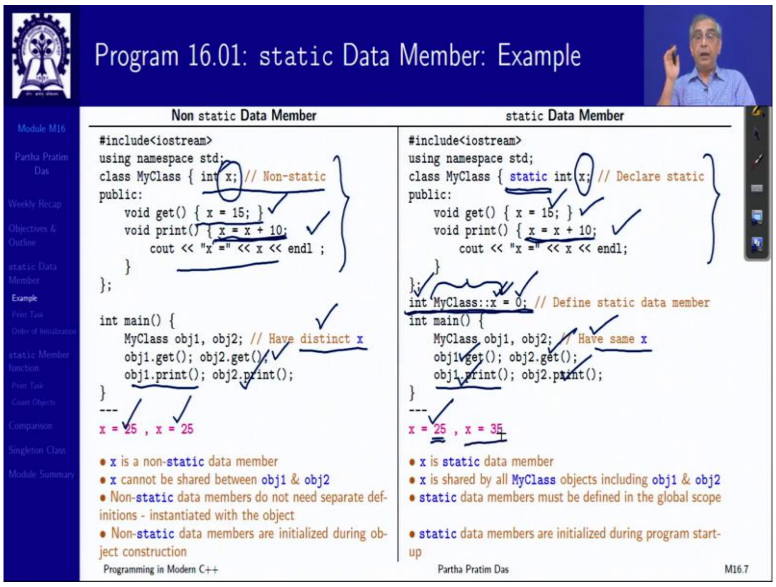
Alternatively I can also access a static data member using an object but just using object . this member, but what compiler actually does, it does not, since it does not relate to the specific instance from the object it will know its class and find out, internally it will find out that what is the class scope where the static data member belongs and therefore, it will be access through that manner.

So, virtually it eliminates the static data members eliminates any need of global variables in an object oriented environment after this, we will not need any other global variable which is defined outside of any function scope or any class scope. So, let us first go ahead and show you an example, where there is a printed task where there is a printer which can be loaded with paper from time to time.

So, there is a tray as if you that you are modeling logically by a variable which says that how much paper is there, and you can load paper into that, and there are several print jobs that are fired on this printer one after the other, and every print job has a number of specified pages, the number of pages that it has to.

So, after that job is done, the number of available pages on the tray will get reduced by the number of pages required for that job. So, this is the basis of this requirement.

(Refer slide time: 6:18)



Program 16.01: static Data Member: Example

Non static Data Member

```
#include<iostream>
using namespace std;
class MyClass { int x; // Non-static
public:
    void get() { x = 15; }
    void print() { x = x + 10;
        cout << "x =" << x << endl ;
    }
};

int main() {
    MyClass obj1, obj2; // Have distinct x
    obj1.get(); obj2.get();
    obj1.print(); obj2.print();
}
---
x = 25 , x = 25
```

- x is a non-static data member
- x cannot be shared between obj1 & obj2
- Non-static data members do not need separate definitions - instantiated with the object
- Non-static data members are initialized during object construction

static Data Member

```
#include<iostream>
using namespace std;
class MyClass { static int x; // Declare static
public:
    void get() { x = 15; }
    void print() { x = x + 10;
        cout << "x =" << x << endl;
    }
};
int MyClass::x = 0; // Define static data member
int main() {
    MyClass obj1, obj2; // Have same x
    obj1.get(); obj2.get();
    obj1.print(); obj2.print();
}
---
x = 25 , x = 35
```

- x is static data member
- x is shared by all MyClass objects including obj1 & obj2
- static data members must be defined in the global scope
- static data members are initialized during program start-up

Programming in Modern C++ Partha Pratim Das M16.7

So, we do the design like this, we have a first we show you example of a simple example and then we go to the print job. So, let us say I have a this is a simple example. So, I have a class MyClass which has a non-static member x, so, I can do a get I can do a print. So, if I instantiate two objects then each object has a separate data member x in it, they are distinct.

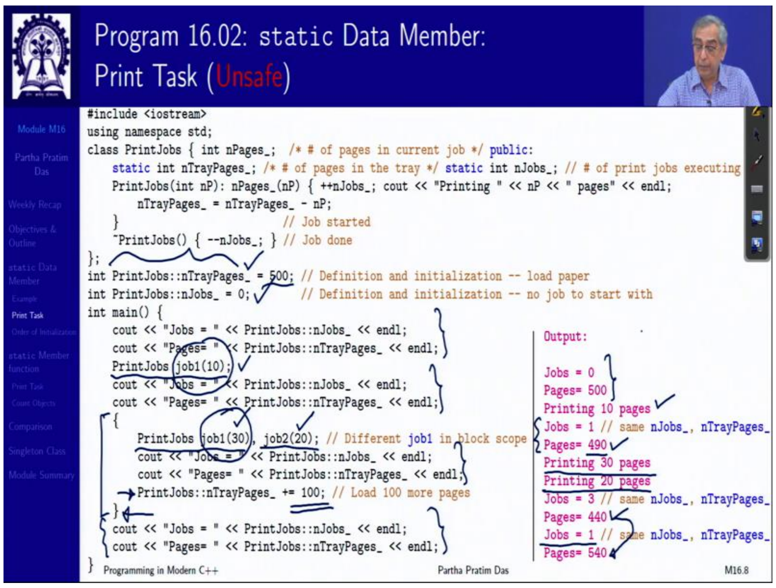
So, when I do get on this, basically that data member is getting set to 15. So, when I do print on that, for object one, this was set to 15. Now, in print this has been incremented, increased by 10. So, it will print 25. For object two, also it will print 25. That is that simple. Now, here, on the right hand side, I have made this a static data member, the difference that I have made is I have put a keyword static before the declaration of the data member which says the static data member.

Which means that now I have only one variable for the class, not object-wise instances, I have the same get set functions. Since I have a static data member, I have to provide a definition for that. So, I provide that definition in the source files outside the scope of the class. This is my fully qualified name of the member variable MyClass::x. I put its type which is int, the initialization value which is zero.

Now, I again instance, I create two objects. Now, in contrast to the previous case, where x was a part of the object, so I had two different x's, here, I have x, which is the same object, which is not a part of any of these objects, it is outside of that it is just belonging to the class. So now if I do get x, this x is set to 15. I do get on the second object again, this is reassigned again 15. So its value is 15. Now, when I do print, the x gets incremented by 10. So it prints 25.

Then when I do print for the next time, x is 25. Because it is the same x here I had a different x, so 15 incremented to 25, 15 incremented to 25 but here it is the same is the same x so it is already incremented to 25. So, now what will happen if I add 10 to it, it will become 35, the second time it prints 35. So, this is this clearly tells you what is the difference between a non-static data member and a static data member. So the explanation is written below for your reference.

(Refer slide time: 9:40)



Program 16.02: static Data Member: Print Task (Unsafe)

```
#include <iostream>
using namespace std;
class PrintJobs { int nPages_;  /* # of pages in current job */ public:
    static int nTrayPages_; /* # of pages in the tray */ static int nJobs_; // # of print jobs executing
    PrintJobs(int nP): nPages_(nP) { ++nJobs_; cout << "Printing " << nP << " pages" << endl;
        nTrayPages_ = nTrayPages_ - nP;
    }                                  // Job started
    ~PrintJobs() { --nJobs_; } // Job done
};
int PrintJobs::nTrayPages_ = 500; // Definition and initialization -- load paper
int PrintJobs::nJobs_ = 0;         // Definition and initialization -- no job to start with
int main() {
    cout << "Jobs = " << PrintJobs::nJobs_ << endl;
    cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
    PrintJobs job1(10);
    cout << "Jobs = " << PrintJobs::nJobs_ << endl;
    cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
    {
        PrintJobs job1(30), job2(20); // Different job1 in block scope
        cout << "Jobs = " << PrintJobs::nJobs_ << endl;
        cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
        PrintJobs::nTrayPages_ += 100; // Load 100 more pages
    }
    cout << "Jobs = " << PrintJobs::nJobs_ << endl;
    cout << "Pages= " << PrintJobs::nTrayPages_ << endl;
}
```

Output:

```
Jobs = 0
Pages= 500
Printing 10 pages
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 490
Printing 30 pages
Printing 20 pages
Jobs = 3 // same nJobs_, nTrayPages_
Pages= 440
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 540
```

So with this, let us go on to the print task that I just described. So I model that there is a print job. I am creating a print job. What is a print job? Print job is print will print something so it has a number of pages which is a non-static member. Then in public, I have two static data members, which is number of nTrayPages, that is how many pages are there in the tray. And another is nJobs, which says how many jobs are there in the printer, has been put to the printer.

So, with that, when I do this print jobs constructor, I set nPages, that is how many pages the job has with the parameter, I increment the static member, because one more job is getting added to it. And I decrement the nTrayPages by nP because this job will print nP pages. So, the tray will be left with nTrayPages minus nP number of pages. So, this is every time a job is placed a constructor is construction is that this will happen and what will happen in the destruction naturally when they destroyed it means that this job is over.

So, now, there will be one less number of job to do right. So, it will become nJobs minus one. So, with that, let us look at some of the initializations so this is a static initialization. So, this is the definition of nTrayPages in the global qualified name as I said, and we have initialized to 500. Assuming that initially 500 pages have been loaded nJobs similarly is initialized to two because there is no job to start with.

So, then I initially print what is the job number of jobs and what are the number of pages zero and 500 is what I get then I create a print jobs with 10 that is 10 pages to be printed, I check these values printing 10 jobs, it says and it is now has jobs one because one job has been submitted and the pages becomes 490 because that job submitted has 10 pages 500 minus 10, 490 is what I get. Now in a scope, I create two more jobs.

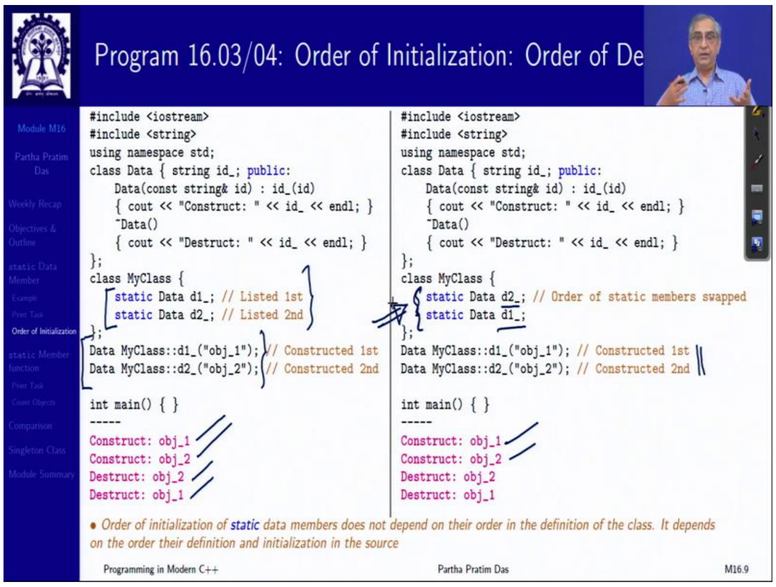
Mind you I have created a job one here, I have created a job one here and the rule of scoping holds. So, the job one created inside the block scope is different from the job one created the outside the block scope. So, the job one prints the printing 30 pages, because it is taken as 30, job two prints printing 20 pages, then I print what is my status of the printer.

I get there are three jobs print job one outside the block scope job one inside the block scope, job two in the block scope, three jobs and the pages remaining are 440, 490 I had minus 30, 460 minus 20, 440 pages remaining. Now, when I before coming out I load in other 100 pages. So, after loading, when I come out of the block, what will happen at this point, you know you have learned that lifetime.

So print jobs, job one and job two inside the block will get destructed at this point as they get destructed number of jobs will decrease for each by one. So, my jobs after this becomes one and the print jobs has now changed increased with the value of 100 so from 440 becomes 540. So, that is how the logic goes. This is I wrote on top that this is not a very safe way of doing it, because these are these static data members are kept in public.

So besides being static, they are also exposed. So the encapsulation the information hiding part is not there. So anybody can mistakenly change that and we will have difficulties in terms of, so we will talk about how to handle this.

(Refer slide time: 14:39)



Program 16.03/04: Order of Initialization: Order of De

```
#include <iostream>
#include <string>
using namespace std;
class Data { string id_; public:
    Data(const string& id) : id_(id)
    { cout << "Construct: " << id_ << endl; }
    ~Data()
    { cout << "Destruct: " << id_ << endl; }
};
class MyClass {
    static Data d1_; // Listed 1st
    static Data d2_; // Listed 2nd
};
Data MyClass::d1_("obj_1"); // Constructed 1st
Data MyClass::d2_("obj_2"); // Constructed 2nd

int main() { }
-----
Construct: obj_1
Construct: obj_2
Destruct: obj_2
Destruct: obj_1
```

```
#include <iostream>
#include <string>
using namespace std;
class Data { string id_; public:
    Data(const string& id) : id_(id)
    { cout << "Construct: " << id_ << endl; }
    ~Data()
    { cout << "Destruct: " << id_ << endl; }
};
class MyClass {
    static Data d2_; // Order of static members swapped
    static Data d1_;
};
Data MyClass::d1_("obj_1"); // Constructed 1st
Data MyClass::d2_("obj_2"); // Constructed 2nd

int main() { }
-----
Construct: obj_1
Construct: obj_2
Destruct: obj_2
Destruct: obj_1
```

- *Order of initialization of static data members does not depend on their order in the definition of the class. It depends on the order their definition and initialization in the source*

Programming in Modern C++ Partha Pratim Das M16.9

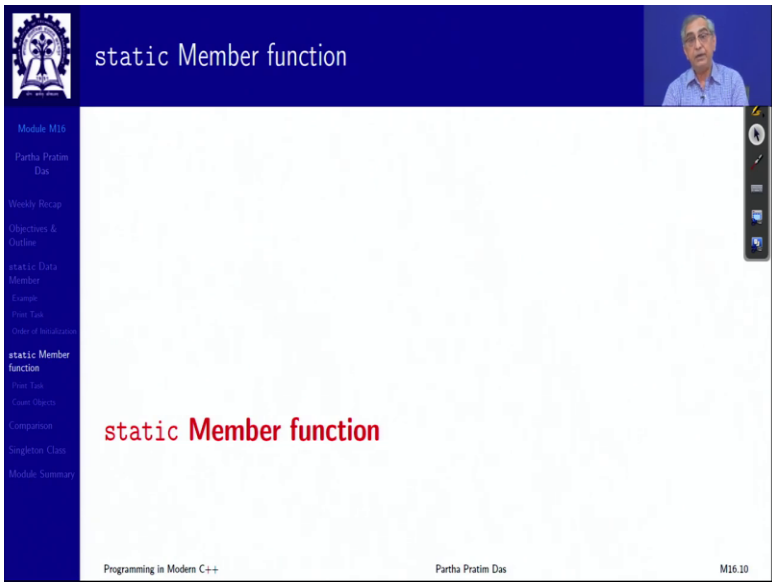
Now comes a question of what is the order of initialization we talked about this for non-static data members and we learned that it does not matter in which order I write the non-static data members in the initializer list. The ordering which I list them in the class is the order in which they are initialized. You will be surprised to see that for static data members the rule gets different. So here I have a class which has two data members, which are static, static data one static data d1, static data d2.

And in the source file, I am initializing constructing them, first d1 with the obj_1, then d2 with obj_2, and I can see that this is the order in which the construction happens, the structure obviously in the reverse order. So now the question is, the order that I am seeing is it because of the order in which I have initialized or the order in which I have listed them in the class.

To check that I just swap, I just swap these two and do not change this as I swap, I find that the order of construction has not changed and so, the order of destruction, but have already swapped this. So, I conclude that it does not matter in which order I list them within the class, but it is the order in which I initialize them in the static area, in the source file is what decides the order in which they be initialized.

Which is different from what we saw for the non-static data member case where the order in the initializer list does not matter, but the order in the listing of the non-static data members in the class actually dictates the order of their initialization right. So, here this rule is different, you will slowly realize why it had to be different and what are the advantages of that, but for now, just know that when you are dealing with static data members, it is the order in which you write the initialization is the order in which they will actually be initialized and constructed.

(Refer slide time: 17:04)



static Member function

Module M16

Partha Pratim Das

Weekly Recap

Objectives & Outline

static Data Member

Example

Print Task

Order of Initialization

static Member function

Print Task

Count Objects

Comparison

Singleton Class

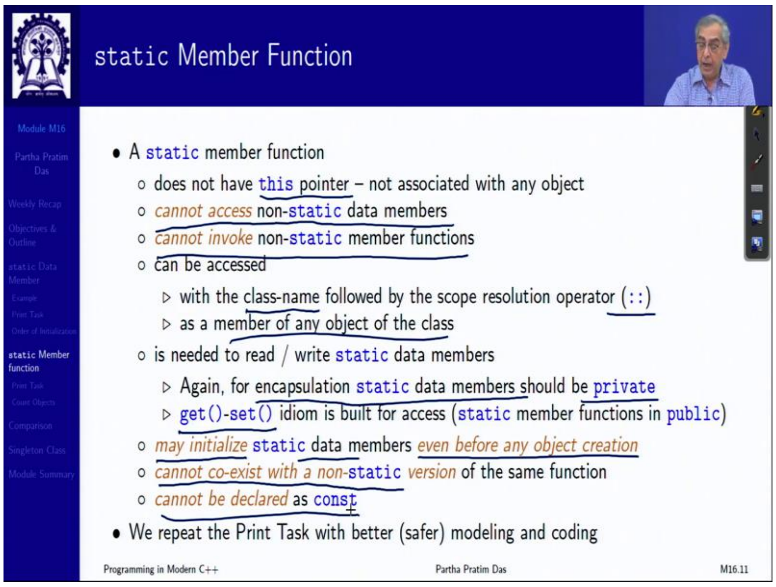
Module Summary

static Member function

Programming in Modern C++ Partha Pratim Das M16.10

Now, let us we have already seen that the encapsulation is broken, there is no information hiding, because if I put the static data members in public, it can be changed without the notion of the class at any point of time. If I put them in private, then I cannot change them. So, along with static data members, I also need static member functions.

(Refer slide time: 17:34)



static Member Function

- A static member function
  - does not have this pointer – not associated with any object
  - cannot access non-static data members
  - cannot invoke non-static member functions
  - can be accessed
    - with the class-name followed by the scope resolution operator (::)
    - as a member of any object of the class
  - is needed to read / write static data members
    - Again, for encapsulation static data members should be private
    - get()-set() idiom is built for access (static member functions in public)
  - may initialize static data members even before any object creation
  - cannot co-exist with a non-static version of the same function
  - cannot be declared as const
- We repeat the Print Task with better (safer) modeling and coding

Programming in Modern C++ Partha Pratim Das M16.11

What is static member functions? Static member functions, the first difference is they do not have a this pointer, unlike the non-static data members, because they are not associated with any object, they are just associated with the class. So, there is no object instance whose this pointer can be or need to be passed here, they cannot be accessed, they cannot access any non-static data member or any non-static member functions.

Why they cannot access because they do not have this pointer. The non-static data member or the non-static member function is specific to an object instance but a static member function has no this pointer. So, it cannot face just like a global function only difference being that it is in the class scope it is qualified by the class name.

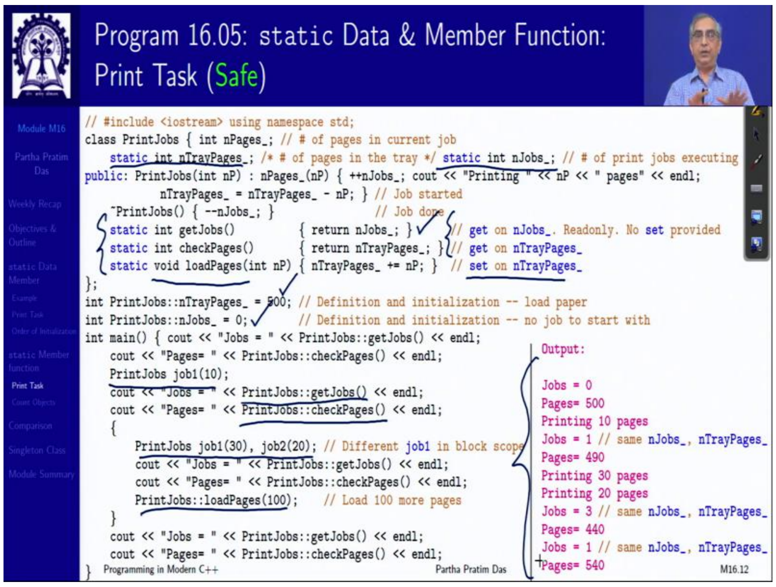
So, it can be accessed with the class name with the :: operator or with any member of the class exactly the same way you could refer to a static data member exactly in the same way you could also invoke you can also invoke a static member function.

So, using this once you have this, you can now again create the same information hiding encapsulation paradigm as we did for the non-static data members, all that you do is encapsulate static data members as private and provide the required get-set idiom in the in terms of static member functions in public. And you can create that same model except that now, this particular encapsulation model is not specific to an object it is specific to the class or the entire ensemble of objects that can be instantiated from a particular class.

So, you may initialize static data members, even before any object creation, because I told you it can potentially it will get initialized before main starts. It is possible that you instantiate some object also before main starts, but it may it is not necessary. Also remember that if I have a static member function, then I cannot have another non-static member function by the same name. That is overloading a member function between static and non-static is not allowed.

Similarly, static member functions cannot be declared as const which you will understand by now, because the consequence of declaring a member function const is to actually change the type of the this pointer, which becomes now a pointer to a constant object, but since static member functions have no this pointer, there is no semantic sense of saying that a static member function is const it cannot be done.

(Refer slide time: 20:36)



Program 16.05: static Data & Member Function: Print Task (Safe)

```
// #include <iostream> using namespace std;
class PrintJobs { int nPages_; // # of pages in current job
    static int nTrayPages_; /* # of pages in the tray */ static int nJobs_; // # of print jobs executing
public: PrintJobs(int nP) : nPages_(nP) { ++nJobs_; cout << "Printing " << nP << " pages" << endl;
            nTrayPages_ = nTrayPages_ - nP; } // Job started
    ~PrintJobs() { --nJobs_; }                     // Job done
    static int getJobs()          { return nJobs_; }       // get on nJobs_. Readonly. No set provided
    static int checkPages()       { return nTrayPages_; }  // get on nTrayPages_
    static void loadPages(int nP) { nTrayPages_ += nP; }   // set on nTrayPages_
};
int PrintJobs::nTrayPages_ = 500; // Definition and initialization -- load paper
int PrintJobs::nJobs_ = 0;        // Definition and initialization -- no job to start with
int main() { cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    PrintJobs job1(10);
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
    {
        PrintJobs job1(30), job2(20); // Different job1 in block scope
        cout << "Jobs = " << PrintJobs::getJobs() << endl;
        cout << "Pages= " << PrintJobs::checkPages() << endl;
        PrintJobs::loadPages(100);    // Load 100 more pages
    }
    cout << "Jobs = " << PrintJobs::getJobs() << endl;
    cout << "Pages= " << PrintJobs::checkPages() << endl;
}
```

```
Output:

Jobs = 0
Pages= 500
Printing 10 pages
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 490
Printing 30 pages
Printing 20 pages
Jobs = 3 // same nJobs_, nTrayPages_
Pages= 440
Jobs = 1 // same nJobs_, nTrayPages_
Pages= 540
```

Programming in Modern C++ Partha Pratim Das M16.12

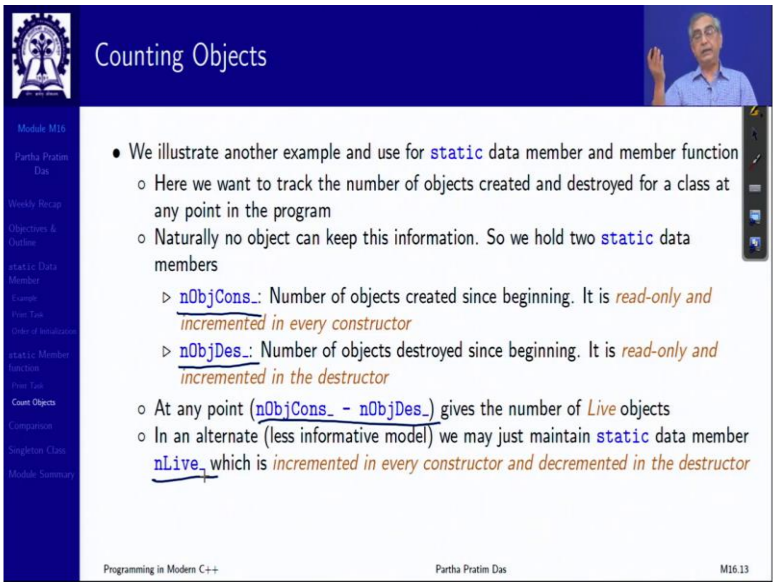
Now, with this with this improvement in the encapsulation and information hiding paradigm, let us take a relook at the print task which is which remains the same except that now, I have made this in the private put them before public. So, these are all private now, you cannot directly access them, and we also provide three static member functions all that you need to do you write the function as you had been doing earlier, except you put the keyword static in the beginning which ensures that these are static functions.

And therefore, they will not have any this pointer. So, this tells you this is a I mean get set kind of So, here what you say that, you are not allowing the set on any of this all that you are doing is you are giving get on the number of jobs and the number of trays number of pages, if you want to set that you have given separate load page function right.

So, it is it is perfectly conforming to the get- set edm based information hiding model that we had done for the non-static data members and non-static member functions. This is same the initialization you create the print jobs in the same way you do everything in the same way. Now, when you are actually trying to find out how many jobs are there and how many pages are there, instead of directly accessing the public static data member which you could now you invoke the public static member function.

Similarly, for loading this, this behavior will remain the same, but now, you have the safety of being encapsulated in terms of your actual static data member values.

(Refer slide time: 22:38)



Counting Objects

- We illustrate another example and use for `static` data member and member function
  - Here we want to track the number of objects created and destroyed for a class at any point in the program
  - Naturally no object can keep this information. So we hold two `static` data members
    - `nObjCons_`: Number of objects created since beginning. It is *read-only and incremented in every constructor*
    - `nObjDes_`: Number of objects destroyed since beginning. It is *read-only and incremented in the destructor*
  - At any point (`nObjCons_ - nObjDes_`) gives the number of *Live* objects
  - In an alternate (less informative model) we may just maintain `static` data member `nLive_` which is *incremented in every constructor and decremented in the destructor*

Module M16
Partha Pratim Das
Weekly Recap
Objectives & Outline
static Data Member
Example
Print Task
Order of Initialization
static Member function
Print Task
Count Objects
Comparison
Singleton Class
Module Summary

Programming in Modern C++ Partha Pratim Das M16.13

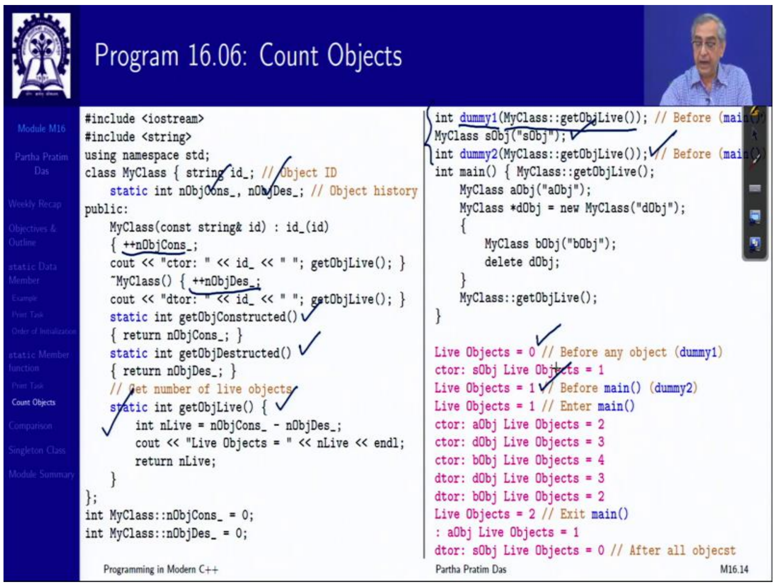
So, this gives you a very nice model to work with where any object in specific information can be kept at this level. So, we will just take a quick look at what could be potential use of that, for example, here and here what we showed in terms of the printer task. If we had to do this without the static data members and static member functions, we would have required to use global variables because a printer is one and does jobs are many they are not specific to that.

So, by it is it is clear as to how by using the static data member and static member functions, how we have nicely encapsulated everything within the class. I will talk about a different example. For example, say if you want to track the number of objects that you have created. Now, the question is obviously, why do you need to do that there may be several reasons for that, for example, your every object could be a lock on a port memory port, you have a memory port object, now, you have three memory ports.

Now, naturally, you cannot create four memory port objects, because the fourth one will not have anything to lock right. So, lot of physical and logical situations will need that you can count the number of objects that you have created. So, how can you implement that could be very simple. So, I assume that there is an nObjCons which is which is incremented every time an object is constructed, and there is a desk which is decremented every time his object is destructed.

At any point of time if you take their difference, you will get the number of objects that are live in a simpler model, you can just make a one single static data member in life incremented in constructor decremented in destructor, but by doing this you will have a more informative model here.

(Refer slide time: 24:40)



Program 16.06: Count Objects

```
#include <iostream>
#include <string>
using namespace std;
class MyClass { string id_; // Object ID
    static int nObjCons_, nObjDes_; // Object history
public:
    MyClass(const string& id) : id_(id)
    { ++nObjCons_;
    cout << "ctor: " << id_ << " "; getObjLive(); }
    ~MyClass() { ++nObjDes_;
    cout << "dtor: " << id_ << " "; getObjLive(); }
    static int getObjConstructed()
    { return nObjCons_; }
    static int getObjDestructed()
    { return nObjDes_; }
    // Get number of live objects
    static int getObjLive() {
        int nLive = nObjCons_ - nObjDes_;
        cout << "Live Objects = " << nLive << endl;
        return nLive;
    }
};
int MyClass::nObjCons_ = 0;
int MyClass::nObjDes_ = 0;
```

```
int dummy1(MyClass::getObjLive()); // Before (mai
MyClass sObj("sObj");
int dummy2(MyClass::getObjLive()); // Before (mai
int main() { MyClass::getObjLive();
    MyClass aObj("aObj");
    MyClass *dObj = new MyClass("dObj");
    {
        MyClass bObj("bObj");
        delete dObj;
    }
    MyClass::getObjLive();
}
```

```
Live Objects = 0 // Before any object (dummy1)
ctor: sObj Live Objects = 1
Live Objects = 1 // Before main() (dummy2)
Live Objects = 1 // Enter main()
ctor: aObj Live Objects = 2
ctor: dObj Live Objects = 3
ctor: bObj Live Objects = 4
dtor: dObj Live Objects = 3
dtor: bObj Live Objects = 2
Live Objects = 2 // Exit main()
: aObj Live Objects = 1
dtor: sObj Live Objects = 0 // After all objecst
```

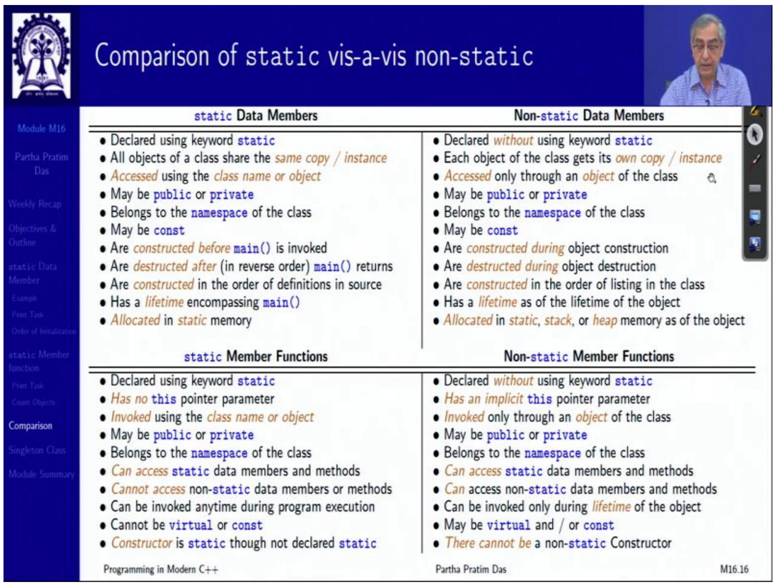
Programming in Modern C++ Partha Pratim Das M16.14

So it is example is very easy to see I have these two in their private I have the way to get obviously I will not have a way to set because I cannot allow the programmer to change either the number of objects constructed or the number of objects destructed that will have to happen either in the constructor or in the destructor right. I can only read them take their difference say how many objects are there.

So with that, this example shows some objects which are created before main. And I had shown you this trick earlier also. So I am I am just as if initializing a dummy integer, I am not interested in that integer, all that I am interested in is to pass it a value, which initially would be zero. But actually, I am not interested in that even all that I am interested in is to invoke the static member functions even before main has started.

So, I get to see how many objects are there then one global object gets created, then I again do that and I get to see how many objects are there then I get into the main see different objects created destroyed, entering the scope leaving the scope actually, you can, you basically get a trace of the object lifetime and the count of live objects at any point of time. So, static could be of great help in terms of that.

(Refer slide time: 26:18)



Comparison of static vis-a-vis non-static

static Data Members	Non-static Data Members
Declared using keyword static	Declared *without* using keyword static
All objects of a class share the *same copy / instance*	Each object of the class gets its *own copy / instance*
Accessed using the *class name or object*	*Accessed* only through an *object* of the class
May be public or private	May be public or private
Belongs to the namespace of the class	Belongs to the namespace of the class
May be const	May be const
Are *constructed before* main() is invoked	Are *constructed during* object construction
Are *destructed after* (in reverse order) main() returns	Are *destructed during* object destruction
Are *constructed* in the order of definitions in source	Are *constructed* in the order of listing in the class
Has a *lifetime* encompassing main()	Has a *lifetime* as of the lifetime of the object
Allocated in *static* memory	*Allocated* in *static*, *stack*, or *heap* memory as of the object

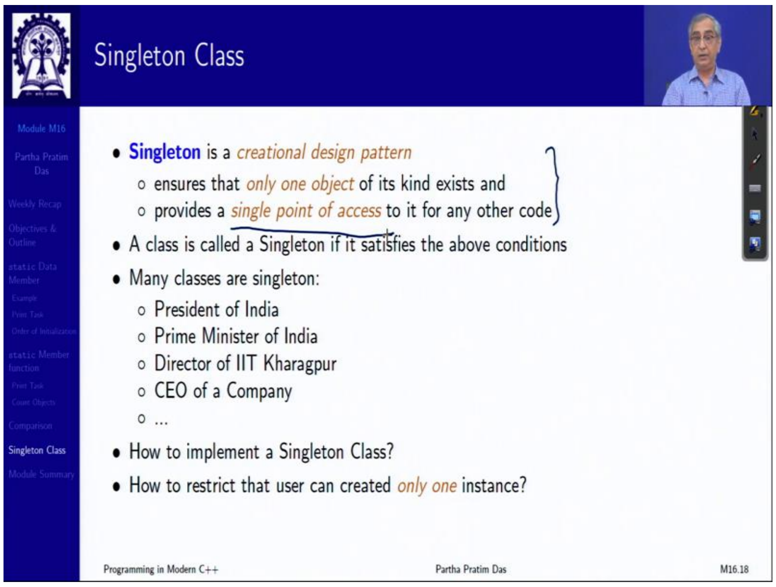
static Member Functions	Non-static Member Functions
Declared using keyword static	Declared *without* using keyword static
Has no this pointer parameter	*Has an implicit* this pointer parameter
Invoked using the *class name or object*	*Invoked* only through an *object* of the class
May be public or private	May be public or private
Belongs to the namespace of the class	Belongs to the namespace of the class
Can access static data members and methods	*Can access* static data members and methods
Cannot access non-static data members or methods	*Can* access non-static data members and methods
Can be invoked anytime during program execution	Can be invoked only during *lifetime* of the object
Cannot be virtual or const	May be virtual and / or const
Constructor is static though not declared static	*There cannot be* a non-static Constructor

Programming in Modern C++ Partha Pratim Das M16.16

To compare static versus non-static, I have given you a complete comparative information all of these I have already discussed in terms of what we have been learning, but this is a compile chart which if you want to refer to if you have a doubt as to well whether Can I can I access non-static data member from a static member function?

You can look up here, no you cannot access, why you cannot access there is no this pointer, but if I have a non-static data member functions, can I access static members? Yes, you can, because static members are always available by the class name right. So, all these differences are listed here, you can you can make references to that study to understand it, better.

(Refer slide time: 27:05)



Singleton Class

- **Singleton** is a *creational design pattern*
  - ensures that *only one object* of its kind exists and
  - provides a *single point of access* to it for any other code
- A class is called a Singleton if it satisfies the above conditions
- Many classes are singleton:
  - President of India
  - Prime Minister of India
  - Director of IIT Kharagpur
  - CEO of a Company
  - ...
- How to implement a Singleton Class?
- How to restrict that user can created *only one* instance?

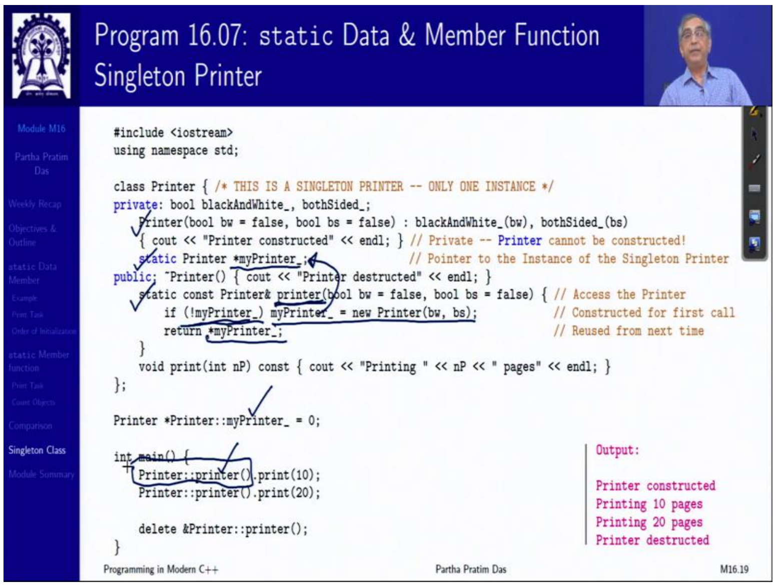
Programming in Modern C++ Partha Pratim Das M16.18

Before I conclude, I will tell you a very simple you know, design, which is which we will talk about separately in tutorial also, it is called a design pattern like this is something which repeatedly occurs in different design scenarios. And therefore, you can have a solution for this pre designed so, that whenever that occurs, you can just use that solution.

So, it is called a singleton pattern, that is one only pattern which says that it is a class which ensures that only one objects kind can exist only one object is funny, we have a class instance in any number of objects, we do not want that you want only one object and then that object there must be a single point of access for that singular object, right.

If these two conditions are satisfied, we say that we have a singleton. Now, conceptually what is a singleton something which is in the real world, which is unique for example, President of India, Prime Minister of India, Director of IIT Kharagpur, CEO of a company and so on, so forth. The question is how do you implement this kind of singleton design in C++.

(Refer slide time: 28:17)



Program 16.07: static Data & Member Function Singleton Printer

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

class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
private: bool blackAndWhite_, bothSided_;
    Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
    { cout << "Printer constructed" << endl; } // Private -- Printer cannot be constructed!
    static Printer *myPrinter_;                 // Pointer to the Instance of the Singleton Printer
public: ~Printer() { cout << "Printer destructed" << endl; }
    static const Printer& printer(bool bw = false, bool bs = false) { // Access the Printer
        if (!myPrinter_) myPrinter_ = new Printer(bw, bs);          // Constructed for first call
        return *myPrinter_;                                          // Reused from next time
    }
    void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};

Printer *Printer::myPrinter_ = 0;

int main() {
    Printer::printer().print(10);
    Printer::printer().print(20);

    delete &Printer::printer();
}
```

```
Output:

Printer constructed
Printing 10 pages
Printing 20 pages
Printer destructed
```

Programming in Modern C++ Partha Pratim Das M16.19

So, two things one is you have to make sure that you can create only one instance now, so, you have to make sure that you cannot let anyone arbitrarily create the object. So, how can you stop creating an object simple ways put the constructor in private nobody can access it other than members in the class.

Now, this might become this becomes a chicken and egg problem if I cannot construct I cannot have an object if I cannot have an object I cannot call a member function. So, how do I go about that is where the static member function come in, because static member functions do not depend on object being constructed.

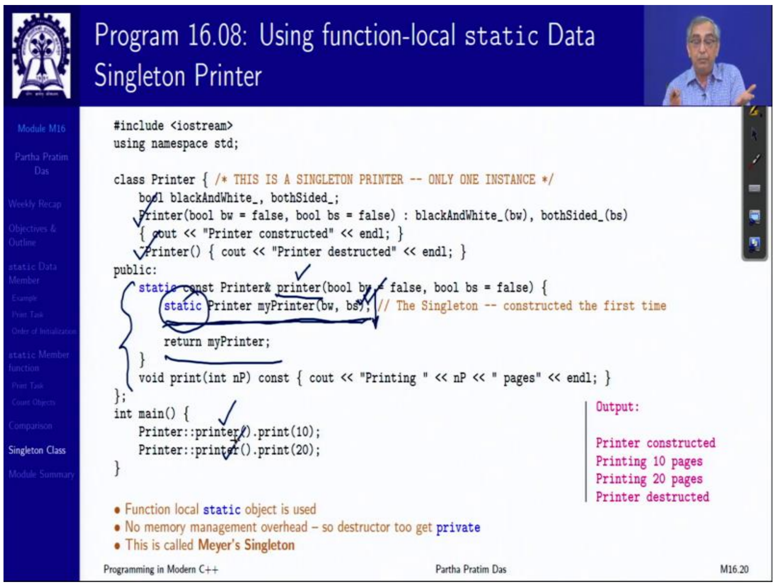
So, I provide a static member function say called printer different from the class name has to be because it is not a constructor which simply will find out that do I have an instance so, how do I know if I have an instance. So, for that I provide in the private a static pointer and initialize that pointer to null. So, what will happen when, when the system starts this pointer is null which tells me that there is no printer.

Now, we ever once calls this printer static member function calls printer, as it calls printer. I go and check is this pointer null. If it is null there is no object constructed so far if it is not null, then object has been constructed. So if it is null I construct an object. I do a new construct the object and put that pointer to this and return that object. Now if it has been constructed that is if the pointer is not null then I do not construct it again, I will just return the same object.

So I get both the points that were asked for in a singleton that is, this will ensure that I can create exactly one object not more than that because the my constructor is private. And there is a single point of access that point of access is this static member function. Every time I need that printer I call that member function, it gives me the same printer. First time it reconstructs and gives, second time onwards it just keep on giving the earlier.

So this is the basic idea of a singleton and this is by using the static data member and by this design paradigm that is you put the constructor in private, you have a marker to remember that object that is in private and you have a single point static member to decide whether you need to construct and object or not and return that will give you a singleton module which here we are showing in terms of printer, it can be done similarly, for any other object type.

(Refer slide time: 31:34)



Program 16.08: Using function-local static Data Singleton Printer

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

class Printer { /* THIS IS A SINGLETON PRINTER -- ONLY ONE INSTANCE */
    bool blackAndWhite_, bothSided_;
    Printer(bool bw = false, bool bs = false) : blackAndWhite_(bw), bothSided_(bs)
    { cout << "Printer constructed" << endl; }
    ~Printer() { cout << "Printer destructed" << endl; }
public:
    static const Printer& printer(bool bw = false, bool bs = false) {
        static Printer myPrinter(bw, bs); // The Singleton -- constructed the first time

        return myPrinter;
    }
    void print(int nP) const { cout << "Printing " << nP << " pages" << endl; }
};
int main() {
    Printer::printer().print(10);
    Printer::printer().print(20);
}
```

```
Output:

Printer constructed
Printing 10 pages
Printing 20 pages
Printer destructed
```

- Function local static object is used
- No memory management overhead – so destructor too get private
- This is called Meyer's Singleton

Programming in Modern C++ Partha Pratim Das M16.20

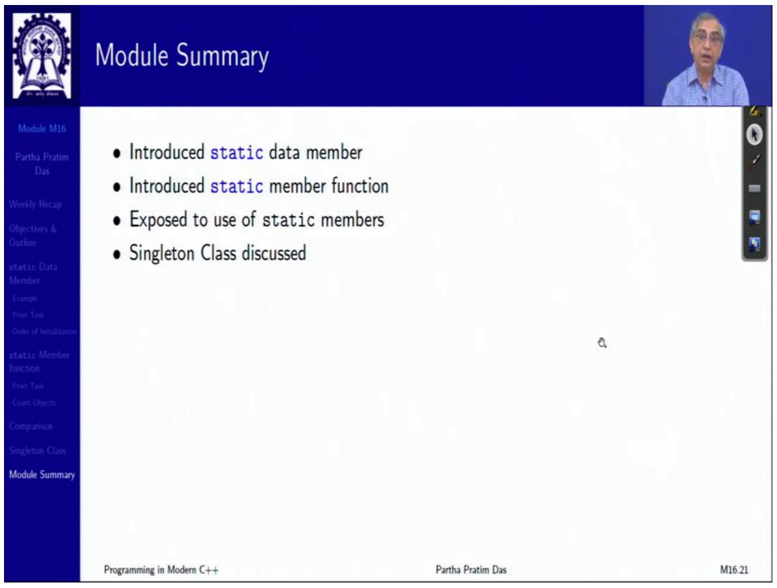
Interestingly there is another alternate design to this where again all that I do is in this access function for the printer. I have defined a function local printer. I am not maintaining, my constructor is private, my destructor is private. Because I am not going to allow any other construction but I am not maintaining a separate pointer as a private member, rather I have a local static variable in the access function which is of the printer type.

So what happens, what is the property of local static variables? Is that, they get constructed the first time the control passes through this point and then they are remembered because they are static, they are remembered across the calls to this function. So the compiler does your job what you have been doing by putting a pointer, checking that whether it is null, setting it to be not null and returning it everytime, you do not have to do any of that.

The compiler will take care of all of that because that is the behaviour of a static local variable of any function. So in the access function I simply make it static local and then I simply return. So what will happen first time I call this here, it has not been constructed, it will get constructed and remembered, second time I call it, it is it will not constructed again because it has been and the same object will be returned.

So it is even a simpler solution this was originally proposed by Scott Meyers. So often this is referred to a Meyer's singleton. You can use this design which is very simple, I mean you do not have to remember anything, all you need to do is put the constructor and destructor in private and then have the access function.

(Refer slide time: 33:47)



Module Summary

- Introduced static data member
- Introduced static member function
- Exposed to use of static members
- Singleton Class discussed

So this brings us to the conclusion of module 16, where we have introduced static data members and static member functions and discuss the use of static member in the C++ design and particularly discuss the singleton class design pattern. Thank you very much and we will meet in the next module.