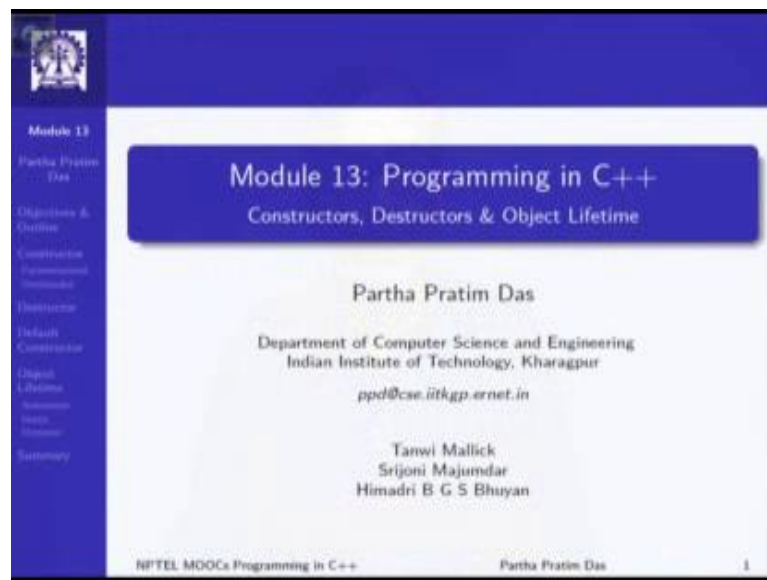


Programming in C++
Prof. Partha Prathim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 25
Constructors, Destructors and Object Lifetime (Contd.)

Welcome to part 3 of module 13.

(Refer Slide Time: 00:24)



Module 13
Partha Prathim Das

Objectives & Outline
Constructors
Destructors
Default Constructors
Object Lifetime
Operator Overloading
Summary

Module 13: Programming in C++
Constructors, Destructors & Object Lifetime

Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

NPTEL MOOCs Programming in C++ Partha Prathim Das 1

In this module, we have already discussed about object construction and destruction through variety of constructors and the destructor. We have talked about the free constructor and destructor that the compiler may provide. Equipped with all these, now we will take a deeper look into what is known as object lifetime. Where, we will answer the basic question of when is an object ready and how long can it be used.

(Refer Slide Time: 00:58)

**Object Lifetime: When is an Object ready?
How long can it be used?**

Application

```
void MyFunc() // E1: Allocation of c on Stack  
{  
    ...  
    Complex c; // E2: Ctor called  
    ...  
    c.norm(); // E5: Use  
    ...  
    return; // E7: Dtor called  
} // E8: De-allocation of c from Stack
```

Class Code

```
Complex::Complex(double re = 0.0, // Ctor  
                 double im = 0.0)  
{ // E4: Object Lifetime STARTS  
    cout << "Ctor:" << endl;  
}  
  
double Complex::norm() // E6  
{ return sqrt(re*re + im*im); }  
  
Complex::~Complex() // Dtor  
{  
    cout << "Dtor:" << endl;  
} // E8: Object Lifetime ENDS
```

Event Sequence and Object Lifetime

Event	Description
E1	MyFunc called. Stackframe allocated. c is a part of Stackframe
E2	Control to pass Complex c. Ctor Complex::Complex(&c) called with the address of c on the frame
E3	Control on initializer list of Complex: Complex(). Data members initialized (constructed)
E4	Object Lifetime STARTS for c. Control reaches the start of the body of Ctor. Ctor executes
E5	Control at c.norm(). Complex::norm(&c) called. Object is being used
E6	Complex::norm() executes
E7	Control to pass return. Dtor Complex::~Complex(&c) called
E8	Dtor executes. Control reaches the end of the body of Dtor. Object Lifetime ENDS for c. return executes. Stackframe including c de-allocated. Control returns to caller

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

So, I will start with a basic abstract chart of an application. So, on the left hand side is my application, which is, let say is one function MyFunc, which does not do anything meaningful. It just a function MyFunc and on the right at snippets of the class code for my complex class, they have necessarily three codes. That I have picked up here; the constructor code, a member function called norm which finds a norm of the number and the destructor code.

So, if we look into what is going to happen? From the instant, the function MyFunc is called by some caller, till the time MyFunc completes its execution and returns the control to the caller. Let us try to see what happens to the objects that are defined in this scope of MyFunc. These are objects like complex c, which are defined locally within this function scope. These are commonly called automatic objects. Now, to trace the sequence of events that happens. In the comments, you will see that I have annotated the comments with some event number E1, E2, like this. These numbers show the order in which the events happen.

So, the first event is E1, where the function has been called; which means as soon as the function has got called, there is a stack frame allocated on the stack. Corresponding to this function and this local variable, this local object c has an allocation on that stack frame, which is, which will give its memory location which eventually will become this pointer. So, this is what happens at E1 and then the control moves on. There is something

in between, which we do not care. And, it comes to the point where the control is about to pass the instantiation of *c*, when the second event E2 happens; that is, the constructor gets called for this class; which means that the control goes over to here with the address of the *c* object as allocated on the stack frame.

As it reaches the constructor, first the initialization has to happen; which means that before the body of the constructor can even start executing, all data members as listed in the initialization list will one after the other be initialized with the value specified. So, first the *r e* gets initialized, then *i m* gets initialized and this is the event three, E3, that will happen. On completion of this initialization, the control will reach the beginning of the body of the construction. And at this point; precisely at this point, we say that the object is actually available. Its lifetime starts. And that is the event E4.

So, if you look into the table below where, whatever I am saying is written down here in the table below. This is the beginning of the life time. So, please bear in mind that the object does not start or its lifetime does not start, when the constructor is called away, the instantiation happens. But, it starts when the constructor has been called and it has completed the initialization part of it.

So, when the lifetime of an object starts, the basic notion is the object has all its data members completely ready, and with initial values as specified in the constructor. Before that, from the time E2 happens and to the time E4 happens, that is, during E3 the object has what is known as an inconsistent state. It may or may not have proper values of the data members. For example, it is quite possible that if I look into the object during this time, may be *r e* component has been initialized to the value given and *i m* component is yet to be initialized. So, the object is said to be in an inconsistent state, as long as the initialization is going on. And, only on completion of that the control reaches the beginning of the constructor body. The event marked as E4, where the object is ready and can now be used.

So in the body of the constructor, actually you write any code, which can assume that the object is already properly ready and going. So, once the construction is over, this constructed body, the control comes back here. Some more stuff keeps on happening and then, again this object is used to invoke some method of the complex class, which is event E5; which gives a call to this method. This execution start; which is event E6 then

it computes the norm of the number and returns that norm back here and again things continue.

So, this basically, if you look into this part in the middle after the construction has happened, then we may have several different use of the object, where the object, where data members are being used, the data members are being changed, different methods are getting invoked and so on and so forth. The object is ready and is getting used. It is in the prime of its lifetime, till it reaches the return statement.

When it reaches the return statement, then you know that at this return statement, you know that as soon as this return statement will get executed, the control will go out of MyFunc. And, if the control goes out of MyFunc, this scope in which this object `c` belongs will no more be available. When the control goes out of MyFunc, the allocation of the frame on the stack for this function will no more be valid. Therefore, the address of `c` on the stack will no more be valid. So, this is the point.

Precisely, the precise point is quiet delicate. It is right before the return, but at the return it is not the previous statement, is not the next statement. But, right at this point the destructor will have to get called. So, here a call to `c` dot the destructor tilde `complex` will get called. So, on E7, the control will go to the destructor. The destructor will go through. The destructor body could do different `d` initialization activities. But, please remember the object is still in use. The object still has got a life time which is valid, till it reaches the end of the destructor body.

Event E8, when it is considered that the lifetime of the object is over and the control comes back to return again. So, when the control comes back, at that point the object is no more a valid one. The object lifetime is already over and then proceeds with the purpose of return, which will return the control back to the caller. And, it will also deallocate the stack frame for this function. A part of which was containing the object `c`, which we were tracking the lifetime for.

So, if we just summarize, it is this point in the construction and it is this point in the destruction. During the execution of the program, which defines the lifetime of an object in the particular context in which we are showing we will show different lifetime constructs; we will show how lifetimes differ. But, it is always between the beginnings of the body of constructor to the end of the body of destructor; is a time during which the

corresponding object is alive and is considered to be its lifetime.

Actually, this understanding will also help us to understand something more specifically about the initialization list. I often get questions from people trying to learn the construction, understand the construction process that why do I need an initialization list? I could have instead in the body of the constructor written something like `r e underscore assigned r e` or `i m underscore assigned i m`. I could have written this as a part of the body of the constructor, instead of doing the initialization here.

So, there are two questions; as to why the initialization list is at all required? And, even if the initialization list is provided, is it necessary to initialize? The answer is as you have understood the object is the moment the control reaches this point; the object is considered live; the object is constructed. The construction is complete.

So, if you put the values of the data members within the body of the constructor, then at this point when the lifetime starts, your data members have inconsistent values. It may have typically garbage values. Now, if you look into the example of a complex class which has just two doubles, it really is not going to matter in your course of program execution as to whether you really, truly initialize or you do not initialize and let the garbage value of `r e` and `i m`, start the object, and then is a body of the constructor. You set new values to `r e` and `i m`. It is not going to make a difference, but later on we will show examples where it does really matter as to whether you have a proper object.

When you reach this point; this is the beginning point of the object. When you reach this point, whether you have a properly initialized object or not for many classes that may make a difference. And when it does, then you really have a problem because there is no way to solve that other than using the initialization list. This is another factor to doing the initialization list. Just think of that there are number of data members in a, in an object. And, in the initialization list or in the process of initialization, I can initialize; I mean whatever order I want. So, if initialization of one data member is dependent on the initialization of the other, it will matter as to which data member is initialized earlier and which is initialized later.

Now, if I write the initialization as assignments in the body of the constructor, then there could be; if there are n different data members, there could be factorial n different ways of initializing data members. But, if I write it as an initialization list, the compiler

follows a unique approach. The compiler initializes them in the order in which you write them in the class, not in the order in which you write them in the initialization list. So, for this example of complex, we have written it as `re` and `im`. We could have written this as; we could have written `im` initialization first, followed by `re` initialization. But, even if I write the initializer list like this, this initialization will happen before this initialization, as long as the data member `re` precedes the data member `im` in the definition of the class.

So, the result of having initialization list is without the user having to put any effort, the process of initialization is unified, in the sense that data members are necessarily initialized from top to bottom, a mechanism which you could not have been able to guarantee, if you were to put initial values in the body of the constructor. So, with this we understand that this is what the lifetime is and this is where it starts and this is where it ends. In the next couple of slides, we will quickly take look into different examples of lifetime and try to understand this better.

(Refer Slide Time: 15:31)

The slide is titled "Object Lifetime" and is part of Module 13, "Programming in C++". It contains the following content:

- Execution Stages**
 - Memory Allocation and Binding
 - Constructor Call and Execution
 - Object Use
 - Destructor Call and Execution
 - Memory De-Allocation and De-Binding
- Object Lifetime**
 - Starts with execution of Constructor Body
 - As soon as Initialization ends and control enters Constructor Body
 - Must follow Memory Allocation
 - Ends with execution of Destructor Body
 - As soon as control leaves Destructor Body
 - Must precede Memory De-allocation
 - For Objects of *Built-in / Pre-Defined Types*
 - No Explicit Constructor / Destructor
 - Lifetime spans from object definition to end of scope

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

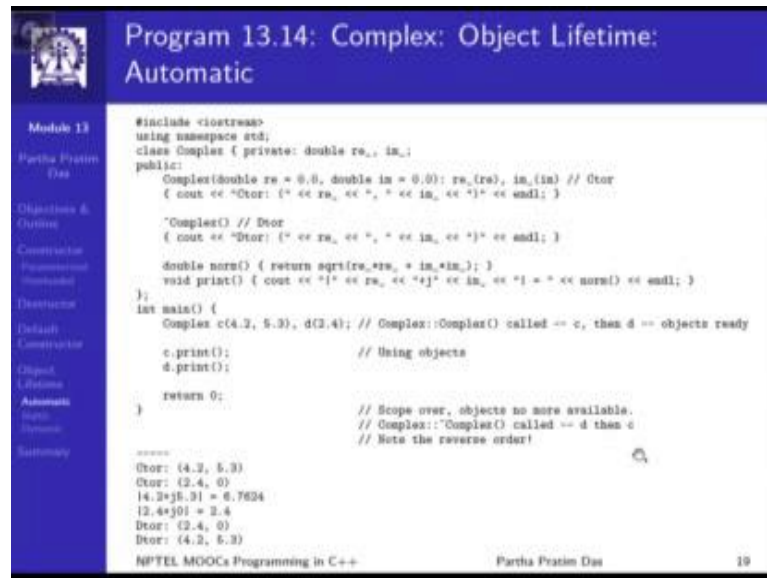
Just before we move on, this is just to summarize what I have discussed so far in that illustration. That an object may be considered to have five different execution stages because first for the object to exist there has to be a memory, which has to get allocated somehow, either on the stack automatically or in the global common area statically or in the heap dynamically.

So, an allocation is required because without the allocation I do not have a memory location where to store the different component values. And, as I do the allocation there has to be a binding. That means, there has to be an association between the name of the object and the address that I am using for that object in the memory. So, this is the first execution state that has to happen. And, for all different types of objects this happens in multiple different ways. You need not get worried about this binding aspect right now. As we go over more and more depth of C++, this binding concept will become more and more clear. But, once a binding has been done, that is, I have a memory to create the object, with that memory we will call the constructor implicitly. And as we have just discussed, it executes through the initialization list. And, the object gets constructed, then the execution continues to the constructor body.

The third stage is all different use of the object, till the destructor call happens. And, the destructor again goes through the body, executes to destroy notionally all the different components. And, at the end of the destructor body we will have the destruction process over. And after that, it is time to deallocate the memory. And, once the memory gets deallocated, the debinding happens. That is, the association between the address and the name is no more valid. So, this is what we have mentioned here.

One point to note at this, here in terms of the built-in or predefined types, you may note that notionally all types in C++ are assumed to have constructor or destructor. But, in reality there is no constructor or destructor for the built-in types. They just follow the same syntax for uniformity. But in reality, they are just simple bitwise assignment of values that happens or initialization of values that happen. So for a built-in type, the lifetime basically spans from the point of definition to the end of scope.

(Refer Slide Time: 18:25)



```
#include <iostream>
using namespace std;
class Complex { private: double re, im;
public:
    Complex(double re = 0.0, double im = 0.0): re(re), im(im) // Dtor
    { cout << "Dtor: (" << re, << ", " << im, << ")" << endl; }

    ~Complex() // Dtor
    { cout << "Dtor: (" << re, << ", " << im, << ")" << endl; }

    double norm() { return sqrt(re*re + im*im); }
    void print() { cout << "(" << re, << "+j" << im, << ") | = " << norm() << endl; }
};

int main() {
    Complex c(4.2, 5.3), d(2.4); // Complex::Complex() called -- c, then d -- objects ready
    c.print();                // Using objects
    d.print();

    return 0;                // Scope over, objects no more available.
                             // Complex::~Complex() called -- d then c
                             // Note the reverse order!
}

=====
Dtor: (4.2, 5.3)
Dtor: (2.4, 0)
(4.2+5.3j) = 6.7624
(2.4+0j) = 2.4
Dtor: (2.4, 0)
Dtor: (4.2, 5.3)
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 19

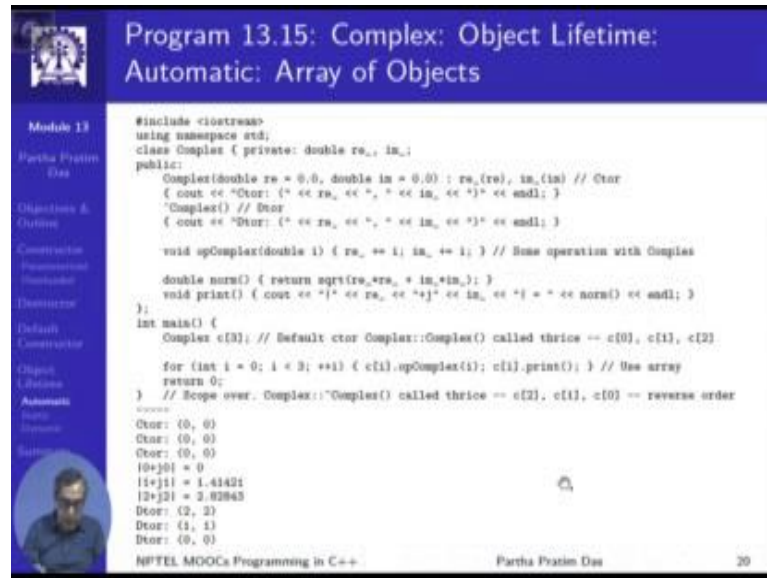
So, this is a more complete example on the lifetime of automatic objects. That is, objects that are necessarily local to a function body or a function parameter and so on. So, this is, these are two objects here. So, if we look into the lifetime, then certainly c will get constructed first. So, the constructor of c will get called first. And, that is why this is what you see at the message statement that when the constructor is called, r e and i m is set to four point two and five point three. So, this is the construction of c.

Subsequently, d is constructed. This is the construction of d, then these are printed. So, that is the use of the objects. And finally, at this point, at the point of return, the destructors will get called. And, I will just minimize this to show you that a destructors have been called and they destroy the two objects.

Now, there is something. One point that you must note very carefully is the destructors are called in the reverse order of construction. That is the default automatic behavior. When the objects are getting constructed, they are getting constructed one after the other. So, c has been constructed first and d has been constructed after that. But, if you look into the destruction, the d is destructed first and c is destructed next. So, you can think of as if the construction, destruction or literally construction, destruction is a LIFO process, where the order in which they are constructed is a reverse order in which they are destructed. As long as you have an automatic or a static object, this construction and destruction rule will have to be followed. So, if c, the object d has a dependence on

object c, then it is logical to construct c and then construct d. But, you will have to remember that d will disappear; have to disappear first as a destruction of d will happen earlier than the destruction of c.

(Refer Slide Time: 20:49)



The slide displays a C++ program named 'Complex' and its execution output. The code defines a class with real and imaginary parts, a constructor, a destructor, a norm function, and an operation function. It creates an array of three objects and prints their details.

```
#include <iostream>
using namespace std;
class Complex { private: double re, im;
public:
    Complex(double re = 0.0, double im = 0.0) : re(re), im(im) // Ctor
    { cout << "Ctor: (" << re << ", " << im << ")" << endl; }
    ~Complex() // Dtor
    { cout << "Dtor: (" << re << ", " << im << ")" << endl; }

    void opComplex(double i) { re, += i; im, += i; } // Sum operation with Complex

    double norm() { return sqrt(re*re + im*im); }
    void print() { cout << "(" << re << "+j" << im << " = " << norm() << endl; }
};

int main() {
    Complex c[3]; // Default ctor Complex::Complex() called thrice -- c[0], c[1], c[2]
    for (int i = 0; i < 3; ++i) { c[i].opComplex(i); c[i].print(); } // Use array
    return 0;
} // Scope over. Complex::~Complex() called thrice -- c[2], c[1], c[0] -- reverse order

Ctor: (0, 0)
Ctor: (0, 0)
Ctor: (0, 0)
(0+j0) = 0
(1+j1) = 1.41421
(2+j2) = 2.82843
Dtor: (2, 2)
Dtor: (1, 1)
Dtor: (0, 0)
```

Output:
Ctor: (0, 0)
Ctor: (0, 0)
Ctor: (0, 0)
(0+j0) = 0
(1+j1) = 1.41421
(2+j2) = 2.82843
Dtor: (2, 2)
Dtor: (1, 1)
Dtor: (0, 0)

Another example, here we specifically show the construction of array objects. So, we were continuing to use the complex as a class. So, we have an array of three of complex objects that I should mention that if you want to construct arrays of objects, then your class must support a default constructor. The reason is simple. If the constructor is not default, its call will require the parameters to be passed.

Now, if I have an array as in here of c 3, which means the name of the array c and I have three objects residing at c 0, c 1 and c 2. So, naturally there are three different constructors that have to be called or rather the constructor will have to be called thrice; once for this object c 0, once at this address c 1 and then finally at this address c 2. So, there has to be three calls of the constructor. And, this is what is what you can see here by tracking the message that the constructor prints. And, since it is not convenient notationally to provide the six possible parameters of these three constructors, C++ assumes that if you are constructing array of objects, then the constructor has to be a default one. Then, you are doing something with this array.

So, for just illustration purpose I have defined a member function, operate complex, which does nothing but takes a double value I and adds that to both the members. So,

kind of it diagonally shifts the complex number on the complex plane. So, I do that on all the array objects one after the other. For every c, for every array object, I applied the complex and do the print. So, these are the different complex numbers that I have now; at the zeroth location, I have 0, 0; at the first location, I have 1, 1; the second location, I have 2, 2. I have just done this to illustrate the order in which the destructors will happen.

So, when the scope gets over finally at this point of return, then certainly the destructors will have to get called. And by the rule, since this is the order in which the construction happened, naturally the destruction will have to happen in the reverse order. That is, c 2 will be destroyed first, then c 1, then c 0. A fact that you can see here that the destructor for the complex number 2, 2, happens first, then the destructor for c 2, c 1, that is, complex number 1,1 and finally c 0. So, this clearly shows what needs to be done in case of array objects, in terms of their lifetime. So, again from this construction to destruction is a lifetime of all the array elements that we have.

(Refer Slide Time: 23:59)

Program 13.16: Complex: Object Lifetime: Static

```

#include <iostream>
using namespace std;

class Complex { private: double re, im;
public:
    Complex(double re = 0.0, double im = 0.0): re(re), im(im) // Ctor
    { cout << "Ctor: (" << re << ", " << im << ")" << endl; }
    ~Complex() // Dtor
    { cout << "Dtor: (" << re << ", " << im << ")" << endl; }
    double norm() { return sqrt(re*re + im*im); }
    void print() { cout << "(" << re << "+j" << im << ") = " << norm() << endl; }
};

Complex c(4.2, 5.3); // Static (global) object
// Constructed before main starts
// Destructed after main ends

int main() {
    cout << "main() Starts" << endl;
    Complex d(2.4); // Ctor for d

    c.print(); // Use static object
    d.print(); // Use local object

    return 0;
} // Dtor for d

// Dtor for c
    
```

----- OUTPUT -----
Ctor: (4.2, 5.3)
main() Starts
Ctor: (2.4, 0)
(4.2+5.3i) = 6.7624
(2.4+0i) = 2.4
Dtor: (2.4, 0)
Dtor: (4.2, 5.3)

NPTEL MOOCs Programming in C++ Partha Pratim Das 21

This is another example with the same class showing lifetime. Now, here is a main function, but we have defined an object statically here. In the static area is a global static object. So, rest of the class is the same. Here is why you can see the output. And, here the output is critical to be able to understand what is happening. And, please track the constructor output message. So, the constructor output message for 4.2, 5.3, this global

static objects you see is printed first, even before main has actually started. So, that is beautiful information. That is a beautiful understanding.

So far as C was concerned, we always understood that everything starts from main. The computation has to start with main; that is the beginning of. In C++, main, still is a entry point function; main, still is a function that you need to provide in a C++ function. And, that is the first function that will get called by the system. But, the computation necessarily does not start with main. The computation starts with the constructors of static objects which have to be constructed, initialized, before main starts. So, all static objects get constructed before main starts.

So, rest of it is constructing a local object as in here. Then, using both these objects the lifetime continues. And, since d is a local object, when the main reaches the end of scope, the destructor for d gets called. And, what is not visible here is this is the point when the main has actually returned. And, the destructor for this static object c happens, after main returns, which is a, which is matching, befitting with the LIFO strategy that the order of destruction has to be reverse of the order of the construction.

So, if I just talk about the simple example, where there is only one static object, the order will construct that static object, call main. Main will construct local objects; use the static as well as local objects, till the scope of main ends. And, at the end of the scope of main, destroy the local objects. The main returns then destroy the static object that was constructed. And, this is; so, actually there is scope for executing code in C++, before main starts and after main has completed.

(Refer Slide Time: 26:54)

The slide displays a C++ program titled "Program 13.17: Complex: Object Lifetime: Dynamic". The code defines a `Complex` class with private members `re_` and `im_`. It includes a constructor, a destructor, and a `print` method. The `main` function demonstrates various ways to create and use `Complex` objects: using `new`, `new []`, and `new (buf)`. It also shows the use of `delete` and `delete []` to release memory. The output of the program is shown on the right side of the slide.

```
#include <iostream>
using namespace std;
class Complex { private: double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
    ~Complex() // Dtor
    { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "(" << re_ << "+j" << im_ << ") = " << norm() << endl; }
};

int main() { unsigned char buf[100]; // Buffer for placement of objects
    Complex* pc = new Complex(4.2, 5.3); // operator new: allocates memory, calls Ctor
    Complex* pd = new Complex[2]; // operator new []: allocates memory,
    // calls default Ctor twice
    Complex* pe = new (buf) Complex(2.6, 3.9); // operator placement new: only calls Ctor
    // no allocation of memory, uses buf
    // Use objects
    pc->print();
    pd[0].print(); pd[1].print();
    pe->print();
    // Release of objects - can be done in any order
    delete pc; // delete: calls Dtor, releases memory
    delete [] pd; // delete[]: calls 2 Dtor's, releases mem
    pe->~Complex(); // No delete: explicit call to Dtor
    // Use with extreme care
    return 0;
}
```

----- OUTPUT -----
Ctor: (4.2, 5.3)
Ctor: (0, 0)
Ctor: (0, 0)
Ctor: (2.6, 3.9)
14.2+5.3j = 6.7624
10+j0 = 0
10+j0 = 0
12.6+3.9j = 4.68722
Dtor: (4.2, 5.3)
Dtor: (0, 0)
Dtor: (0, 0)
Dtor: (2.6, 3.9)

The last example is in terms of a dynamic allocation. We have seen how to; the different dynamic allocation operators, operator new and so on. So, we again use complex to illustrate that. This is using operator new. I can create a complex object. I can use; create operator new array form to create an array of complex objects. I can do a operator placement, I can do a new placement into a given buffer for an object.

Now, certainly when new happens, then first the new will allocate memory and then the construction will happen. So, like in cases of automatic or static objects, the construction was implicit. Similarly, in new also the construction is implicit. That is the only difference being that before the construction, the new actually will allocate the memory dynamically. So, both these forms of new and array new, will allocate memory and then will call the corresponding necessary constructor.

So, when these are released, like I do delete p c, that is, I am trying to release this object. Then, this delete will call the destructor, corresponding to the call of the constructor and then it will release the memory. So, new and delete are not; we had said earlier that new and delete are like malloc and free. But, they are not exactly malloc and free because malloc only allocates memory, but new allocates memory and also calls the constructor, free only deallocates memory, but delete, calls the destructor and then deallocates memory. So, this difference has to be kept in mind. The total whole output of this program is shown here. You can; you should carefully trace to understand what is going

on here.

Since, it is dynamic allocation, so the user has full freedom as to when to allocate; when to create the object by new and when to destroy it by delete. And, the lifetime is limited between them. Only exception that you will have to remember is if you are doing a placement new, as we are explained earlier the memory is not to be allocated. It is coming from, provided by the user as in case of buffer. Therefore, you cannot do a delete on a pointer that has been created by placement new.

So, for this object p e, which is created by placement new, you will have to do something which is explicit destruction. That is, on that pointer you will have to actually call the destructor of complex. So, this is one of the very rare cases, where you explicitly call the destructor because here it cannot be packaged within the delete operation because delete will need to release the memory, which you do not have here. And, it cannot be implicit because it is a dynamic process, which you want to manage yourself.

(Refer Slide Time: 30:05)

Module Summary

- Objects are initialized by Constructors
- Constructors can be Parameterized and can be Overloaded
- Default Constructor does not take any parameter. It is necessary for defining arrays of objects
- Objects are cleaned-up by Destructors. Destructor for a class is unique
- Compiler provides *free* Default Constructor and Destructor, if not provides by the program
- Objects have a well-defined lifetime spanning from execution of the beginning of the body of a constructor to the execution till the end of the body of the destructor
- Memory for an object must be available before its construction and can be released only after its destruction

NITEL MOOCs Programming in C++ Partha Pratim Das 23

So, this is the whole process of object lifetime. Please, go through this carefully couple of times because construction, destruction and associated object lifetime is all of the very, one of the core concept of object management in C++. And, whole of what we will do in the subsequent modules in the subsequent weeks, will critically depend on your understanding of the construction and the destruction process and the validity of the lifetime. All that we have that we have discussed in this module are given in this

summary. So, you could just check back that you have understood all these points.

And, thank you very much.