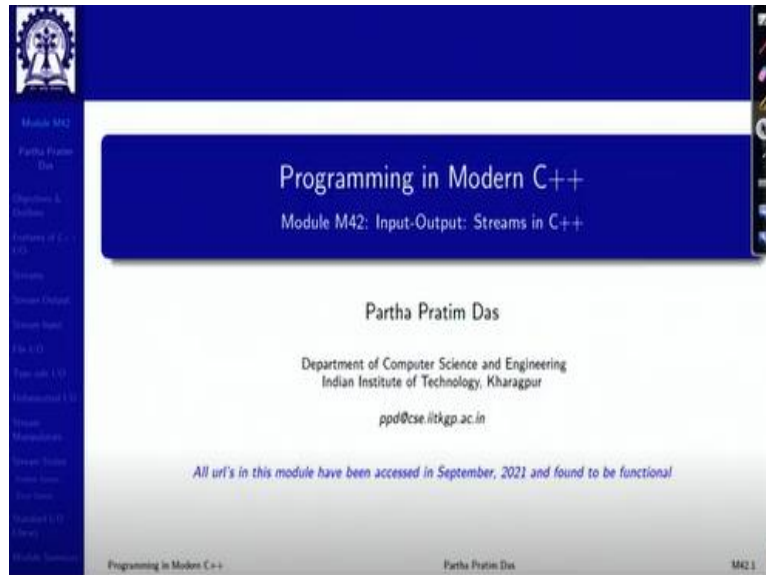


Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 42
Input-Output: Streams in C++

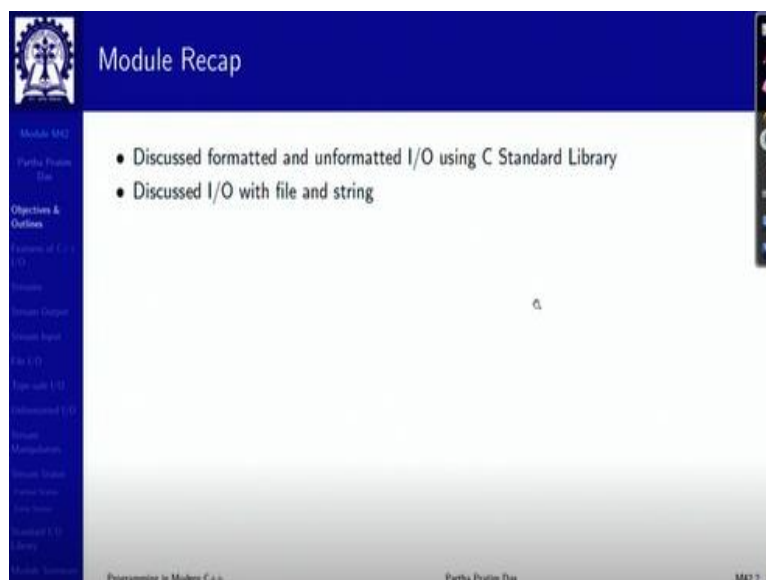
(Refer Slide Time: 0:31)



The slide features a blue header with the IIT KGP logo and a navigation menu on the left. The main content area has a blue title bar with the text "Programming in Modern C++" and "Module M42: Input-Output: Streams in C++". Below this, the presenter's name "Partha Pratim Das" is centered, followed by his affiliation: "Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur" and his email "ppd@cse.iitkgp.ac.in". A note at the bottom states: "All url's in this module have been accessed in September, 2021 and found to be functional". The footer contains "Programming in Modern C++", "Partha Pratim Das", and "M42.1".

Welcome to Programming in Modern C++, we are in week 9 and we are going to discuss module 42 - M42.

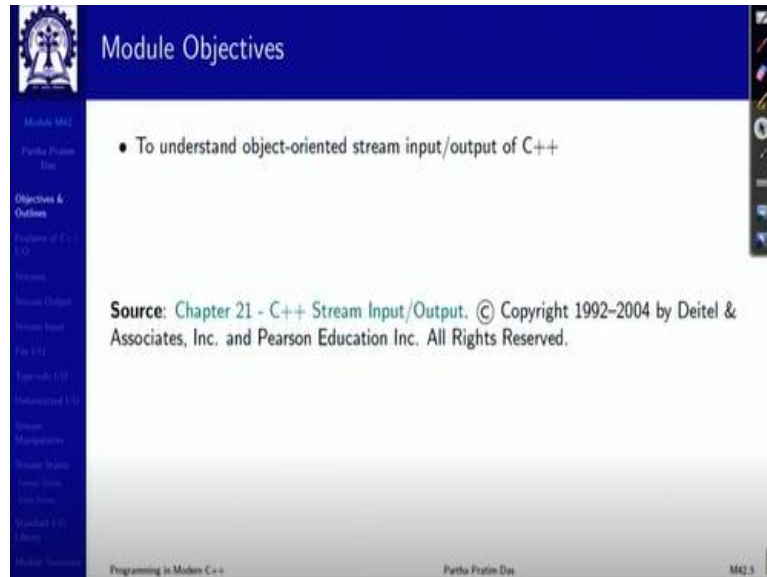
(Refer Slide Time: 0:35)



The slide has a blue header with the IIT KGP logo and a navigation menu on the left. The title "Module Recap" is in the top right of the header. The main content area lists two bullet points: "Discussed formatted and unformatted I/O using C Standard Library" and "Discussed I/O with file and string". The footer contains "Programming in Modern C++", "Partha Pratim Das", and "M42.2".

So, in the last module we have talked about doing formatted and unformatted I/O using C standard library, particularly the component `stdio.h`. We discussed I/O with file and string and specifically introduced the notion of files and streams and their association.

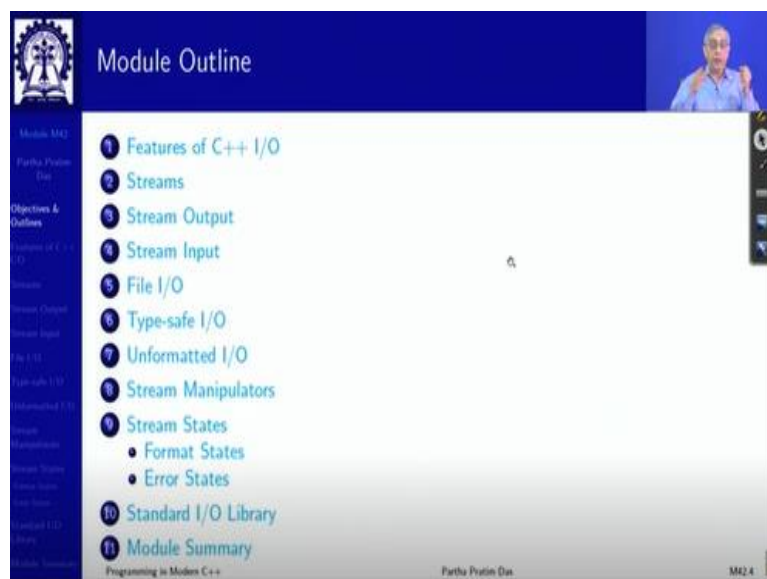
(Refer Slide Time: 0:53)



The slide titled "Module Objectives" features a blue header with a logo on the left. The main content area is white with a blue border. It contains a single bullet point: "To understand object-oriented stream input/output of C++". Below this, the source is cited as "Chapter 21 - C++ Stream Input/Output. © Copyright 1992-2004 by Deitel & Associates, Inc. and Pearson Education Inc. All Rights Reserved." The footer includes "Programming in Modern C++", "Partha Pratim Das", and "MM2.3".

In this module, we are going to discuss about the object-oriented stream input-output of C++. How does C++ do it? Obviously, `stdio.h` is also available in C++ as you know as `cstdio` in the `std` namespace. You can use all those functions, but certainly you can do things in a lot better way in terms of using the object-oriented stream operations in C++.

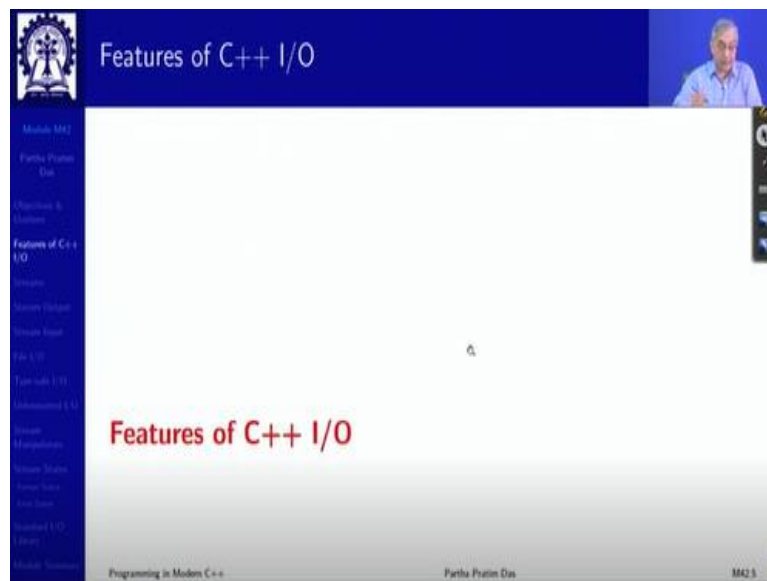
(Refer Slide Time: 1:22)



The slide titled "Module Outline" features a blue header with a logo on the left and a small video inset of a speaker on the right. The main content area is white with a blue border. It contains a numbered list of 11 items: 1. Features of C++ I/O, 2. Streams, 3. Stream Output, 4. Stream Input, 5. File I/O, 6. Type-safe I/O, 7. Unformatted I/O, 8. Stream Manipulators, 9. Stream States (with sub-points: Format States, Error States), 10. Standard I/O Library, 11. Module Summary. The footer includes "Programming in Modern C++", "Partha Pratim Das", and "MM2.4".

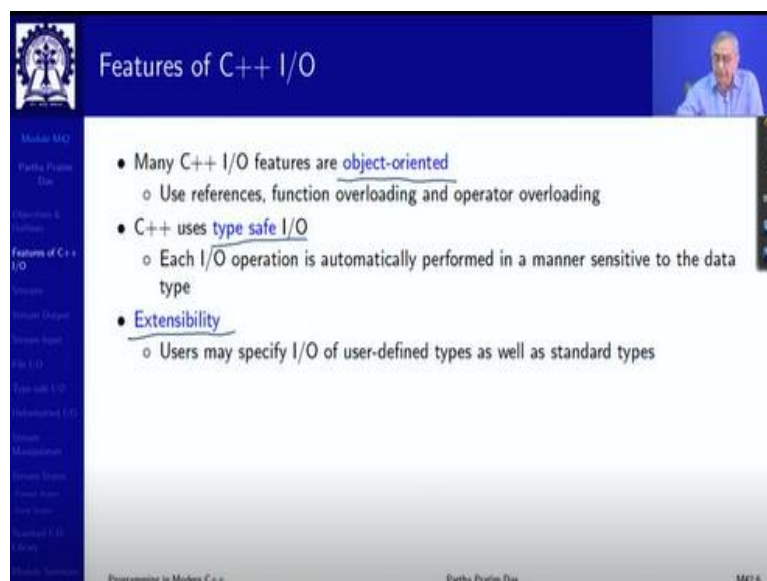
This is the module outline which will be there on your left panel all the time.

(Refer Slide Time: 1:34)



So, let us identify what are the key features of C++ I/O.

(Refer Slide Time: 1:40)



There are three key features: one is most of the C++ I/O features are object-oriented, like in the C stdio.h, you had everything as a collection of functions, procedural. Here, you will have most of the things as classes and objects instantiated for those classes, and you will deal with the member functions of those classes. So, that is, that is what makes it really-really flexible and you know, gives you all the advantages.

The second is, C++ uses type safe I/O that is in C when you are trying to say write a value say you are doing trying to write an integer value (int type of value) you will have to rightly

specify the format in the corresponding position. If you write %d it is good, but if you write %s and try to write an integer value, you will have unpredicted results.

And this correspondence of the type of the value you are writing and the format specification that you have given is to be managed by the programmer, which is a big overhead and in the source of huge number of errors that have happened over time and keeps happening. But in C++ it is type safe, you do not have to specify the format to write as you have seen. The format is picked up based on the type of the data you are going to stream we are going to write. You can just really say that I want to write this I want to read this, compiler knows the type of the variable and accordingly chooses the right format.

And if you want to change that default choice, there are also additional features given to you So, type safety is a big feature. The third that you have is extensibility that is in C. If you have to write or read a particular structure that you have defined, user-defined data type So, to say like complex, you know, you cannot have any specific function specific format string for doing that, because the format strings were created before your particular your program is done. So, you have to take it component by component always break it down to the lowest level of built in types and do the read write this is extremely inconvenient.

Whereas in terms of C++, you can actually use the function overloading we have discussed enough of that already. Particularly the operator overloading in terms of the streaming operators and define your own input and output operators for your own type. And then just keep using it. And it will have all the behaviors of object orientation and type safety as we have for the building types. So, these are the three major features of C++ type of I/O and that makes it really powerful.

(Refer Slide Time: 4:56)

The slide is titled "Streams" and features a blue header with a logo on the left and a small video inset of a speaker on the right. The main content is on a white background with a blue sidebar on the left. The sidebar lists various topics, with "Streams" highlighted. The main text is as follows:

- Stream
 - A transfer of information in the form of a sequence of bytes
 - The term stream is an abstraction of a construct that allows you to send or receive an unknown number of bytes. The metaphor is a stream of water. You take the data as it comes, or send it as needed. Contrast this to an array, for example, which has a fixed, known length.
- I/O Operations
 - Input: A stream that flows from an input device (that is, keyboard, disk drive, network connection) to main memory
 - ▷ `istream` ✓
 - ▷ `ifstream` ✓
 - Output: A stream that flows from main memory to an output device (that is, screen, printer, disk drive, network connection)
 - ▷ `ostream` ✓
 - ▷ `ofstream` ✓

At the bottom of the slide, there is a footer with the text "Programming in Modern C++", "Partho Pratim Das", and "M42 8".

We have already talked about streams So, just a quick reminder that it is a transfer of information in the form of a sequence of bytes. The term stream I said it comes from the natural you know, what a stream and for doing I/O operations here what you have is you have certain stream classes defined. So, if it is an input device then it was represented by `istream`, if it is I mean typically `stdin` or you can have a file defined or associated as an `ifstream` object. You can understand that `i` stands for input `f` stands for file.

Similarly, for output you have `ostream` and `ofstream`. So, these are the classes given in the library, so that you have all that you need to do is to just specify the physical file object or the physical device with the corresponding stream type and everything else will then come from that stream type class. So, you have already seen this but not very consciously though.

(Refer Slide Time: 6:08)

Streams

- I/O operations are a bottleneck
 - The time for a stream to flow is many times larger than the time it takes the CPU to process the data in the stream
- Low-level I/O
 - Unformatted
 - Individual byte unit of interest
 - High speed, high volume, but inconvenient for people
- High-level I/O
 - Formatted
 - Bytes grouped into meaningful units: integers, characters, etc.
 - Good for all I/O except high-volume file processing

Programming in Modern C++ Parthiv Pravin Das MA2 8

Now, again a reminder is I/O operation is a bottleneck because I/O with the disk is very-very slow. So, you can work around this if you have a low level you that is what you do you for individual bytes of interest, you can just put them in the buffer and flush them out high speed high volume can do. But high level I/O which is typically formatted needs to be grouped in terms of the typical data types and they are good for all kinds of meaningful use, because you cannot go down to low level and do things all the time.

So, the streams give you a mechanism by which you can do a high level I/O with its advantages, but underlying it uses the buffers and all the padding and all those to make sure that you get the efficiency of the low level I/O that is a basic process of the stream.

(Refer Slide Time: 7:12)

<iostream> Header Files

- **iostream library**
 - **<iostream>**: Contains `cin`, `cout`, `cerr` and `clog` objects
 - **<iomanip>**: Contains parameterized stream manipulators

Programming in Modern C++ Parthiv Pravin Das MA2 10

So, to start with, let me introduce the stream libraries that we have. You are already familiar with I/O stream which is this library. You can see here in this diagram. I am trying to show the where basically there ISA hierarchy of the major classes that are involved. So, you can see that you have your I/O stream is one com. So, let me explain the notation. So, whatever you see in white, these are your classes. So, you have a hierarchy diagram amongst them. What you see in terms of boxed and red names are the standard library components.

So, these are the headers that include the definitions of these corresponding classes. Whatever is in blue is what we will typically include in a user program. Whatever is green is included from these files, but we do not need to directly include them.

So, if we if you recall into this, you will know that we have always started by including iostream, So, what do you get? If you include iostream then you actually get four streams, which we have always used the cin for input, cout for output, and there are two more cerr for error reporting and clog for log reporting.

Since we have not worked with the files, we have not included this. But this will be required to be included if I want to work with files, input and output both. So, it has fstream, it has ifstream which is input file stream, it is output file stream and iostream. When you include iostream it actually in turn includes these two headers which we do not have to specify which has the istream and the iostream for input as well as output.

Now I have mentioned here another which is iomanip. It is a little unnatural name what it means input-output manipulation (manip comes from manipulator). So, if you include iostream we for every data type you get a default formatting information in terms of what is the size, what is the weight, written alignment, and so on so forth.

But if you want to change that, you want to specifically widen the size shorten the size, you want a left justification, you want a right justification all of this, then you need to do this stream manipulation through this iomanip since we have not done this before, we did not require this earlier. So, as you can see, to summarize that, you will all that you will need to remember is including iostream for standard input-output, including fstream if you are doing file input-output and including iomanip if you want to manipulate the formatting for finer details.

(Refer Slide Time: 10:43)

Stream I/O Classes and Objects

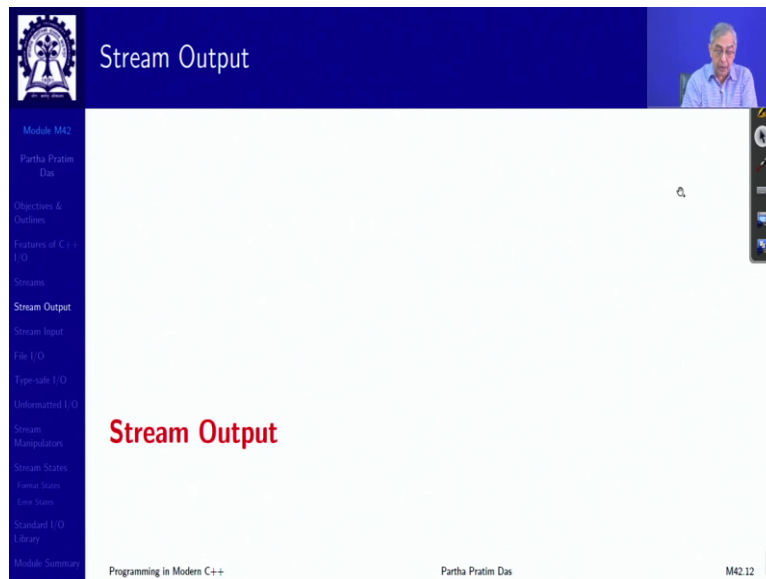
- **ios:**
 - `istream` and `ostream` inherit from `ios`
 - ▷ `iostream` inherits from `istream` and `ostream`
- **<< (left-shift operator)**
 - Overloaded as stream insertion operator
- **>> (right-shift operator)**
 - Overloaded as stream extraction operator
 - Both operators used with `cin`, `cout`, `cerr` and `clog`, and with user-defined stream objects
- **istream: input streams**
 - `cin >> grade;`
 - ▷ `cin` knows what type of data is to be assigned to `grade` (based on the type of `grade`)
- **ostream: output streams**
 - `cout << grade;`
 - ▷ `cout` knows the type of data to output
 - `cerr << errorMessage;` ✓
 - ▷ **Unbuffered** - prints `errorMessage` immediately
 - `clog << errorMessage;` ✓
 - ▷ **Buffered** - prints `errorMessage` as soon as output buffer is full or flushed

Programming in Modern C++ Partha Pratim Das MA2.11

So, with that, let us take a quick look into the stream classes and objects. So, at the top you have `ios` which is the basic `istream` and `ostream` come from there you have, in that you have for the two operators which look like left shift operator for output streaming right shift operator for the input streaming. Then you have `istream`, which has `cin`. Then you have with that you have `ostream` with which you can do up to `cout` you have seen this several times.

You can at the same time do `cerr` for printing unbuffered messages immediately, like `cout`, as I said is buffered. So, you may write quite a few things, but it comes from the console, maybe at a later point of time. But `cerr` it will come immediately because error is very important to report. Whereas there is a logging `clog` where you can just log the things that are happening, so, it is not unbuffered it is buffered because you do not need to look at the log at every point, whenever the buffer is full, like the `cout` when the buffer is full or when needs to be flushed, it is put together.

(Refer Slide Time: 11:56)



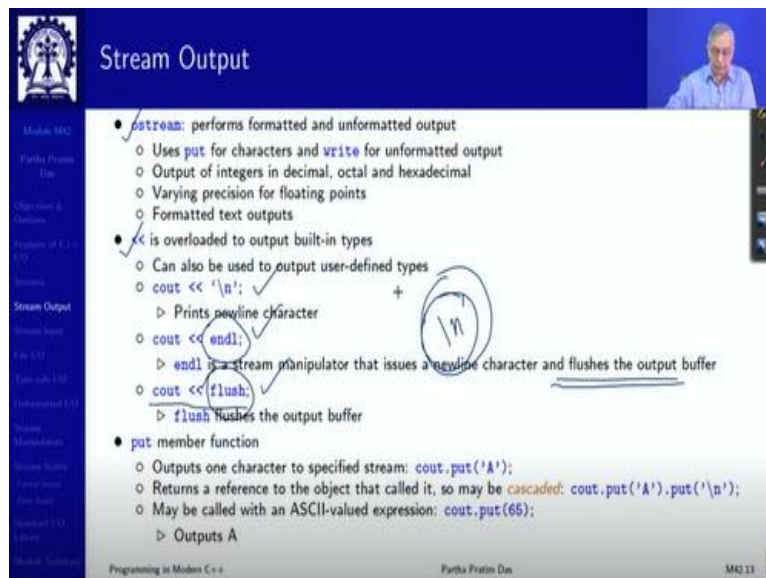
Slide 1: Stream Output

Module M42
Partha Pratim Das

Objectives & Outlines
Features of C++
I/O
Streams
Stream Output
Stream Input
File I/O
Type-safe I/O
Unformatted I/O
Stream Manipulators
Stream States
Error States
Low States
Standard I/O Library
Module Summary

Stream Output

Programming in Modern C++ Partha Pratim Das M42.12



Slide 2: Stream Output

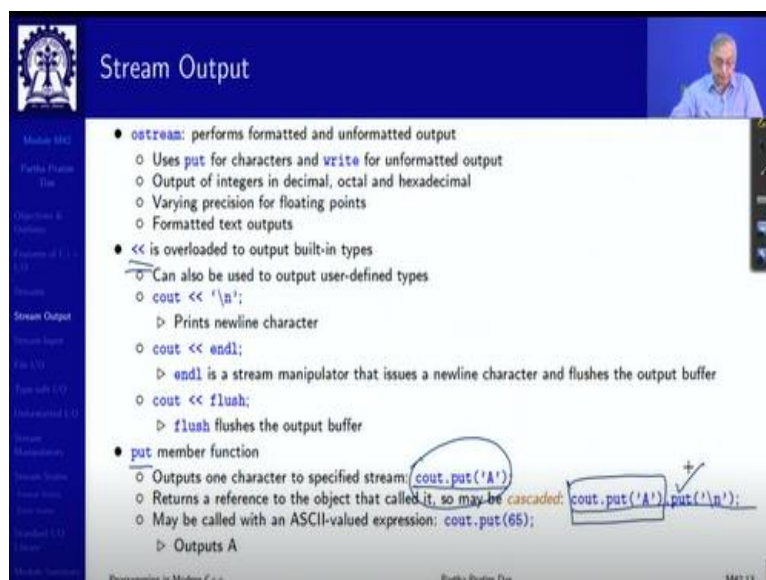
Module M42
Partha Pratim Das

Objectives & Outlines
Features of C++
I/O
Streams
Stream Output
Stream Input
File I/O
Type-safe I/O
Unformatted I/O
Stream Manipulators
Stream States
Error States
Low States
Standard I/O Library
Module Summary

Stream Output

- **ostream**: performs formatted and unformatted output
 - Uses `put` for characters and `write` for unformatted output
 - Output of integers in decimal, octal and hexadecimal
 - Varying precision for floating points
 - Formatted text outputs
- **<<** is overloaded to output built-in types
 - Can also be used to output user-defined types
 - `cout << '\n';`
 - ▷ Prints newline character
 - `cout << endl;`
 - ▷ `endl` is a stream manipulator that issues a newline character and flushes the output buffer
 - `cout << flush;`
 - ▷ `flush` flushes the output buffer
- **put** member function
 - Outputs one character to specified stream: `cout.put('A');`
 - Returns a reference to the object that called it, so may be cascaded: `cout.put('A').put('\n');`
 - May be called with an ASCII-valued expression: `cout.put(65);`
 - ▷ Outputs A

Programming in Modern C++ Partha Pratim Das M42.13



Slide 3: Stream Output

Module M42
Partha Pratim Das

Objectives & Outlines
Features of C++
I/O
Streams
Stream Output
Stream Input
File I/O
Type-safe I/O
Unformatted I/O
Stream Manipulators
Stream States
Error States
Low States
Standard I/O Library
Module Summary

Stream Output

- **ostream**: performs formatted and unformatted output
 - Uses `put` for characters and `write` for unformatted output
 - Output of integers in decimal, octal and hexadecimal
 - Varying precision for floating points
 - Formatted text outputs
- **<<** is overloaded to output built-in types
 - Can also be used to output user-defined types
 - `cout << '\n';`
 - ▷ Prints newline character
 - `cout << endl;`
 - ▷ `endl` is a stream manipulator that issues a newline character and flushes the output buffer
 - `cout << flush;`
 - ▷ `flush` flushes the output buffer
- **put** member function
 - Outputs one character to specified stream: `cout.put('A');`
 - Returns a reference to the object that called it, so may be cascaded: `cout.put('A').put('\n');`
 - May be called with an ASCII-valued expression: `cout.put(65);`
 - ▷ Outputs A

Programming in Modern C++ Partha Pratim Das M42.13

So, coming to stream output, there is not much to add. It is on ostream this operator. These are the typical ways to write just I would like to mention that flush. Suppose, if you do if you are writing, then it is being written on the buffer. And you are not being able to see it on the console, because it is just being written on the buffer. So, if you want to make sure that whatever you have written is shown on the console, then you can do it flush and the way to do this is to just output flush like here you do a flush along with a newline.

So, endl is just not a newline, endl is not equivalent to this. endl or end line is actually a manipulator which puts a newline, but after that it flushes the output buffer. If you just use new line then it you may find that actually the newline has been put to the cout, but you are not seeing it on the console.

Similarly, for you know kind of unformatted or character I/O you have put function member function which you can call on the cout. As you can cascade your formatted I/O your object I/O you can also cascade the character I/O like this cout.put. So, basically you can make out that the result this will put A on to the console and it will return cout again. So, this again becomes cout. Exactly the same behavior, which we see in terms of this operator you have seen we have written that we have seen that. So, you can again do a putc and keep on doing this.

(Refer Slide Time: 14:01)

```
#include <iostream>
using namespace std;
int main() {
    int i = 17; long l = 0x012a78cb; // 19660661
    long long unsigned int 164 = 0x012a78cb2597ac3d; // 84012356964166717
    float f = 15.0 / 7; double d = 15.0 / 7;
    char c = 'x'; const char *s = "ppd";
    int *p = &i;

    cout << i << " "; // int // 17 Optional dec can be used
    cout << hex << i << endl; // hex // 11
    cout << oct << i << endl; // oct // 21
    cout << l << " "; // long // 19660661
    cout << 164 << " "; // int 64 // 84012356964166717
    cout << f << " "; // float // 2.14286
    cout << d << " "; // double // 2.14286
    cout << c << " "; // char // x
    cout << s << " "; // string // ppd
    cout << (void*)(s) << endl; // pointer // 0x55c82522009 // Address of 1st character of the string
    cout << p << " "; // pointer // 0x7fff9a17cf68
}
```

• An integer (int) may be printed in decimal (dec. by default), octal (oct) or hexadecimal (hex) format
• A char* pointer prints the string. To print the pointer value, cast to void* by static_cast<const void*> or (void*)

Programming in Modern C++ Partha Pratim Das MA2.14

```

#include <iostream>
using namespace std;
int main() {
    int i = 17; long l = 0x012a78cb; // 19560651
    long long unsigned int l64 = 0x012a78cb2597ac3d; // 84012356964166717
    float f = 15.0 / 7; double d = 15.0 / 7;
    char c = 'x'; const char *s = "ppd";
    int *p = &i;

    cout << i << " "; // int // 17 Optional dec may be used
    cout << hex << i << endl; // hex // 11
    cout << oct << i << endl; // oct // 21
    cout << l << " "; // long // 19560651
    cout << l64 << " "; // int 64 // 84012356964166717
    cout << f << " "; // float // 2.14286
    cout << d << " "; // double // 2.14286
    cout << c << " "; // char // x
    cout << s << " "; // string // ppd
    cout << (void*)(s) << endl; // pointer // 0x7ffff9a17c708 // Address of 1st character of the string
    cout << p << " "; // pointer // 0x7ffff9a17c708
}

```

• An integer (int) may be printed in decimal (dec; by default), octal (oct) or hexadecimal (hex) format
 • A char* pointer prints the string. To print the pointer value, cast to void* by static.cast<const void*> or (void*)

So, this is the basic streaming output. The earlier example I showed in the module 41 regarding using the printf the same example I have now written in terms of cout and you can see what all are getting printed. The only two points to note here like this you will get with %d in printf, and this you will get with %x, this will get with %o. You do not have these in C++ because it is being derived from that type.

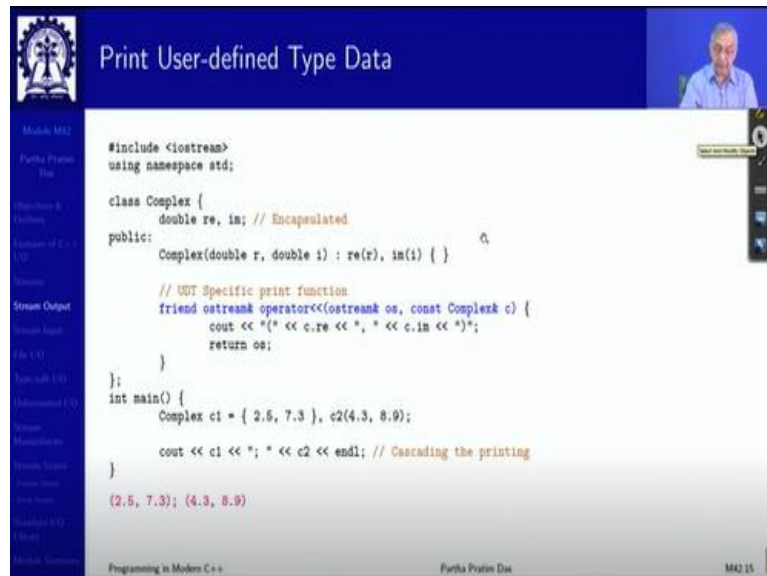
So, if I write i then the default format is %d. And whatever %d will do write it in decimal. So, if I want to write it as hex what do I do? I just do this very nicely just yes cout hex than this. So, what it does is? It streams hex first onto the cout, sets it in the hexadecimal mode and then when i comes then i will be printed in the hexadecimal mode. So, you will get the printout, print as 11, rather not 11, one one because it is a, it is a hexadecimal value.

So, these are again the stream manipulators. So, you can very simply like we have seen endl, we have seen flush, hex, oct or other manipulators that you can very frequently use when you have the same type of data to be printed in different other forms that you want.

Here, if you have this character pointer, you can print it as a string by using %s. But if you want that pointer value, you can print this that by %p. But here, when you do it with cout in C++, when you do it ostream the type it knows for s is char* and char* is a string C style string understanding for here. So, if you output s, then it will be output as a string, you will get this string. So, how do you get the value of s value of this pointer which you could get as %p. You cannot get that directly, so all that you need to do is to erase the type information from this pointer.

So, you can use a C style casting to void* which again you should not do, you should do a C++ style casting cast the constant char* to const void* and then use it and in that case it will not know what type of value it is pointing to. So, it will not try to interpret that it will simply print that value and you will get the pointer that is address of the first character. So, these are the two basic differences from whatever we know in C.

(Refer Slide Time: 17:20)



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

class Complex {
public:
    double re, im; // Encapsulated
    Complex(double r, double i) : re(r), im(i) {}

    // UDT Specific print function
    friend ostream operator<<(ostream os, const Complex c) {
        cout << "(" << c.re << ", " << c.im << ")";
        return os;
    }
};

int main() {
    Complex c1 = { 2.5, 7.3 }, c2(4.3, 8.9);

    cout << c1 << "; " << c2 << endl; // Cascading the printing
}

(2.5, 7.3); (4.3, 8.9)
```

In terms of user-defined type, obviously, you have a big advantage because now you can define overload your output streaming operator and put it as a friend function in the class and cascade and write the user-defined objects exactly as the built in objects are written. I will not I am not going through these details, because you have already we have already discussed this at length earlier.

(Refer Slide Time: 17:49)

```
#include <cstdio>
#include <iostream>
using namespace std;
int main() {
    int i = 17;
    long l = 0x012a78cb; // 19560651
    long long unsigned int l64 = 0x012a78cb2597ac3d; // 84012356964166717
    float f = 15.0 / 7;
    double d = 15.0 / 7;
    char c = 'x';
    const char *s = "ppd";
    int *p = &i;
    cout << i << " ";
    cout << hex << i << endl;
    cout << oct << i << endl;
    cout << l << " ";
    cout << l64 << " ";
    cout << f << " ";
    cout << d << " ";
    cout << c << " ";
    cout << s << " ";
    cout << p << " ";

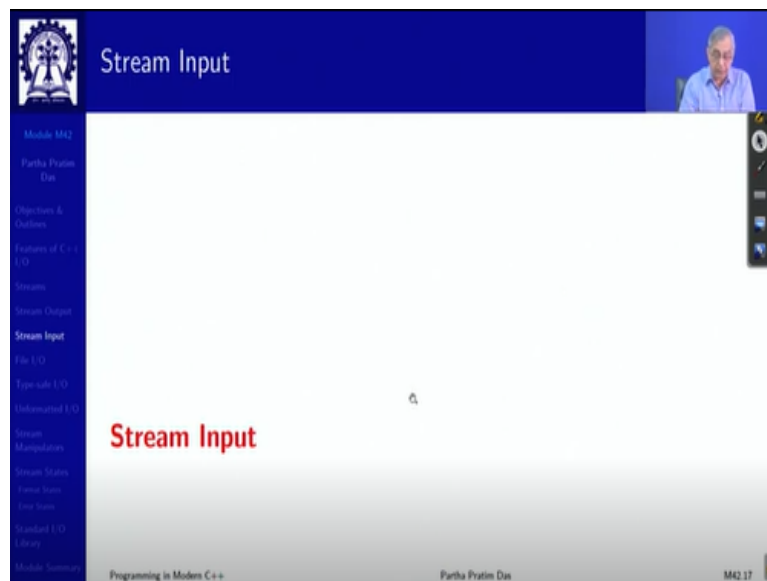
    printf("%d\n", i); // dec // 17 17 Opt. dec may be used in C++
    printf("%x\n", i); // hex // 11 11
    printf("%o\n", i); // oct // 21 21
    printf("%ld\n", l); // long // 19560651 19560651
    printf("%lld\n", l64); // int 64 // 84012356964166717 84012356964166717
    printf("%f\n", f); // float // 2.14286 2.142857
    printf("%lf\n", d); // double // 2.14286 2.142857
    printf("%c\n", c); // char // 'x'
    printf("%s\n", s); // string // p p d
    printf("%p\n", p); // pointer // 0x7ffc28102988 0x7ffc28102988
}
```

• Note the use of hex and oct in C++ and the difference in default precision for float and double between C++ and C.

This is just a comparison of how it writes between C++ and C. If you write the same similar things in both, you will find that, these all actually write identically. Though there are some differences in cases. Here when you are writing float or double the default precision of %f for %lf is different from the default precision being taken.

Of course, you can change that by streaming an appropriate manipulator of precision or set precision. There is a set precision manipulated by setting that you can change this precision and make them identical otherwise, all of this will look very similar and you can you can very easily when you write it side by side, you can very easily see that all these information which you will have to have in your mind or to go back to module 41 slides and check the tables that have given and carefully construct you do not need to have any one of those. Here are very simple you just say these are value I want to output that is it. So, that gives you a big-big advantage.

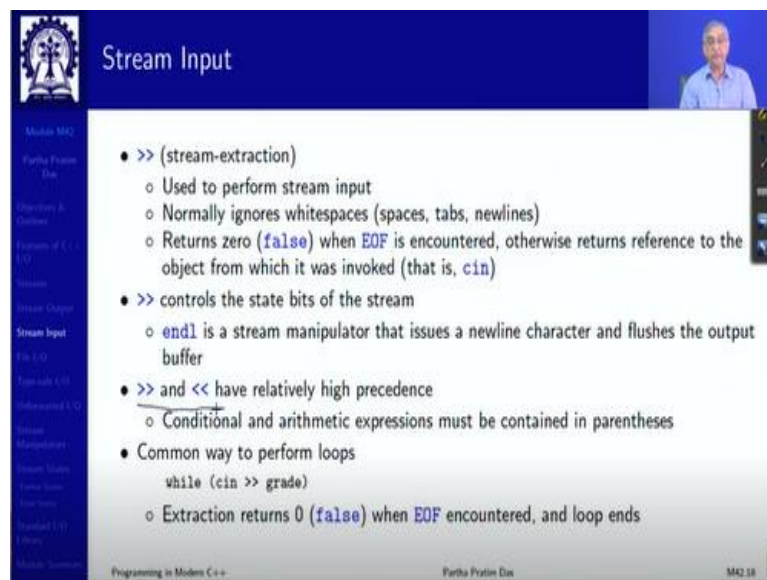
(Refer Slide Time: 19:20)



Stream Input

Module MA2
Partha Pratim Das
Objectives & Outcomes
Features of C++
Streams
Stream Output
Stream Input
File I/O
Template I/O
Advanced I/O
Stream Manipulators
Stream States
User-Defined Stream States
Standard I/O Library
Module Summary

Programming in Modern C++ Partha Pratim Das MA2.17



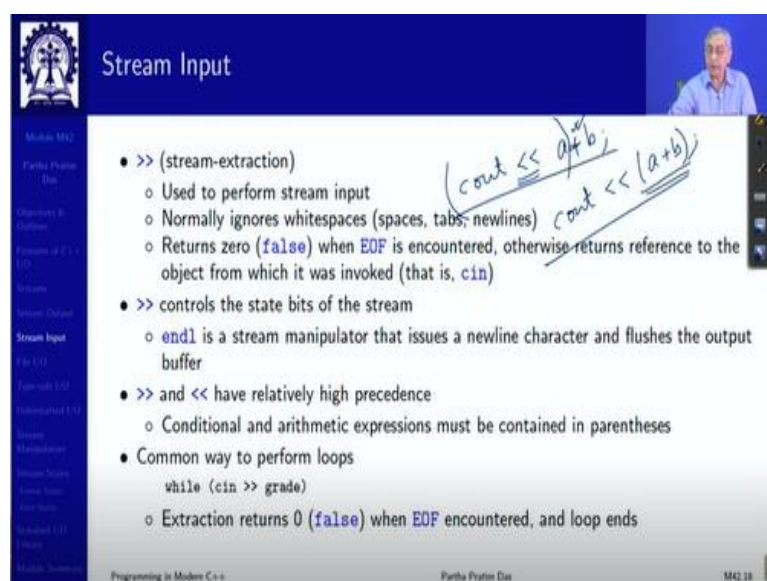
Stream Input

- `>>` (stream-extraction)
 - Used to perform stream input
 - Normally ignores whitespaces (spaces, tabs, newlines)
 - Returns zero (`false`) when EOF is encountered, otherwise returns reference to the object from which it was invoked (that is, `cin`)
- `>>` controls the state bits of the stream
 - `endl` is a stream manipulator that issues a newline character and flushes the output buffer
- `>>` and `<<` have relatively high precedence
 - Conditional and arithmetic expressions must be contained in parentheses
- Common way to perform loops

```
while (cin >> grade)
```

 - Extraction returns 0 (`false`) when EOF encountered, and loop ends

Programming in Modern C++ Partha Pratim Das MA2.18



Stream Input

- `>>` (stream-extraction)
 - Used to perform stream input
 - Normally ignores whitespaces (spaces, tabs, newlines)
 - Returns zero (`false`) when EOF is encountered, otherwise returns reference to the object from which it was invoked (that is, `cin`)
- `>>` controls the state bits of the stream
 - `endl` is a stream manipulator that issues a newline character and flushes the output buffer
- `>>` and `<<` have relatively high precedence
 - Conditional and arithmetic expressions must be contained in parentheses
- Common way to perform loops

```
while (cin >> grade)
```

 - Extraction returns 0 (`false`) when EOF encountered, and loop ends

Handwritten notes:
`cout << a+b;`
`cout << (a+b);`

Programming in Modern C++ Partha Pratim Das MA2.18

Anyway, if it comes to input, it is obviously very similar. It is a stream extraction, as long as you do not get to the end of the stream or end of file, the extraction will happen successfully and the input streaming operator can be can control the state of the bits that are coming in. And obviously at this point, it will be good to note that both of these operators have pretty high precedence. So, if you have some conditional or some, you know expressions arithmetic expression.

For example, if you want to suppose I want to write `cout << a + b`. If I write this it will be interpreted as because this streaming operator has higher precedence than plus. So, this obviously is not what I am meaning. So, it is always good that when you write expressions in your input or output streaming, always put them in parenthesis to make it safe. So, now this will happen and only that result will be output to the stream. Similar thing will happen for input as well.

(Refer Slide Time: 20:48)

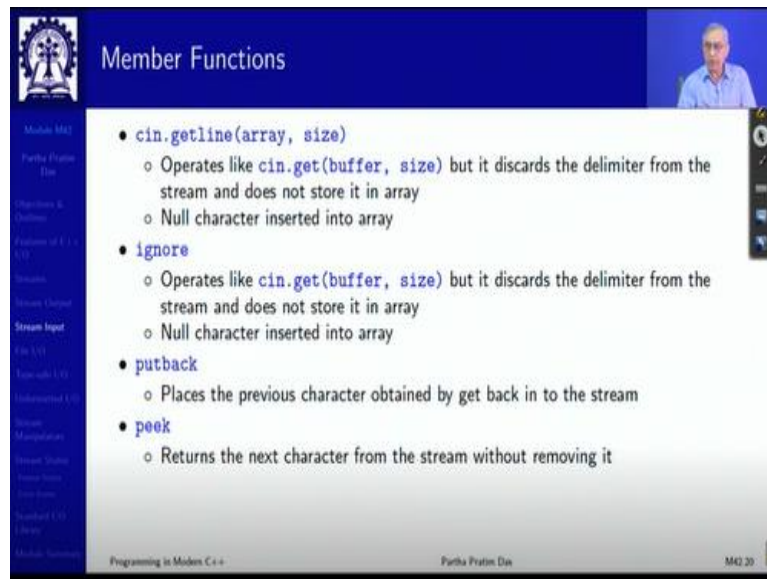
Member Functions

- `cin.eof()`
 - returns true if end-of-file has occurred on `cin`
- `cin.get()`
 - inputs a character from stream (even white spaces) and returns it
- `cin.get(c)`
 - inputs a character from stream and stores it in `c`
- `cin.get(array, size)`
 - Accepts 3 arguments: array of characters, the size limit, and a delimiter (default of `'\n'`)
 - Uses the array as a buffer
 - When the delimiter is encountered, it remains in the input stream
 - Null character is inserted in the array
 - Unless delimiter flushed from stream, it will stay there

Programming in Modern C++ | Partha Pratim Das | MA2.18

There are a number of member functions for `cin` as well, which, like `eof` tells you, if you are at the end of the file, which you need to know, you can have character I/O using `get` or `getc`, `get array`. So, these are the different ways to do the corresponding operations in the `istream` domain.

(Refer Slide Time: 21:11)



The slide is titled "Member Functions" and lists several functions with their descriptions:

- `cin.getline(array, size)`
 - Operates like `cin.get(buffer, size)` but it discards the delimiter from the stream and does not store it in array
 - Null character inserted into array
- `ignore`
 - Operates like `cin.get(buffer, size)` but it discards the delimiter from the stream and does not store it in array
 - Null character inserted into array
- `putback`
 - Places the previous character obtained by `get` back in to the stream
- `peek`
 - Returns the next character from the stream without removing it

The slide also features a navigation menu on the left, a logo in the top left, and a small video feed of the presenter in the top right. The footer contains the text "Programming in Modern C++", "Partha Pratim Das", and "M42 20".

You can read a line by `getline`, you can ignore some, you can you have read character, you can put it back, you can see what is the character you are about to read without actually removing it from the stream and so on. So, there are number of these functions, you can use them as an when you so, whenever you need to do something just look up, you will find that in I mean almost certainty I can say that it will be available in the standard library in the `istream` or `ostream`, `ifstream` or `ofstream` for you directly.

(Refer Slide Time: 21:32)



The slide is titled "File I/O" and is mostly blank, with the title "File I/O" displayed in red text in the center. The slide features a navigation menu on the left, a logo in the top left, and a small video feed of the presenter in the top right. The footer contains the text "Programming in Modern C++", "Partha Pratim Das", and "M42 21".

The slide is titled "Input / Output with Files" and features a small video inset of a speaker in the top right corner. The main content is a list of bullet points and code snippets:

- Open**
 - Like in C, files need to be first opened and associated with a stream


```
ofstream myfile; // Output stream
myfile.open("example.txt"); // Open: Associate file example.txt to output stream myfile

ofstream myfile("example.txt"); // Output stream opened and associated
myfile.is_open(); // Check if open has worked correctly
```
 - Unlike C (where stream is a pointer), stream is an object in C++
 - Unlike C (where mode is specified by a string flag), stream object itself is of input or output types
- Read / Write**
 - Like in C, we perform formatted or unformatted I/O on an open stream (file)
 - Unlike C (where functions for formatted I/O are variadic and needs explicit format specification), objects are read / written using streaming operators for the data types
- Close**
 - Like in C, streams need to be closed when done and disassociated from the file


```
myfile.close(); // Close: Flush stream to file and disassociate from stream
```
- Binary Files**
 - Use `ios::binary` flag in the opening mode.

At the bottom of the slide, it says "Programming in Modern C++" and "Partha Pratim Das" on the left, and "MA2 22" on the right.

So, now, let me just show you the final total example with working with the file in C++. So, like in C, you need to open the file that is you need to associate the file with the stream. So, if I am trying to do this, say for output, I am showing the example for output. I declare my stream object myfile and that is of type ofstream, output file stream very simple. And then my file has a method open to which I can pass a name.

So, this simply does the association if it cannot, then it will throw an exception. An alternate way of doing that would be to do it at the time of construction itself that is you can make out this construction is happening by default constructed, but ofstream also has a constructor which takes the name of the file as a string. So, you can directly to the opening along with the creation of the stream object. So, this will happen and then you can check if it has been if a file has been open correctly, this will give you a true if it has been opened correctly otherwise false.

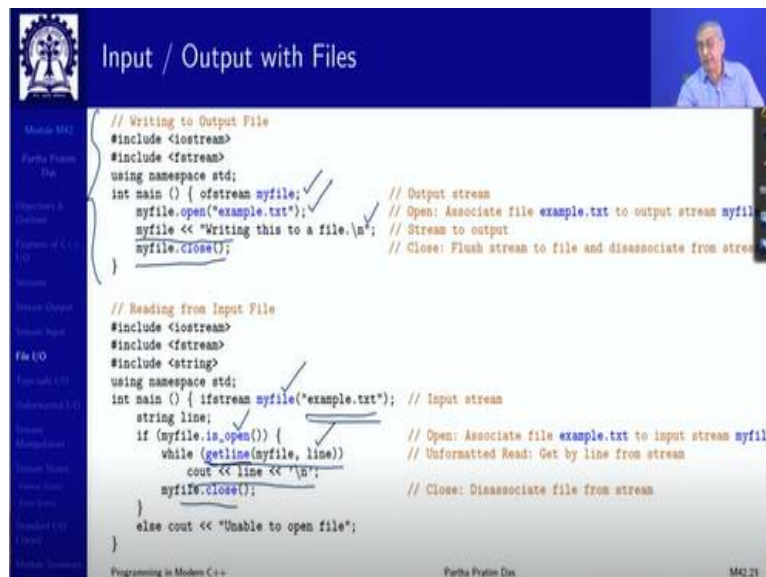
So, this makes it really very simple. You do not have to do all these modes and all that in terms of see if you see that the stream was just a file pointer, which did not have any specific type for input or output. So, it is possible that you have multiple file pointers open and you mistakenly use an input file pointer for doing a fprintf or the vice versa these kinds of things will not happen here because your corresponding types are radically different.

So, you do not need to set a mode because this itself tells you the mode. You do not need to say that this is output, because this will itself tell you that this is output if you have to do input, then you have to create a stream which is an object of ifstream class and the mode will be the input one.

So, this is a I mean the clean object-oriented solution. After the opening has been done, that is association of the file and the stream has happened then you keep on using the stream keep on writing or reading whatever you have planned for in the formatted or unformatted manner to that using the streaming operators.

And finally, when you are done then you close. So, like in C we had had the close fclose function which takes the FILE* pointer and closes dissociates here you have the closed member function on the corresponding type of the stream object, and you just invoke that. If you want to specifically open a binary file, then you have a flag ios::binary, which you will have to specify when you are opening the file or you are creating the file as a second parameter, if not specified, it opens it as a text file that is a default behavior.

(Refer Slide Time: 25:22)



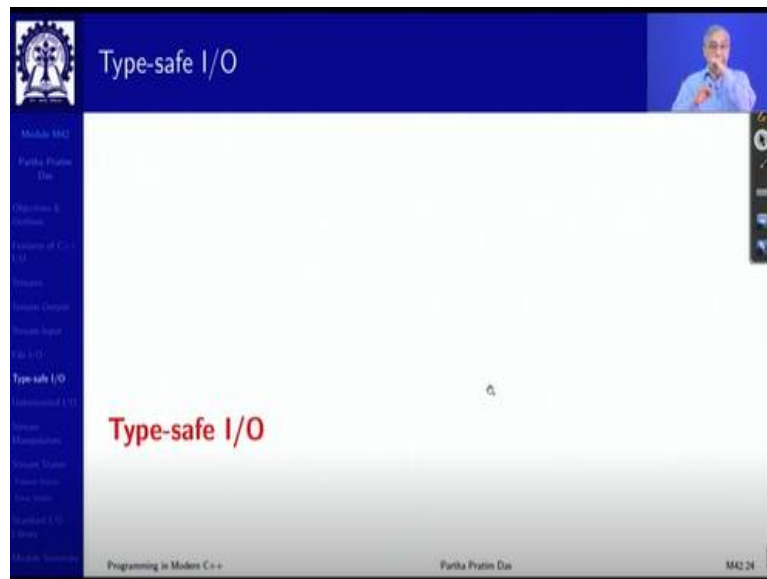
```
// Writing to Output File
#include <iostream>
#include <fstream>
using namespace std;
int main () { ofstream myfile; // Output stream
  myfile.open("example.txt"); // Open: Associate file example.txt to output stream myfile
  myfile << "Writing this to a file.\n"; // Stream to output
  myfile.close(); // Close: Flush stream to file and dissociate from stream
}

// Reading from Input File
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main () { ifstream myfile("example.txt"); // Input stream
  string line;
  if (myfile.is_open()) { // Open: Associate file example.txt to input stream myfile
    while (getline(myfile, line)) // Unformatted Read: Get by line from stream
      cout << line << '\n';
    myfile.close(); // Close: Dissociate file from stream
  }
  else cout << "Unable to open file";
}
```

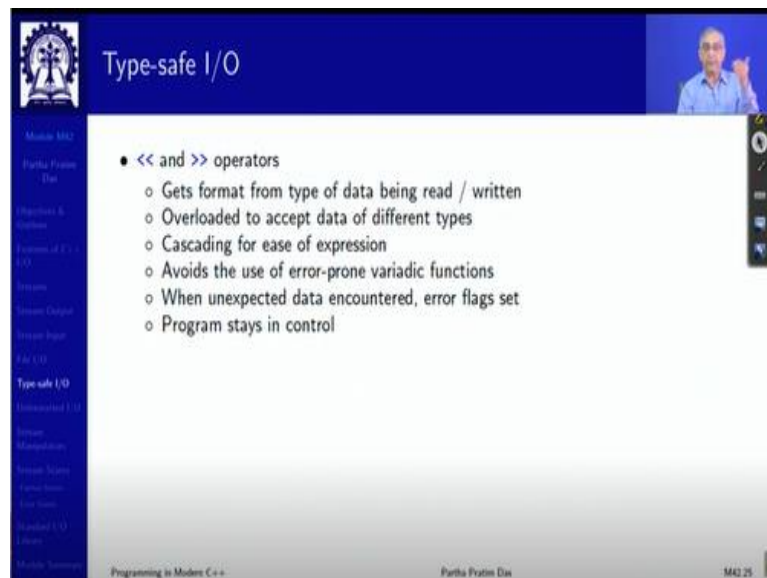
So, here is a small example, actually two examples one is writing to output file. So, I create a default stream object. I open a file and do the association. I do a streaming to it exactly as I do in terms of cout. And that is written to that file, I do a close which will by which the association is broken, and the buffer will be flushed and the file will be closed.

Similarly, here, I am opening a stream with my given file at the construction itself. I check if it is open. If it is open, then I am reading from my file, that is what I have written here, I am reading that and I output that so, I am reading it as a as a line. So, this is a line buffer, I read it as a line buffer and output that to the stdout. When I am done, I close my input file. This is a simple way everything else besides these few lines, everything else is exactly as you did in case of cout or cin.

(Refer Slide Time: 26:36)



The slide is titled "Type-safe I/O" and features a blue header with a logo on the left and a small video inset of the speaker on the right. A vertical navigation menu is on the left side. The main content area is white with the text "Type-safe I/O" in red. At the bottom, it says "Programming in Modern C++", "Purba Pratim Das", and "MA2 24".

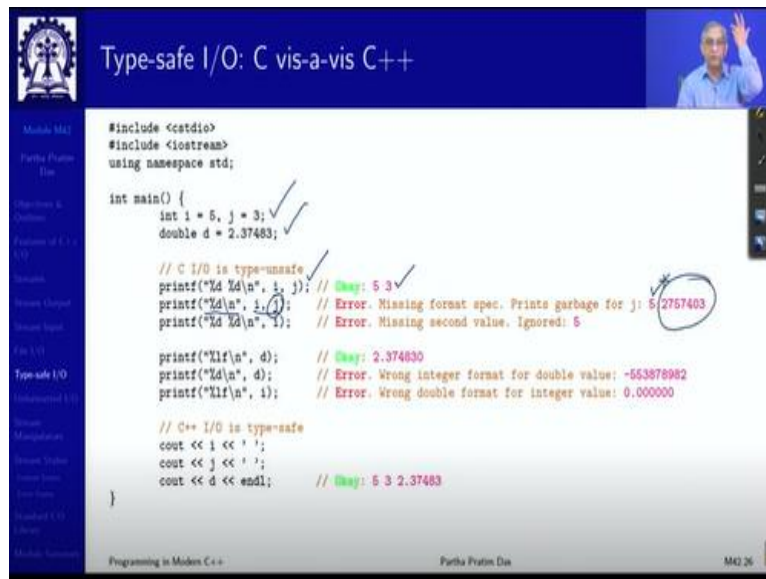


The slide is titled "Type-safe I/O" and features a blue header with a logo on the left and a small video inset of the speaker on the right. A vertical navigation menu is on the left side. The main content area is white with a bulleted list of points. At the bottom, it says "Programming in Modern C++", "Purba Pratim Das", and "MA2 25".

- << and >> operators
 - Gets format from type of data being read / written
 - Overloaded to accept data of different types
 - Cascading for ease of expression
 - Avoids the use of error-prone variadic functions
 - When unexpected data encountered, error flags set
 - Program stays in control

Now, the most critical thing is that the I/O in C++ is type safe that means it is very difficult to make errors in the I/O. It gets the format from the type of the data I have already mentioned, and for all built in types, the overloads of this operators are already given. Cascading makes the ease of expression you do not have to correspond the format with the type of you know data listed you just keep on saying one data after the other it happens automatically. So, it avoids the use of the error-prone variadic that is variable number of argument functions like printf, scanf and so on so, makes it and makes it much easier to use.

(Refer Slide Time: 27:26)



Slide 1: Type-safe I/O: C vis-a-vis C++

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

int main() {
    int i = 5, j = 3;
    double d = 2.37483;

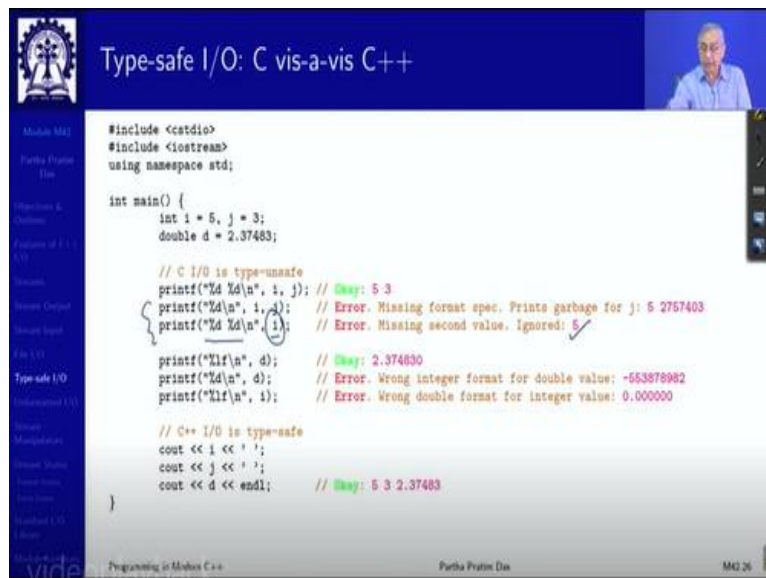
    // C I/O is type-unsafe
    printf("%d %d\n", i, j); // Okay: 5 3
    printf("%d\n", i); // Error: Missing format spec. Prints garbage for j: 5 2757403
    printf("%d %d\n", i); // Error: Missing second value. Ignored: 5

    printf("%lf\n", d); // Okay: 2.374830
    printf("%d\n", d); // Error: Wrong integer format for double value: -553878982
    printf("%lf\n", i); // Error: Wrong double format for integer value: 0.000000

    // C++ I/O is type-safe
    cout << i << ' ';
    cout << j << ' ';
    cout << d << endl; // Okay: 5 3 2.37483
}
```

Navigation: Multiple MCQ, Partha Pratim Das, Algorithms & Patterns, Features of C++, I/O, Streams, Stream Output, Stream Input, File I/O, Type-safe I/O, Underpinned I/O, Stream Manipulators, Stream Types, User Types, User Types, Standard C++ Library, Multiple Questions

Programming in Modern C++ Partha Pratim Das MCQ 26



Slide 2: Type-safe I/O: C vis-a-vis C++

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

int main() {
    int i = 5, j = 3;
    double d = 2.37483;

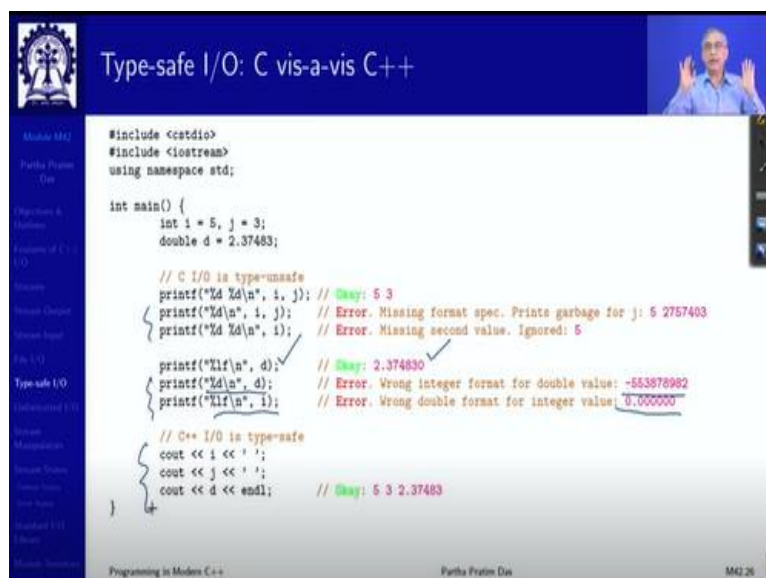
    // C I/O is type-unsafe
    printf("%d %d\n", i, j); // Okay: 5 3
    printf("%d\n", i); // Error: Missing format spec. Prints garbage for j: 5 2757403
    printf("%d %d\n", i); // Error: Missing second value. Ignored: 5

    printf("%lf\n", d); // Okay: 2.374830
    printf("%d\n", d); // Error: Wrong integer format for double value: -553878982
    printf("%lf\n", i); // Error: Wrong double format for integer value: 0.000000

    // C++ I/O is type-safe
    cout << i << ' ';
    cout << j << ' ';
    cout << d << endl; // Okay: 5 3 2.37483
}
```

Navigation: Multiple MCQ, Partha Pratim Das, Algorithms & Patterns, Features of C++, I/O, Streams, Stream Output, Stream Input, File I/O, Type-safe I/O, Underpinned I/O, Stream Manipulators, Stream Types, User Types, User Types, Standard C++ Library, Multiple Questions

Programming in Modern C++ Partha Pratim Das MCQ 26



Slide 3: Type-safe I/O: C vis-a-vis C++

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

int main() {
    int i = 5, j = 3;
    double d = 2.37483;

    // C I/O is type-unsafe
    printf("%d %d\n", i, j); // Okay: 5 3
    printf("%d\n", i); // Error: Missing format spec. Prints garbage for j: 5 2757403
    printf("%d %d\n", i); // Error: Missing second value. Ignored: 5

    printf("%lf\n", d); // Okay: 2.374830
    printf("%d\n", d); // Error: Wrong integer format for double value: -553878982
    printf("%lf\n", i); // Error: Wrong double format for integer value: 0.000000

    // C++ I/O is type-safe
    cout << i << ' ';
    cout << j << ' ';
    cout << d << endl; // Okay: 5 3 2.37483
}
```

Navigation: Multiple MCQ, Partha Pratim Das, Algorithms & Patterns, Features of C++, I/O, Streams, Stream Output, Stream Input, File I/O, Type-safe I/O, Underpinned I/O, Stream Manipulators, Stream Types, User Types, User Types, Standard C++ Library, Multiple Questions

Programming in Modern C++ Partha Pratim Das MCQ 26

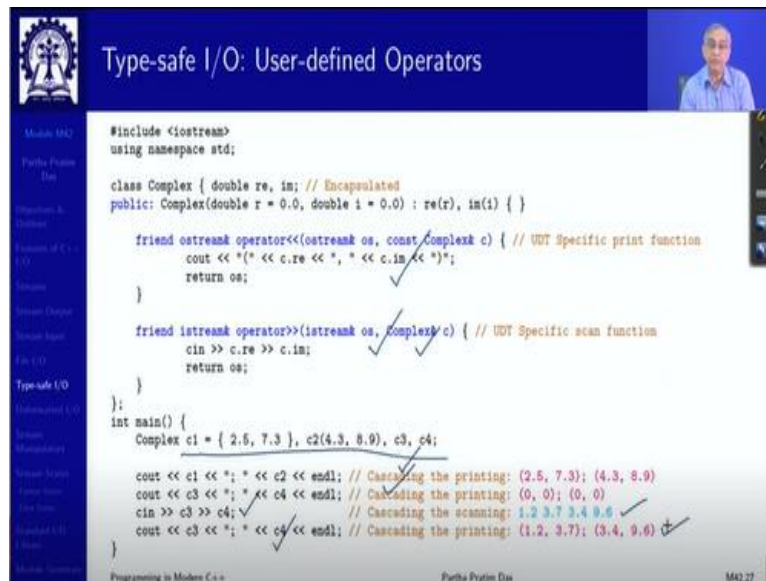
So, here is an example of type safety C versus C++. So, there are two types of two integer variables and one double variable this is what I wanted to write for the integer %d %d it prints okay. Here I have missed out on one %d which can always happen and I write two. So, since it is it is missing and there are two values given it prints the first one it knows the percentage the for the second value j it does not know what is a format. So, it prints in some god knows in what format.

A problem of the reverse kind can also happen for example, you have written two format strings and but given only one input, which will not give you an error or garbage but you will simply ignore. So, %d it has got i it has printed that it does not have a second variable to print. So, it does not do anything. So, all these kinds of errors are possible. What is even more dangerous is if you use the wrong format. You want to write the double value you have to use %lf it prints the value correctly.

Suppose you have done it with d it will not tell you anything it will silently print something. You know what does it printing it is I mean this entire double is interpreted, reinterpreted as if as an int in some way and a negative value is generated. If you happen to do the reverse, you are trying to print an integer but using the format for double %lf you again get something meaningfulness. So, all these errors will keep on happening.

If you do it in the C++ style, there is no scope of error because all these specifying the format the order the matching corresponding correspondence between the position of the format string and the position of the variable all these do not exist. So, you can just keep on cascading and it is guaranteed to work it is safe, type wise it will always work.

(Refer Slide Time: 29:43)



The slide displays C++ code for a `Complex` class with user-defined operators for cascading I/O. The code includes the following:

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

class Complex { double re, im; // Encapsulated
public: Complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }

    friend ostream& operator<<(ostream& os, const Complex& c) { // UDT Specific print function
        cout << "(" << c.re << ", " << c.im << ")";
        return os;
    }

    friend istream& operator>>(istream& is, Complex& c) { // UDT Specific scan function
        cin >> c.re >> c.im;
        return is;
    }
};

int main() {
    Complex c1 = { 2.5, 7.3 }, c2(4.3, 8.9), c3, c4;

    cout << c1 << " "; * << c2 << endl; // Cascading the printing: (2.5, 7.3); (4.3, 8.9)
    cout << c3 << " "; * << c4 << endl; // Cascading the printing: (0, 0); (0, 0)
    cin >> c3 >> c4; // Cascading the scanning: 1.2 3.7 3.4 9.6
    cout << c3 << " "; * << c4 << endl; // Cascading the printing: (1.2, 3.7); (3.4, 9.6)
}
```

What is more is the type safety extends to user-defined operators this is just for your reminder. So, here is the operator for output is the operator for input which I have defined for the `Complex` class you have seen this before. And here, we have some `Complex` class objects which we can cascade and write, cascade and write. Then we can read into them using the read operator and again right to check that you are getting correct values.

This is for completeness, because we have already discussed this, but for completeness in the context of the I/O is very important that it is type safety is not only available for the built in types, but the same safety and the same syntax and the same cascading is available for any user-defined type which makes everything very uniform.

(Refer Slide Time: 30:38)



The slide is titled "Unformatted I/O" and features a large, bold, red text "Unformatted I/O" centered on a white background. The slide also includes a navigation sidebar on the left and a video feed of the presenter in the top right corner.

Unformatted I/O

- `read` and `write` member functions
 - Unformatted I/O
 - Input/output raw bytes to or from a character array in memory
 - Since the data is unformatted, the functions will not terminate at a newline character for example
 - Instead, like `getline`, they continue to process a designated number of characters
 - If fewer than the designated number of characters are read, then the `failbit` is set
- `gcount`
 - Returns the total number of characters read in the last input operation

Programming in Modern C++ Partha Pratim Das MA2.29

Certainly, you can do unformatted I/O there are member functions like `read`, `write`, `getline`. We have already seen total number of characters that you have done and so on.

(Refer Slide Time: 30:53)

Stream Manipulators

Stream Manipulators

Programming in Modern C++ Partha Pratim Das MA2.30

Functionality of Stream Manipulators

- Setting field widths
- Setting precisions
- Setting and unsetting format flags
- Setting the fill character in fields
- Flushing streams
- Inserting a newline in the output stream and flushing the stream
- Inserting a null character in the output stream and skipping whitespace in the input stream

Programming in Modern C++ Partha Pratim Das MA2 31

Stream manipulators, we have seen already for example, we have seen how to flush the stream and so, on how to put a new line we already know.

(Refer Slide Time: 31:07)

Integral Stream Base

- `dec` (default), `oct` or `hex` ✓
 - Change base of which integers are interpreted from the stream

```
int n = 15;
cout << hex << n;
```

 - Prints "F"
- `setbase`:
 - Changes base of integer output
 - Load `<iomanip>`
 - Accepts an integer argument (10, 8, or 16)

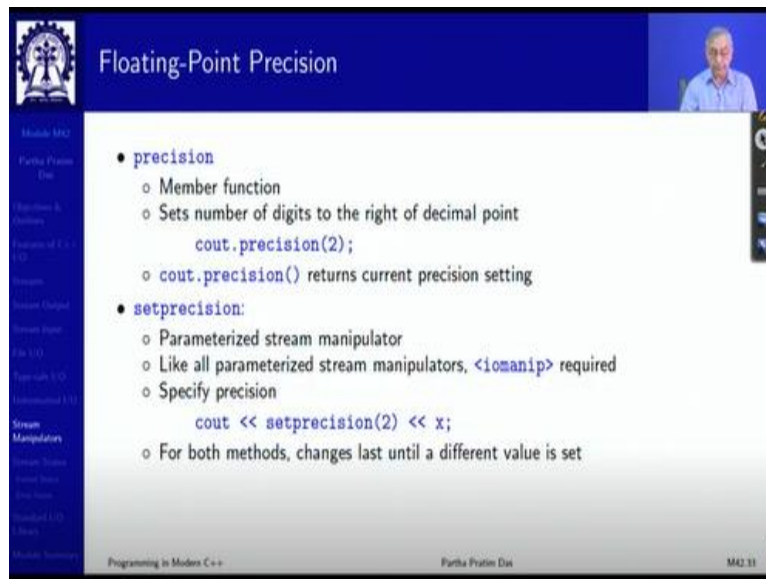
```
cout << setbase(16) << n;
```

 - Parameterized stream manipulator - takes an argument

Programming in Modern C++ Partha Pratim Das MA2 32

So, these you have seen you can change the base also you can say that I want to print this value in the hexadecimal base and so on.

(Refer Slide Time: 31:21)



The slide is titled "Floating-Point Precision" and features a blue header with a logo on the left and a small video feed of the presenter on the right. The main content area is white with a blue sidebar on the left containing a navigation menu. The slide lists two methods for controlling floating-point precision:

- **precision**
 - Member function
 - Sets number of digits to the right of decimal point

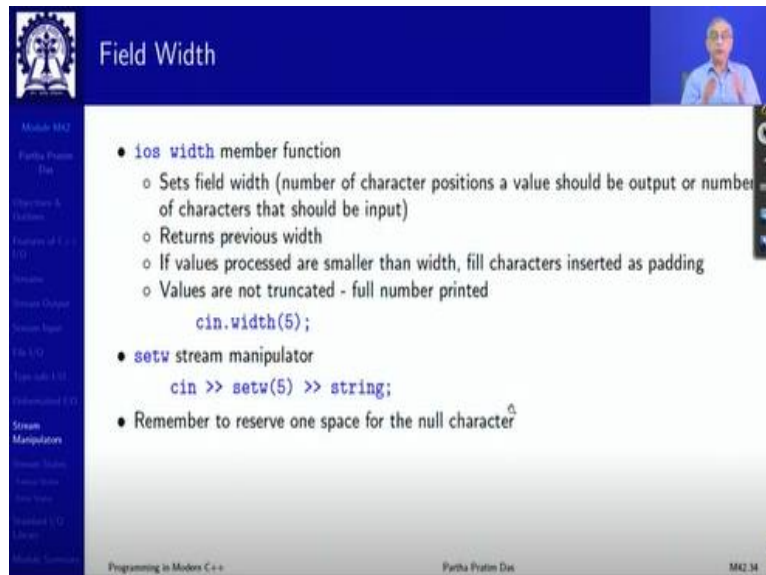
```
cout.precision(2);
```
 - `cout.precision()` returns current precision setting
- **setprecision:**
 - Parameterized stream manipulator
 - Like all parameterized stream manipulators, `<iomanip>` required
 - Specify precision

```
cout << setprecision(2) << x;
```
 - For both methods, changes last until a different value is set

At the bottom of the slide, it says "Programming in Modern C++", "Partha Pratim Das", and "M42.31".

So, there are several manipulators like precision is something which is very-very important because you can buy precision you can say how many points after the decimal that you want. You can clearly say that by invoking the precision member function of cout and then basically stream to it or you can use the setprecision function as well.

(Refer Slide Time: 31:45)



The slide is titled "Field Width" and features a blue header with a logo on the left and a small video feed of the presenter on the right. The main content area is white with a blue sidebar on the left containing a navigation menu. The slide lists two methods for controlling field width:

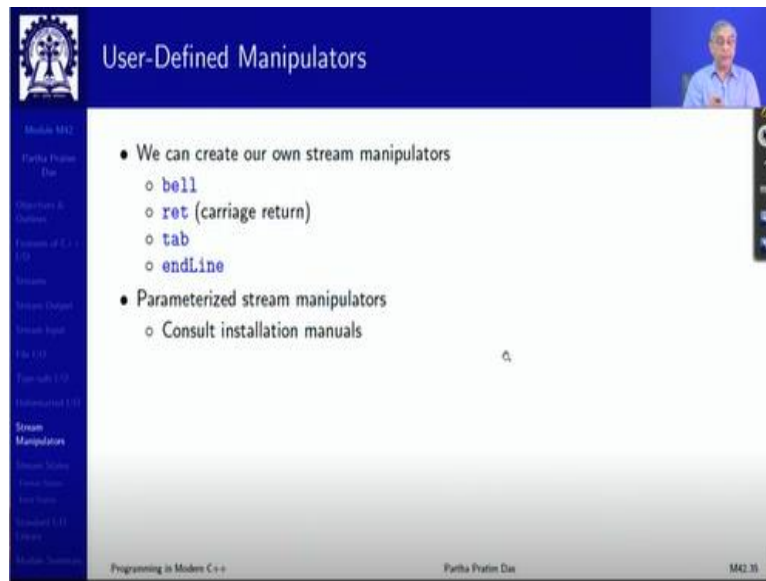
- **ios width** member function
 - Sets field width (number of character positions a value should be output or number of characters that should be input)
 - Returns previous width
 - If values processed are smaller than width, fill characters inserted as padding
 - Values are not truncated - full number printed

```
cin.width(5);
```
- **setw** stream manipulator
 - ```
cin >> setw(5) >> string;
```
  - Remember to reserve one space for the null character

At the bottom of the slide, it says "Programming in Modern C++", "Partha Pratim Das", and "M42.34".

You can control the field width these I mean there are number of them, you just need to I mean whatever you need, you just look up and apply that the style is the same that either you can invoke it as a member function on the stream object or you can invoke it you can have it as a global function wrapper in a `iomanip`.

(Refer Slide Time: 32:07)



The slide is titled "User-Defined Manipulators" and features a blue header with a logo on the left and a small video feed of the presenter on the right. The main content area is white and contains a bulleted list:

- We can create our own stream manipulators
  - `bell`
  - `ret` (carriage return)
  - `tab`
  - `endl`
- Parameterized stream manipulators
  - Consult installation manuals

The slide footer includes "Programming in Modern C++", "Partha Pratim Das", and "M42.11".

In for all this, you will have to certainly include the `iomanip` library component.

(Refer Slide Time: 32:14)



The slide is titled "Stream States" and features a blue header with a logo on the left and a small video feed of the presenter on the right. The main content area is white and contains the text "Stream States" in a large, bold, red font.

The slide footer includes "Programming in Modern C++", "Partha Pratim Das", and "M42.11".

Now, in the next couple of slides, I am not going to go through at all because these are for completeness of information in terms of what are the different states that exist for formats of files, as well as in case of error states. So, those are just like data. So, there is nothing no concept involved particularly. So, you can just look at them and refer to them as and when you need.

(Refer Slide Time: 32:43)

## Standard I/O Library

Standard I/O Library

Programming in Modern C++ Parthiv Pratim Das MC2 48

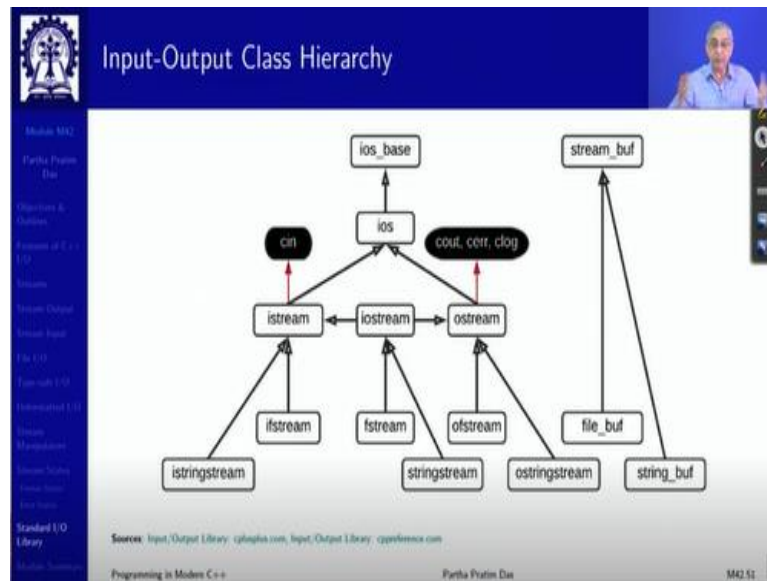
## Library Organization

- `<ios>`, `<iostream>`, `<ostream>`, `<streambuf>` and `<iosfwd>` are not usually included directly in most C++ programs. They describe the base classes of the hierarchy and are automatically included by other header files of the library that contain the derived classes.
- `<iostream>` declares the objects used to communicate through the standard input and output (including `cin` and `cout`)
- `<fstream>` defines the file stream classes (like template `basic_ifstream` or class `ofstream`) as well as the internal buffer objects used (`basic_filebuf`). These classes are used to manipulate files with streams.
- `<sstream>`. The classes defined in this file are used to manipulate STL string objects as if they were streams.
- `<iomanip>` declares some standard manipulators with parameters to be used with extraction and insertion operators to modify internal flags and formatting options.

Programming in Modern C++ Parthiv Pratim Das MC2 50

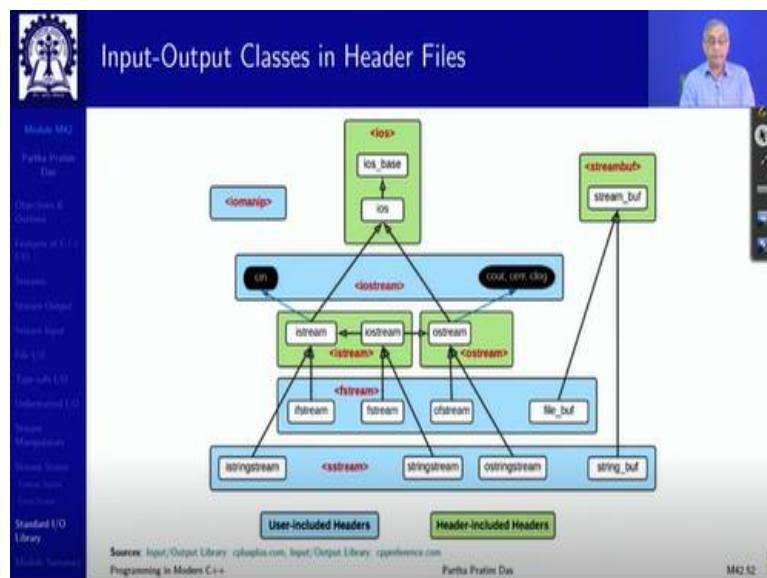
Before I conclude, I will just summarize on the standard I/O library for C++. So, in terms of the organization you can see that what we typically need to include is `iostream` and `sstream` if you want to do with strings and `fstream` or specifically `fstream` or `ofstream`. We can do `fstream` itself like you are doing `iostream` and `iomanip` everything else like these all different headers are included through them, you do not directly need to include them, particularly as you do the application programs.

(Refer Slide Time: 33:24)



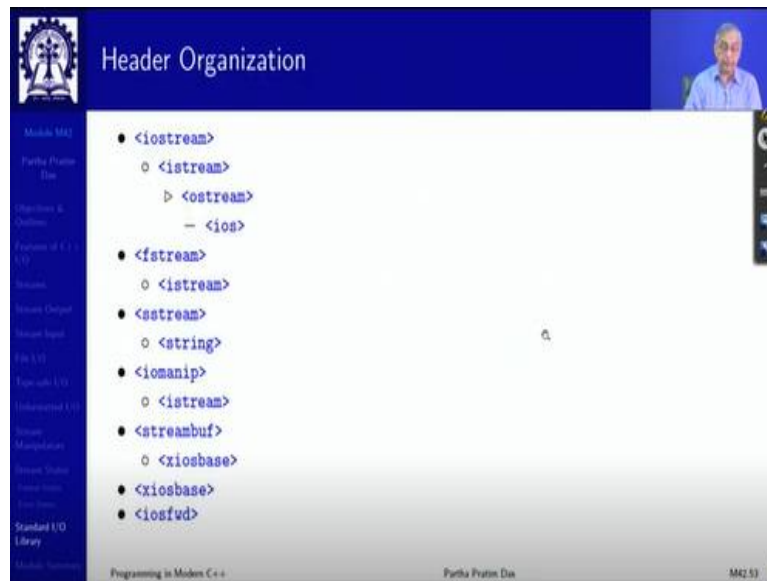
This is the overall class hierarchy for your understanding. So, you can say that ios base is what lies in the stop, but ios is a main implementation of the input-output system from which istream and ostream both are separately inherited and iostream is multiple inherited from both and then fstream and string stream are come from them. So, that is how this whole structure goes on.

(Refer Slide Time: 33:56)



And this is the complete diagram of classes and the different files, remember the blue ones are what you will need to include the four iostream, fstream, sstream, sstream and iomanip others get included as it is.

(Refer Slide Time: 34:14)



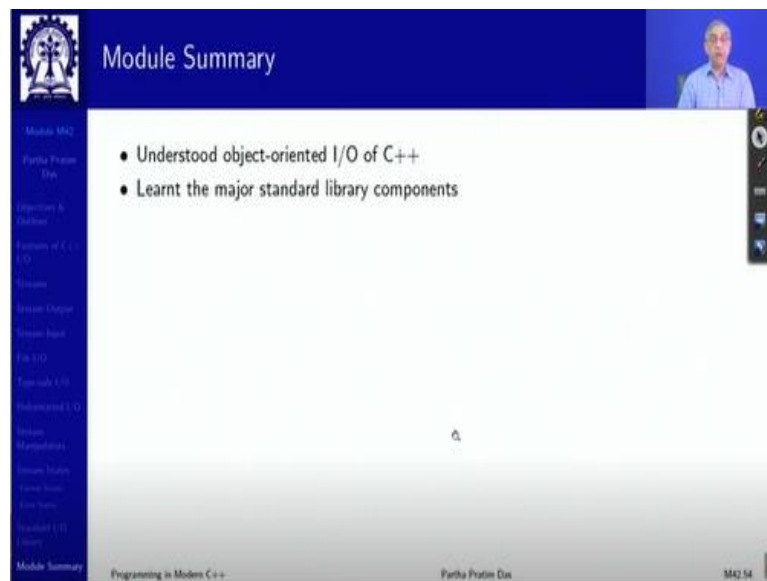
The slide titled "Header Organization" displays a hierarchical list of C++ headers. The list is as follows:

- `<iostream>`
  - `<iostream>`
    - ▷ `<ostream>`
      - `<ios>`
- `<fstream>`
  - `<iostream>`
- `<sstream>`
  - `<string>`
- `<iomanip>`
  - `<iostream>`
- `<streambuf>`
  - `<xiosbase>`
- `<xiosbase>`
- `<iosfwd>`

The slide also features a navigation sidebar on the left with items like "Module M2", "Part 1", "Streams", "Stream Output", "Stream Input", "File I/O", "Text I/O", "Formatted I/O", "Streams", "Manipulators", "Stream States", "Stream States", "Stream States", "Standard I/O Library", and "Module Summary". The footer includes "Programming in Modern C++", "Part 1: Part 1: Part 1", and "M2.53".

And this is an organization of the header for your reference which includes switch header.

(Refer Slide Time: 34:20)



The slide titled "Module Summary" lists the following learning objectives:

- Understood object-oriented I/O of C++
- Learnt the major standard library components

The slide features the same navigation sidebar as the previous slide. The footer includes "Programming in Modern C++", "Part 1: Part 1: Part 1", and "M2.54".

So, we have a kind of taken a look at the object-oriented style of I/O in C++. We have seen how it gives us very compact type safe mechanism which is extendable to user-defined data types. Thank you very much for your attention see you in the next module.