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

**Lecture – 24**
**Constructors, Destructors and Object Lifetime (Contd.)**

Welcome to module 13, part-2 of Programming in C++.

In the first part of this module, we have seen how objects can be constructed using the constructor of classes, how implicitly constructors get called and they can be used to initialize data members of objects by specifying through the initialization list, and we have also seen how we can have parameterized constructors? How we can use default values for those parameters? And how we can have overloaded constructors for all varieties of construction and initialization mechanism to be provided?

(Refer Slide Time: 01:16)

Program 13.10/11: Stack: Destructor

Module 13
Partha Pratim Das
Objectives & Outline
Constructor
Parameterized
Overloaded
Destructor
Default Constructor
Object Lifetime
Automatic
Static
Dynamic
Summary

Automatic Array

```
#include <iostream> using namespace std;
class Stack { private:
    char *data_; int top_; // Dynamic
public: Stack(); // Constructor
    void de_init() { delete [] data_; }
    // Stack methods
};
Stack::Stack(): data_(new char[10]), top_(-1)
{ cout << "Stack::Stack() called\n"; }

int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call

    // Reverse string using Stack
    de_init();
    return 0;
}
-----
Stack::Stack() called
EDCBA
```

- Dynamically allocated data_ leaks unless released before program loses scope of s
- Application may forget to call de_init(); Also, when should de_init() be called?

Dynamic Array

```
#include <iostream> using namespace std;
class Stack { private:
    char *data_; int top_; // Dynamic
public: Stack(); // Constructor
    ~Stack();    // Destructor
    // Stack methods
};
Stack::Stack(): data_(new char[10]), top_(-1)
{ cout << "Stack::Stack() called\n"; }
Stack::~Stack() {
    cout << "\nStack::~Stack() called\n";
    delete data_;
}
int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call

    // Reverse string using Stack

    return 0;
} // De-Init by Stack::~Stack() call
-----
Stack::Stack() called
EDCBA
Stack::~Stack() called
```

- Can de-initialization (release of data_) be a part of scope rules?
- Yes. Destructor is implicitly called at end of scope

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

Now, we will look at the other side of the story. We will look at the process of what happens when an object reaches the end of it is lifetime or it is time to destruct. So, again we go back to the stack example. We are looking into the stack having private data and the container that contains the elements of the stack is dynamically allocated array therefore, we just have a pointer.

So, in terms of the stack we know what needs to be done, in terms of the construction we

know what needs to be done, we have seen this. A dynamically allocation is done with operator new for an array of 10 characters and that pointer is stored in data.

So, the stack will work, it will initialize through this construction and then it will do the reverse. So, just for clarity of presentation I have skipped all the details of the stack methods and the reversing string code they are same as before, but having done that when we reach to this point, we are done with this stack and we should return. But the point is, as we have already noted there is a dynamically allocated array whose pointer is held by the data component of the stack object s.

Now, if we return at this point; we return from main and the control goes away, then certainly this dynamically created array remain becomes inaccessible because quiet clearly s, the object s that is instantiated here is available only within the scope of this function. So, if I do return and go out of it then there is no way that I can access s, there is no way that I can therefore access s dot data or release the memory that we have actually acquired through the new process done in the construction. So, to avoid this problem to be able to manage our resources right at release them, whenever we are done with using the resource there has to be a matching mechanism of d initialization, which will undo what we had done at the initialization time.

So, we make a similar assumption as we are done earlier for initialization, let there be a function d in it, why do we need a function? And why do we need that function to be put here? Because to release the allocated array, we need to access the data, underscore data member of the stack class which is private. So, we put d in it method in the public and then call that method, style similar to what we did in initialization and the problems this will cause the problems which is similar to what happened for initialization also. That is precisely what if the application would forget to call the init that is one part.

Second, in this case we have a bigger problem because it is not only about forgetting to call de init, but there is a bigger issue is even if I remember, what is up exact location where I should call de init, if I call de init and after calling d init I try to use the stack. Then I will have evaluation because the stack does not have the container any more. I have already released it, but de init is within the same scope where the stack is defined. So, it is possible that after calling de init here may try to use the stack. So, I have to be very careful that every call to de init must ensure that after that there is no possible use of

s.

Second from a function; this is a very simple case we are showing here, from a function there could be multiple places from where I actually return, if I am returning from multiple places I do not know before and which particular return statement will be taken by the control flow. So, I will need to remember and put de init at each one of these places. So, it is a complete message, it is a complete mess in. It is one of the major reasons is a known source of problem in C that resources allocated are not properly de-allocated or not properly released.

In C++, we have a beautiful solution in terms of what is known as destructor. Let us look at that solution; what we do, we introduce another member function in the class. This member function too have the same name as the class, but the difference being it is prefixed to with the special symbol tilde and it is called the destructor of the class. So, what happens is as at the time of abstract instantiation, the constructor is getting called.

Similarly, when the object goes out of scope here that is, this is the curly brace on which the object becomes goes out of scope. In the sense that after this curly brace has been passed by the control, there is no way to talk about this s that is a scope rule of C that is the scope rule of C++. So, right at this point when it is trying to cross the end of this scope, a call will be made to the destructor of the class for this object s.

At this point s dot tilde stack, this function will get called and the beauty of the whole thing is that the user does not, or the programmer does not need to remember and make the call. The compiler implicitly would compute that, this is the end of scope for s and implicitly put this call at the end of the scope. So, if we now look at the whole mechanism together we do not need this kind of d init is no more, is not required here.

The stack gets initialized at this point to the constructor call to the call to this function and that is the message that it prints. Then the stack is used to reverse the string. The reverse string is output and then the control goes out of scope at this point and implicitly the destructor function is called and you can make out that the destructor function has been called by finding out that the message within the destructor function has been printed at the output.

So, this ensures that the data that was dynamically held up in this data member of the

stack class can be released at this point. So, whenever no matter in what context I use the stack, whenever I instance situate a class, the constructor ensures that proper allocation will be done to the data, proper initialization will happen to stop and whenever that instantiated stack object gets out of scope, a compiler will also ensure that the constructor is called so that the proper release of the allocated data can happen. So, this combined pair of constructor and destructor gives us a completely well defined structured way of managing the life time of every object in a very clean manner in C++.

(Refer Slide Time: 09:53)



So, to look at formal properties of destructor as we have already seen destructor is also a member function. It has at this pointer like any other member function, its name is special, it is a tilde followed by the name of the class, like the constructor destructor too does not have a return type because certainly we have seen it is called at the end of the scope and therefore, if it were to return something then there is no taker for that returned value. There is no meaning of having a return computed from the destruction of an object, so destructors also do not have any return type - not even void.

Certainly, consequently there is no return statement in a destructor and as we have already seen which is the most significant and important part is the destructors are called implicitly at the end of the scope and can be used seamlessly for any automatic object.

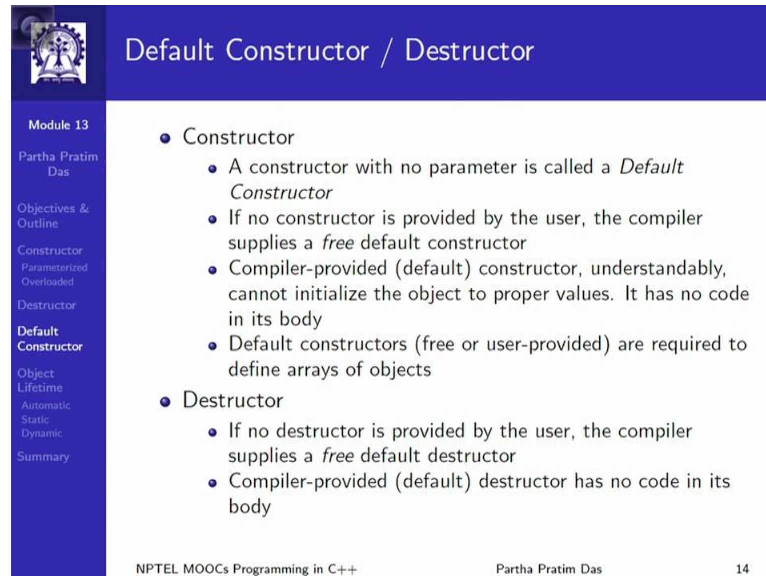
There are other ways to call the destructor also, we will illustrate that, but usually always the call to a destructor is implicit by the compiler through the measurement of this

automatic scope in which the particular object was constructed. In contrast to constructor, a destructor is not allowed any parameter. Since, it is not allowed, any parameter you will understand by now that the consequence of that is exactly, you cannot overload the destructor. So, if you cannot have any parameter and if you cannot overload the destructor which means that the destructor of a class is unique.

So, the scenario is a class may have one or more constructors as many as it wants through different mechanisms of overloading, but it must have only one way to destruct and there is a profound reason that the things are defined this way because obviously, once when you are about to construct an object, you certainly have choice, you would like choice in terms of what kind of object what initial values for parameters and all that, you want to set and based on that it is quiet logical that you may need different mechanisms or different parameters sets to construct the object and therefore, you need a number of overloaded constructors.

But, once we have constructed the object, there is no trace of the fact is to which constructor was used to construct it is object. There is no trace of whether a data member was initialized or it was subsequently set a value and so on. Therefore, when you want to destruct, all objects are similar. All that he as saying that I am done with this object, I want to release all the allocated resources that are held by my data members and I want to free up the memory in which the object currently resides. So, I need to comply with the destruction process and therefore, the destruction process for an object in a class has to be unique and therefore, there is unique single destructor for every class in the C++ design.

(Refer Slide Time: 13:17)



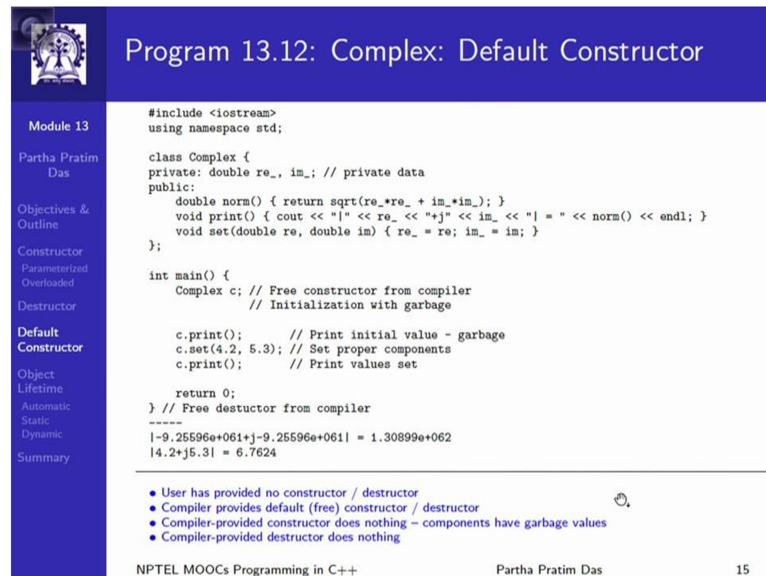

At this point, once we have seen the constructor and destructor, let us also note that constructors can be default. A constructor which has no parameter is called a default constructor. So, user actually has two options, user can provide write a constructor or it may not write a constructor, but the C++ has a mechanism must have a constructor for every class. So, what the compiler does is, if the user provides a constructor it will use that constructor, but if the user does not provide a constructor for a class then the compiler will supply a free default constructor.

Now, if the compiler supplies a constructor, naturally the compiler do not know what parameters you might want. The compiler does not know how to initialize your data members. So, just to make the whole process coherent, the compiler will give you a free default constructor and the code will compile assuming that you have a free default constructor, you have default constructor. So, it is usually good to provide a constructor even if it does not have a parameter because once you provide a constructor, even if it is default, your constructor will be used and compiler will not provide the free constructor because if you write the default constructor or if you write any other overloaded constructor then you really know what are you are initializing.

If the compiler provides it, then it will just have a whole lot of garbage values for your data members possibly, but the mechanisms exist that the compiler will give you a free default constructor. Similarly for destruction, if the user has not provided a destructor

then the compiler will supply a free default destructor which certainly does nothing it has got a empty body, but there is no code in their body because that compiler does not know what possibly needs to be released or if it is nothing to be released, but to make the whole mechanism work the compiler will indeed provide a free default destructor.

(Refer Slide Time: 15:42)




So, here we show an example of default constructor. Please take a look into this class. It has our complex class, the old front of us. What I have added for illustration, I have added a new member function method set which takes two double values and sets two values into the two data members of the class which can be invoked at point of time to set a complex value to my class, but what is missing now, you know is that there is no constructor and there is no destructor provided. So, what will happen if we; when this instantiation is happening at this point there will be a call to the default constructor that has freely been provided by the compiler?

So, this call will happen even though no such function has been defined by you in the body of the class. So, what will this constructor initialize re and im with, it does not know what it needs to be initialized and therefore, it will just leave some garbage values at these points. So, to understand that, if after this object has been constructed if you immediately after that if you print this object. I mean this is just one case, if you try this experiment yourself, you will get some different values possibly. So, just prints this which is some garbage bit pattern which existed where re and im should be there, but

once I have used the set function that I have provided here to set the component values to re and im and then I do the print, I do get the proper values again.

Simple advice that, if you do not provide the constructor then unfortunately the compiler will not give an error, compiler will provide a default constructor default destructor and go ahead with that. So, he will run the risk of not having proper values, proper initialization syntax. So, whenever you write a class, make sure that you write a constructor and the destructor.

(Refer Slide Time: 18:03)

Program 13.13: Complex: Default Constructor

Module 13

Partha Pratim Das

Objectives & Outline

Constructor
Parameterized
Overloaded

Destructor

Default Constructor

Object Lifetime
Automatic
Static
Dynamic

Summary

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

class Complex { private: double re_, im_;
public:
    Complex(): re_(0.0), im_(0.0) // Default 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; }
    void set(double re, double im) { re_ = re; im_ = im; }
};
int main() {
    Complex c; // Default constructor -- user provided

    c.print();       // Print initial values
    c.set(4.2, 5.3); // Set components
    c.print();       // Print values set

    return 0;
} // Destuctor
-----
Ctor: (0, 0)
|0+j0| = 0
|4.2+j5.3| = 6.7624
Dtor: (4.2, 5.3)
```

- User has provided a default constructor

NPTEL MOOCs Programming in C++ Partha Pratim Das 16

Here is another example where we are showing the default constructor, but in this case the only difference is the default constructor is not provided by the computer, but the default constructor is provided by the user. So, user has written this. So, when you do the first printing after the initialization naturally, since there was an initialization with proper values 0 and 0, you get that there is proper initial values and not the garbage kind of garbage that we saw in the last example.

Similarly, a destructor is also given here. In this case, the destructor prints this message. In reality a destructor for a complex class may not do anything, but it is always good to provide one with a blank body.

With this we will close on the construction and destruction process. We have learnt that every class will have a destructor which is unique and which will get invoked for

automatic objects, it will get invoked at the end of this scope and within this destructor, we can release clean up any kind of resources that we have been holding on to, and we have also seen that the compiler provides a default constructor and a default destructor provided the user has not written a destructor or a constructor for the class.