

Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture – 14
Copy Constructor and Copy
Assignment Constructor

Welcome to programming in modern C++, we are in week three. And we will be discussing module 14.

(Refer Slide Time: 00:32)

Module Recap

- 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 M14.2

In the last module, we have taken a look into the constructor and destructor that the basic process of creating an object and releasing it with taking care of all dynamic data and so on, to have a very well defined lifecycle. And we have shown examples of the lifetime of variables that are objects, that are automatic, that are static and that are dynamic.

(Refer Slide Time: 01:04)

The slide is titled "Module Objectives" and features a dark blue header with a logo on the left and a small video inset of the speaker on the right. The main content area is white and contains a bulleted list of four objectives. A vertical navigation menu is visible on the left side of the slide.

- More on Object Lifetime
- Understand Copy Construction
- Understand Copy Assignment Operator
- Understand Shallow and Deep Copy

At the bottom of the slide, there is a footer with the text "Programming in Modern C++", "Partha Pratim Das", and "M14.3".

In the, in this module, we will take a couple of more examples of object lifetime, which you should practice. And we will primarily deal with the issue of copy, how do you make copies of objects? And what does that mean, in C++ and why is it important.

(Refer Slide Time: 01:24)

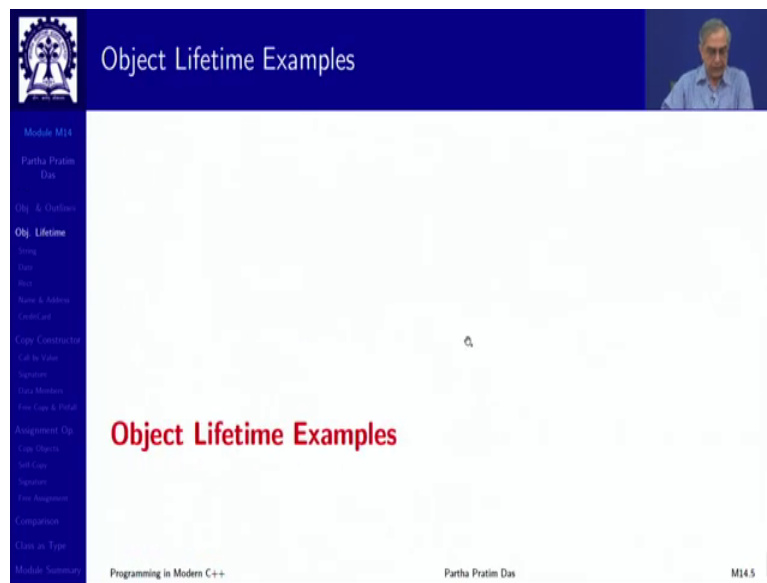
The slide is titled "Module Outline" and features a dark blue header with a logo on the left and a small video inset of the speaker on the right. The main content area is white and contains a numbered list of six items. The first three items have sub-bullets. A vertical navigation menu is visible on the left side of the slide.

- 1 Object Lifetime Examples
 - String
 - Date: Practice
 - Rect: Practice
 - Name & Address: Practice
 - CreditCard: Practice
- 2 Copy Constructor
 - Call by Value
 - Signature
 - Data Members
 - Free Copy Constructor and Pitfalls
- 3 Copy Assignment Operator
 - Copy Objects
 - Self-Copy
 - Signature
 - Free Assignment Operator
- 4 Comparison of Copy Constructor and Copy Assignment Operator
- 5 Class as a Data-type
- 6 Module Summary

At the bottom of the slide, there is a footer with the text "Programming in Modern C++", "Partha Pratim Das", and "M14.4".

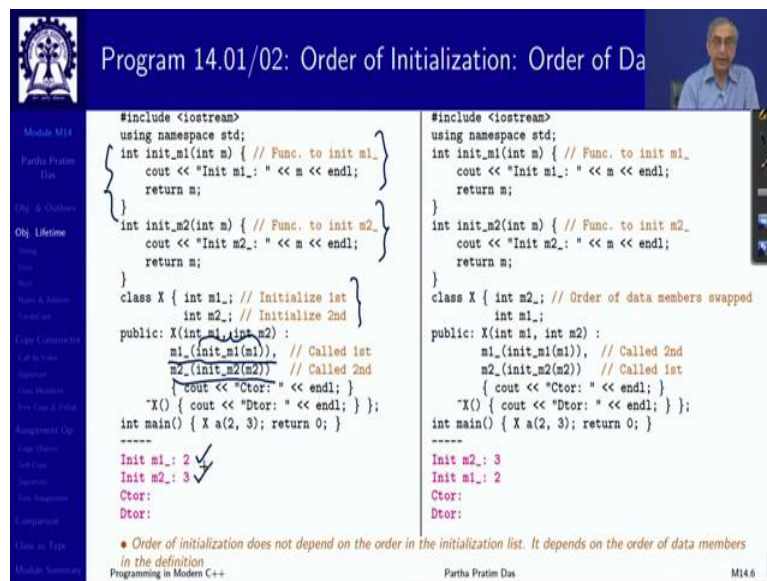
As you will see, in several slides in this presentation, the title is prefixed with practice in blue. And those slides are not what I will discuss in the presentation, but those are included so that you can study them in depth and possibly try them out to have a better understanding of the respective issues.

(Refer Slide Time: 01:49)



Here is the module outline as every time.

(Refer Slide Time: 01:51)



So, in terms of object lifetime, we want to answer a very specific question in this slide as to what is the order in which the different data members of a class is constructed, when the object of that class is constructed? So, you have an object and you have different data members. So, to construct the object, you need to construct each and every data member. Those may also be a user defined types, so they will have data members.

So, the process goes recursively in that manner. But with a simple example, we try to illustrate what is the order in which they are constructed, is it by the ordering which you write

them in the initializer list, or is it by the order in which you list them in the class definition. So, to start with, we have a sample class containing two integer data members.

And to be able to understand as to which 1 is actually getting constructed, we have created two insertable dummy functions. So, instead of directly copying the value, we will pass the value that we want to initialize into this into data member to the init function, corresponding init function. So, that I can print the message and know in which order they are happening.

So, these are the two data members and I have that in the initialization list, I put the initialization of m1 first. And I initialize it with the value returned by this function that is, that is, this is just an instrument to tell us that we get to see in in which order they are happening, because the print messages will happen in that way. So, I do that and when I see I see that init m1 has happened first and init m2 is happening second.

Now, the question is, is it because is init m1 initialization of m1 construction of m1 is happening first is it because m1 is earlier than m2 in the list of data members in the class? Or is it because in the initialization list, it has been initialized first? So, to test that, let us change the let us change the order of data member list in the class.

I now list m2 first and m1 next and I do not change the order they are ordered in the initialization list. Now as I do this, I find that in the invocations are in the same way using the functions. Now I find that init m2 is happening first. So, this clearly tells us the init of m2 is actually listed later in the initializer, but it is happening first because it is the first data member it occurs before m1.

So, this is this simple program tells us that the data members will always be constructed in the order in which they are listed in the class and not in the order in which they are given in the initializer list, which could be anything. And this is very important to understand because a particular class may have multiple different constructors as we have seen.

So, they may list the data members, they may need to put the data members in any order whatsoever they want to. So, if that were detected, that were directing the initialization order, then there would be ambiguity. So, when it is in terms of the order in which they are listed in the class, obviously, there is no ambiguity because there is only one list, irrespective of how many constructors you may have for the class. So, let us see the consequence of this.

(Refer Slide Time: 05:53)

Program 14.03/04: A Simple String Class

C Style	C++ Style
<pre>#include <iostream> #include <cstring> #include <cstdlib> using namespace std; struct String { char *str_; // Container size_t len_; // Length }; void print(const String& s) { cout << s.str_ << " : " << s.len_ << endl; } int main() { String s; // Init data members s.str_ = strdup("Partha"); s.len_ = strlen(s.str_); print(s); free(s.str_); } ----- Partha: 6</pre>	<pre>#include <iostream> #include <cstring> #include <cstdlib> using namespace std; class String { char *str_; // Container size_t len_; // Length public: String(char *s) : str_(strdup(s)), // Uses malloc len_(strlen(str_)) { cout << "ctor: "; print(); } ~String() { cout << "dctor: "; print(); free(str_); // To match malloc() in strdup() } void print() { cout << "(" << str_ << " : " << len_ << ")" << endl; } size_t len() { return len_; } }; int main() { String s = "Partha"; // Ctor called s.print(); } ----- ctor: (Partha: 6) (Partha: 6) dctor: (Partha: 6) len_. What if we swap them?</pre>

● Note the order of initialization between `str_` and `len_`. What if we swap them?

And for that we pick up a simple string class having a `char*` pointer for the containing the string and another variable `len` to keep the length of the string. And this is the order in which they are given. And there is a print here. So, I first in, in the C style I first do a copy of the given string, put the length of that string.

And finally, I free up the space created by `strdup`. In the C++, actually, I put them in this same order. And I initialize `str` first and then `len`, in terms of the initializer list. And what I get to see is, well, it is it is working fine. I can see the call to the constructor and the call to the destructor. But that is not dependent on the order.

(Refer Slide Time: 06:52)

Program 14.05: A Simple String Class: Fails for wrong order of data members

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

class String {
    size_t len_; // Swapped members cause garbage to be printed or program crash (unhandled exception)
    char *str_;
public:
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { cout << "ctor: "; print(); }
    ~String() { cout << "dctor: "; print(); free(str_); }
    void print() { cout << "(" << str_ << " : " << len_ << ")" << endl; }
};

int main() { String s = "Partha";
            s.print();
}
----- // May produce garbage or crash
ctor: (Partha: 20)
(Partha: 20) // Garbage ✓
dctor: (Partha: 20)
```

- `len_` precedes `str_` in list of data members
- `len_.strlen(str_)` is executed before `str_.strdup(s)`
- When `strlen(str_)` is called `str_` is still uninitialized
- May causes the program to crash

Now, let us question that what happens if I change the order of these data members? If I swap the order of the data member, I first put len, and then put the char* str. If I just change these two, and as you will know that the order of construction will be in the order the data members are listed. And you will see that this will cause a disaster. Because even though here, this is given first, that is you should copy the str, copy the string s into str.

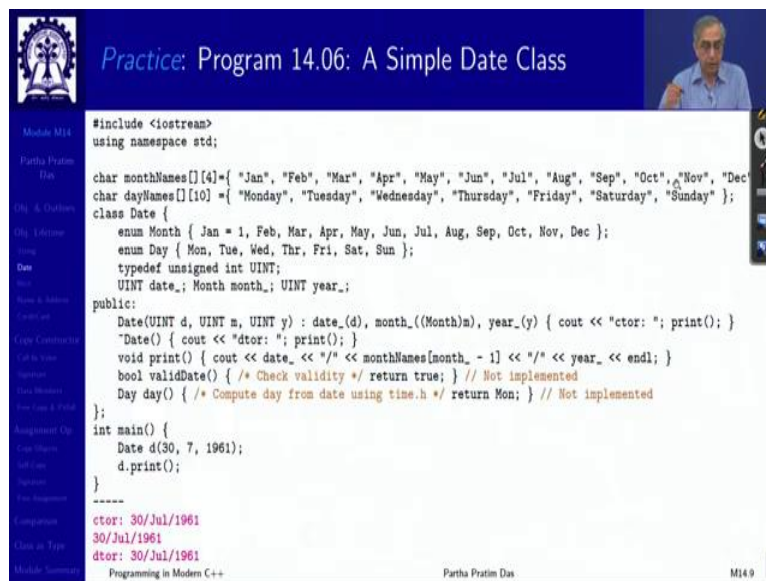
And then you are saying that from the str you will construct, you will compute the length to, this is the order given, but the order in which they will execute is first the len. So, this will be first to execute and this will be second to execute. Now obviously, when you are executing this first, you are passing on str, the second data member, which is still not been constructed is still not been initialized.

So, it is, it is, it is a garbage pointer, basically. So, you are passing a garbage pointer to strlen, so you do not know what it is going to do. So, if you if you run this program, then there are several possibilities. One is what I found on the mingw compiler that I am using on my Windows machine is that it does not crash it constructs, destructs but it arbitrarily takes some values 20.

Well, the actual length of the string, partha is 6. And when I run it on my visual studio, windows, visuals Microsoft Visual Studio, then the program simply crashes saying that there is an unhandled exception, because you have passed the garbage to strlen. And so you will have to be very careful in terms of the order in which you if there is dependency between the values of one initialized data member to the other.

You have to put them in the right order in the listing of the data members. So, that is what so it says that the lifetime of I mean the reason I am discussing it here is it says that the lifetime of str has to start before the lifetime of len is expected to start, because len depends on the str.

(Refer Slide Time: 09:06)



Practice: Program 14.06: A Simple Date Class

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

char monthNames[][4]={"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
char dayNames[][10]={"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};

class Date {
    enum Month { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
    enum Day { Mon, Tue, Wed, Thr, Fri, Sat, Sun };
    typedef unsigned int UINT;
    UINT date_; Month month_; UINT year_;
public:
    Date(UINT d, UINT m, UINT y) : date_(d), month_((Month)m), year_(y) { cout << "ctor: "; print(); }
    ~Date() { cout << "dtor: "; print(); }
    void print() { cout << date_ << "/" << monthNames[month_ - 1] << "/" << year_ << endl; }
    bool validDate() { /* Check validity */ return true; } // Not implemented
    Day day() { /* Compute day from date using time.h */ return Mon; } // Not implemented
};

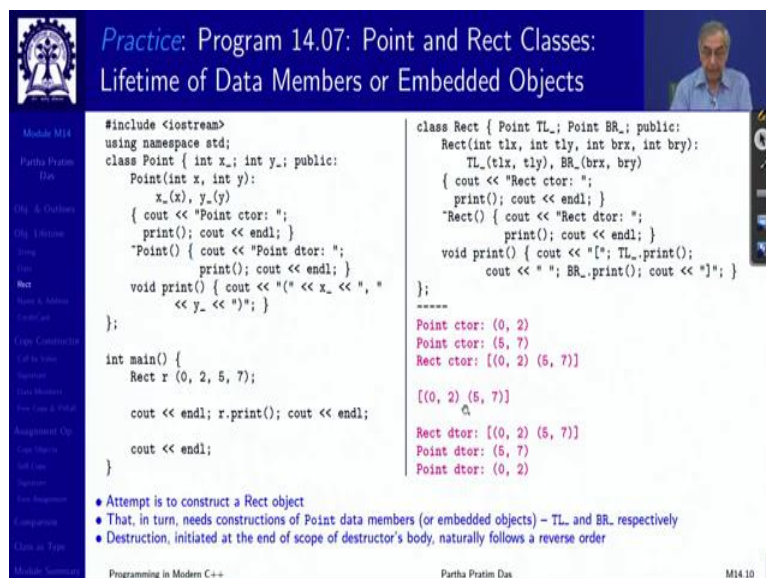
int main() {
    Date d(30, 7, 1961);
    d.print();
}

-----
ctor: 30/Jul/1961
30/Jul/1961
dtor: 30/Jul/1961
```

Programming in Modern C++ Partha Pratim Das M14.9

So, now, I have put a number of examples for you to actually trace the constructor destruction and the use of object lifetime. And as you can see here, the title is prefixed with practice. So, we will not get into the details. We will possibly come back using them in, in different contexts later on. But I I insist that you try out these programs, read every line of it and understand it well.

(Refer Slide Time: 09:36)



Practice: Program 14.07: Point and Rect Classes: Lifetime of Data Members or Embedded Objects

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

class Point { int x_, int y_; public:
    Point(int x, int y):
        x_(x), y_(y)
    { cout << "Point ctor: ";
      print(); cout << endl; }
    ~Point() { cout << "Point dtor: ";
             print(); cout << endl; }
    void print() { cout << "(" << x_ << ", " <<
                 << y_ << ")"; }
};

int main() {
    Rect r (0, 2, 5, 7);

    cout << endl; r.print(); cout << endl;

    cout << endl;
}

-----
Point ctor: (0, 2)
Point ctor: (5, 7)
Rect ctor: [(0, 2) (5, 7)]

[(0, 2) (5, 7)]
Rect dtor: [(0, 2) (5, 7)]
Point dtor: (5, 7)
Point dtor: (0, 2)
```

- Attempt is to construct a Rect object
- That, in turn, needs constructions of Point data members (or embedded objects) – TL_ and BR_ respectively
- Destruction, initiated at the end of scope of destructor's body, naturally follows a reverse order

Programming in Modern C++ Partha Pratim Das M14.10

So, there is a, this is the design of a date class. This is our good old friend, point and rec class and I have shown the expected output everywhere so you can check and get convinced that things are happening in the right way.

(Refer Slide Time: 09:48)

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

#include "String.h" // Containing class String from slide 14.7
#include "Date.h"

class Name { String firstName_, lastName_;
public: Name(char* fn, char* ln) : firstName_(fn), lastName_(ln)
    { cout << "Name ctor: "; print(); cout << endl; }
    ~Name() { cout << "Name dtor: "; print(); cout << endl; }
    void print() { firstName_.print(); cout << " "; lastName_.print(); }
};

class Address { unsigned int houseNo_;
    String street_, city_, pin_;
public: Address(unsigned int hn, char* sn, char* cn, char* pin) :
    houseNo_(hn), street_(sn), city_(cn), pin_(pin)
    { cout << "Address ctor: "; print(); cout << endl; }
    ~Address() { cout << "Address dtor: "; print(); cout << endl; }
    void print() {
        cout << houseNo_ << " ";
        street_.print(); cout << " ";
        city_.print(); cout << " ";
        pin_.print();
    }
};
```

Then there is a name, class and address class.

(Refer Slide Time: 09:52)

```
typedef unsigned int UINT;
class CreditCard { char cardNumber_[17];
    Name holder_; Address addr_;
    Date issueDate_, expiryDate_; UINT cvv_; };
class Name { String firstName_, lastName_; };
class Address { unsigned int houseNo_;
    String street_, city_, pin_; };
class Date { enum Month;
    UINT date_; Month month_; UINT year_; };

Construction of Objects
String: Sherlock
String: Holmes
Name: Sherlock Holmes
String: Baker Street
String: London
String: NW1 6XE
Address: 221 Baker Street London NW1 6XE
Date: 1/Jul/2014
Date: 1/Dec/2016
CC: 5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811

Use of Object
5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811

Destruction of Objects
*CC: 5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811
*Date: 1/Dec/2016
*Date: 1/Jul/2014
*Address: 221 Baker Street London NW1 6XE
*String: NW1 6XE
*String: London
*String: Baker Street
*Name: Sherlock Holmes
*String: Holmes
*String: Sherlock
```

These are all for your practice including the credit card class which uses all of these.

(Refer Slide Time: 09:58)

Practice: Program 14.08: CreditCard Class: Lifetime Chart

```
typedef unsigned int UINT;
class CreditCard { char cardNumber_[17];
                  Name holder_; Address addr_;
                  Date issueDate_, expiryDate_; UINT cvv_ };
class Name { String firstName_, lastName_ };
class Address { unsigned int houseNo_;
               String street_, city_, pin_ };
class Date { enum Month;
             UINT date_, Month month_, UINT year_ };

Construction of Objects
String: Sharlock
String: Holmes
Name: Sharlock Holmes
String: Baker Street
String: London
String: NW1 6XE
Address: 221 Baker Street London NW1 6XE
Date: 1/Jul/2014
Date: 1/Dec/2016
CC: 5321711934640027 Sharlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811

Use of Object
5321711934640027 Sharlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811

Destruction of Objects
*CC: 5321711934640027 Sharlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811
*Date: 1/Dec/2016
*Date: 1/Jul/2014
*Address: 221 Baker Street London NW1 6XE
*String: NW1 6XE
*String: London
*String: Baker Street
*Name: Sharlock Holmes
*String: Holmes
*String: Sharlock
```

Programming in Modern C++ Partha Pratim Das M14.13

And if you if you do that, if you run that, then this is the kind of the, the magenta is the kind of output that you expect from this, while you have on the right top the design of different classes. So please practice them out to get more confidence.

(Refer Slide Time: 10:15)

Copy Constructor

Copy Constructor

Programming in Modern C++ Partha Pratim Das M14.14

Now, let me move on to the copy issues.

(Refer Slide Time: 10:19)

Copy Constructor

- We know:
`Complex c1(4.2, 5.9);` ✓
invokes
`Constructor Complex::Complex(double, double);` ✓
- Which constructor is invoked for?
`Complex c2(c1);` ✓
- Or for?
`Complex c2 = c1;` ✓
- It is the **Copy Constructor** that takes an object of the same type and constructs a copy:
`Complex::Complex(const Complex &);`

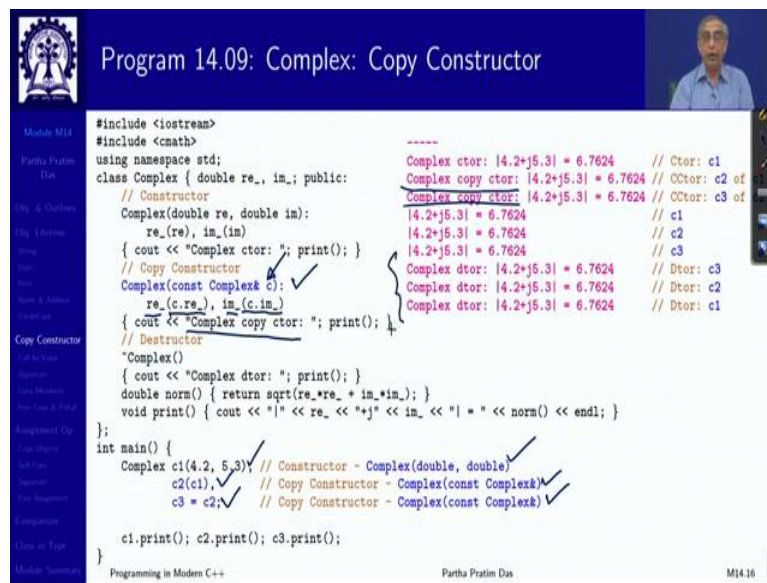
Programming in Modern C++ Partha Pratim Das M14.15

The first is a copy constructor. So, what we do is, this is a normal construction, we all have seen this. So, when we do this, we can we can do it by this or we can actually write it as 4.2 5.9 like this also. So, it constructs it expects a constructor which double double and construct by that. Now, what if you construct, invoke it like this. That c1 is already constructed as a complex object, and you are trying to construct c2, putting c1 in the parameter of this.

Or putting c1 in the initialization value. This, this is the point where copy construction is taking place. Why? Because you already have an object of the complex type here or here. And you want to create another object of the complex type as a copy of this object. So, the copy constructor takes place. So, in the copy constructor, naturally, what you are passing as a parameter is an object of the same type.

So, naturally, it is it has to take a parameter which is complex. We will avoid using the I mean, we would avoid using the call by value as you know, so we will put it as call by reference. And we do not want the source object to be changed by this copy constructor. So, we will put a const. So, that is the that is a basic copy constructor design, that we will have.

(Refer Slide Time: 11:50)



```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
// Constructor
Complex(double re, double im):
re_(re), im_(im)
{ cout << "Complex ctor: "; print(); }
// Copy Constructor
Complex(const Complex& c):
re_(c.re_), im_(c.im_)
{ cout << "Complex copy ctor: "; print(); }
// Destructor
~Complex()
{ cout << "Complex dtor: "; print(); }
double norm() { return sqrt(re_*re_ + im_*im_); }
void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};

int main() {
Complex c1(4.2, 5.3) // Constructor - Complex(double, double) ✓
c2(c1), // Copy Constructor - Complex(const Complex&) ✓
c3 = c2; // Copy Constructor - Complex(const Complex&) ✓

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

Complex ctor: |4.2+j5.3| = 6.7624 // Ctor: c1
Complex copy ctor: |4.2+j5.3| = 6.7624 // Ctor: c2 of
Complex copy ctor: |4.2+j5.3| = 6.7624 // Ctor: c3 of
|4.2+j5.3| = 6.7624 // c1
|4.2+j5.3| = 6.7624 // c2
|4.2+j5.3| = 6.7624 // c3
Complex dtor: |4.2+j5.3| = 6.7624 // Dtor: c3
Complex dtor: |4.2+j5.3| = 6.7624 // Dtor: c2
Complex dtor: |4.2+j5.3| = 6.7624 // Dtor: c1

So, here is an example of this, where this is the copy constructor. So, what we do is, I am passing a object c of complex type to this constructor. And I take the real component of c copy to the real component of the object to be constructed similarly for. So, basically, item by item, I am data member by data member I copy. And if those data members are user defined types, then their respective copy constructor in turn will again get invoked.

It is, it is the same process as as the construction where construction in general can take parameters of any type, or may not have a parameter for default. In copy constructor specifically, you take an object of the same type. So, when you write this, you are invoking a constructor, and when you are writing like this or this you are invoking the copy constructor. And I have put messages here specifically.

So, that in the output trace, you can find out which constructor is being called and in which order destructors of course, will happen in the reverse order as we have seen. So, this is a, this is the basic story of copy construction. So, it is like copying variables we have.

(Refer Slide Time: 13:11)

Why do we need Copy Constructor?

- Consider the **function call mechanisms** in C++:
 - **Call-by-reference**: Set a reference to the actual parameter as a formal parameter. Both the formal parameter and the actual parameter share the same location (object). **No copy is needed**
 - **Return-by-reference**: Set a reference to the computed value as a return value. Both the computed value and the return value share the same location (object). **No copy is needed**
 - **Call-by-value**: Make a **copy or clone** of the actual parameter as a formal parameter. This needs a **Copy Constructor**
 - **Return-by-value**: Make a **copy or clone** of the computed value as a return value. This needs a **Copy Constructor**
- **Copy Constructor** is needed for *initializing the data members* of a UDT from an existing value

Programming in Modern C++ Partha Pratim Das M14.17

Now, we have given ourselves a mechanism to copy any object that we have created for our user defined types. Now, the question is, when do you need to do this copy construction? The primary reason you need copy construction is to support call by value. As you know, if you do call by reference or return by reference, then only the reference is being passed, no object copies as we have seen already.

But if you do a call by value, then naturally you need to copy the given actual parameter object to the expected formal parameter object. So, you need a copy and that copy is what is done by the copy constructor. Similarly, if you are returning something by value, then you will need the copy constructor. So, copy constructor is needed for initializing the data members of a UDT from an existing value.

(Refer Slide Time: 14:06)

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
    Complex(double re, double im): re_(re), im_(im) // Constructor
    { cout << "ctor: "; print(); }
    Complex(const Complex& c): re_(c.re_), im_(c.im_) // Copy Constructor
    { cout << "copy ctor: "; print(); }
    ~Complex() { cout << "dctor: "; print(); }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
void Display(Complex c_param) { // Call by value
    cout << "Display: "; c_param.print();
}
int main() { Complex c(4.2, 5.3); // Constructor - Complex(double, double)
    Display(c); // Copy Constructor called to copy c to c_param
}
-----
ctor: |4.2+j5.3| = 6.7624 // Ctor of c in main()
copy ctor: |4.2+j5.3| = 6.7624 // Ctor c_param as copy of c, call Display()
Display: |4.2+j5.3| = 6.7624 // c_param
dctor: |4.2+j5.3| = 6.7624 // Dtor c_param on exit from Display()
dctor: |4.2+j5.3| = 6.7624 // Dtor of c on exit from main()
-----
Programming in Modern C++ Partha Pratim Das M14.10
```

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
    Complex(double re, double im): re_(re), im_(im) // Constructor
    { cout << "ctor: "; print(); }
    Complex(const Complex& c): re_(c.re_), im_(c.im_) // Copy Constructor
    { cout << "copy ctor: "; print(); }
    ~Complex() { cout << "dctor: "; print(); }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
void Display(Complex c_param) { // Call by value
    cout << "Display: "; c_param.print();
}
int main() { Complex c(4.2, 5.3); // Constructor - Complex(double, double)
    Display(c); // Copy Constructor called to copy c to c_param
}
-----
✓ctor: |4.2+j5.3| = 6.7624 // Ctor of c in main()
✓copy ctor: |4.2+j5.3| = 6.7624 // Ctor c_param as copy of c, call Display()
✓Display: |4.2+j5.3| = 6.7624 // c_param
✓dctor: |4.2+j5.3| = 6.7624 // Dtor c_param on exit from Display()
✓dctor: |4.2+j5.3| = 6.7624 // Dtor of c on exit from main()
-----
Programming in Modern C++ Partha Pratim Das M14.10
```

So, this is the reason without the copy constructor, you will not be able to call functions by value. So, here I give an example, this is a copy constructor. And I am trying to write a global function, which is displayed which takes a complex parameter by value. I am not putting the &. So, as I take it by value, and I call it so my actual parameters c needs to be copied to the formal parameters c_param.

And when I do that, this copy process will invoke the copy constructor because it has to copy the fields of c into the fields of c_param. So, as you can see here is really that initially when you do, initially when you are doing this, you have a constructor call. And then when you have not made any explicit copy constructor call here, but when you call display try to call display, you have a copy constructor called copying c_param as a copy of c in display.

And then the display is actually called it does the display process. And when you are done, when you are done, you are here you have constructed a an object here. So, the lifetime of that object has started, that formal parameter object has started with the call of display. So, when you get to the end of display, your control is going back, you are out of that scope.

And by that same rule, the destructor will be called for c_param. So, the destructor is called and with the same set of values. And then finally the destructor of c is called at the end of main. So, this is how the call by value is supported.

(Refer Slide Time: 15:56)

Signature of Copy Constructors

- Signature of a *Copy Constructor* can be one of:
 - `MyClass(const MyClass& other);` ✓ // Common // Source cannot be changed
 - `MyClass(MyClass& other);` // Occasional // Source needs to change. Like in smart pointers
 - `MyClass(volatile const MyClass& other);` // Rare
 - `MyClass(volatile MyClass& other);` // Rare
- None of the following are copy constructors, though they can copy:
 - `MyClass(MyClass* other);`
 - `MyClass(const MyClass* other);`
- Why the parameter to a copy constructor must be passed as Call-by-Reference?
 - `MyClass(MyClass other);`

The above is an infinite recursion of copy calls as the call to copy constructor itself needs to make copy for the Call-by-Value mechanism

Programming in Modern C++ Partha Pratim Das M14.19

Signature of Copy Constructors

- Signature of a *Copy Constructor* can be one of:
 - `MyClass(const MyClass& other);` // Common // Source cannot be changed
 - `MyClass(MyClass& other);` // Occasional // Source needs to change. Like in smart pointers
 - `MyClass(volatile const MyClass& other);` // Rare
 - `MyClass(volatile MyClass& other);` // Rare
- None of the following are copy constructors, though they can copy:
 - `MyClass(MyClass* other);`
 - `MyClass(const MyClass* other);`
- Why the parameter to a copy constructor must be passed as Call-by-Reference?
 - `MyClass(MyClass other);`

The above is an infinite recursion of copy calls as the call to copy constructor itself needs to make copy for the Call-by-Value mechanism

Programming in Modern C++ Partha Pratim Das M14.19

You can see the same thing if you do return by value as well. So, what is what should be the signature for the constructor function. So, this is the most common signature that you pass the

parameter to the copy constructor by reference by reference, and you make that a constant so that the source cannot be changed. It is also possible that you write a copy constructor where you have not used constant.

In it, you might wonder that while I am copying, why do I need to change the source? See there is a subtle difference between what we say is a copy and what you say is a move. If I want to realize move, which does not exist in C++ 03, then what I want is not only make a copy but actually invalidate the source if required. So, that there is only 1 because if you are just to copy there are two copies of the variable.

If I just want one, that the source has moved to the destination, then I will need to use this I will talk about this in more details when I talk about smart pointers and how they use this. And we will come back to that in C++ 11 when we talk about the move semantics, which is specifically supported. So, this is the most common 99.9 percent of copy constructors are like this, some are like this and very rarely you may use other qualifiers like volatile and all that.

So, it may also be noted that if you pass a, if you pass a pointer, like if you try to do a call by address for the copy constructor, you can, you will still be able to make a copy. But that is not a copy constructor, that is if you provide this, the compiler will not use it in the context of call by value or call by reference. So, do not consider that these are functions. These are obviously some overloaded constructors, but they are not copy constructor.

Now the final question is why do I need to pass the parameter to the copy constructor as reference? What if I pass it as a value? Now what will happen? If I pass it as a value, then to be able to call the copy constructor needs to copy this value, which in turn needs the copy constructor itself. So, I am passing it called by value, so call by value needs, the value must be copied that needs the copy constructor must be invoked.

And if I do that for the copy constructor, then the copy constructor itself will have to be invoked. So, it will have to be called. And to call that I need to copy the actual parameter again. So, the copy constructor will be called again. So, I end up having an infinite recursion. So, this is not if you write this, then you will basically have an infinite recursion you will never be able to end. So, the only way to write copy constructed is by reference and preferably by constant reference.

(Refer Slide Time: 19:04)

Program 14.11: Point and Rect Classes: Embedded Of Default, Copy and Overloaded Constructors

```

#include <iostream>
using namespace std;
class Point { int x_; int y_; public:
    Point(int x, int y): x_(x), y_(y) { cout << "Point ctor: "; print(); cout << endl; } // Ctor
    Point(): x_(0), y_(0) { cout << "Point ctor: "; print(); cout << endl; } // Dctor
    Point(const Point& p): x_(p.x_), y_(p.y_) { cout << "Point ctor: "; print(); cout << endl; } // Cctor
    Point() { cout << "Point dtor: "; print(); cout << endl; } // Dctor
    void print() { cout << "(" << x_ << ", " << y_ << ")"; } // Class Point
};
class Rect { Point TL_; Point BR_; public:
    Rect(int tlx, int tly, int brx, int bry): TL_(tlx, tly), BR_(brx, bry) // Ctor of Rect: 4 coords
    { cout << "Rect ctor: "; print(); cout << endl; } // Uses Ctor for Point
    Rect(const Point& p_tl, const Point& p_br): TL_(p_tl), BR_(p_br) // Ctor of Rect: 2 Points
    { cout << "Rect ctor: "; print(); cout << endl; } // Uses Cctor for Point
    Rect(const Point& p_tl, int brx, int bry): TL_(p_tl), BR_(brx, bry) // Ctor of Rect: Point + 2 coords
    { cout << "Rect ctor: "; print(); cout << endl; } // Uses Cctor for Point
    Rect() { cout << "Rect ctor: "; print(); cout << endl; } // Dctor of Rect: // Dctor Point
    Rect(const Rect& r): TL_(r.TL_), BR_(r.BR_) // Cctor of Rect
    { cout << "Rect ctor: "; print(); cout << endl; } // Uses Cctor for Point
    ~Rect() { cout << "Rect dtor: "; print(); cout << endl; } // Dtor
    void print() { cout << "[" << TL_.print(); cout << " "; BR_.print(); cout << "]""; } // Class Rect
};

```

- When parameter (tlx, tly) is set to TL_ by TL_(tlx, tly): parameterized Ctor of Point is invoked
- When parameter p_tl is set to TL_ by TL_(p_tl): Cctor of Point is invoked
- When TL_ is set by default in Dctor of Rect: Dctor of Point is invoked
- When member r.TL_ is set to TL_ by TL_(r.TL_) in Cctor of Rect: Cctor of Point is invoked

M14.20

So, these are again, our point and rect class the different instances. For example, if you see that in the initializer list, you are constructing tl the top left object point by x and y the integer value, so this will invoke the normal constructor. But if you are doing this, that is where your ptl is already a point object and you are trying to create the tl member of the rec class by copying it then you will have the copy constructor invoked.

So, you can study it well. The notes are all given explaining what every stage is doing and understand. Where does the default constructor come in? Where does the copy constructor come in? And what does the other overloaded constructors come in?

(Refer Slide Time: 19:57)

Practice: Program 14.11: Rect Class: Trace of Object

Code	Output	Lifetime	Remarks
int main() {			
Rect r1(0, 2, 5, 7);	Point ctor: (0, 2) Point ctor: (5, 7) Rect ctor: (0, 2) (5, 7)	Point r1.TL_ Point r1.BR_ Rect r1	
Rect r2(Point(3, 5), Point(6, 9));	Point ctor: (6, 9) Point ctor: (3, 5) Rect ctor: (3, 5) (6, 9) Point dtor: (3, 5) Point dtor: (6, 9)	Point t1 Point t2 Rect r2 Point t2 Point t1	Second parameter First parameter Copy to r2.TL_ Copy to r2.BR_
Rect r3(Point(2, 2), 6, 4);	Point ctor: (2, 2) Point ctor: (2, 2) Point ctor: (6, 4) Rect ctor: (2, 2) (6, 4) Point dtor: (2, 2)	Point t3 r3.TL_ = t3 Point r3.BR_ Rect r3 Point t3	First parameter Copy to r3.TL_
Rect r4;	Point ctor: (0, 0) Point ctor: (0, 0) Rect ctor: (0, 0) (0, 0)	Point r4.TL_ Point r4.BR_ Rect r4	First parameter
return 0;	Rect dtor: (0, 0) (0, 0) Point dtor: (0, 0) Point dtor: (0, 0) Rect dtor: (2, 2) (6, 4) Point dtor: (6, 4) Point dtor: (2, 2) Rect dtor: (3, 5) (6, 9) Point dtor: (6, 9) Point dtor: (3, 5) Rect dtor: (0, 2) (5, 7) Point dtor: (5, 7) Point dtor: (0, 2)	Rect r4 Point r4.BR_ Point r4.TL_ Rect r3 Point r3.BR_ Point r3.TL_ Rect r2 Point r2.BR_ Point r2.TL_ Rect r1 Point r1.BR_ Point r1.TL_	

M14.21

So, if you run that program and you will see that actually, this is the kind of output that you get. And it will be a good exercise that you do not look into this, right hand side column first, keep it blank on your copy, guess take output, and then try to put the lifetime information as to when things are getting created, when things are getting copied, when things are getting destroyed, when things are getting used and so on, that will give you a good practice. So, this is a, this is yet another practice slide that I have put for you.

(Refer Slide Time: 20:29)

Free Copy Constructor

- If no copy constructor is provided by the user, the compiler supplies a *free* one
- *Free* copy constructor cannot initialize the object to proper values. It performs *Shallow Copy*
- **Shallow Copy** aka *bit-wise copy, field-by-field copy, field-for-field copy, or field copy*
 - An object is created by simply *copying the data of all variables of the original object*
 - Works well if *none of the variables of the object are defined in heap / free store*
 - For dynamically created variables, the *copied object refers to the same memory location*
 - Creates *ambiguity* (changing one changes the copy) and *run-time errors* (dangling pointer)
- **Deep Copy** or its variants *Lazy Copy* and *Copy-on-Write*
 - An object is created by copying data of all variables except the ones on heap
 - Allocates similar memory resources with the same value to the object
 - **Need to explicitly define the copy constructor and assign dynamic memory as required**
 - **Required to dynamically allocate memory to the variables in the other constructors**

Diagram: The diagram compares 'Shallow Clone' and 'Deep Clone'. In 'Shallow Clone', the 'Original Object' and 'Cloned Object' both point to the same 'Referenced Object' box. In 'Deep Clone', the 'Original Object' points to its own 'Referenced Object', and the 'Cloned Object' points to a separate 'Referenced Clone' box.

Free Copy Constructor

- If no copy constructor is provided by the user, the compiler supplies a *free* one
- *Free* copy constructor cannot initialize the object to proper values. It performs *Shallow Copy*
- **Shallow Copy** aka *bit-wise copy, field-by-field copy, field-for-field copy, or field copy*
 - An object is created by simply *copying the data of all variables of the original object*
 - Works well if *none of the variables of the object are defined in heap / free store*
 - For dynamically created variables, the *copied object refers to the same memory location*
 - Creates *ambiguity* (changing one changes the copy) and *run-time errors* (dangling pointer)
- **Deep Copy** or its variants *Lazy Copy* and *Copy-on-Write*
 - An object is created by copying data of all variables except the ones on heap
 - Allocates similar memory resources with the same value to the object
 - **Need to explicitly define the copy constructor and assign dynamic memory as required**
 - **Required to dynamically allocate memory to the variables in the other constructors**

Diagram: The diagram compares 'Shallow Clone' and 'Deep Clone'. In 'Shallow Clone', the 'Original Object' and 'Cloned Object' both point to the same 'Referenced Object' box. In 'Deep Clone', the 'Original Object' points to its own 'Referenced Object', and the 'Cloned Object' points to a separate 'Referenced Clone' box.

Now, like the constructor, if you do not provide a copy constructor, then the compiler will give you a copy constructor. Now, the copy constructor, the compiler provides, because if I mean, if you have not given a copy constructor, and you are using that object of that class, by

in call by value or call by reference, I mean call by value or return by value, then certainly the compiler needs a copy constructor.

So, it will provide you a free one. Now, it does not know what to do, what so what it does, it just copies a bit pattern, whatever the object had it copies a bit pattern. So, this is called the bitwise copy field wise copy, field for field copy, field copy, field copy and so on. So, if you have data members, which are just, built in types, whose actually they do not have any specific thing to construct, their values will come properly.

But if you have some reference variable, some pointer variable, which has allocated a location, then what will happen this is your original object. Now, you have if you have copied that pointer value, then in your copied object, the pointer value will be the same. So, you have two copied objects having the pointed values, which are same, so they are basically referring to the same object.

So, this is a very ambiguous copy. And this might lead to unexpected results because, for example, after cloning, if I clone is another process used for referring to the copied object. So, after cloning, if I use that pointer to change this reference object, then the same value will be reflected in the original object. So, this type of copy is called shallow copy, where you are just copying the pointers, because you are doing a bit copy.

That is what the compiler has given you. The other which is which is basically the preferred or the correct way of doing this is to have a deep copy. So, what you do in a deep copy in a deep copy, you for the, every built in type, you copy the value. For every other data members, you call the copy constructor of the corresponding class and for pointers, you actually make a fresh allocation and copy the pointed value as well.

It is not enough to just copy the pointer that is not semantically correct. So, you have to explicitly define the copy constructor assign dynamic memory as required. And put the so we have in terms of a deep clone or deep copy your original object is giving you the clone object. You have a reference object, you are not copying this pointer, you are not copying this pointer rather, you are doing a fresh allocation and copying this object by that object's copy constructor. So, that is a, that is a big pitfall of having a free copy constructor from the compiler or the pitfalls of shallow copy.

(Refer Slide Time: 23:43)

Pitfalls of Bit-wise Copy: Shallow Copy

- Consider a class:

```
class A { int i_; // Non-pointer data member
        int* p_; // Pointer data member
public:
    A(int i, int j) : i_(i), p_(new int(j)) { // Init. with pointer to dynamically created object
    }
    ~A() { cout << "Destruct " << this << ": "; // Object identity
          cout << " i_ = " << i_ << " p_ = " << p_ << " *p_ = " << *p_ << endl; // Object state
          delete p_; // Release resource
    };
```
- As no copy constructor is provided, the implicit copy constructor does a bit-wise copy. So when an A object is copied, p_ is copied and continues to point to the same dynamic int:

```
int main() { A a1(2, 3); A a2(a1); // Construct a2 as a copy of a1. Done by bit-wise copy
            cout << "a1 = " << a1 << " a2 = " << a2 << endl;
```
- The output is wrong, as a1.p_ = a2.p_ points to the same int location. Once a2 is destructed, a2.p_ is released, and a1.p_ becomes dangling. The program may print garbage or crash:

```
a1 = 008FF838 a2 = 008FF828 // Identities of objects
Destruct 008FF828: i_ = 2 p_ = 00C15440 *p_ = 4 // Dtor of a2. Note that a2.p_ = a1.p_
Destruct 008FF838: i_ = 2 p_ = 008FF838 *p_ = -17891602 // Dtor of a1. a1.p_ = a2.p_ points to garbage
```
- The bit-wise copy of members is known as **Shallow Copy**

Programming in Modern C++ Partha Pratim Das M14.23

So, to just to illustrate, I have shown here a class with one integer value and one pointer two integer. So, naturally in the constructor, a value in a dynamic allocation is made and that will be deleted later on. Now, I have not provided any copy constructor. So, what the compiler will do? It will provide a free one which will copy everything. So, what happens I have two objects, one is originally created, the other is a copy of that which has been created by bit copy, shallow copy.

So, when I print the addresses of these two objects, I find that different objects. But when I go deep and print the pointers, this particular p_ pointer that has also got copied. So, they have same values and both of them will try to point to the same location. Now what is happening, what when I do this, then at this point, you can see that my destructor is actually making the print.

So, when I come to the end of main, this object will be the first to get destroyed. The last created first destroyed. So, this object, when this object is destroyed, this p_ pointer is deleted. So, now it becomes a dangling pointer. So, when you go to destroying a1, which is a first object created the second to be destroyed, then you are again trying to do a delete p_ or you are trying to print the value of pointed to by this.

But p_ has already been deleted, because it was pointed to by a2 and a2 has done the destruction. And it is deleted that pointer is a dangling pointer, so, you get a garbage value you might get crash also. So, this is a problem with the shallow copy.

(Refer Slide Time: 25:41)

Pitfalls of Bit-wise Copy: Deep Copy

- Now suppose we provide a user-defined copy constructor:

```
class A { int i_; // Non-pointer data member
        int* p_; // Pointer data member
public:
    A(int i, int j) : i_(i), p_(new int(j)) {} // Init. with pointer to dynamically created object
    A(const A& a) : i_(a.i_), // Copy Constructor
                 p_(new int(*a.p_)) {} // Allocation done and value copied - Deep Copy
    ~A() { cout << "Destruct " << this << " "; // Object identity
          cout << "i_ = " << i_ << " p_ = " << p_ << " *p_ = " << *p_ << endl; // Object state
          delete p_; // Release resource
    };
};
```

- The output now is correct, as $a1.p_ \neq a2.p_$ points to the different int locations with the values $*a1.p_ = *a2.p_$ properly copied:

```
&a1 = 00B8F9E0 &a2 = 00B8F9D0 // Identities of objects
Destruct 00B8F9D0: i_ = 2 p_ = 00C95480 *p_ = 3 // Dtor of a2. a2.p_ is different from a1.p_
Destruct 00B8F9E0: i_ = 2 p_ = 00C95440 *p_ = 3 // Dtor of a1. Works correctly!
```

- This is known as **Deep Copy** where every member is copied properly. Note that:
 - In every class, provide copy constructor to adopt to deep copy which is always safe
 - Naturally, shallow copy is cheaper than deep copy. So some languages support variants as *Lazy Copy* or *Copy-on-Write* for efficiency

Programming in Modern C++ Partha Pratim Das M14.24

Pitfalls of Bit-wise Copy: Deep Copy

- Now suppose we provide a user-defined copy constructor:

```
class A { int i_; // Non-pointer data member
        int* p_; // Pointer data member
public:
    A(int i, int j) : i_(i), p_(new int(j)) {} // Init. with pointer to dynamically created object
    A(const A& a) : i_(a.i_), // Copy Constructor
                 p_(new int(*a.p_)) {} // Allocation done and value copied - Deep Copy
    ~A() { cout << "Destruct " << this << " "; // Object identity
          cout << "i_ = " << i_ << " p_ = " << p_ << " *p_ = " << *p_ << endl; // Object state
          delete p_; // Release resource
    };
};
```

- The output now is correct, as $a1.p_ \neq a2.p_$ points to the different int locations with the values $*a1.p_ = *a2.p_$ properly copied:

```
&a1 = 00B8F9E0 &a2 = 00B8F9D0 // Identities of objects
Destruct 00B8F9D0: i_ = 2 p_ = 00C95480 *p_ = 3 // Dtor of a2. a2.p_ is different from a1.p_
Destruct 00B8F9E0: i_ = 2 p_ = 00C95440 *p_ = 3 // Dtor of a1. Works correctly!
```

- This is known as **Deep Copy** where every member is copied properly. Note that:
 - In every class, provide copy constructor to adopt to deep copy which is always safe
 - Naturally, shallow copy is cheaper than deep copy. So some languages support variants as *Lazy Copy* or *Copy-on-Write* for efficiency

Programming in Modern C++ Partha Pratim Das M14.24

So, do not do that always do deep copy. So, as in the copy constructor, now, you can see that you copy the integer variable because it is not a pointer. For a pointer, you do an allocation and initialize it with the value it was pointing to. Once you do that, and you write the same code, now, what you have is in the copy it is not a bitwise copy, it is no more that you have these two pointers same.

So, in the two different objects that you have got the two pointers are of different value. And therefore, when a2 is destroyed, and that pointer is released, the pointer in the a1 does not get affected, and both of them have the same value because that is how you have created. So, this is what is deep copy. There are alternate terms also which is used, some use lazy copy, some use copy on right.

There, they are not exactly same, there are semantic differences and at an appropriate point I will explain that. But the whole idea of deep copy is copy the pointed variable values separately with allocation.

(Refer Slide Time: 26:54)

Practice: Program 14.12: Complex: Free Copy Constr

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
    Complex(double re, double im) : re_(re), im_(im) { cout << "ctor: "; print(); } // Ctor
    // Complex(const Complex& c) : re_(c.re_), im_(c.im_) { cout << "copy ctor: "; print(); } // Cctor: Free
    ~Complex() { cout << "dtor: "; print(); } // Dtor
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "| " << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
void Display(Complex c_param) { cout << "Display: "; c_param.print(); }
int main() { Complex c(4.2, 5.3); // Constructor = Complex(double, double)
    Display(c); // Free Copy Constructor called to copy c to c_param
}
```

User-defined Cctor	Free Cctor
ctor: 4.2+j5.3 = 6.7624	ctor: 4.2+j5.3 = 6.7624
copy ctor: 4.2+j5.3 = 6.7624	No message from free Cctor
Display: 4.2+j5.3 = 6.7624	Display: 4.2+j5.3 = 6.7624
dtor: 4.2+j5.3 = 6.7624	dtor: 4.2+j5.3 = 6.7624
dtor: 4.2+j5.3 = 6.7624	dtor: 4.2+j5.3 = 6.7624

- User has provided **no copy constructor**
- Compiler provides **free copy constructor**
- Compiler-provided copy constructor **performs bit-wise copy** - hence there is no message
- Correct in this case** as members are of built-in type and there is no dynamically allocated data

Programming in Modern C++ Partha Pratim Das M14.25

So, again, practice examples given here with a free copy constructor to carry out through them. There is a user defined copy constructor for a string object class, please try that with the free copy constructor.

(Refer Slide Time: 27:02)

Practice: Program 14.14: String: Free Copy Construct

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String { public: char *str_; size_t len_;
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // Ctor
    // String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // Cctor: Free only
    ~String() { free(str_); } // Dtor
    void print() { cout << " " << str_ << " : " << len_ << " " << endl; }
};
void strToUpper(String a) { // Make the string uppercase
    for (int i = 0; i < a.len_; ++i) { a.str_[i] = toupper(a.str_[i]); } cout << "strToUpper: "; a.print();
} // a.String() is invoked releasing a.str_ and invalidating s.str_ = a.str_
int main() { String s = "Partha"; s.print(); strToUpper(s); s.print(); } // Last print fails
```

User-defined Cctor	Free Cctor
(Partha: 6)	(Partha: 6)
strToUpper: (PARTHA: 6)	strToUpper: (PARTHA: 6)
(Partha: 6)	(?????????????????????????????????????: 6)

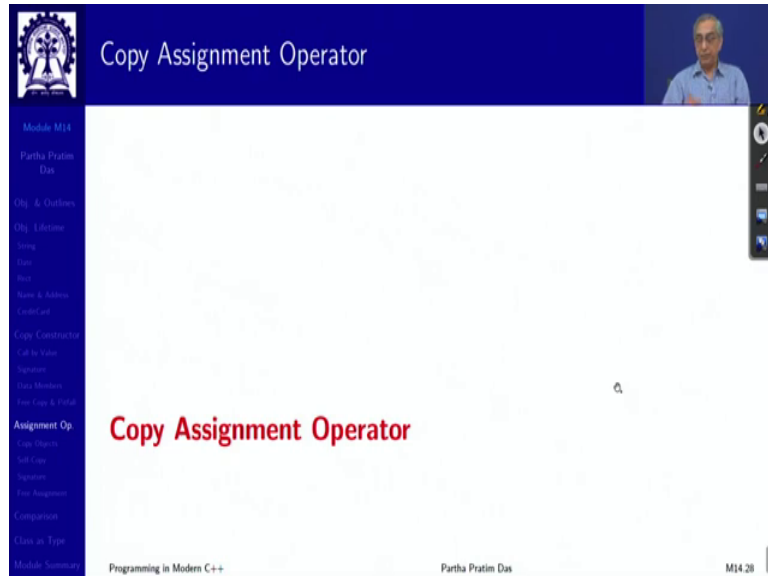
- User has provided **no copy constructor**. Compiler provides **free copy constructor**
- Free copy constructor performs **bit-copy** - hence no allocation is done for str_ when actual parameter s is copied to formal parameter a. s.str_ is merely copied to a.str_ and both continue to point to the same memory. On exit from strToUpper, a is destructed and a.str_ is deallocated. Hence in main access to s.str_ is dangling. Program prints garbage and / or crashes
- Shallow Copy** With bit-copy, only the pointer is copied - not the pointed object. **This is risky**

Programming in Modern C++ Partha Pratim Das M14.27

And in the, in the notes, I would always suggest that do not read the notes, first understand the program. If you cannot figure it out, or after you have figured out what has happened and

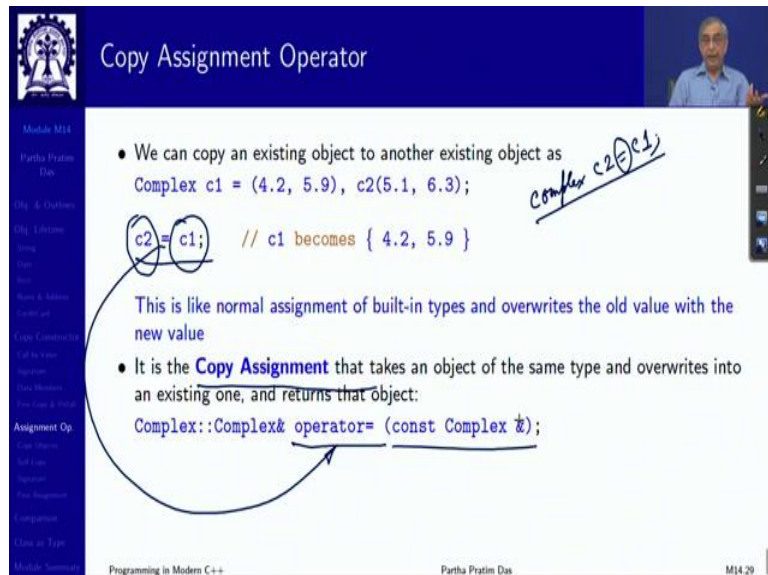
try it build and try to run it. Once you have seen what is happening, then you go to the notes and confirm that your understanding is correct.

(Refer Slide Time: 27:32)



So, this is how the practice ones must be done. Now, let me talk about the copy assignment.

(Refer Slide Time: 27:39)



So, other than the construction, which was we were doing, the other that we do regularly is copy 1 value into another. So, when I write it, when I write something like complex c2 then I made a copy construction, because this is an initialization.

But if I simply write a statement `c1 c2` both are available, and if we simply write a statement `c1` is copied to `c2`, then I need a copy to be done which means that I would like to erase the values in `c2` and overwrite them with the values in `c1`. So, this is an assignment. So, this is called copy assignment or assignment.

Now, we know operators can be overloaded. So, this operator of assignment is Operator Assignment. Naturally it takes the object of this complex type, the source type, which will come as `a`, as a constant reference. And what does it return? It has to return an object of the same time because you have copied. So, it also returns by reference the same type of object.

(Refer Slide Time: 29:03)

```

#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
    Complex(double re, double im) : re_(re), im_(im) { cout << "ctor: "; print(); } // Ctor
    Complex(const Complex& c) : re_(c.re_), im_(c.im_) { cout << "cctor: "; print(); } // Cctor
    Complex() { cout << "dctor: "; print(); } // Dctor
    Complex& operator=(const Complex& c) // Copy Assignment Operator
    { re_ = c.re_; im_ = c.im_; cout << "copy: "; print(); return *this; } // Return *this for chainable
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; } }; // Class Complex
int main() { Complex c1(4.2, 5.3), c2(7.9, 8.5); Complex c3(c2); // c3 Copy Constructed from c2
    c1.print(); c2.print(); // Copy Assignment Operator
    c1 = c2 = c3; c1.print(); c2.print(); c3.print(); // Copy Assignment Chain
}
ctor: |4.2+j5.3| = 6.7624 // c1 - ctor
cctor: |7.9+j8.5| = 11.6043 // c2 - ctor
cctor: |7.9+j8.5| = 11.6043 // c3 - ctor
copy: |4.2+j5.3| = 6.7624 // c1
copy: |7.9+j8.5| = 11.6043 // c2
copy: |7.9+j8.5| = 11.6043 // c3
dctor: |7.9+j8.5| = 11.6043 // c3 - dctor
dctor: |7.9+j8.5| = 11.6043 // c2 - dctor
dctor: |7.9+j8.5| = 11.6043 // c1 - dctor

```

• Copy assignment operator should return the object to make chain assignments possible

So, here is a copy assignment here, the copy assignment for this. And it returns, returns this, it is not only enough to copy and make changes. Like you are copying and making changes here, which is fine. But it is also important that it returns the object of the same type. Why do we need that because we may want to change the assignment. For example, here we have changed the assignment.

So, `c3` so this operator is called first because this is right associative. So, `c3` is copied to `c2` and whatever is copied that object is returned. And that then is copied to `c1`. If you did not return that object that you have copied into. Then the second one you will not be able to write. So, to support this continuity of semantics of assignment being chainable, you need to return the same object.

(Refer Slide Time: 30:13)

Program 14.16: String: Copy Assignment

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String { public: char *str_; size_t len_;
String(char *s) : str_(strdup(s)), len_(strlen(str_)) { } // Ctor
String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // Cctor
String() { free(str_); } // Dtor
String& operator=(const String& s) { // Copy Assignment Operator
    free(str_); // Release existing memory
    str_ = strdup(s.str_); // Perform deep copy
    len_ = s.len_; // Copy data member of built-in type
    return *this; // Return object for chain assignment
}
void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() { String s1 = "Football", s2 = "Cricket"; s1.print(); s2.print(); s2 = s1; s2.print(); }
---
```

(Football: 8)
(Cricket: 7)
(Football: 8)

- In copy assignment operator, `str_ = s.str_;` should not be done for two reasons:
 - 1) Resource held by `str_;` will leak
 - 2) Shallow copy will result with its related issues
- What happens if a self-copy `s1 = s1;` is done?

Programming in Modern C++ Partha Pratim Das M14.31

So, this is what is the copy assignment, you can see the copy assignment in string. So, if in the, in the string, you have a pointer and the length, so the string is pointing to an allocated memory where you have doped. So, what you will have to do is when you are copying, you cannot simply copy that pointer from your source to the destination object in the copy assignment.

Because if you do that, then whatever the your destination object was pointing to will leak, because that memory gets lost. So, you have to free that up, then do a fresh strdup, to do a deep copy, as you have understood by now, copy the length and return the same object. So, that is a that is a simple way of doing this. This works pretty fine and if you do this with s1 being assigned to s2, you will have a very nice working.

Now, what if you are doing a self-copy? Instead of this, what if you are now you might ask why should I do a self-copy, but it is always possible because, you do not in the whole set of programming, you do not want to create an operator. Because in a built in type of int x, if you assign x to x, then it has it it works perfectly, but here it should also work perfectly, but does it?

Suppose you have done this, this is the only change that I have made. Now, what will happen? You have released the memory once you have released the memory here, here you actually wanted to release the memory here in the destination object but is same as the source

object. So, it has got released in that as well. So, when you are doing this strdup for performing grip copy, it is a garbage value.

So it will create a either a garbage value or a program crash. So, self-copy is a problem in this strategy. And so it has to be detected and somehow taken care of. So it is very easy to do that. Because if you are doing a self-copy, then all that you need to detect is the object that you are getting as a source and the current object on which the copy assignment operator has been invoked.

That is that this pointer, these are the same object. So, what is identity the address, so this is the destination object. And &s is a source object. If they are identical, then they are the same object and therefore, there is nothing to be done for this copy. So, if this is equal to &s, you just return this, that is all. Otherwise, you do this, where you know you are, they are different and it is it is going to be always correct.

(Refer Slide Time: 33:15)

Signature and Body of Copy Assignment Operator

- For class `MyClass`, typical copy assignment operator will be:

```
MyClass& operator=(const MyClass& s) {  
    if (this != &s) { // Check if the source and destination are same  
        // Release resources held by *this  
        // Copy members of s to members of *this  
    }  
    return *this; // Return object for chain assignment  
}
```
- Signature of a *Copy Assignment Operator* can be one of:

```
MyClass& operator=(const MyClass& rhs); // Common. No change in Source  
MyClass& operator=(MyClass& rhs); // Occasional. Change in Source
```
- The following *Copy Assignment Operators* are occasionally used:

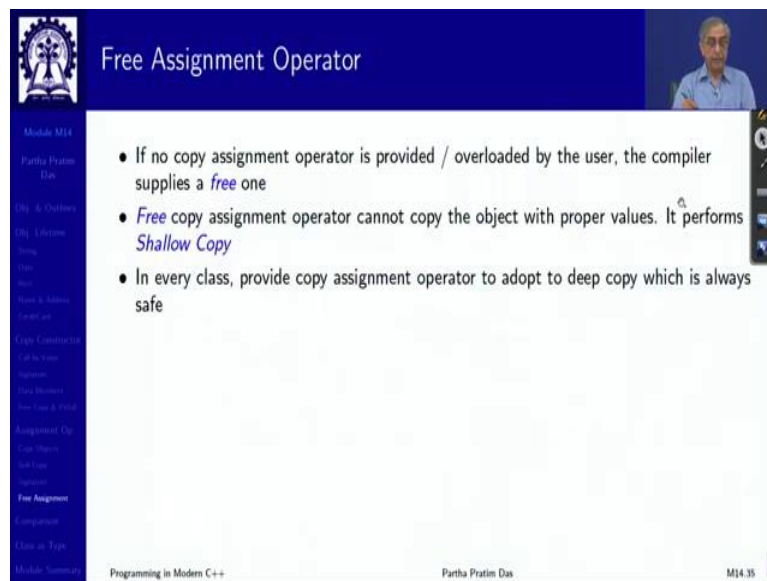
```
MyClass& operator=(MyClass rhs);  
const MyClass& operator=(const MyClass& rhs);  
const MyClass& operator=(MyClass& rhs);  
const MyClass& operator=(MyClass rhs);  
MyClass operator=(const MyClass& rhs);  
MyClass operator=(MyClass& rhs);  
MyClass operator=(MyClass rhs);
```

Programming in Modern C++ Partha Pratim Das M14.34

And this is the self-copy needs to be taken care of. And otherwise, so, this is what turns out to be what is the basic signature of the copy assignment operator, where you check for self-copy, do whatever is required for the deep copy, and then you return this.

You can copy assignment operator could also be of this type, like the copy constructor that is in the first 1 which is common, if the source is not changed. In the second one, the source will get changed. So, I might want to move this, there could be several other signatures, but they are rarely used. So, do not try to use them.

(Refer Slide Time: 33:54)



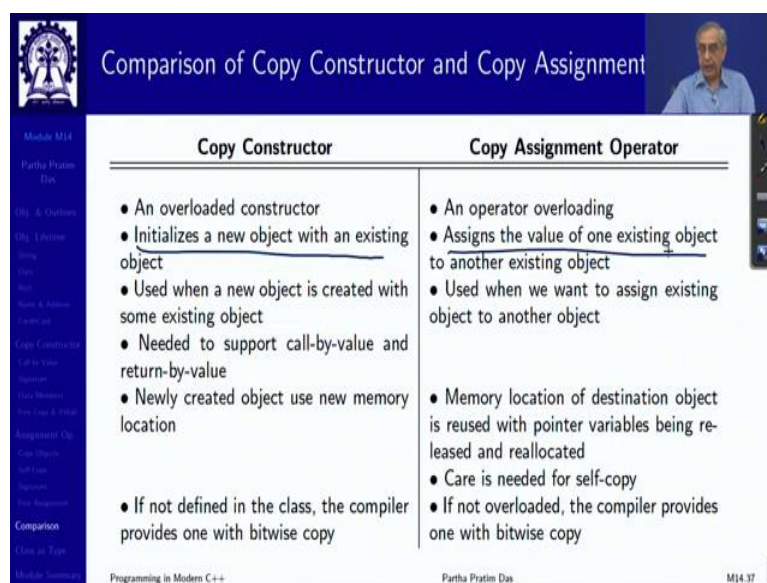
Free Assignment Operator

- If no copy assignment operator is provided / overloaded by the user, the compiler supplies a *free* one
- *Free* copy assignment operator cannot copy the object with proper values. It performs *Shallow Copy*
- In every class, provide copy assignment operator to adopt to deep copy which is always safe

Programming in Modern C++ Partha Pratim Das M14.35

Copy assignment operator could also be, will also be provided free by the compiler and therefore it will come with all the issues of shallow copy that we have talked off.

(Refer Slide Time: 34:06)



Comparison of Copy Constructor and Copy Assignment

Copy Constructor	Copy Assignment Operator
<ul style="list-style-type: none">• An overloaded constructor• <u>Initializes a new object with an existing object</u>• Used when a new object is created with some existing object• Needed to support call-by-value and return-by-value• Newly created object use new memory location• If not defined in the class, the compiler provides one with bitwise copy	<ul style="list-style-type: none">• An operator overloading• <u>Assigns the value of one existing object to another existing object</u>• Used when we want to assign existing object to another object• Memory location of destination object is reused with pointer variables being released and reallocated• Care is needed for self-copy• If not overloaded, the compiler provides one with bitwise copy

Programming in Modern C++ Partha Pratim Das M14.37

Now, before I conclude the just to compare what is the difference both our copying constructor and assignment operators. Now constructor is overloaded, the operator is also overloaded. The basic differences is when you do copy construction, the object does not exist. So, that object has to be created as a copy.

Whereas when you are doing copy assignment, there you already have two objects and you are changing the destination object according to the source object. So, that is a fundamental difference, rest of it is whatever we have discussed in the slides so far.

(Refer Slide Time: 34:50)

Class as a Data-type

- We add the copy construction and assignment to a class being a composite data type in C++

<pre>// declare i to be of int type int i; // initialise i int i = 5; int i = i; int k(j); // print i cout << i; // add two ints int i = 5, j = 6; i+j; // copy value of i to j int i = 5, j; j = i;</pre>	<pre>// declare c to be of Complex type Complex c; // initialise the real and imaginary components of c Complex c = (4, 5); // Ctor Complex c1 = c; // Cctor Complex c2(c1); // Cctor // print the real and imaginary components of c cout << c.re << c.im; OR c.print(); // Method Complex::print() defined for printing OR cout << c; // operator<<() overloaded for printing // add two Complex objects Complex c1 = (4, 5), c2 = (4, 6); c1.add(c2); // Method Complex::add() defined to add OR c1+c2; // operator+() overloaded to add // copy value of one Complex object to another Complex c1 = (4, 5), c2 = (4, 6); c2 = c1; // c2.re <- c1.re and c2.im <- c1.im by copy assignment</pre>
--	---

Programming in Modern C++ Partha Pratim Das M14.39

And finally, you have the class as a, as a type. So, this we had seen earlier. Now with this and this coming in, we have the copy construction which is also common, also available for the built in type. So, you can see that your assignments and like in assignments here you can see that we are actually extending the class and making it more like a perfect type.

(Refer Slide Time: 35:22)

Module Summary

- Copy Constructors**
 - A new object is created
 - The new object is initialized with the value of data members of another object
- Copy Assignment Operator**
 - An object is already existing (and initialized)
 - The members of the existing object are replaced by values of data members of another object
 - Care is needed for self-copy
- Deep and Shallow Copy for Pointer Members**
 - Deep copy allocates new space for the contents and copies the pointed data
 - Shallow copy merely copies the pointer value – hence, the new copy and the original pointer continue to point to the same data

Programming in Modern C++ Partha Pratim Das M14.40

So, with so with this I conclude on the on the module with different semantics of copy construction assignment, and deep and shallow copy. Do practice this very, very thoroughly because this will be critically important in all kinds of things, programs that we write in future. Thank you very much for your attention and we will meet in the next module.