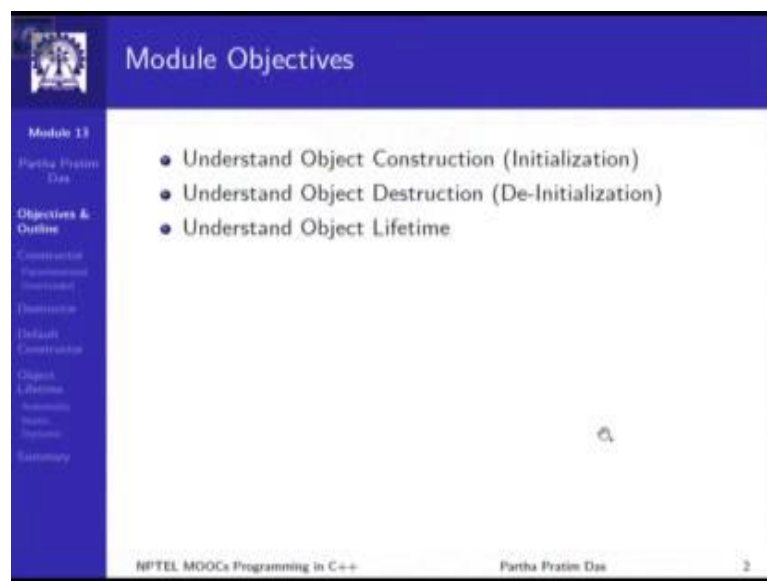


Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 23
Constructors, Destructors and Object Lifetime

Welcome to module 13 of Programming in C++.

(Refer Slide Time: 00:30)



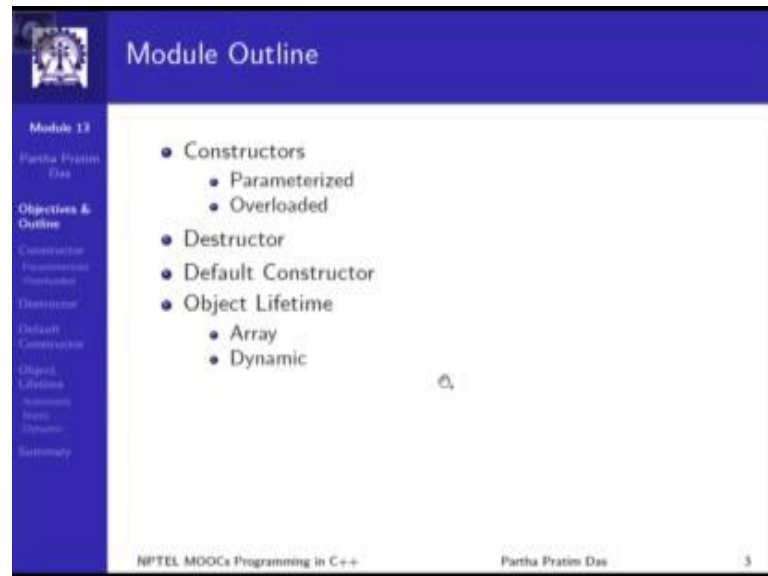
The screenshot shows a presentation slide with a blue header and footer. The header contains the text "Module Objectives". The main content area is white and contains a bulleted list of three objectives: "Understand Object Construction (Initialization)", "Understand Object Destruction (De-Initialization)", and "Understand Object Lifetime". On the left side, there is a vertical navigation menu with the following items: "Module 13", "Partha Pratim Das", "Objectives & Outline", "Constructor", "Destructor", "Default Constructor", "Object Lifetime", "Access Specifier", "Friendship", and "Virtuality". The footer contains the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "2".

In this module, we will talk about understanding how objects are constructed or initialized, and after their use how they are destructed or deinitialized, and in this whole process what is the life time that the object will have. In terms of the object oriented futures in C++ we have already seen how classes can be defined and how objects of a certain class can be instantiated.

We have already had a look in the data members and methods and we have talked about the whole paradigm of information hiding the basic design process of access specification and visibility restriction to make data members particularly private in a design, and provide an interface to methods which is public. Now, we will extended on that and talk more, this was more on the design aspect in terms of the structure of a class

or the blue print of different objects. Now in the current module will look specifically on the runtime execution, runtime behavior of an object starting trying to find out when does it get into life, and how long does it remain in life? And what happens at the end of life of an object?

(Refer Slide Time: 02:07)



We will get started with all these discussions, this is outline does it can see on the left of your panel also.

(Refer Slide Time: 02:13)

Program 13.01/02: Stack: Initialization

Public Data	Private Data
<pre>#include <iostream> using namespace std; class Stack { public: char data_[10]; int top_; public: int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } }; int main() { char str[10] = "ABCDE"; Stack s; s.top_ = -1; // Exposed initialization for (int i = 0; i < 5; ++i) s.push(str[i]); // s.top_ = 2; // RISK while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre>	<pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_; public: void init() { top_ = -1; } int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } }; int main() { char str[10] = "ABCDE"; Stack s; s.init(); // Class initialization for (int i = 0; i < 5; ++i) s.push(str[i]); // s.top_ = 2; // Compile error - RISK while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre>

- Splits data structure codes into application
- public data reveals the internals
- In switch container, application needs to change
- Application may corrupt the stack!

- No code in application, but `init()` to be called
- private data protects the internals
- Switching container is seamless
- Application cannot corrupt the stack

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

So, let us refer back to one of the earlier examples of stack we have already introduced this earlier. This stack has two data members and array of characters to store the stack elements a top marker which will keep the index of the top element and in the public it has the four LIFO methods, LIFO interface, empty, push, pop and top to use this stack and here we show the use of this stack to reverse a string.

Now, if we look into this then in a closer scrutiny you will find that here we are using this instantiating this stack to create a stack object, but as soon as we create this stack object we cannot start using it for doing the reverse string operation that we are having to do here, in between we need to make sure that the stack has a proper initialization, in the sense that once `s` is instantiated I need to know a certain what is the value of the top marker at this point.

This is a point where no operation has happened so far, so you need to make sure that the stack just created conceptually has to be a null stack containing nothing an empty stack and therefore, its top marker must be one less than the first element that can go into the array that is the top marker must be minus 1. So, for this code to work it is critical that we add the initial value of top marker through an assignment after the stack variable as been instantiated.

If you look closely further you will realize that we are able to do this because I have defined the stack variables that data members as public. Therefore, I can easily access the top underscore data member and do this assignment and so therefore we what we observed here is in this solution we have an initialization which exposes the internal of the stack in contrary to the information hiding principles that we had invite earlier.

So, let us move on to the right column where in contrasts to the public data that we are used here, in contrasts we use private data we go back to our information hiding paradigm so we make the data private and if we do that naturally, we cannot write this anymore here because as you can very well understand that if we try to write `s.top` assign minus 1 then the compiler will not compile this program it will give us an error saying that top is a private data member and cannot be accessed. We solve that problem by introducing a separate function say by the name of `init` in it which we put on the interface as a public method and in it basically initializes top to minus 1, and here instead of doing `s.top` assign minus 1 we give a call to the `init` function. They should serve the same purpose.

With this we can still maintain the information hiding principle that the object oriented design must follow and ensure that the solution will work. And a side benefit of doing this is also you can compare here in this two lines, for example when the data member was public as we had illustrated earlier in terms of access specification module that if the data member is public it is possible to create some potential risk inadvertently `s.top` is assign to some value in the middle. But, when we are back to the information hiding principle now this kind of a code cannot be return with a private data because it will become compilation error. In that way making the data member private and providing a unit function is a better proposition of solution for the initialization problem, and certainly we would like to work on with this private data further.

(Refer Slide Time: 07:10)

Program 13.02/03: Stack: Initialization

Using init()	Using Constructor
<pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_; public: void init() { top_ = -1; } int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } }; int main() { char str[10] = "ABCDE"; Stack s; s.init(); // Clean initialization for (int i = 0; i < 5; ++i) s.push(str[i]); // s.top_ = 2; // Compile error - SAFE while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre>	<pre>#include <iostream> using namespace std; class Stack { private: char data_[10]; int top_; public: Stack() : top_(-1) {} // Initialization int empty() { return (top_ == -1); } void push(char x) { data_[++top_] = x; } void pop() { --top_; } char top() { return data_[top_]; } }; int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call for (int i = 0; i < 5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; }</pre>
<ul style="list-style-type: none">• <code>init()</code> serves no stable purpose - application may forget to call• If application misses to call <code>init()</code>, we have a corrupt stack	<ul style="list-style-type: none">• Can initialization be made a part of instantiation?• Yes. Constructor is implicitly called at instantiation as set by the compiler

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

Now let us move on. On the left column you see the same code that you just saw which as the private data and public methods including and in it method to initialize top. As we have observed that provides a clean initialization, but the question is certainly what if the application forgets to call in it, or what if the application would call in it at a wrong position? If the application misses to call this in it which for the application as actually no purpose because the application is trying to reverse a string and it knows that it is a LIFO structure which has empty, top, pop and push, these four methods but, for all this to work it as to call in it but if it misses on that then naturally we have an arbitrary value in top to start with, and therefore the whole program will give a very unpredictable behavior.

So you ask a question that can we do better than this, can we when the stack is actually instantiated when it is defined is it possible that right at that instantiation point if this initialization call can be somehow done. The mechanism of constructor basically provides this solution. C++ provides a mechanism by which if you instantiate an object a special function call the constructor is invoked right at this point, and this constructor can be used to initialize the values of data members as are required. Just looking into this example, then we do not have the in it call anymore this is not there instead we have introduced a new method which is called the constructor of this object. How do you

know this is the constructor? It has a typical signature of having the same name as the name of the class, so by the name you know this is a constructor it has a little bit of different way of putting things and we will discuss that more.

But, what we say here is necessarily we are saying that take the data member top and put minus 1 as a initial value. The other, it is not required to initialize the array because it will get used as and when push and pop happens. But the advantage is, if we define a constructor then as soon as the control will pass this particular point that is when the object gets instantiated immediately an automatic call an implicit call will happen to this method, and that call will make sure that top is indeed assign to minus 1. So when we return from this call at this point we already have the top of yes initialize to minus 1 and that is a this implicit initialization is a beauty of the constructor.

(Refer Slide Time: 10:51)

Program 13.04/05: Stack: Constructor

Automatic Array	Dynamic Array
<pre>#include <iostream> using namespace std; class Stack { private: char data[10]; int top; // Automatic public: Stack(); // Constructor // Stack methods }; Stack::Stack() { top_(-1) { // Init list cout << "Stack::Stack() called" << endl; } } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call for (int i=0; i<5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; } ----- Stack::Stack() called ABCDE</pre>	<pre>#include <iostream> using namespace std; class Stack { private: char *data; int top; // Dynamic public: Stack(); // Constructor // Stack methods }; Stack::Stack() : data(new char[10]), // Init top_(-1) { // list cout << "Stack::Stack() called" << endl; } } int main() { char str[10] = "ABCDE"; Stack s; // Init by Stack::Stack() call for (int i=0; i<5; ++i) s.push(str[i]); while (!s.empty()) { cout << s.top(); s.pop(); } return 0; } ----- Stack::Stack() called ABCDE</pre>
<ul style="list-style-type: none"> • top, initialized to -1 in list • data[10] initialized by default (automatic) • Stack::Stack() called automatically when control passes Stack s; - Guarantee initialization 	<ul style="list-style-type: none"> • top, initialized to -1 in list • data, initialized to new char[10] in list • Stack::Stack() called automatically when control passes Stack s; - Guarantee initialization

NPTEL MOOCs Programming in C++ Partha Pratin Das 6

Let us look at more examples say here, let us now we again have the same kind of example with the stack where the constructor is there and we know on the left hand side that the constructor will initialize top all that I am showing here is earlier the constructor was written as a part of the class right at this point you will remember. But now, we have written it outside of the class it is written as stack colon colon stack, this naming also you

will recall is any class member as a name which is qualified by the name of the class. So this is the name of the class stack, this is the name of constructor.

What we have is the body of the constructor, and in that body first we have initialized top to minus 1 and then rest of the C out statement. If we do this then as soon as the stack is instantiated here they call is made to this point top is initialized the C out happens which you see here control comes back and then the reversal of the string happens. A very seamless procedure by which the top is getting explicitly initialized, of course the other data member which is a array is at automatic default initialization because it is a given array of certain size.

Now, let us see if we just make a small change in state of having an automatic array if you just have a pointer to character, therefore what we are trying to do is to dynamically create this array. So naturally in this stack code the initialization to top remains same, but we now also have to initialize the array pointer which we do by dynamically allocating it. At this point let us understand this way of writing the initialization. Just note carefully that in the initialization we first write the name of the data member and then within a pair of parenthesis we write the value that we want to use it as an initialization. If we look at data this is the data pointer and this expression as you know will use the operator new allocate and array of character having 10 elements and return a pointed to that array having char star as a pointer type and that pointer is what is being set as a initial value in data.

This is something which you we had not seen earlier. The earlier styles where of assignment where we will say top, assigned one, or we will say data assigned new char tan like this. But here this special way of writing the initialization which is possible only when you are using a constructor and note that after the signature of the constructor you have a separator as colon and you need to write this whole list of initializations of data members between this colon and the opening curly brackets of the constructor body.

Here you write the data member one after the other separated by coma and after every data member within a pair of parenthesis you write what value you want to initialize them with. Certainly every data member can occur here only once and it is not necessary

that all data members will have to be initialized, but we can initialize all of them also if we want as we done in this case.

What will happen for this particular version of the stack? As soon as the control passes this point, where the stack as being instantiated this constructor will get called, so for the stack object s the top will get initialize to minus 1 there will be a dynamic location of the character array of size 10 and its pointer will be set to data, and then the stack call this message will get printed and on completion the control will come back to this point where the string reverse will proceed.

With this we can see that the constructor can be used to initialize the data member in multiple different ways and the mechanism of C++ will ensure that instantiation itself is initialization of the object as well. And the application has no over head of remembering or trying to make sure that the initialization happen for the stack data part or for the stack top marker part, it will be the responsibility of the constructor and the compiler will call this constructor implicitly we do not have to remember to call it at an appropriate point it will get called every time.

(Refer Slide Time: 16:44)

Constructor

- Is a member function with this pointer
- Name is same as the name of the class
- Has no return type
- No return, hence has no return statement
- Initializer list to initialize the data members
- Implicit call by instantiation / operator new
- May have any number of parameters
- Can be overloaded

```
class Stack { public:
    Stack();
};

Stack::Stack() // Not even void
{ } // Returns implicitly

Stack::Stack() // Initializer list
data(new char[10]), // Init data,
top_(-1) // Init top,
{ }
```

Member Function

- Has implicit this pointer
- Any name different from name of class
- Must have a return type
- Must have at least one return statement
- Not applicable
- Explicit call by the object
- May have any number of parameters
- Can be overloaded

```
class Stack { public:
    int empty();
};

int Stack::empty()
{ return (top_ == -1); }

void pop()
{ --top_; } // Implicit return
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 7

Now, just to sum up what does a constructor do as we have seen is a necessarily a member function, but it has a number of differences with the member function. So, being a member function it has this pointer like any member function, but its name as I already mentioned is specifically the name of the class which is not so for the other member functions.

Compiler you may have noticed already that there is no return type being shown because the job of the constructor is to construct the object and initialize it, it is not meant for doing a computation and giving you a value back. Even though at the constructor is a function it does not return any value and it does not have a return type not even void. You have seen function which have void return type which say that I am not returning anything, but in constructor that is not the case. In the constructor what you say there is no mechanism to return any value at all, so there is no return type to be specified.

Since there is no return type there is no return statement either so you will not find any return statement a like a common member function where there can be return statement or the return could be implicit if your return type is void, but a constructor will not need any return statement it will return implicitly. In addition the constructor will have the initialization list which I have just explained the list of data members within parenthesis the initialization value. The list starts with a colon ends with the beginning of the constructor body, and this is something which as no parallel for any other member functions. The constructor does not have any explicit call it is implicitly called by instantiation as soon as I instantiate this object the constructor will get called, where as the call objects are necessarily explicit in nature.

We will see examples, but finally a constructor may have any number of parameters in the example that we have seen right now there is no parameter, but a constructor like any other member function may have any number of parameters, and like other member functions a constructor can be over loaded as well.

(Refer Slide Time: 19:20)

The slide displays a C++ program for a complex number class. The code includes headers, a class definition with a parameterized constructor, a norm function, and a print function. The main function creates two complex objects and prints their values and norms. The output shows the complex numbers in standard form and their corresponding norms.

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

class Complex { private: double re, im;
public:
    Complex(double re, double im) { // Ctor w/ params
        re(re), im(im)           // Params used to initialize
    }
    double norm() { return sqrt(re*re + im*im); }

    void print() {
        cout << "r = " << re << "j" << im << "i = " << endl;
        cout << norm() << endl;
    }
};

int main() {
    Complex c(4.2, 5.3);           // Complex:Complex(4.2, 5.3)
    Complex d(1.6, 2.9);         // Complex:Complex(1.6, 2.9)

    c.print();
    d.print();

    return 0;
}

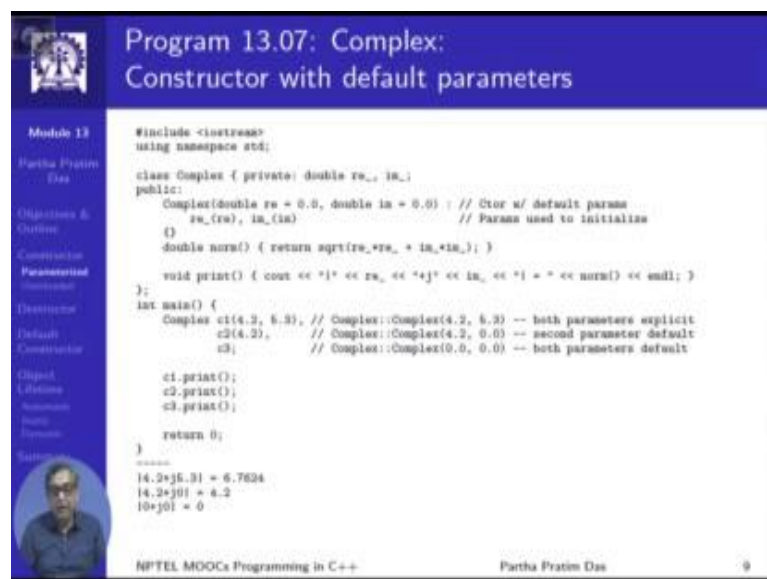
-----
14.2+5j5.3i = 6.7824
11.6+2j2.9i = 3.3121
```

A constructor is a special function which basically will help us in initializing objects implicitly all the time. We quickly flip through a couple of slides looking at variety of constructors that you can define in C++, you can have parameters in the constructor these are called Parameterized constructor. We are showing the double type again which we have already seen so this parameter values can be used to initialize the data members through the initialization list. Then the constructor will get invoked implicitly by instantiation and there are two syntax typically that can be used for it.

One is a typical function call kind of syntax, where you put the object name and with in parenthesis you put the parameters of the constructor in the same order in which they are define for the constructor. So when I do write this particular object instantiation it means that when control passes this point the constructor of complex will get called where 4.2 will get go as the value of r e, 5.3 will go as a value of i m and with that the object will get constructed and set as the object C. Beyond this if you do c point r e, c dot r e you should get a value 4.2 that is what this print statement show here where we print the complex number complex notation and compute its norm. Similarly, if I do i m component of c I will get 5.3.

In the second in case of d we just show the same thing in a different alternate notation this is called list initialization. That is, if you have multiple parameters in a constructor you could put the initial values of these parameters in terms of a pair of (Refer Time: 21:22) in the list notation and use an initialization symbol after the object name. The effect of both are same, they are just alternate notations for doing the same operation. So, constructors which have one or more parameters is known as Parameterized constructors.

(Refer Slide Time: 21:45)



The slide displays the following C++ code for a `Complex` class:

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

class Complex { private: double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0) // ctor w/ default params
        re_(re), im_(im) // Params used to initialize
    {}
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "i" << re_ << " + j" << im_ << " | = " << norm() << endl; }
};

int main() {
    Complex c1(4.2, 6.3); // Complex::Complex(4.2, 6.3) -- both parameters explicit
    Complex c2(4.2); // Complex::Complex(4.2, 0.0) -- second parameter default
    Complex c3; // Complex::Complex(0.0, 0.0) -- both parameters default

    c1.print();
    c2.print();
    c3.print();

    return 0;
}
```

The output of the program is:

```
14.2+6.3i = 6.7624
14.2+0i = 4.2
0+0i = 0
```

The slide also includes a sidebar with navigation options: Module 13, Partha Pratim Das, Objectives & Outline, Constructor, Parameterized Constructor, Destructor, Default Constructor, Object Lifetime, Name, Operator, and Summary. The footer contains 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the page number '9'.

Now, constructor in every respect is just another member function, or specifically just another function in C++ therefore, if I have parameters I can also have default values for parameters. So, I can have constructor with different default values so you show an example here again with the two parameters `re` and `im` having default values `0.0`. Then based on all the rules of default parameter for functions and default parameters for invocation of functions, we can use this same constructor to construct objects in three different ways I can put both the parameters I can just specify the first parameter or I can specify none of the parameter. Parameters that are not specified will take the default values and accordingly if you print out you will be able to see the corresponding objects as they have got created. It is just showing that the whole mechanism of defaulting parameter values all rules has we have learnt for function with default parameters will also apply to constructors.

(Refer Slide Time: 23:06)

The slide displays a C++ program for a Stack class. The class has a private member `char *data_;` and a private member `int top_;`. The public methods include a constructor `Stack(size_t = 10)`, `int empty()`, `void push(char x)`, `void pop()`, and `char top()`. The `main` function creates a stack `s` with the string `"ABCDE"` and pushes each character onto the stack, then prints the top element and pops it until the stack is empty. The output shows the stack size as 5 and the characters in reverse order: EDCBA.

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

class Stack { private: char *data_; int top_;
public:
    Stack(size_t = 10); // Size of data_ defaulted

    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};

Stack s(Stack(size_t s) : data_(new char[s]), // Array of size s allocated
        top_(-1)
        { cout << "Stack created with max size = " << s << endl; }

int main() {
    char str[] = "ABCDE";
    Stack s(strlen(str)); // Create a stack large enough for the problem

    for (int i = 0; i<6; ++i) s.push(str[i]);
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
    return 0;
}

=====
Stack created with max size = 5
EDCBA
```

This is just another example here, as we are going back to the stack, we show a constructor for a stack with a default size value of 10, size `t` 10. In the definition of the constructor, we have a parameter, so if I want, I can pass this parameter value. For example, here, we are constructing the object taking the length of the string that we want to reverse because we know that if this stack is meant to reverse the given string `STR`, then it is never going to need more size than the length of the string, so we can use that and make a stack which is just large enough to accommodate the string. Here, you can see that the constructor is printing as to what is the size of the stack that has got created. If I do not use this, if we do not provide all this information of what should be the size, if we just write `Stack s`, then certainly it will take the default parameter value 10 and create a stack of 10 elements.

So these are the different examples of constructors that we can have.

(Refer Slide Time: 24:25)

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

class Complex { private: double re, im;
public:
    Complex(double re, double im): re(re), im(im) {} // Two parameters
    Complex(double re): re(re), im(0.0) {} // One parameter
    Complex(): re(0.0), im(0.0) {} // No parameter

    double norm() { return sqrt(re*re + im*im); }

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

int main() {
    Complex c1(4.2, 6.3), // Complex::Complex(4.2, 6.3)
           c2(4.2), // Complex::Complex(4.2)
           c3; // Complex::Complex()

    c1.print();
    c2.print();
    c3.print();

    return 0;
}

-----
14.2+6.3j = 6.7824
14.2+0j = 4.2
10+0j = 0
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

Next like any other C++ group of functions the constructors can also be overloaded, which mean that x class can more than one constructor because necessarily being a constructor it has the same name as the class name, so if I want to write two constructor they will necessarily have the same name, but it is permitted to do so as long as they deferred in the number of parameters or types of parameters and so on. All rules of function overloading as we are learnt will apply exactly in the same way in the terms of over loaded constructor. So here we are showing the three constructors for complex which one which takes two paramters re and im both, one which just takes array and one which takes nothing.

And we are using these parameters in terms of the three possible instantiation of the number of parameters deciding which particular form of the constructor that will get invoked. For example, if we look at this instantiation of c 2, when c 2 will gets instantiated then naturally this constructor which has one parameter will get invoked. Whereas, if c 1, when c one is being instantiated then the constructor having two parameters will get invoked, so all rules of over loaded constructors apply in this case.

In this way constructors can be over loaded and we can write construction process in terms of variety of different parameters and their combinations and so on and as we go

forward we will see several more examples of how overloaded constructions becomes important for writing very effective construction mechanism for the different classes that we built.