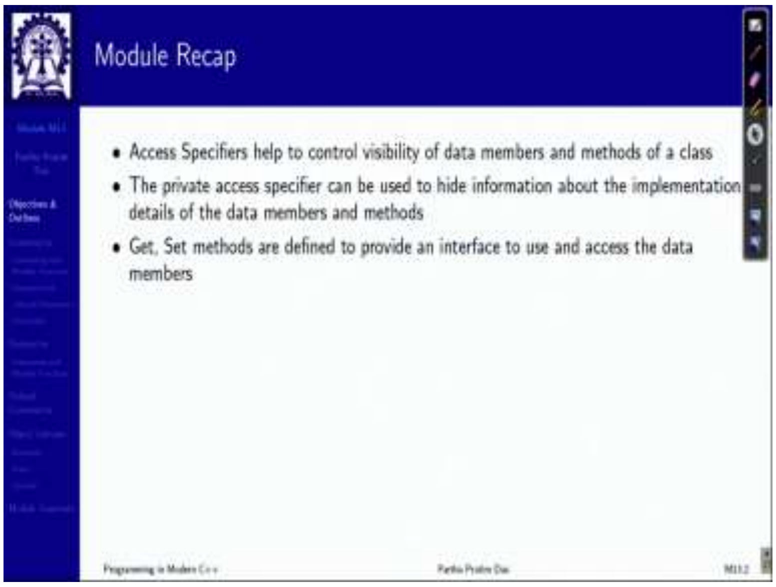
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 13
Constructors, Destructors & Object Lifetime

Welcome to programming in modern C++, we are in week 3, and going to discuss module 13.

(Refer Slide Time: 0:36)

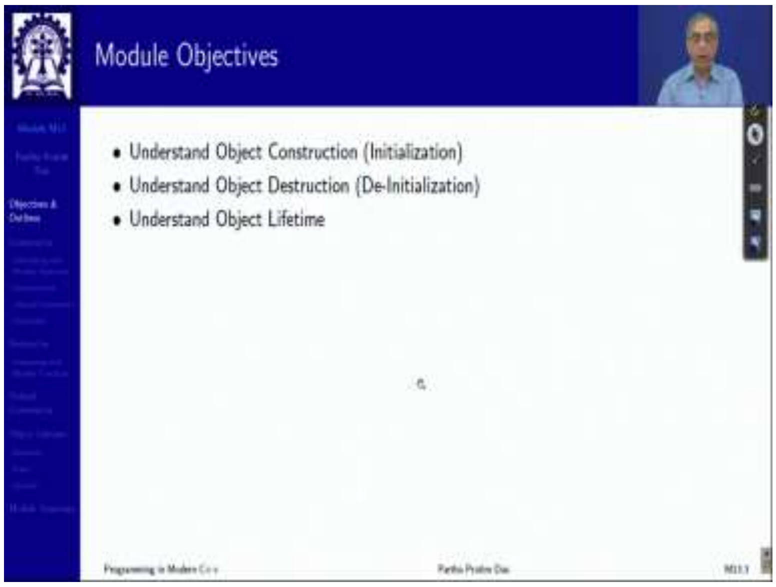


Module Recap

- Access Specifiers help to control visibility of data members and methods of a class
- The private access specifier can be used to hide information about the implementation details of the data members and methods
- Get, Set methods are defined to provide an interface to use and access the data members

In the last module, we have primarily covered the information hiding feature of object oriented programming, how to implement that in C++ classes, by making the implementation, the data members private, and exposing the interface, the member functions that people need to use making them public. And we have talked about get set idiom.

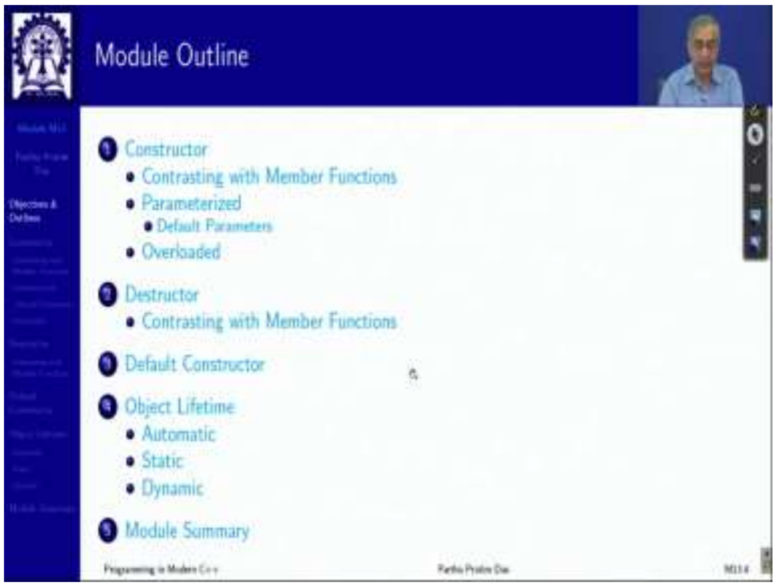
(Refer Slide Time: 1:12)



Module Objectives

- Understand Object Construction (Initialization)
- Understand Object Destruction (De-Initialization)
- Understand Object Lifetime

Programming in Modern C++ Partha Pratim Das M13.3



Module Outline

1. Constructor
   - Contrasting with Member Functions
   - Parameterized
     - Default Parameters
   - Overloaded
2. Destructor
   - Contrasting with Member Functions
3. Default Constructor
4. Object Lifetime
   - Automatic
   - Static
   - Dynamic
5. Module Summary

Programming in Modern C++ Partha Pratim Das M13.4

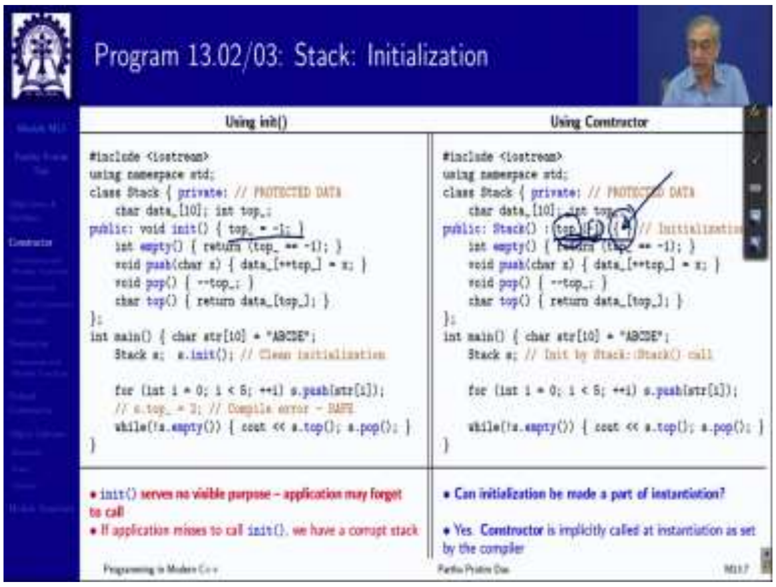
We will now get into the very depth of how do we start the whole process of object creation? And how do we end that life of an object. So, we will talk about the constructor and destructor, which I have already mentioned couple of times in my earlier modules.

(Refer Slide Time: 1:32)

Constructor

Constructor

Programming in Modern C++ Partha Pratim Das M13.5



Program 13.02/03: Stack: Initialization

Using init()	Using Constructor
see below	see below

Using init()

```
#include <iostream>
using namespace std;
class Stack { private: // PROTECTED DATA
    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(); }
}
```

- init() serves no visible purpose – application may forget to call
- If application misses to call init(), we have a corrupt stack

Using Constructor

```
#include <iostream>
using namespace std;
class Stack { private: // PROTECTED DATA
    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(); }
}
```

- Can initialization be made a part of instantiation?
- Yes. Constructor is implicitly called at instantiation as set by the compiler

Programming in Modern C++ Partha Pratim Das M13.7

But now you will actually learn what it does. So, starting with the constructor, so, this is what we had seen that this is what we should not be doing. And the exposed initialization, the risk, we have seen all of that. So, we are in this style of information, hidden design, where the data members are in private, and only the member functions which the interface needs are in public.

Now, there is a now in this a there is a border, the border is that when I say stack s, obviously, I will have an array of data, I will have the top. But what about the value of top that needs an initialization, which I was doing here, directly. So, instead of doing that, which exposes the initialization, I will add another function, I may add another function to the interface called init. And so using that function, the user can initialize a struct.

So, I do a s.init(). Now that this obviously is cleaner than what you had here better. But it still has a couple of very critical problems one is, this has to be called before the first of any of these operations in the stack can be done. User has to remember doing that. Second, it should never be called after that, because the moment to call this is like flushing the whole stack out. So, that again, I mean this with the information hiding.

This does give us a programming way to handle the situation of initialization, but it is not a very preferred way because it can still cause a lot of possible errors and pick things unsafe. So, if we compare this so this is the init we had set and this is what has to be called what you just saw. Now, the C++ allows that you can write a constructor, a constructor is a special function, which has the same name as the class.

So, why do we have that there is not a simple. I can I need to initialize and I need to initialize every object of every class that I create. Now, if I let users give name to the initialization function, then somebody will call it init, somebody will call it start. Somebody will call it initialization. There will be all confusion and how will the compiler know.

So, give it the name which is same as the name of the class because the name of the class is unique. And the important part is, when this is declared, the control is passing this point then this constructor is automatically called, what is automatically called? The compiler knows at that point that unless that s has been declared, s dot push cannot be done, because this is no more, I am not even an object.

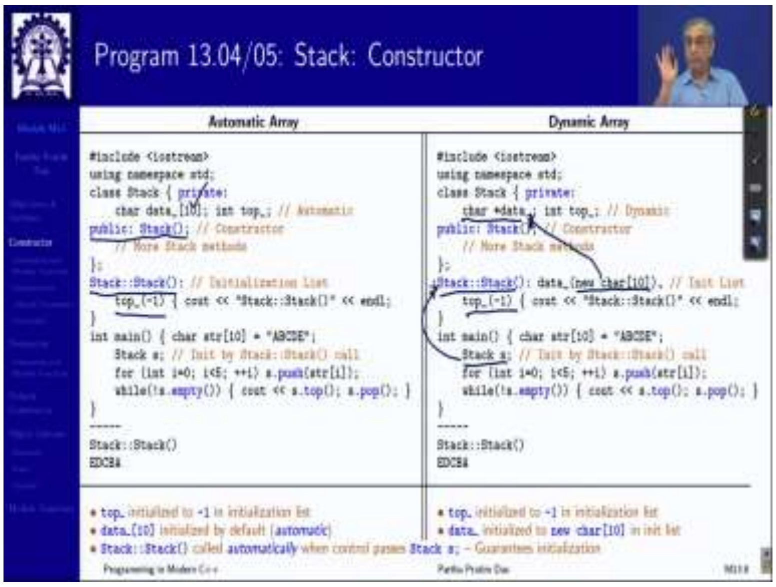
But once this has been declared, from the next point onwards, it can be used. So, at the declaration point it puts, which I we do not get to see puts a call to that constructed. So, if you put all your initialization within the constructor, then it will automatically get constructed, you do not have to do anything, there is no question of forgetting it, there is no question of doing it multiple times.

There is no question of doing it at a wrong point, any of this everything gets sorted. So, that is the basic idea of the constructor, minute in the constructor we have we are using here this was an assignment, we are assigning this. Here what we say we are doing an initialization this actually is a constructor body which has nothing it can have something but in this case, it has nothing.

So, what do you say we put the name of the data member, and we put the value within parenthesis that we want to initialize it with. That way, it gets a lot of advantages, we will see those advantages slowly, more and more, but the biggest advantage is that it is very clearly available at one point and very clearly I cannot actually get into the executing the body of the constructor, because till all data members that need to be initialized or properly initialized, I do not have a proper state of the object.

So, that is the purpose of initializer list. So, we will come to discuss that more in depth later.

(Refer Slide Time: 7:08)



Program 13.04/05: Stack: Constructor

Automatic Array

```
#include <iostream>
using namespace std;
class Stack { private:
    char data_[10]; int top_; // Automatic
public: Stack(); // Constructor
    // More Stack methods
};
Stack::Stack(): // Initialization List
    top_(-1) { cout << "Stack::Stack()" << 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(); }
}
-----
Stack::Stack()
EDCBA
```

Dynamic Array

```
#include <iostream>
using namespace std;
class Stack { private:
    char *data_; int top_; // Dynamic
public: Stack(); // Constructor
    // More Stack methods
};
Stack::Stack(): data_(new char[10]), // Init List
    top_(-1) { cout << "Stack::Stack()" << 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(); }
}
-----
Stack::Stack()
EDCBA
```

- top_ initialized to -1 in initialization list
- data_[10] initialized by default (automatic)

- top_ initialized to -1 in initialization list
- data_ initialized to new char[10] in init list

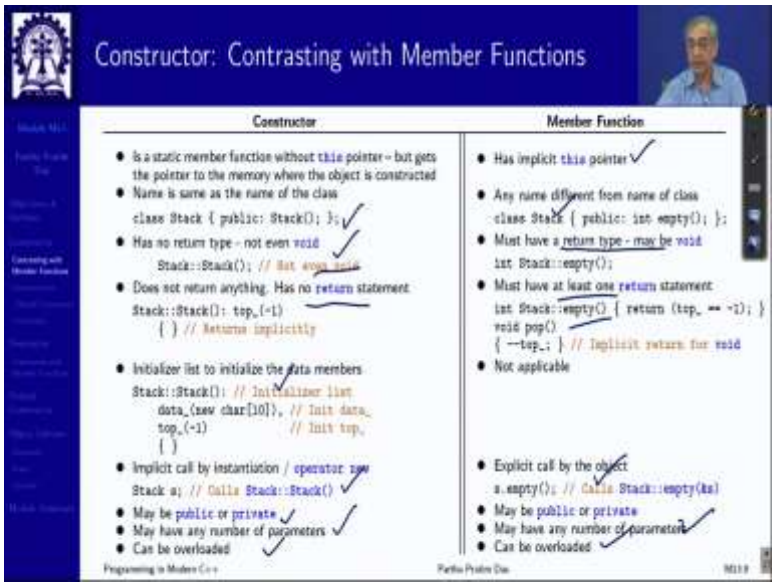
- Stack::Stack() called automatically when control passes Stack s; – Guarantees initialization

Programming in Modern C++ Partha Pratim Das M13.8

So, for example, examples of stack constructor here I have given the constructor outside of the class it is qualified by the class name, I initialize top because this is an automatic array whereas here it is a dynamic array. So, in the constructor, I initialized top, but I also need to dynamically create an array and put that pointer put that pointer to data.

So, whatever internal structure I may have, whether I have a static array, automatic airy, or they have a dynamic array, or I have a linked list, whatever I do, I do not need that to be exposed. All that the compiler knows is a stack s. So, the constructor will have to get hold, that is it and it will do the job makes it a very, very clean solution.

(Refer Slide Time: 8:17)



Constructor: Contrasting with Member Functions

Constructor	Member Function
• Is a static member function without `this` pointer – but gets the pointer to the memory where the object is constructed	• Has implicit `this` pointer
• Name is same as the name of the class `class Stack { public: Stack(); };`	• Any name different from name of class `class Stack { public: int empty(); };`
• Has no return type - not even `void` `Stack::Stack(); // Not even void`	• Must have a return type - may be `void` `int Stack::empty();`

• Does not return anything. Has no `return` statement
`Stack::Stack(): top_(-1)`
`{ } // Returns implicitly`
	• Must have at least one `return` statement
	`int Stack::empty() { return (top_ == -1); }`
	`void pop()`
	`{ --top_; } // Implicit return for void`

• Initializer list to initialize the data members
`Stack::Stack(): // Initializer list`
`data_(new char[10]), // Init data_`
`top_(-1) // Init top_`
`{ }`
	• Not applicable

• Implicit call by instantiation / `operator new` `Stack s; // Calls Stack::Stack()`	• Explicit call by the object `s.empty(); // Calls Stack::empty(&s)`
• May be `public` or `private`	• May be `public` or `private`
• May have any number of parameters	• May have any number of parameters
• Can be overloaded	• Can be overloaded

Programming in Modern C++ Partha Pratim Das M11.9

So, if you just to see what is the difference between a constructor and a member function, because constructor is also a member of the class, because it is specific to that class has that specific name. The first thing to remember is and we will talk about this more later is it is a static member function. In the sense that see, the problem with the constructor in comparison to other member functions is.

All other member functions are invoked after the object has been constructed, all initialization is done. When you are calling the constructor, the object is not constructed, because you are that is the purpose for which you are calling it, the object there is no initialization.

So, but the constructor needs to know the address where the object will recite because without that address, it cannot put values to data members like it is saying top_ is will be initialized with minus 1 how will it put it in which memory so, it has to know that memory when the object will come.

But that memory pointer has still not become at this point. It becomes at this point and only when the object is been born has been created by the constructor. So, the constructor is implicitly independently called and that is what is the important thing and data members do not have that. So, this we have already said these are the same name as the class member functions must have different names. Constructors do not return anything.

Not only that, they do not have a return type even it is not even void because there is it is construction is a process which establishes the object and then the control has to come back.

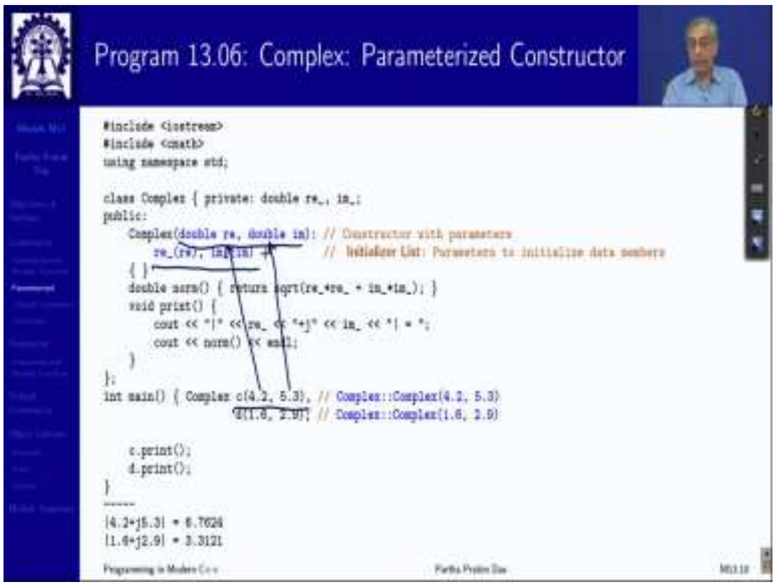
There is nothing else, we cannot come back with anything or there is nothing required to say that there is there is a void return nothing, nothing is returned.

So, there is no return therefore, there is no return statement either the controller returns implicitly whereas, in case of member functions the return may be there it may not may be implicit, it may be explicit whatever. So, initializer list is there and this is specific to the constructor such lists are not available to other functions not member functions, it is implicit by instantiation whereas member functions have to be explicitly called.

Constructor may also be private. And we will I mean that that is another paradigm of information hiding that we will talk of that it might wander you as to what will happen if constructor is private? What am I trying to say? I am trying to say that I do not allow you to construct an object where if I if you do not allow the user to construct an object, then well, then how will the user at all use the service?

Well, there are answers to that question. But that is a that is another very strong paradigm of programming that we will get into. So, typically, in the initial part 2 will always have constructors which are in public, but it can be private also member functions can be public or private, both can have any number of parameters and both can be overloaded. We will see these things slowly.

(Refer Slide Time: 11:46)



Program 13.06: Complex: Parameterized Constructor

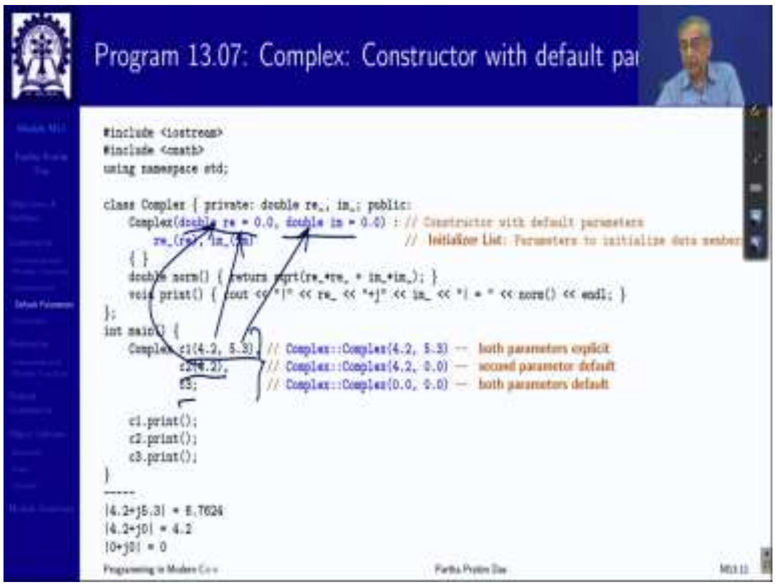
```
#include <iostream>
#include <cmath>
using namespace std;

class Complex { private: double re_, im_;
public:
    Complex(double re, double im): // Constructor with parameters
        re_(re), im_(im)           // Initializer List: Parameters to initialize data members
    { }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() {
        cout << "|" << re_ << "+j" << im_ << "| = ";
        cout << norm() << endl;
    }
};
int main() { Complex c(4.2, 5.3), // Complex::Complex(4.2, 5.3)
                     d(1.6, 2.9); // Complex::Complex(1.6, 2.9)

    c.print();
    d.print();
}
-----
|4.2+j5.3| = 6.7624
|1.6+j2.9| = 3.3121
```

Programming in Modern C++ Partha Pratim Das M13.18

So, this is the basic I just wanted to outline the differences between a common member function and the. So, you can have parameters to the constructor like any other than the initializer list and not having a return type and having a fixed name other than that, it every rule of the function applies there, you can have parameters.

So I can pass the specific values when I write c 4.2, 5.3, then this goes as the first parameter, this goes as a second parameter. In stack I did not need that. So, there was no parameter because in stack the initialization is by known values always whereas I here I need them. So, I can put this parameters.

(Refer Slide Time: 12:37)



Program 13.07: Complex: Constructor with default pa

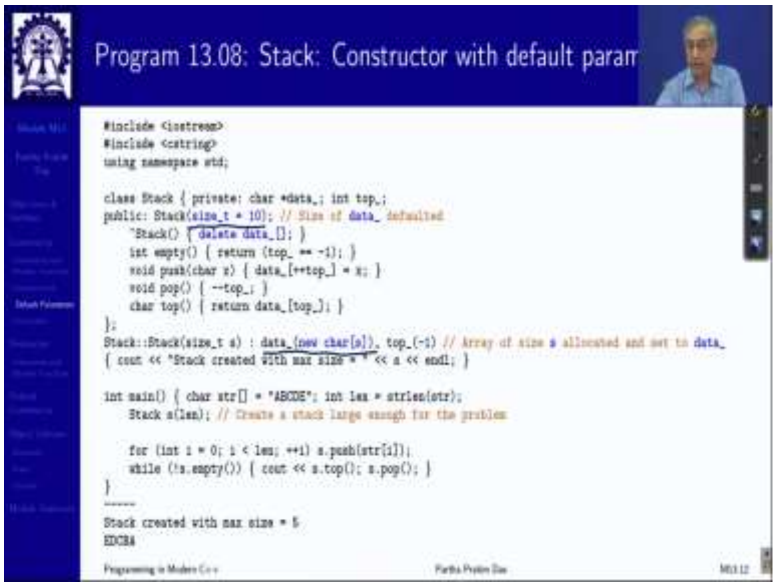
```
#include <iostream>
#include <cmath>
using namespace std;

class Complex { private: double re_, im_; public:
    Complex(double re = 0.0, double im = 0.0) : // Constructor with default parameters
        re_(re), im_(im)                         // Initializer List: Parameters to initialize data member
    { }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() {
    Complex c1(4.2, 5.3), // Complex::Complex(4.2, 5.3) -- both parameters explicit
            c2(4.2),      // Complex::Complex(4.2, 0.0) -- second parameter default
            c3;           // Complex::Complex(0.0, 0.0) -- both parameters default

    c1.print();
    c2.print();
    c3.print();
}
-----
|4.2+j5.3| = 6.7624
|4.2+j0| = 4.2
|0+j0| = 0
```

Programming in Modern C++ Partha Pratim Das M13.11

And the moment I can put parameters and all related stories will come stories of function will come with that data, if I can put parameters and parameters can be defaulted. If parameters can be defaulted, then a constructor can be invoked in multiple ways, this is just like following the rules of function, default parameters in functions.

So, if I have defaulted both the parameters, so I can just instantiate it, instantiate it with only 1 which will be the re or instantiate with both which will be re and im. So, this there is there is nothing new in this over what we have already seen in the functions.

(Refer Slide Time: 13:21)



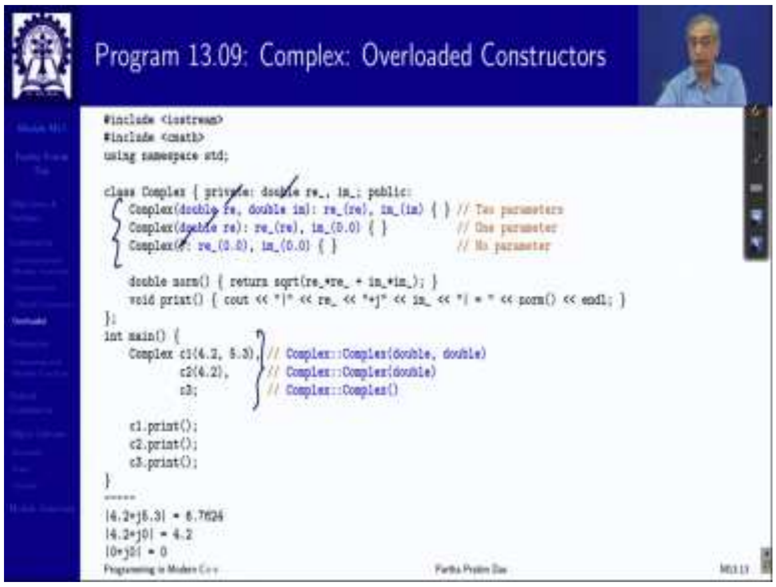
Program 13.08: Stack: Constructor with default param

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

class Stack { private: char *data_; int top_;
public: Stack(size_t = 10); // Size of data_ defaulted
    ~Stack() { delete data_[]; }
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
Stack::Stack(size_t s) : data_(new char[s]), top_(-1) // Array of size s allocated and set to data_
{ cout << "Stack created with max size = " << s << endl; }

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

    for (int i = 0; i < len; ++i) s.push(str[i]);
    while (!s.empty()) { cout << s.top(); s.pop(); }
}
-----
Stack created with max size = 5
EDCBA
```

Programming in Modern C++ Partha Pratim Das M13.12

Now, here is a stack with a default parameter, which is the initial size of the stack that I provide and accordingly in the constructor, the allocation is done for that size. So, these are different ways you can use it the parameters also.

(Refer Slide Time: 13:46)



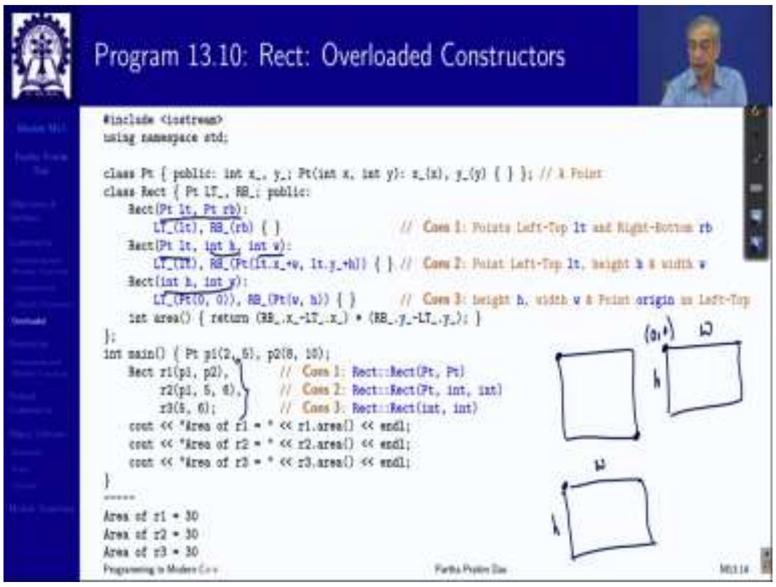
Program 13.09: Complex: Overloaded Constructors

```
#include <iostream>
#include <cmath>
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, 5.3), // Complex::Complex(double, double)
            c2(4.2),      // Complex::Complex(double)
            c3;           // Complex::Complex()

    c1.print();
    c2.print();
    c3.print();
}
-----
|4.2+j5.3| = 6.7624
|4.2+j0| = 4.2
|0+j0| = 0
```

Programming in Modern C++ Partha Pratim Das M13.13

Now, once you have the default parameters, you can understand that like in functions, you can have overloading. So, it is not necessary that a constructor will have only will be unique constructor, there could be multiple constructors for example, here I have shown 3 constructors, one which takes 2 parameters, one, one and one, two, it does not take any accordingly we have 3 possible invocations or instantiations of the objects. So, all overloading rules all that as we had seen earlier will apply.

(Refer Slide Time: 14:23)



Program 13.10: Rect: Overloaded Constructors

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

class Pt { public: int x_, y_; Pt(int x, int y): x_(x), y_(y) { } }; // A Point
class Rect { Pt LT_, RB_; public:
    Rect(Pt lt, Pt rb):
        LT_(lt), RB_(rb) { }                     // Cons 1: Points Left-Top lt and Right-Bottom rb
    Rect(Pt lt, int h, int w):
        LT_(lt), RB_(Pt(lt.x_+w, lt.y_+h)) { } // Cons 2: Point Left-Top lt, height h & width w
    Rect(int h, int w):
        LT_(Pt(0, 0)), RB_(Pt(w, h)) { }         // Cons 3: height h, width w & Point origin as Left-Top
    int area() { return (RB_.x_-LT_.x_) * (RB_.y_-LT_.y_); }
};
int main() { Pt p1(2, 5), p2(8, 10);
    Rect r1(p1, p2),        // Cons 1: Rect::Rect(Pt, Pt)
         r2(p1, 5, 6),      // Cons 2: Rect::Rect(Pt, int, int)
         r3(5, 6);          // Cons 3: Rect::Rect(int, int)
    cout << "Area of r1 = " << r1.area() << endl;
    cout << "Area of r2 = " << r2.area() << endl;
    cout << "Area of r3 = " << r3.area() << endl;
}
-----
Area of r1 = 30
Area of r2 = 30
Area of r3 = 30
```

Programming in Modern C++ Partha Pratim Das M13.14

So, these are overloaded constructors for rectangles. So, here what we are seeing, we are passing 2 points. This becomes a rectangle. Here what I am saying? I am passing a point and a height and a width. So, this will also become a rectangle, a very different way of passing rectangle. Here, I am just passing height and width, height and weight, not the left up. So, I have to assume a default. So, I am assuming that leftovers.

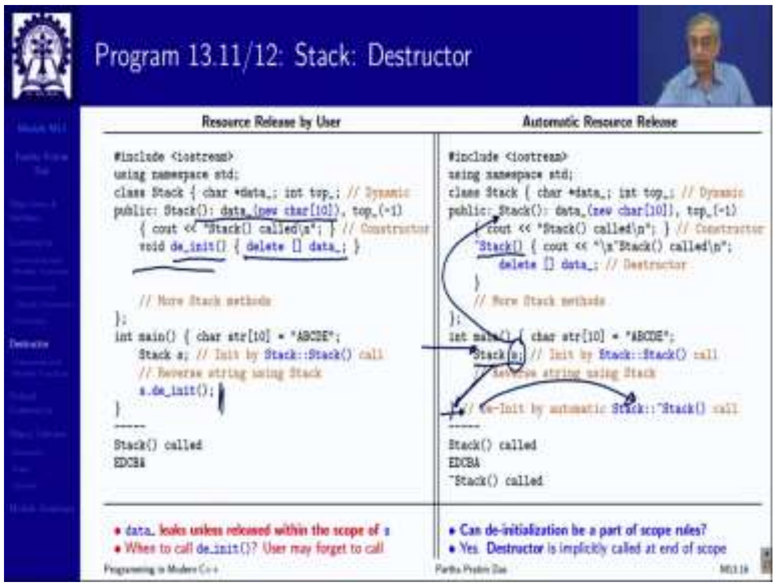
So, there are different ways I can construct a rectangle and there are different constructors for that, I have to make sure that they are distinguishable by the number and types of parameters that same issue of overload resolution will come in here. And here I have different invocations of these constructors for building the rectangle objects.

(Refer Slide Time: 15:37)

Destructor

Destructor

Programming in Modern C++ Partha Pratim Das M13.15



Program 13.11/12: Stack: Destructor

Resource Release by User

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

    // More Stack methods
};
int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call
    // Reverse string using Stack
    s.de_init();
}
-----
Stack() called
EDCBA
```

- data_ leaks unless released within the scope of s
- When to call de_init()? User may forget to call

Automatic Resource Release

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

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

Programming in Modern C++ Partha Pratim Das M13.16

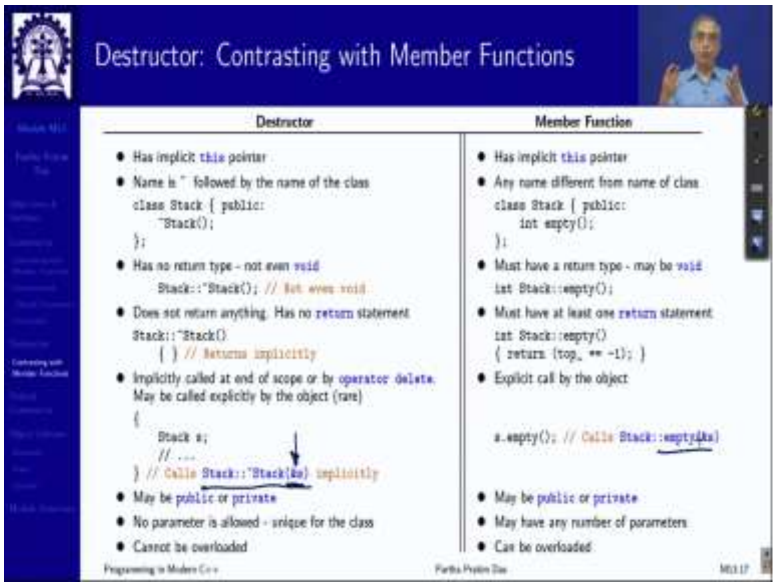
So, that is about the constructors. Now, let us move on to the destructors. Destructor does the other thing, constructors constructed the object. So, if you think in terms of in a C style, then you will say okay, I had constructed, so, this allocation was done. So, I need to release that. So, I have a deinitializer, which should release that memory and that should be called at the end. Again, the issues are these are the user will forget.

And if they issue if the user forgets the data leaks or if the user calls it twice, then you may have different kinds of pointer referencing issues and so on. So, what the C++ provides? It provides a destructor which also is the class name preceded by the tilde symbol, that is a peculiar way of naming which you do not use anywhere else, and that will get called at an appropriate time. So, you have declared the stack here.

So, here the constructor has got called. Now, the compiler knows the scope up to which the s can be accessed. Before this s does not exist. So, there is no object either. After this point, after the control passes this point, you go out of the scope, you go out of this function, so naturally there is no s. So, it is only up to that point.

At this point, it will invoke the destructor automatically, absolutely clean solution, you have to do nothing. You have to do nothing other than just declaring that you are using a stack s and then go and use it the compiler automatically constructed, compiler will automatically destruct, beautiful solutions.

(Refer Slide Time: 17:40)



Destructor: Contrasting with Member Functions

Destructor

- Has implicit this pointer
- Name is ~ followed by the name of the class

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

- Has no return type - not even void

```
    Stack::~Stack(); // Not even void
```

- Does not return anything. Has no return statement

```
Stack::~Stack()
    { } // Returns implicitly
```

- Implicitly called at end of scope or by operator delete. May be called explicitly by the object (rare)

```
{
    Stack s;
    // ...
} // Calls Stack::~Stack(&s) implicitly
```

- May be public or private
- No parameter is allowed - unique for the class
- Cannot be overloaded

Member Function

- Has implicit this pointer
- Any name different from name of class

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

- Must have a return type - may be void

```
int Stack::empty();
```

- Must have at least one return statement

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

- Explicit call by the object

```
s.empty(); // Calls Stack::empty(&s)
```

- May be public or private
- May have any number of parameters
- Can be overloaded

Programming in Modern C++ Partha Pratim Das M13.17

So, if you just want to compare with the main common member functions, then both destructor has a this pointer because the object is existing. The name is as I said, till day followed by the class name for member functions, you do not have that. Like constructor destructor also does not have a return type neither not even void, because it is because you will destroy after you have destroyed, the object is gone.

So, what will you return and there is no concept of a data whereas all those are standard for member functions. And it is implicitly called at the end of the scope or by operator delete rules. So, we will see that kind of scenario soon. And so, you can make a note that since the destructor will have to actually tell which object it is destroying.

Constructor did not have to do that. This constructor did not have an object he just needed a memory which compiler makes a mechanism to pass that memory pointer to the constructor but when you are destroying or destroying the object say s. So, you have to pass that this pointer of s which by the implicit mechanism the compiler will pass exactly like it passes through the member functions.

Again destructors could be public or private, it does not allow parameters for the simple reason that what will a parameter mean? Parameter mean that you are trying to say something to be done. But when this is being called even nobody is calling this no user code is calling this it is being called implicitly and at the end.

So, in the implicit call, how do you pass a parameter? Like in the constructor it was possible because it is it is automatically called but it is explicit. You know when it will be called. In case of distracted it will be called, but there is no call site that you can see, it is not visible, end of scope compiler has put this in it may have put in hundreds of such destructors.

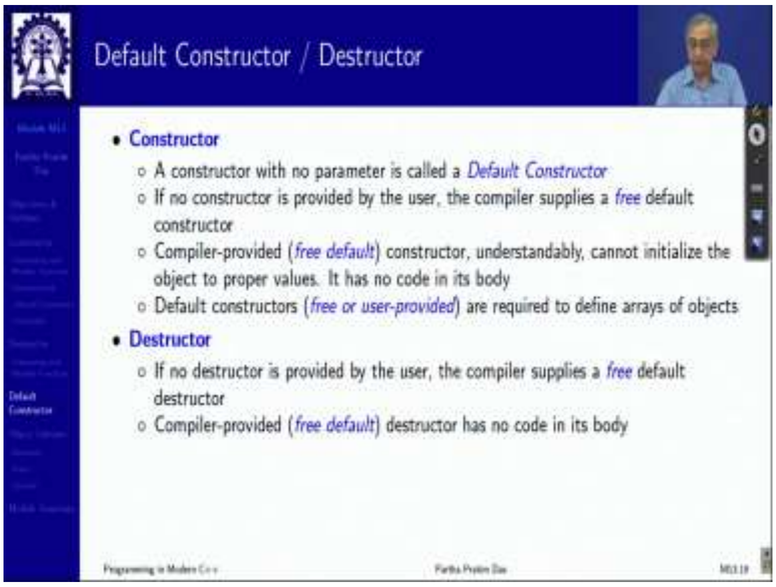
So, there is no sense neither a mechanism neither possible syntax to pass parameters to it. So, the destructors will never allow parameters. So, and obviously, it cannot be overloaded, you cannot have 2 different destructors because again how will the compiler know which one to call. If you overload you need parameters. So, you cannot have parameters it cannot overload.

(Refer Slide Time: 20:42)

Default Constructor

Default Constructor

Programming in Modern C++ Partha Pratim Das M13.18



Default Constructor / Destructor

- **Constructor**
  - A constructor with no parameter is called a *Default Constructor*
  - If no constructor is provided by the user, the compiler supplies a *free* default constructor
  - Compiler-provided (*free default*) constructor, understandably, cannot initialize the object to proper values. It has no code in its body
  - Default constructors (*free or user-provided*) are required to define arrays of objects
- **Destructor**
  - If no destructor is provided by the user, the compiler supplies a *free* default destructor
  - Compiler-provided (*free default*) destructor has no code in its body

Programming in Modern C++ Partha Pratim Das M13.19

So, these are some of the basics about destructors. There is a special type of constructor; a constructor without any parameter is called a default constructor. Now, so, actually, if you define a constructor and default all its parameter values, then also you are actually having a default constructor because it is possible to call it without any parameter. But you can choose not to have a default constructor.

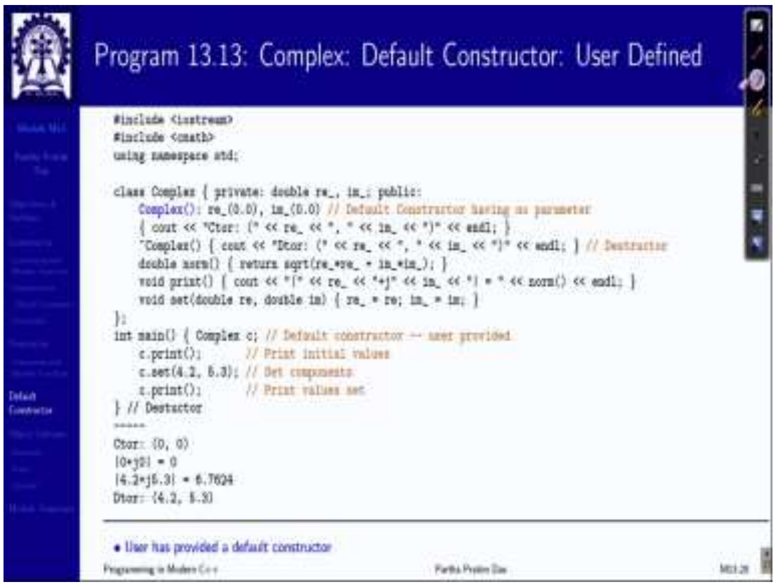
Now, what happens is when you do not, if you write a class and you do not give a constructor? Now, there could have been 2 choices one is this could have been made not allowed at all. So, that the code will not compile unless you give a constructor but that has not been done because there are several small situations where the constructor is so trivial that you will not like to take the pain of writing it.

So, what does the system do is, if you do not provide any constructor, the compiler will supply a default constructor, a constructor without any parameters, without any code, without any initialization, but it will provide that placeholder function. And once you write a constructor, which is either I should say default or is non default that it has parameters only, once you write that the compiler will not provide it.

Similarly, for the destructor. If you write one, then the destructor will not be provided by the compiler. But if you do not, then the compiler will provide it instructed, it is very common that we do not write destructor because if there is no dynamic allocation, usually, then there is no need to specifically do anything when the object is going out of scope. The automatic scoping will take care of it.

So, but so the compiler will in those in every such case, compiler will provide default destructor. So, you can whether or not you write a constructor whether or not to write a destructor. This guarantees that every class when it is after its compilation, has a destructor, has a constructor, at least 1 and has a unique destructor. So, that is a.

(Refer Slide Time: 23:25)

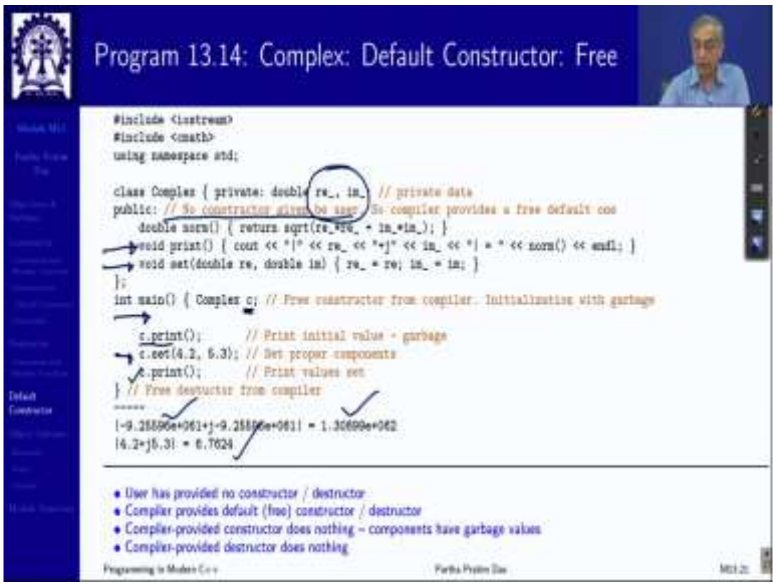


Program 13.13: Complex: Default Constructor: User Defined

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

class Complex { private: double re_, im_; public:
    Complex(): re_(0.0), im_(0.0) // Default Constructor having no parameter
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
    ~Complex() { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } // Destructor
    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
} // Destuctor
-----
Ctor: (0, 0)
|0+j0| = 0
|4.2+j5.3| = 6.7624
Dtor: (4.2, 5.3)
```

- User has provided a default constructor

Programming in Modern C++ Partha Pratim Das M13.20



Program 13.14: Complex: Default Constructor: Free

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

class Complex { private: double re_, im_; // private data
public: // No constructor given be user. So compiler provides a free default one
    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; // Free constructor from compiler. Initialization with garbage
    c.print();        // Print initial value - garbage
    c.set(4.2, 5.3); // Set proper components
    c.print();        // Print values set
} // Free destuctor from compiler
-----
|-9.25596e+061+j-9.25596e+061| = 1.30899e+062
|4.2+j5.3| = 6.7624
```

- User has provided no constructor / destructor
- Compiler provides default (free) constructor / destructor
- Compiler-provided constructor does nothing – components have garbage values
- Compiler-provided destructor does nothing

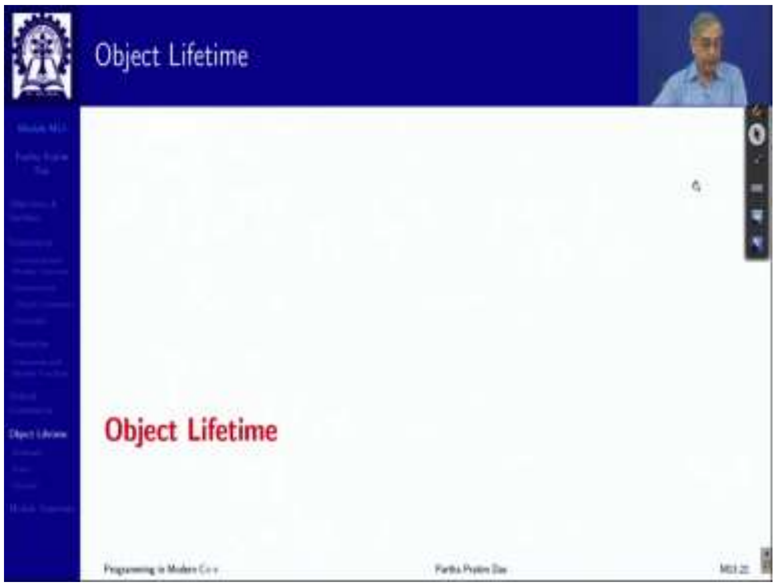
Programming in Modern C++ Partha Pratim Das M13.21

Now, so this is the example of a default constructor by the user. I have explicitly written that. So, the construction is done here by default values. Now, let us see if I do not provide it. Here I have not no constructor have given. Since I have not given a constructor, what will happen at this point? The compiler will give a default constructor.

Now that default constructor does not know what to put here. As I said, there is no initializer list there is nothing in the function, it is just a blank function with no parameters. So, after the object has been constructed, just for experiment, if you try to print to see what kind of things it has, if you do that print, you will find something like this, garbage values, of course.

So, then you will have to then use your member function to set it to proper values. Doing this set on re and im and then do the print you will get the proper one. So, remember this that the free constructor that you get, may not always often satisfy what you what your actual needs are?

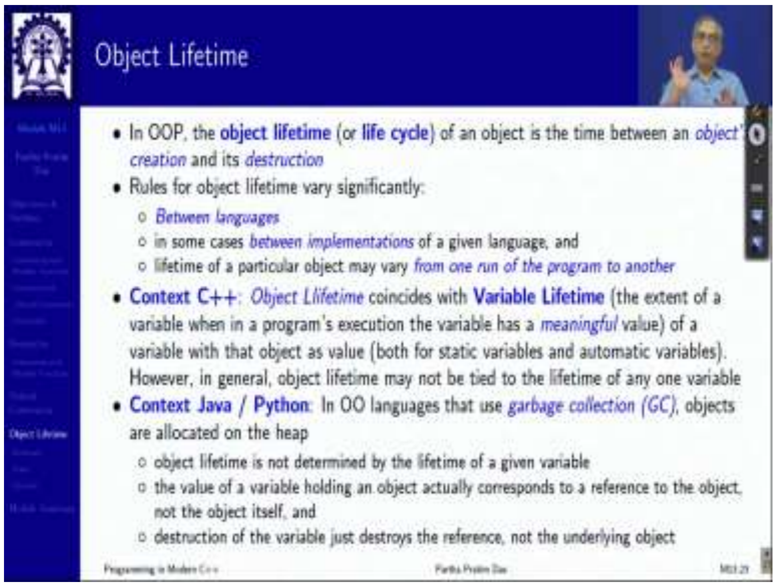
(Refer Slide Time: 24:57)



Object Lifetime

Object Lifetime

Programming in Modern C++ Partha Pratim Das M03.22



Object Lifetime

- In OOP, the **object lifetime** (or **life cycle**) of an object is the time between an *object's creation* and its *destruction*
- Rules for object lifetime vary significantly:
  - *Between languages*
  - in some cases *between implementations* of a given language, and
  - lifetime of a particular object may vary *from one run of the program to another*
- **Context C++:** *Object Llifetime* coincides with **Variable Lifetime** (the extent of a variable when in a program's execution the variable has a *meaningful* value) of a variable with that object as value (both for static variables and automatic variables). However, in general, object lifetime may not be tied to the lifetime of any one variable
- **Context Java / Python:** In OO languages that use *garbage collection (GC)*, objects are allocated on the heap
  - object lifetime is not determined by the lifetime of a given variable
  - the value of a variable holding an object actually corresponds to a reference to the object, not the object itself, and
  - destruction of the variable just destroys the reference, not the underlying object

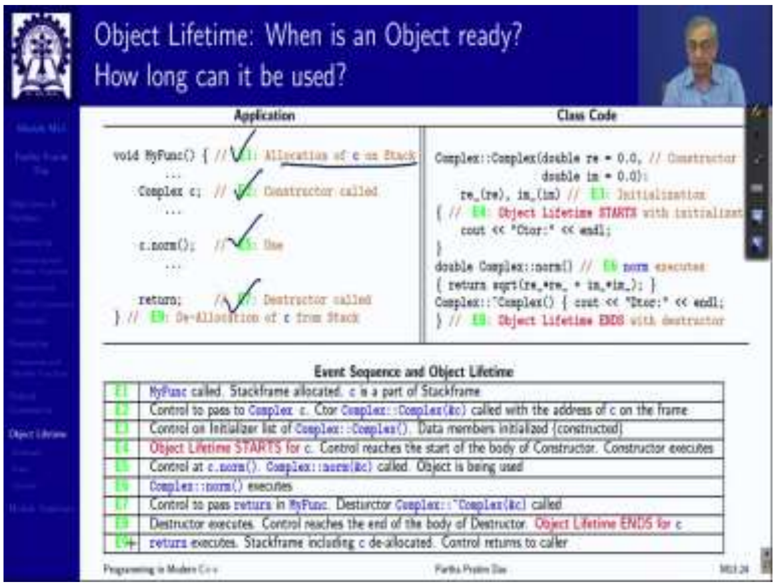
Programming in Modern C++ Partha Pratim Das M03.23

Now with this, let me get into that basic lifetime issues that is it is clear that an object does not exist in the system all the time. The object exists between object creation and object destruction. Now, naturally this is this notion is not specific to C++, it has in several languages this concept is there it is called object lifetime or life cycle, but the meaning of that may vary from time to time.

For example, in C++ the object lifetime primarily denotes I mean primarily relates to coincides with that variable lifetime, the variable is in the scope, the object is there. When the variable is no more accessible object is not accessible. But, that may not be true again if you have dynamically created objects, because the variable which created that object held that pointer to that object may go out of scope, but you may have passed that pointer to someone else. So, you continue to have that object.

So, the lifetime will be as the lifetime is from object creation to object destruction. Similarly, other languages like Java, Python have garbage collectors. So, the object goes out of scope, it is notionally not there, but it actually exists till the garbage collected removes it.

(Refer Slide Time: 26:40)



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: Constructor called
    ...
    c.norm();   // E5: Use
    ...
    return;     // E7: Destructor called
} // E9: De-Allocation of c from Stack
```

Class Code

```
Complex::Complex(double re = 0.0, // Constructor
                 double im = 0.0):
    re_(re), im_(im) // E3: Initialization
{ // E4: Object Lifetime STARTS with initializat
    cout << "Ctor:" << endl;
}
double Complex::norm() // E6 norm executes
{ return sqrt(re_*re_ + im_*im_); }
Complex::~Complex() { cout << "Dtor:" << endl;
} // E8: Object Lifetime ENDS with destructor
```

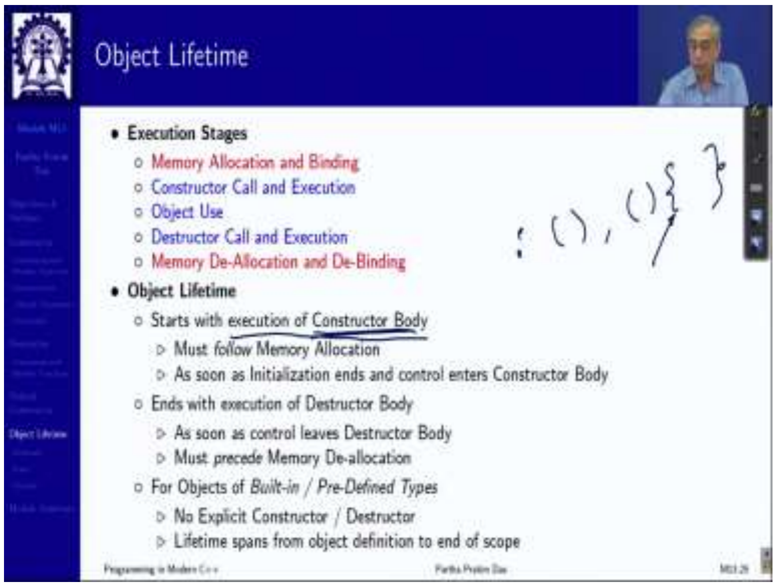
Event Sequence and Object Lifetime

E1	MyFunc called. Stackframe allocated. c is a part of Stackframe
E2	Control to pass to 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 Constructor. Constructor executes
E5	Control at c.norm(). Complex::norm(&c) called. Object is being used
E6	Complex::norm() executes
E7	Control to pass return in MyFunc. Desturctor Complex::~Complex(&c) called
E8	Destructor executes. Control reaches the end of the body of Destructor. Object Lifetime ENDS for c
E9+	return executes. Stackframe including c de-allocated. Control returns to caller

Programming in Modern C++ Partha Pratim Das M13.26

So, based on that there are so, in the C++ it typically is certainly you have allocation, the construction, allocation of the stack frame, the construction, the use and destruction. So, it will I have marked by E1 to through E9 in this code as to exactly how the object is taking part in the flow. So, I would request you to carefully study each one of these from this chart and make a clear understanding of what the objects are going through.

(Refer Slide Time: 27:15)



Object Lifetime

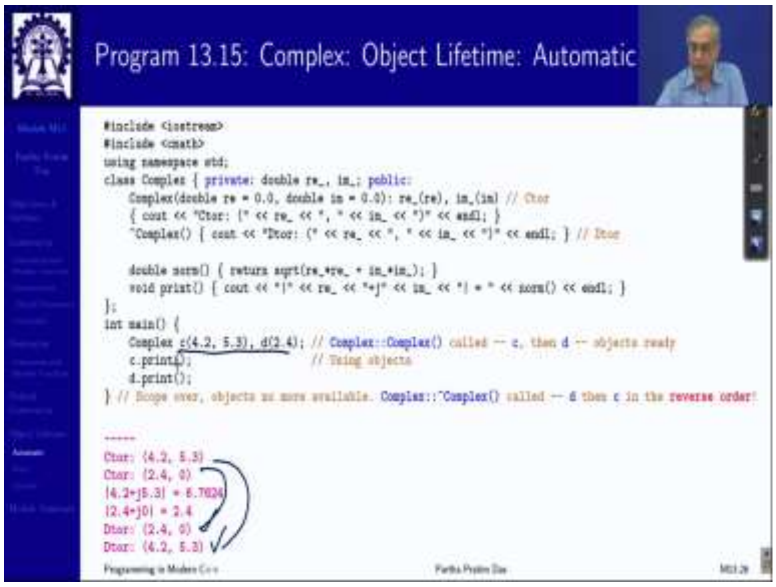
- **Execution Stages**
  - Memory Allocation and Binding
  - Constructor Call and Execution
  - Object Use
  - Destructor Call and Execution
  - Memory De-Allocation and De-Binding
- **Object Lifetime**
  - Starts with execution of Constructor Body
    - Must *follow* Memory Allocation
    - As soon as Initialization ends and control enters Constructor Body
  - 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

So, at the top level, first the memory has to get allocated and bound for this object. Then the constructor has to get called and constructor will execute. Then the object is ready object will be used then the destructor will be called and executed. So, destruction part is done then it has to be de binding and de allocation for the whole thing.

So, the basic construction object lifetime considers is the it starts with the execution of the constructor body following the memory allocation but constructed body mind. The initializer when as long as the control is in the initializer list, it is not considered that the object lifetime has started for the simple reason that if you have that lifetime there you still do not have a valid state. So, say your error has been allocated, but in for a stack but top has not been initialized. So, the object is actually not there.

So, after the initialization list, so, you have this after the initialization list, when you are at the beginning of the constructor body is when your object construction is started. And your object destruction will be considered to have happened when as soon as the control leaves the destructor body. So, this is the simple notion of the lifetime.

(Refer Slide Time: 28:59)



Program 13.15: Complex: Object Lifetime: Automatic

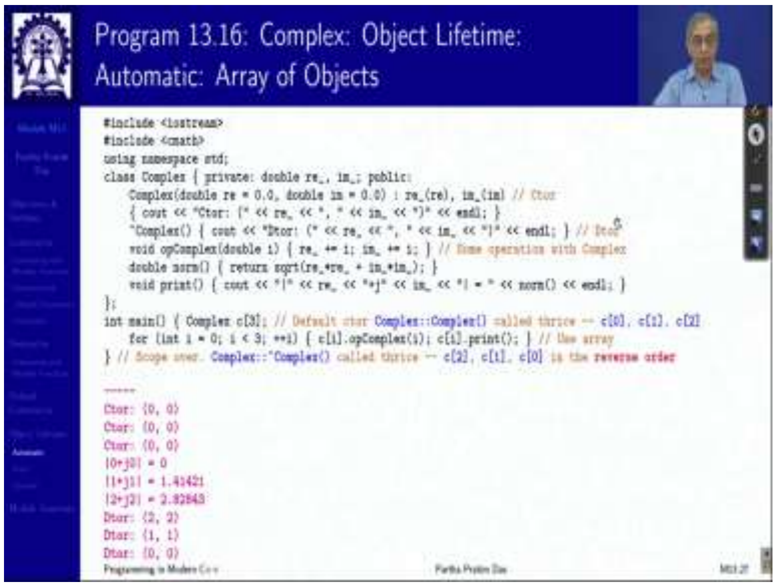
```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { private: double re_, im_; public:
    Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
    ~Complex() { cout << "Dtor: (" << re_ << ", " <<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

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

-----
Ctor: (4.2, 5.3)
Ctor: (2.4, 0)
|4.2+j5.3| = 6.7624
|2.4+j0| = 2.4
Dtor: (2.4, 0)
Dtor: (4.2, 5.3)
```

Programming in Modern C++ Partha Pratim Das M13.26

In the following slides I have provided the examples for what happens in case of automatic variables like in here these are automatic. So, you are this you have already seen. Naturally it is constructed here and destructed and you remember the destruction will always happen in the reverse order. If there are 2, it is constructed as cd and it is destructed as dc.

So, you can think of it as a as it as a stack. So, it is last constructed first destructed is always the mechanism. That is all through C++.

(Refer Slide Time: 29:42)



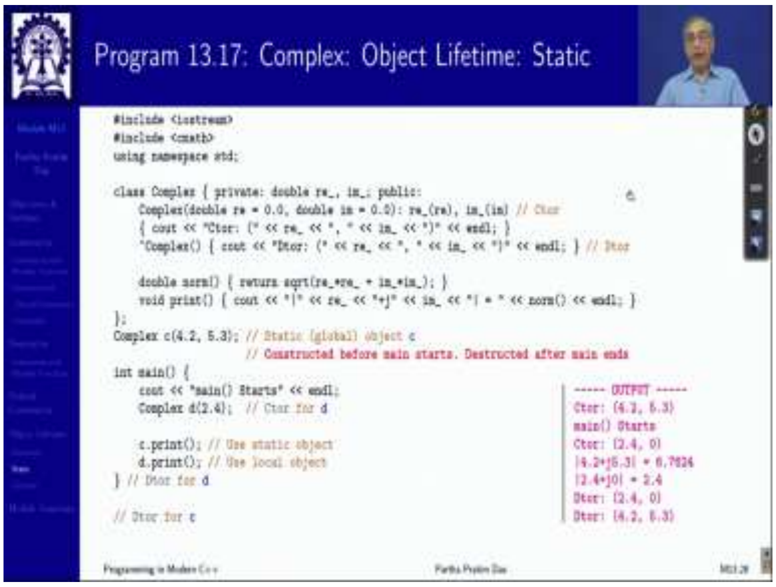
Program 13.16: Complex: Object Lifetime: Automatic: Array of Objects

```
#include <iostream>
#include <cmath>
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() { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } // Dtor
    void opComplex(double i) { re_ += i; im_ += i; } // Some 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
} // Scope over. Complex::~Complex() called thrice -- c[2], c[1], c[0] in the 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)
```

Programming in Modern C++ Partha Pratim Das M13.27

So, it could be automatic like this. It could be an array of objects. If you have an array of 3 objects and there are 3 construction calls. And one point to note is if you want to construct an array of objects, you must provide a default constructor, so 3 bad constructors called c 0, c 1, c 2. So, it will destruction will be c 2, c 1, c 0.

(Refer Slide Time: 30:10)



Program 13.17: Complex: Object Lifetime: Static

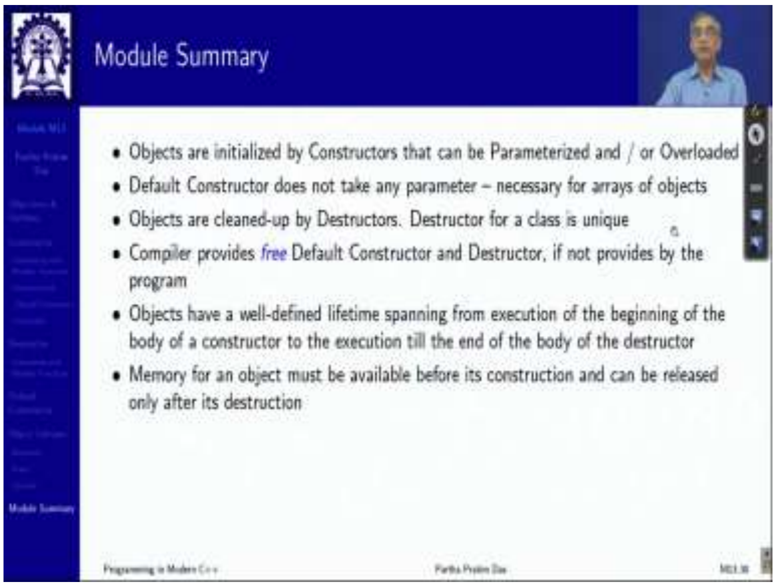
```
#include <iostream>
#include <cmath>
using namespace std;

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() { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } // Dtor

    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 c
                     // 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
} // Dtor for d

// Dtor for c
```

```
----- OUTPUT -----
Ctor: (4.2, 5.3)
main() Starts
Ctor: (2.4, 0)
|4.2+j5.3| = 6.7624
|2.4+j0| = 2.4
Dtor: (2.4, 0)
Dtor: (4.2, 5.3)
```

Programming in Modern C++ Partha Pratim Das M13.29

If it is static, then it will be constructed before pain starts this is something which is possibly new to you. But remember that static all static variables have to get initialized before main start so, now there is a construction so the static object will get constructed. So, I have given the programs in a way and the output will clearly tell you where the lifetime is going. And finally, if you have dynamic created objects, then it is up to your control as to when you after, when you construct by new and when you distract by delete.

(Refer Slide Time: 30:48)



Module Summary

- Objects are initialized by Constructors that can be Parameterized and / or Overloaded
- Default Constructor does not take any parameter – necessary for 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

Programming in Modern C++ Partha Pratim Das M13.30

So, please trace this codes carefully to understand the more about the object lifetime. We have talked about the constructor and destructor, the 2 basic most important functions in a class. So, I hope you have understood this and you will need to revise on this very frequently. Thank you very much for your attention and we will meet in the next module.