

Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Lecture 41
Input-Output: File Handling in C

(Refer Slide Time: 0:36)

Programming in Modern C++
Module M41: Input-Output: File Handling in C

Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Many diagrams in this module are taken from *Computer Science: A Structured Programming Approach Using C*

Programming in Modern C++ Partha Pratim Das M41.1

Weekly Recap

- Introduced the concept of exceptions
 - Discussed error handling in C with various language features and library support in C for handling errors
 - Discussed exception (error) handling in C++ with `try-throw-catch` feature in C++ for handling errors
- Introduced the templates in C++
 - Discussed function templates as generic algorithmic solution for code reuse
 - Discussed class templates as generic solution for data structure reuse
 - Explained partial template instantiation and default template parameters
 - Demonstrated templates on inheritance hierarchy
- Introduced Function Objects or Functors
 - Illustrated functors with several simple examples and examples from STL

Programming in Modern C++ Partha Pratim Das M41.2

Welcome to Programming in Modern C++. We are in week 9 we are starting it with Module M41. In the last week, we have discussed some key concepts of C++ particularly in terms of exceptions and error handling, introducing the try throw catch mechanism. We introduced generic programming or meta programming through templates in C++. And we specifically took a look at a special type of objects or classes called function objects or functors. And there we will see extensive use of those in the modules coming hereafter.

(Refer Slide Time: 1:17)

Module Objectives

- Understand file handling and I/O in C
- To understand Text and Binary I/O

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open / Close
Formatted I/O
Output
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

Programming in Modern C++ Partha Pratim Das M41.3

Module Outline

- 1 Weekly Recap
- 2 Standard Library for I/O
- 3 Files and Streams
 - File Open / Close
- 4 Formatted I/O
 - Output
 - Read
- 5 Unformatted I/O
- 6 Direct IO
- 7 File Positioning
- 8 Module Summary

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open / Close
Formatted I/O
Output
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

Programming in Modern C++ Partha Pratim Das M41.4

Standard Library for I/O

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open / Close
Formatted I/O
Output
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

Programming in Modern C++ Partha Pratim Das M41.5

In the module today, we are going to discuss about file handling and input output in C and particularly understand the text and binary input output, how it is to be done in C? This is an outline and will be available on the left pane as always. So, in C, the language by itself does not have any support for input output, input output is provided totally in terms of C standard library.

(Refer Slide Time: 1:57)

Standard C I/O Functions

- The C programming language provides many standard library functions for file input and output. These functions make up the bulk of the C standard library header `<stdio.h>`
- **Categories of I/O Functions**
 - File Open/Close
 - Formatted Input/Output
 - Character Input/Output
 - Line Input/Output
 - Block Input/Output
 - File Positioning
 - System File Operations
 - File Status

Source: [C file input/output](#), Wikipedia

Navigation sidebar (left): Module M41, Partha Pratim Das, Weekly Recap, Objectives & Outlines, Standard Library for I/O, Files and Streams, File Open - Close, Formatted I/O, Open, Read, Unformatted I/O, Direct IO, File Positioning, Module Summary

Footer: Programming in Modern C++, Partha Pratim Das, M41.6

And you know that it has a header `stdio.h` which needs to be included in a source file for doing any kind of file or terminal or printer input output operations. It provides many functions and the functions can be broadly categorized in terms of opening and closing files, formatted as well as unformatted input output, block input output, file positioning, certain checking of file status, system file operations and so on. There is a long list and we will discuss some of the representative functions here which are more commonly used in regular programming.

(Refer Slide Time: 2:45)

Files and Streams

Module M41
Partha Pratim Das

Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open / Close
Formatted I/O
Lines
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

Files and Streams

Programming in Modern C++ Partha Pratim Das M41.7

Files and Streams

Module M41
Partha Pratim Das

Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open / Close
Formatted I/O
Lines
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

- **A file is an external collection of related data treated as a unit.** The primary purpose of a file is to keep a record of data. Since the contents of primary memory are lost when the computer is shut down, we need files to store our data in a more permanent form.
- **Data is input to and output from a stream.** A stream can be associated with a physical device, such as a terminal, or with a file stored in auxiliary memory.

Programming in Modern C++ Partha Pratim Das M41.8

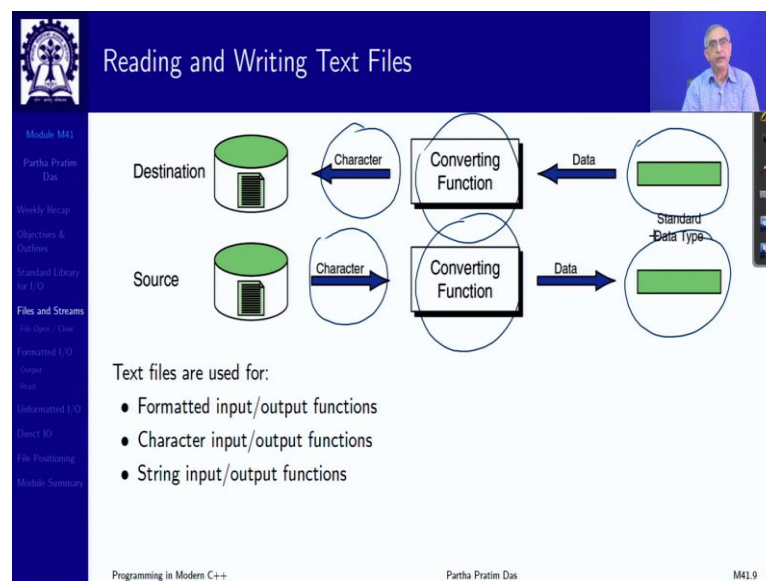
So, the first thing we introduce here is what is file and what is stream? The basic notion of stream is very critical. And actually C was a pioneer in introducing the concept of stream which eventually got imbibed into Unix and it is all there everywhere. We understand what is a file, it is an external collection of related data, which is treated as a unit and more often a file is treated as a sequential data structure, which means that starting from the beginning of the file, you can keep on moving forward reading or writing and you get the subsequent data elements that exist, though it is possible to treat it, also as a kind of random access by positioning the particular reader.

Now, stream in contrast is kind of an abstract concept, which the name naturally derives from the natural English word of stream which keeps on flowing. So, it is something which can be associated with a physical device like a terminal or a printer or a file stored in the auxiliary

memory. So, when we have for example, these input devices like key board or a file, then from that we create an input text stream, because these are input devices, and that brings in the data into the memory.

Similarly, the other way, if the data residing in the memory can be put through the output text stream, which is kind of a continuous one in this, in a stream you do not go back, you just keep on doing things forward. And that can be subsequently stored in a file or displayed in a monitor or maybe printed on a printer and so on.

(Refer Slide Time: 4:55)

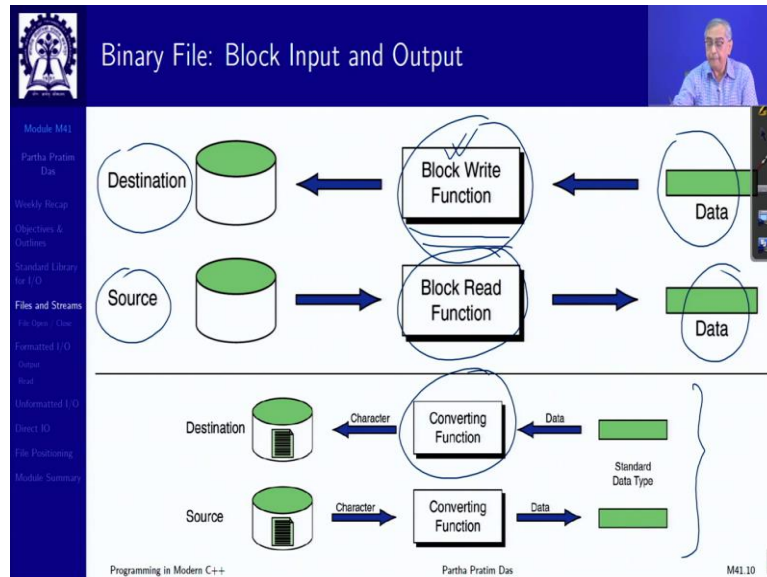


Now, how do we read or write text files? We are already familiar to a good extent doing it particularly on the console. So, for doing a text file, we have the data represented in some standard in the form of some standard type, it could be an integer data, character data, string data and so on. So, you take that and you apply a conversion function, a conversion function, So, that the data can be suitably represented as a sequence of characters.

This is important, because a text file is simply a sequence of characters and then you put that in your destination or the or in the reverse direction if you have a text file which is a sequence of characters, you take each character one by one and then apply the appropriate conversion function you are you will be more familiar with this in the name of Format Specification and convert that into a form which is representable in the memory according to the corresponding type and you stored that in the type.

So, this is the basic idea they have of doing io with a text file, it can be used for formatted input output or unformatted ones like character input output or even free input output can be done with the text files.

(Refer Slide Time: 6:24)



The other form of input output which is available through the C standard library function are kind of direct or block input output. So, you have the data represented in the memory as before, then you have a block write function, that is it checks takes the data as the chunk as it exists and puts it to the destination, and the difference here is it is not unlike here below is what you do for the text file. So, unlike doing a conversion, it is not doing a conversion it takes the data in the binary form and puts it to the destination directly without trying to do any kind of conversion.

So, this could be formatted, this could be unformatted, this could be written simply in the binary form and so on. Similarly, the reverse happens from a source, you do a block transfer of whatever bit patterns you get, and put that in the memory. So, these are the two broad types of input output that is available in terms of C functions.

(Refer Slide Time: 7:31)

Text and Binary Files

- `ascii(7) = 55 = 0b 0011 0111`
- `ascii(6) = 54 = 0b 0011 0110`
- `ascii(8) = 56 = 0b 0011 1000`
- `ascii('A') = 65 = 0b 0100 0001`
- `768 = 0b 0000 0011 0000 0000`

- Text files store data as a sequence of characters
- Binary files store data as they are stored in primary memory

Programming in Modern C++ Partha Pratim Das M41.11

Text and Binary Files

- `ascii(7) = 55 = 0b 0011 0111`
- `ascii(6) = 54 = 0b 0011 0110`
- `ascii(8) = 56 = 0b 0011 1000`
- `ascii('A') = 65 = 0b 0100 0001`
- `768 = 0b 0000 0011 0000 0000`

NotePad

- Text files store data as a sequence of characters
- Binary files store data as they are stored in primary memory

Programming in Modern C++ Partha Pratim Das M41.11

So, just to understand the text and binary files in little bit more depth, suppose I have a short integer 768. And I have a character A capital A uppercase A. Now, if you represent it in terms of a text file, what you will do is, you will take each one of these 7, 6 and 8, each one you take as a character and you put the ASCII code of the character in the file. So, here is the ASCII code of 7.

For your convenience I have written the ASCII codes here you can see, these ASCII code of 7 being written in 8 bits, next is the ASCII code of 6 being written in 8 bits, next is ASCII code of 8 being written in 8 bits and then the code of ASCII code of uppercase A which is 65 is written here.

So, it is being written character by character. So, a text file every component of the text file is a character typically it is 8 bit character, that is 1 byte character, it could be Unicode 2 byte characters as well that those kinds of text files are also possible. In contrast, if I say a file is a binary one, then it will represent this number in its binary form, that is as it is stored in the memory. How will it be stored in the memory? This is the entire binary representation.

So, 768 as we can see, it needs 2 bytes because one byte can keep up to 255. So, 768 is more than that, it will keep, it will need two bytes which can keep up to 16k numbers. And therefore, if you see this is the higher order byte which is here, this is a lower order byte which is here.

So, in a binary file, you are not representing these characters, but you are representing the value as a whole here the value 768, whereas when it comes to the character A, since character A is an individual data, it has this ASCII code. So, the character A is represented as before there is no difference in that. So, that is the basic difference between text file and binary file.

Examples are like any program source we are writing is a text file naturally, but an image that we click, image that we display is a binary file because it does not have characters, it just has the bit patterns of different intensity values. Similarly, even say something I mean just to be clear, suppose you have a doc file, doc or docx file the word file. Now, you when you visualize it through the Word, you will find that it displays you text, but that is what actually a text file.

The viewer is showing you as a text, but along with that, it is also showing you different annotations, like some words may be bold, some words may be italicized, there are alignment pads, there is paragraph separators and all that which are actually not text character information. So, if you actually ask a doc file or a docx file is actually a binary file too. Just to check what you can do is, you can in say windows, you can use a notepad application to open a docx file. You will find that you are not seeing the text as you are familiar to see you will see a whole lot of you know garbage nonsense characters, because notepad necessarily is a text editor.

So, it tries to, any file it opens it tries to separate it, punctuate it in 8 bits together and represent it as a character, but the representation inside the word file is not in terms of the ASCII codes, they are in terms of other symbols and binary values. So, you will see that

garbage, but if you open a program source file through notepad, you will see a very nice, you know, nice text that comes in. So, that is the basic difference. And that is a reason that we do need both of these to be available.

(Refer Slide Time: 12:11)

File Modes

Mode	r	w	a	r+	w+	a+
Open state	read	write	write	read	write	write
Read allowed	yes	no	no	yes	yes	yes
Write allowed	no	yes	yes	yes	yes	yes
Append allowed	no	no	yes	no	no	yes
File must exist	yes	no	no	yes	no	no
Contents of existing file lost	no	yes	no	no	yes	no

For read/write of binary files, use 'b' with one of the above modes

File Opening Modes

Read Mode (r, r+) Write Mode (w, w+) Append Mode (a, a+)

Now, every file is associated with a mode, that is you can either do input to a file, do input from a file or you can do output to a file. So, there are this is specified by certain flags, read, write flags. So, when you associate a stream with a file, you have to specify which mode are you using. So, you can read, write at the two dominant modes.

So, if the file has been open, which means that it has been associated with a stream, then in that you can do a read if you have opened it in the read mode, that is you are taking inputs from it. You can do write if you have opened it in the write mode, when the read be allowed, if you have opened it in the read mode, but if you have opened a file in the write mode, you will not be allowed to read from that. It will be an error.

Similarly, write will be allowed if you have opened the file in the write mode, but read will not be allowed. If the file is being opened in the read mode, it must exist otherwise you will have an error, but in the write mode it will not need to exist, if it exists, it will be cleared, all contents will be cleared and overwritten new and if it does not exist a file will be created.

So, this is the basic difference between the read and write mode. There are other modes like append which is pretty much like write, but it does some different behavior as you can see in terms of here. For example, if you open in append mode, you will not be able to write, you will be able to append only that is it will happen. So, you can see how that goes on.

This is the view of the stream that you have here. And you have opened means you have put a marker on it, the marker, which remembers in the sequential order where you are. So, when you do it in the read mode, it will always put the marker at the beginning because you have to start reading from the beginning. When you do it in the write mode.

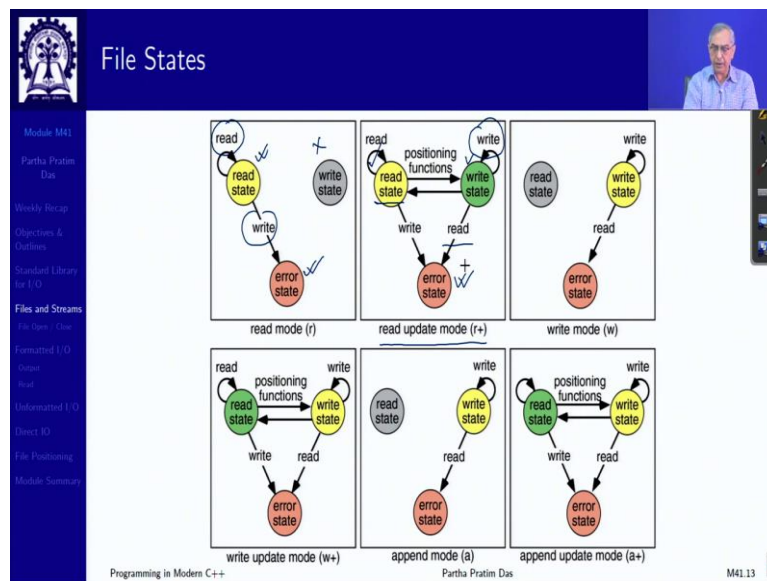
It will also have it in the beginning, but you can see in the read mode, we expect content to be there in the right mode, there will be no content. If there were contents when you open it in the write mode, that content is purged out. Whereas if you open it in the append mode, it already has content and you are putting mode to it. So, it is like append mode is important, because if you do not write the entire file in one go you write some data.

And then maybe some other program write some more data then maybe again your program write some more data, you cannot do it by the write mode, because the moment you open it by the write mode, everything that you had written earlier it purged in append mode that purging is not done. But as much as has been written, the file marker will go right after that and start writing from there.

So, these are the basic three modes, then there are + modes on those, which says that read and something more. So, you can see what are the different behaviors I have written down here in some cases, like if something is opened in r+, then read of course is allowed, but write we if it is open in w+, then also you can you are able to read. So, you can use the + modes to do both read and write, append and read these kinds of things together.

So, these are the basic modes and particularly, if you have a binary, so, if you open a file in these modes, it will be opened as a text file, if you want to open a binary file, you will have to write b along with the mode. So, you write this it opens as a text file, you write this it opens as a binary file, that is a simple rule that you will have.

(Refer Slide Time: 16:46)



Now, naturally, depending on in which mode you have opened it the file will maintain a state based on whether you are reading or you are writing and here in this diagram, I have just tried to summarize the states, like if you have opened something in the read mode, you do every read it continues to remain in the read state, but if you try to do a write it goes to an error and the write state is not reachable because you cannot write there.

Whereas if you do it in the read+ state mode, read update mode, then in the read state it can keep on reading, you can reposition your marker, I will show how to reposition the marker, and come to the write mode where you can keep on writing you remain in this state it can again reposition go back to the read mode. So, you can write some data go back read it again, go forward write again and so on.

But if you try to do read on a write state or write on a read state, then you will come to an error, repositioning properly is absolutely necessary for doing the other state of operation. So, that is the basic principle you can go through each of these diagrams and based on the mode chart I gave you in the last slide you can understand these state transitions.

(Refer Slide Time: 18:06)

File Open and Close

- To write to or read from a file, we need to **open** it: `FILE *fopen(const char *filename, const char *mode);`
 - Each FILE object denotes a C stream and keeps the state during I/O
 - filename: file name to associate the file stream to
 - mode: null-terminated character string determining file access mode
- If successful, returns a pointer to the new file stream. The stream is fully buffered unless filename refers to an interactive device.
- On error, returns a null pointer
- On successful opening, we **write** and/or **read** data using I/O functions
- Once the write or read file over, we need to **close** it:
`int fclose(FILE *stream); // stream: the file stream to close`
 - Returns 0 on success, EOF otherwise // EOF is special End-of-File marker
 - Closes the file stream, flushes unwritten buffered data, and discards unread buffered data
 - The stream is no longer associated with a file, the buffer is disassociated and deallocated
 - The behavior is undefined if the value of the pointer stream is used after fclose returns

Diagram: A box labeled 'FILE' contains a 'Stream' box and a buffer. The buffer is a horizontal array of cells with an arrow pointing to the first cell. An 'EOF' marker is shown at the end of the buffer.

File Open and Close

- To write to or read from a file, we need to **open** it: `FILE *fopen(const char *filename, const char *mode);`
 - Each FILE object denotes a C stream and keeps the state during I/O
 - filename: file name to associate the file stream to
 - mode: null-terminated character string determining file access mode
- If successful, returns a pointer to the new file stream. The stream is fully buffered unless filename refers to an interactive device.
- On error, returns a null pointer
- On successful opening, we **write** and/or **read** data using I/O functions
- Once the write or read file over, we need to **close** it:
`int fclose(FILE *stream); // stream: the file stream to close`
 - Returns 0 on success, EOF otherwise // EOF is special End-of-File marker
 - Closes the file stream, flushes unwritten buffered data, and discards unread buffered data
 - The stream is no longer associated with a file, the buffer is disassociated and deallocated
 - The behavior is undefined if the value of the pointer stream is used after fclose returns

Diagram: Similar to slide 1, but with a handwritten arrow pointing from the 'Stream' box to the 'FILE' box.

Now the basic form of io is like this. Let us say you have a file. So, that file, let us say physically exist in the disk system or needs to be created in the disk system. What do you do? You visualize this in the program as a stream and that stream type in C is a data structure called capital FILE, which is defined in the stdio.h library, this has a buffer to do your io and it has a marker on that buffer.

So, what happens when you say, you have open it, this is how you open it, fopen, give the file name which is this entity and give the mode you want to open for writing. So, he will give it w here. Then what you get? You get a pointer to this file structure. And everything that you do, you actually refer to this pointer, pointer to the file which means that whenever you are

trying to write something you are doing a `printf` rather `fprintf` because you are writing to a file, then you actually the data keeps on going to the buffer.

When the buffer gets full, it will write it to the file and then flush itself and again keep on. The reason it is done in this way, in this buffered way is simple that the file being on the disk actual writing to the disk could be very, very slow. So, you want to avoid during that, at every instant you keep on accumulating things in a buffer, which is in the memory, which is very fast and only periodically when the buffer has become full or when you are done with the file and you want to close everything you write that actually to the file.

The same thing happens in the case of read, again, you again you will open, let me just clear this out this part. So, what you have is, you are opening it in the read mode. So, you will do read to provide the file name this must exist now, because you want to read and what you get is a file pointer, pointer to this data structure file, which contains this buffer.

So, as you start reading from it, which you will do by `fscanf` a chunk of data is actually taken from the physical disk file and the buffer is filled up and the marker remains here at the beginning. Now, as you keep on reading data, the marker keeps on progressing and when it is exhausted the buffer the next chunk will again be brought from the file.

So, you can understand that this particular transfer part is expensive, whether you are reading or writing because it is making accesses to the disk. So, you avoid that by doing this buffered stuff, which is happening automatically inside the library functions. So, the steps are simple you have to open a file, which means associate a stream with a file then you do write or read depending on the mode or mixture of that read update, write update depending on the mode in which you have opened and then you do a close, by close you dissociate.

Now, you say that no more this file structure a pointer to the file structure will mean the file that you had associated it with. So, with that, if you are writing then whatever data was there in the buffer is flushed onto the disk file and you are your disk file is saved it will remain there forever and the file structure is released, if you are doing a read then if anything left in the buffer that will simply be discarded and the linkage will be broken and this file structure will be released. So, this is the basic process of io, rest of it are nothing but simple function call.

(Refer Slide Time: 22:46)

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    // ...
    FILE* spTemps; // Declarations for file handler
    // ...
    if ((spTemps = fopen("TEMPS.DAT", "r")) == NULL) {
        printf("aERROR opening TEMPS.DAT\n");
        exit(100);
    } // if open

    // Perform I/O

    if (fclose(spTemps) == EOF) {
        printf("aERROR closing TEMPS.DAT\n");
        exit(102);
    } // if close

    // ...
} // main
```

So, for example, here we are showing how to open. We are opening this file name in this mode with fopen. It will give you the stream pointed and if the opening has an error suppose we are opening with the read mode, so the file must exist. Suppose the file does not exist, then it will not be able to open then it will not be able to associate. So, if it does that, if that happens, then it returns a value 0 in null pointer. And by that you know that you could not open it successfully, So, you exit.

Similarly, when you are closing it returns you a value typically a value 0 to mean success, otherwise, you read it returns with end of file marker, which is a special character, which is the marker put at the end of every file meaning that this is a point where there is nothing more in that file structure. So, suppose you could not close because your disk is full. So, it is not possible to flush out the buffer and write the remaining data on to the file because this does not have space. So, in that case, your fclose will fail and you will get EOF as a return value.

(Refer Slide Time: 24:00)

Formatted I/O

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open - Close
Formatted I/O
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

Formatted I/O

Programming in Modern C++ Partha Pratim Das M41.16

Formatted I/O: File Write and Read

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open - Close
Formatted I/O
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

- To write, we use:
 - `int printf(const char *format, ...);` // Writes to output stream `stdout`
 - `int fprintf(FILE *stream, const char *format, ...);` // Writes to output stream `stream`
 - `int sprintf(char *buffer, const char *format, ...);` // Writes to a string buffer
- `format`: A null-terminated multibyte string specifying interpretation of the data
- `...`: arguments specifying data to print - a variadic function
- `stream` must be open before writing (`stdout` stays open) or reading (`stdin` stays open)
- `buffer` must be allocated before writing
- If successful, number of characters transmitted to the output stream or number of characters written to buffer (not counting the terminating null character) is returned
- A negative value is returned for an output error or an encoding error
- To read, we use:
 - `int scanf(const char *format, ...);` // Reads from input stream `stdin`
 - `int fscanf(FILE *stream, const char *format, ...);` // Reads from input stream `stream`
 - `int sscanf(char *buffer, const char *format, ...);` // Reads from a string buffer
- Number of receiving arguments successfully assigned (which may be zero in case a matching failure occurred before the first receiving argument was assigned), or EOF if input failure occurs before the first receiving argument was assigned.

Programming in Modern C++ Partha Pratim Das M41.17

So, this is the basic you know process. Now, I will quickly go through the functions. So, you know the printf, so, I will not detail it any more, you know the use of formats. And so just you know remember that in this library, the function names are done in a very systematic manner. So, f at the end of this name, whether you are doing print, or we are doing scan is meant to represent format.

So, it shows that formatted. You are doing it with formatting which means that it will have a format string to say how the conversion has to happen, whether it is for output or is for input. And if you have a f or an s at the beginning, f means that you are doing it with the file, s means you are doing it with a string buffer. A string buffer could also act as a source or a destination of your read write operation exactly in the same way. Rest of it is exactly like

printf as you have used where by default, your file is stdout, which is always open you do not need to defer, you never need it to open the file because stdout is always open.

Similarly, when you do scanf when you are doing reading, you have the similar format stuff specified and you do not need to open it because it will always use stdin and that is always open when your program starts. So, that is the basic process.

(Refer Slide Time: 25:38)

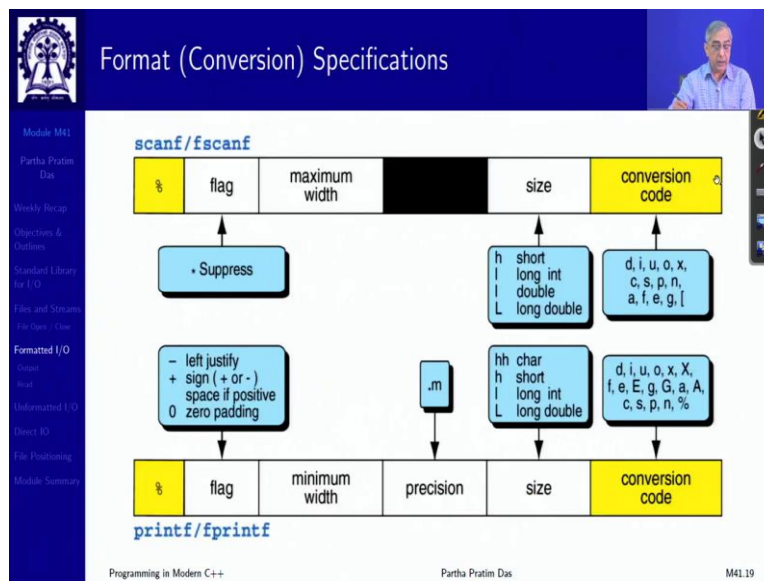
The slide is titled "Side Effect and Value of Formatted I/O Function". It features a table of contents on the left side with the following items: Module M41, Partha Pratim Das, Weekly Recap, Objectives & Outlines, Standard Libraries for I/O, Files and Streams (File Open, Close), Formatted I/O (printf, scanf), Unformatted I/O (Direct IO), File Positioning, and Module Summary. The main content area contains the following text:

- **printf, fprintf etc.**
 - **Side Effect**
 - ▷ Converts internal data, as required, to strings of characters and writes the converted values to a file, which may be the standard output or error file
 - **Value**
 - ▷ Returns the number of characters written to the output file. In case of an error, it returns EOF
- **scanf, fscanf etc.**
 - **Side Effect**
 - ▷ Reads and converts a stream of characters from the input file, and stores the converted values in the list of variables found in the address list
 - **Value**
 - ▷ Returns the number of successful data conversions. If end of file is reached before any data are converted, it returns EOF

At the bottom of the slide, it says "Programming in Modern C++", "Partha Pratim Das", and "M41.18".

Naturally both printf and I mean all kinds of printf and scanf they do the operation of writing or reading and they return a value, for printf the return value gives you how many characters you have written and for scanf it gives you how many data elements that you have converted and read.

(Refer Slide Time: 26:01)



So, these are the, so, this is the different kinds of format conversion you have this app shown in terms of a diagram. So, it always starts with a percentage then you have a flag for alignment and so on. You can specify maximum width for `scanf`, minimum width for `printf` you can provide precision for a `printf` and then you provide the size which is mean how much size you want basically. And then you have the conversion code which you have to align with the type of the data that you are printing that you already have seen in the `printf`.

(Refer Slide Time: 26:40)

```
#include <stdio.h>

int main() {
    int i = 17;
    long l = 0x012a78cb; // 19560651
    long long unsigned int i64 = 0x012a78cb2597ac3d; // 84012356964166717
    float f = 15.0 / 7;
    double d = 15.0 / 7;
    char c = 'x';
    const char *s = "ppd";
    int *p = &i;

    printf("%d\n", i); // dec 17
    printf("%x\n", i); // hex 11
    printf("%o\n", i); // oct 21
    printf("%ld\n", l); // long 19560651
    printf("%llu\n", i64); // int 64 84012356964166717
    printf("%f\n", f); // float 2.142857
    printf("%lf\n", d); // double 2.142857
    printf("%c\n", c); // char x
    printf("%s\n", s); // string ppd
    printf("%p\n", p); // pointer 0x7ffc28102988
}
```

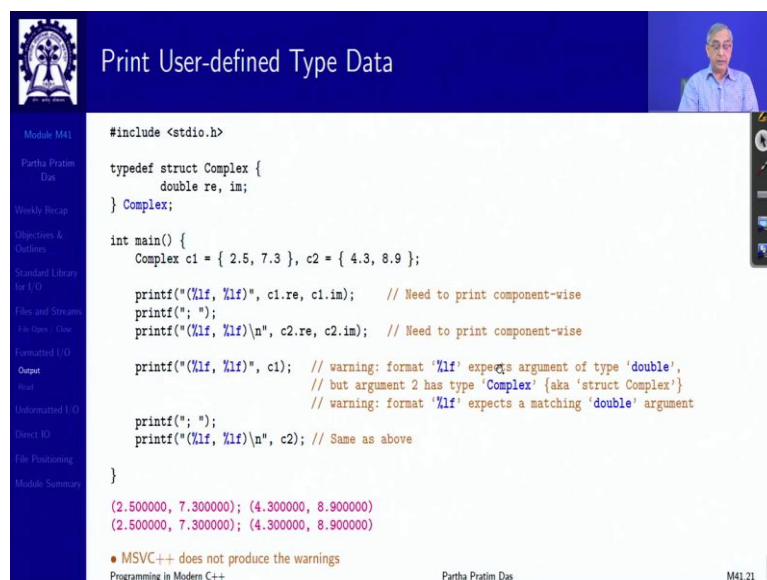
The slide also includes a navigation sidebar on the left and a video feed of the presenter in the top right corner. At the bottom, it says "Programming in Modern C++", "Partha Pratim Das", and "M41.20".

So, here is an example which you can try to run these are different types of data and these are `printf` on those and you can see what are the values that are being printed. Just be careful with this particular line because unless you have a 64 bit machine this particular data type

declaration and the corresponding percentage `%llu` writing will not work. It works only in the 64 bit type. So, if your system does not have that, then just comment out these two lines and rest of the code will work you can see what kind of data you get.

And you have all different types of formats that are possible both in terms of the data type as well as the way you want to visualize. For example, `%d`, `%x`, `%o`, all work with each type, but they write the data in different forms. So, same 17 with `%d` is written as 17, but with `%x` is written as 11 because it is hexadecimal. So, 11 is $1*16+1$, 17. Similarly, if you do percentage `o` it will be written as 21, in the octal system two times 8 is $16+1$ like this. So, you can you can easily use that.

(Refer Slide Time: 28:03)



```
#include <stdio.h>

typedef struct Complex {
    double re, im;
} Complex;

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

    printf("(%lf, %lf)", c1.re, c1.im); // Need to print component-wise
    printf("\n");
    printf("(%lf, %lf)\n", c2.re, c2.im); // Need to print component-wise

    printf("(%lf, %lf)", c1); // warning: format '%lf' expects argument of type 'double',
    // but argument 2 has type 'Complex' {aka 'struct Complex'}
    // warning: format '%lf' expects a matching 'double' argument

    printf("\n");
    printf("(%lf, %lf)\n", c2); // Same as above
}

(2.500000, 7.300000); (4.300000, 8.900000)
(2.500000, 7.300000); (4.300000, 8.900000)

● MSVC++ does not produce the warnings
Programming in Modern C++ Partha Pratim Das M41.21
```

You can use these, but if you have user defined data type like a complex as we have seen, then you will not be able to extend printf for doing that you have to take them component by component and use them you have seen this before.

(Refer Slide Time: 28:18)

Flags, Sizes, and Conversion Code for printf family

Argument Type	Flag	Size Specifier	Code
integer	-, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	d, i
unsigned int	-, +, 0, space	hh (char), h (short), none (int), l (long), ll (long long)	u
integer (octal)	-, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	o
integer (hex)	-, +, 0, #, space	hh (char), h (short), none (int), l (long), ll (long long)	x, X
real	-, +, 0, #, space	none (double), l (double), L (double)	f
real (scientific)	-, +, 0, #, space	none (double), l (double), L (double)	e, E
real (scientific)	-, +, 0, #, space	none (double), l (double), L (double)	g, G
real (hex)	-, +, 0, #, space	none (double), l (double), L (double)	a, A
character	-	none (char), l (wchar_t)	c
string	-	none (char string), l (wchar_t string)	s
pointer	-	-	p
integer (for count)	-	none (int), h (short), l (long)	n
to print %	-	-	%

Programming in Modern C++ Partha Pratim Das M41.22

Flag Formatting Options

Flag Type	Flag Code	Formatting
Justification	none	right justified
	-	left justified
Padding	none	space padding
	0	zero padding
Sign	none	positive value: no sign negative value: -
	+	positive value: + negative value: -
	space	positive value: space negative value: -
Alternate	#	print alternative format for scientific, hexadecimal, and octal

Programming in Modern C++ Partha Pratim Das M41.23

Here I have given some charts, which certainly I am not going to go through right away. This chart is for your reference, So, that if you want to use a particular type of format, you can get all the information in this charts of different flags, formatting options of padding justification and so on.

(Refer Slide Time: 28:40)

Read Built-in Type Data

```

#include <stdio.h>
int main() {
    int i; long l;
    long long unsigned int i64; // For a 64-bit machine
    float f; double d;
    char c; char *s = (char*)malloc(10); // Space needs to be allocated to store the string to be read
    int *p;

    // Input shown in magenta and Output shown in gray

    scanf("%d\n", &i);    printf("%d\n", i);    // dec // 17 17
    scanf("%x\n", &i);    printf("%x\n", i);    // hex // 11 11
    scanf("%o\n", &i);    printf("%o\n", i);    // oct // 21 21
    scanf("%ld\n", &l);    printf("%ld\n", l);    // long // 19560651 19560651
    scanf("%llu\n", &i64); printf("%llu\n", i64); // int 64 // 84012356964166717 84012356964166717
    scanf("%f\n", &f);    printf("%f\n", f);    // float // 2.142857 2.142857
    scanf("%lf\n", &d);    printf("%lf\n", d);    // double // 2.142857 2.142857
    scanf("%c\n", &c);    printf("%c\n", c);    // char // x x

    // Used just 's', not %s, as it is a pointer
    scanf("%s\n", s);    printf("%s\n", s);    // string // ppd ppd

    scanf("%p\n", &p);    printf("%p\n", p);    // pointer // 008FFC0C 008FFC0C
}
    
```

Programming in Modern C++ Partha Pratim Das M41.24

Similar exercise will also apply to read in terms of scanf exactly as you have done, the formats are more or less similar, the format codes are more or less similar. And here I have shown just to be confirmed that what you have read, what you have read is what you are getting, I have shown the corresponding printf as well, you can just run it and get comfortable with the common formats of data that you have.

(Refer Slide Time: 29:10)

Sizes and Conversion Code for scanf family

Argument Type	Size Specifier	Code
integral	hh (char), h (short), none (int), l (long), ll (long long) h (short), none (int), l (long), ll (long long)	i
integer	h (short), none (int), l (long), ll (long long)	d
unsigned int	hh (char), h (short), none (int), l (long), ll (long long)	u
character octal	hh (unsigned char)	o
integer hexadecimal	h (short), none (int), l (long), ll (long long)	x
real	none (double), l (double), L (double)	f
real (scientific)	none (double), l (double), L (double)	e
real (scientific)	none (double), l (double), L (double)	g
real (hexadecimal)	none (double), l (double), L (double)	a
character	none (char), l (wchar_t)	c
string	none (char string), l (wchar_t string)	s
pointer		p
integer (for count)	none (int), hh (char), h (short), l (long), ll (long long)	n
set	none (char), l (wchar_t)	[

Programming in Modern C++ Partha Pratim Das M41.25

The slide is titled "Checking scanf Results" and features a video feed of the presenter in the top right corner. The main content is a C++ code snippet. A red circle highlights the return value of the scanf function, which is assigned to the variable ioResult. The code includes error handling for incorrect input counts.

```

#include <stdio.h>

#define FLUSH while (getchar() != '\n')
#define ERR1 "\aPrice incorrect. Re-enter both fields\n"
#define ERR2 "\aAmount incorrect. Re-enter both fields\n"

int main() {
    int amount;
    double price;
    int ioResult;

    // Read price and amount
    do {
        printf("\nEnter amount and price: ");
        ioResult = scanf("%d%f", &amount, &price);
        if (ioResult != 2) {
            FLUSH;
            if (ioResult == 1) ✓
                printf(ERR1);
            else
                printf(ERR2);
        } // if
    } while (ioResult != 2);
}

```

Navigation icons are visible on the right side of the slide, and the footer contains "Programming in Modern C++", "Partha Pratim Das", and "M41.26".

Again the similar conversion chart for the scanf which you can refer to and when you do scanf you could for example, you have given scanf for two values as we are here. So, as I said the scanf will return how many data elements you have converted. So, if you have given for two and you have provided only one data, then certainly you will have an error or so, if it is one or you have not provided any data, you have just tried to kind of say that there is no data.

So, then also he will have error. So, with scanf you can use this kind of checks to see that you have gotten as much data as you had actually needed to have from the input.

(Refer Slide Time: 30:04)

The slide is titled "Unformatted I/O" and features a video feed of the presenter in the top right corner. The main content area is mostly blank, with the text "Unformatted I/O" displayed in red at the bottom center. The left sidebar shows a navigation menu with "Unformatted I/O" selected. The footer contains "Programming in Modern C++", "Partha Pratim Das", and "M41.27".

Character I/O

- Terminal Only
 - Input: *getchar*
 - Output: *putchar*
- Any Stream
 - Input: *getc / fgetc*
 - Output: *putc / fputc*
 - Push Back: *ungetc*

Programming in Modern C++ Partha Pratim Das M41.28

IO could be unformatted also like you can read character wise and these are the different types of functions that are available. If you are doing it with terminal, it is `getchar` and `putchar` which does input-output of individual character otherwise with any stream you will have `getc` or `fgetc`, (now, the naming convention). So, `getc` will get it from a steady in `fgetc` will get it from a file similarly, `putc`, `fputc`, `ungetc` and so on.

(Refer Slide Time: 30:35)

```

/* This program creates a text file */
#include <stdio.h>
int main() {
    FILE* spText;    // Stream
    int c, closeStatus;

    printf("This program copies input to a file.\n");
    printf("When you are through, enter <EOF>.\n\n");

    if (!(spText = fopen("My_New_Text_File.txt", "w"))) {
        printf("Error opening My_New_Text_File.txt for writing");
        return (1);
    } // if open

    while ((c = getchar()) != EOF) // Read characters from stdin. Use ^Z for EOF
        fputc(c, spText);        // Write characters to file

    closeStatus = fclose(spText);
    if (closeStatus == EOF) {
        printf("Error closing file\n");
        return 100;
    } // if

    printf("\n\nYour file is complete\n");
}

```

Programming in Modern C++ Partha Pratim Das M41.29

Here is a text file example given which I will not go through, please try it on your system and see how you get.

(Refer Slide Time: 30:46)

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open - Close
Formatted I/O
Open
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

Direct Input/Output

Programming in Modern C++ Partha Pratim Das M41.30

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open - Close
Formatted I/O
Open
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

File Write Operation

4 * 3 = 12 bytes

before write

after write

outArea

```
fwrite(outArea, sizeof(int), 3, spOut);
```

Programming in Modern C++ Partha Pratim Das M41.31

Besides this, you can also have direct input-output that is particularly what you want to do with a binary file is take a chunk of data and directly write it. So, here for that you use different functions possibly fwrite, which takes a buffer like this. So, you need to specify that what is the units in that buffer. So, I said there are three units: total sizes, its size of int,

So, every chunk has to be size say 4 bytes, and there will be three such and it has to go to this stream. So, it takes in this way, it takes three chunks of ints and put it to the output. So, that is our basic operation, you can see that this is where your marker was before you started doing it and this is where it goes after you are done with it. So, directly using fwrite you can directly do that. Similarly, there is a version called write which works which does not need the spout which will work with stdout.

You can do similar stuff with directly taking a structure and writing it in using the fwrite. You can do the inverse by reading using fread again you need to have the buffer into which he will read, the size of every unit and how many units you want to read. And then finally the stream these diagrams M41 will obviously corresponding to fread there is a read which can happen with the stdin.

(Refer Slide Time: 32:28)

Reading a Structure

The diagram illustrates the state of memory buffers before and after a `fread` operation. It shows three horizontal buffers, each divided into four segments. The middle segment of each buffer is highlighted in yellow. In the 'Before Read' state, the middle segment of the top buffer contains a question mark, and a red arrow labeled 'oneStudent' points to this segment. A red arrow labeled 'before read' points to the boundary between the first and second segments. In the 'After Read' state, the middle segment of the bottom buffer is filled with green vertical bars, and a red arrow labeled 'oneStudent' points to this segment. A red arrow labeled 'after read' points to the boundary between the second and third segments.

File Write / Read

```
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);
```

// buffer: pointer to the array where the read objects are stored
 // size: size of each object in bytes
 // count: the number of the objects to be read

- Writes count of objects from the given array buffer to the output stream stream. The objects are written as if by reinterpreting each object as an array of unsigned char and calling `fputc` size times for each object to write those unsigned chars into stream, in order. The file position indicator for the stream is advanced by the number of characters written.
- Returns number of objects written successfully, which may be less than count if an error occurs.

```
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

- Reads up to count objects into the array buffer from the given input stream stream as if by calling `fgetc` size times for each object, and storing the results, in the order obtained, into the successive positions of buffer, which is reinterpreted as an array of unsigned char. The file position indicator for the stream is advanced by the number of characters read.
- Returns number of objects read successfully, which may be less than count if an error or end-of-file condition occurs.

You can read by structures also. So, these are all, here you are not parsing it in terms of specific characters you are just you know taking things in block and reading or writing them. So, here are the two major functions fwrite and fread which I have already explained here are the details you can go through.

(Refer Slide Time: 32:51)

File Positioning

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open / Close
Formatted I/O
Getenv
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

File Positioning

Programming in Modern C++ Partha Pratim Das M41.36

File Read Operation

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open / Close
Formatted I/O
Getenv
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

3 * 4 = 12 bytes

before read

after read

inArea

```

fread (inArea, sizeof (int), 3, spData);

```

Programming in Modern C++ Partha Pratim Das M41.33

Rewind File

Module M41
Partha Pratim Das
Weekly Recap
Objectives & Outlines
Standard Library for I/O
Files and Streams
File Open / Close
Formatted I/O
Getenv
Read
Unformatted I/O
Direct IO
File Positioning
Module Summary

Before Rewind

file marker

After Rewind

file marker

Programming in Modern C++ Partha Pratim Das M41.37

Naturally for you know, for using the + mode, the update mode, you need to reposition the marker. So, it is possible that you have done some tasks and you can reposition. So, there are at certain functions to do that you can rewind, which will bring the file marker from wherever it was to the beginning of the file, but it can be done specifically at different points also.

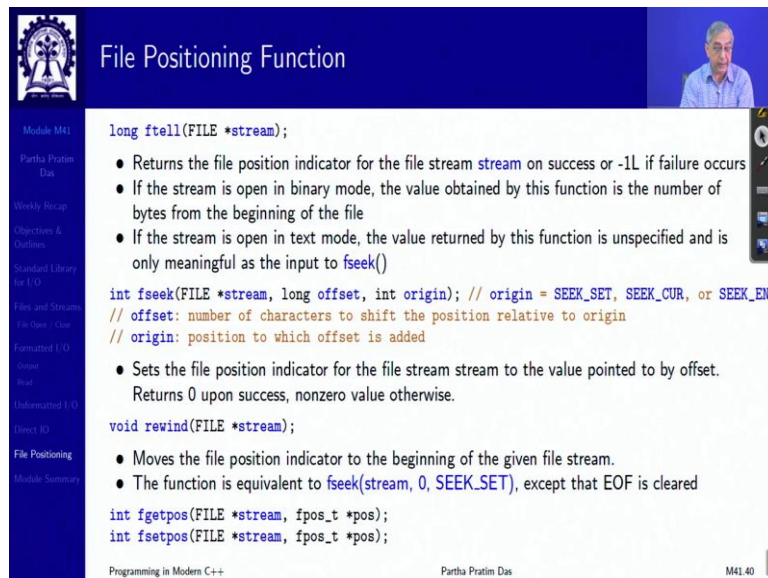
(Refer Slide Time: 33:20)

The slide is titled "Current Location (ftell) Operation". It features a diagram of a file structure represented as a horizontal bar divided into several green rectangular blocks. A yellow callout bubble labeled "Beginning of File" points to the first block. A red arrow labeled "Current location (16)" points to the fourth block. Below the diagram, the text "Number of Bytes" is written. The slide includes a navigation sidebar on the left with items like "Module M41", "Partha Pratim Das", "Weekly Recap", "Objectives & Outlines", "Standard Library for I/O", "Files and Streams", "File Open - Close", "Formatted I/O", "Open", "Read", "Unformatted I/O", "Direct I/O", "File Positioning", and "Module Summary". The footer contains "Programming in Modern C++", "Partha Pratim Das", and "M41.38".

The slide is titled "File Seek Operation". It shows three examples of file seek operations. Each example consists of a diagram of a file structure with a red arrow indicating the new current location, followed by a C code snippet. The first example shows the current location at the beginning of the file with the code `fseek (sp, 4 * sizeof(STRUCTURE_TYPE), SEEK_SET);`. The second example shows the current location at the end of the file with the code `fseek (sp, - 4 * sizeof(STRUCTURE_TYPE), SEEK_END);`. The third example shows the current location at the current position with the code `fseek (sp, 2 * sizeof(STRUCTURE_TYPE), SEEK_CUR);`. The slide includes the same navigation sidebar as the previous slide. The footer contains "Programming in Modern C++", "Partha Pratim Das", and "M41.39".

You can actually note the current position of the file marker also by a function called ftell, you can reposition the file marker anywhere by doing it fseek operation you can set it to a point you can set it to the end you can set it to the beginning and so on. So, rewind is basically a special case of seek operation. So, these by this you can reposition your marker and then again restart read or write as we had explained in the update operation.

(Refer Slide Time: 33:54)



The slide is titled "File Positioning Function" and features a small video inset of the presenter in the top right corner. The left sidebar contains a navigation menu with the following items: Module M41, Partha Pratim Das, Weekly Recap, Objectives & Outlines, Standard Library for I/O, Files and Streams, File Open / Close, Formatted I/O, Open, Read, Unformatted I/O, Direct IO, File Positioning, and Module Summary. The main content area includes the following text:

```
long ftell(FILE *stream);
```

- Returns the file position indicator for the file stream `stream` on success or `-1L` if failure occurs
- If the stream is open in binary mode, the value obtained by this function is the number of bytes from the beginning of the file
- If the stream is open in text mode, the value returned by this function is unspecified and is only meaningful as the input to `fseek()`

```
int fseek(FILE *stream, long offset, int origin); // origin = SEEK_SET, SEEK_CUR, or SEEK_END
// offset: number of characters to shift the position relative to origin
// origin: position to which offset is added
```

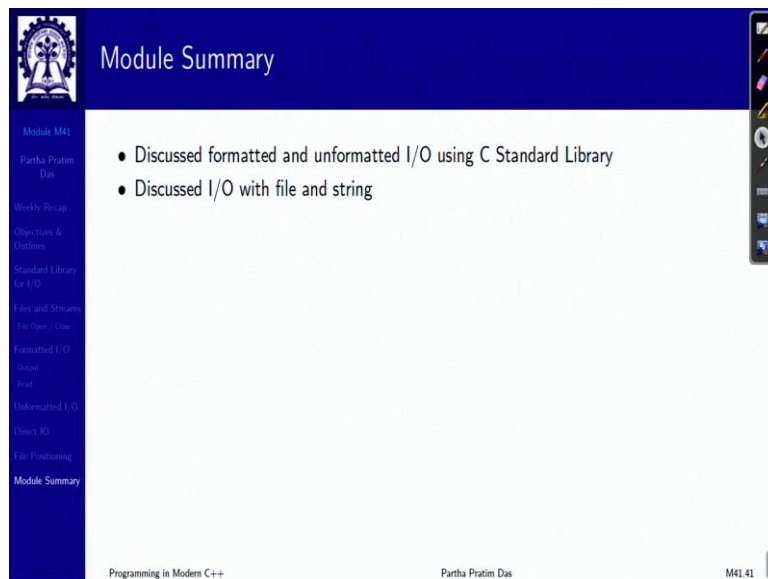
- Sets the file position indicator for the file stream `stream` to the value pointed to by `offset`. Returns `0` upon success, nonzero value otherwise.

```
void rewind(FILE *stream);
```

- Moves the file position indicator to the beginning of the given file stream.
- The function is equivalent to `fseek(stream, 0, SEEK_SET)`, except that EOF is cleared

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);
```

At the bottom of the slide, it says "Programming in Modern C++", "Partha Pratim Das", and "M41.40".



The slide is titled "Module Summary" and features a small video inset of the presenter in the top right corner. The left sidebar contains a navigation menu with the following items: Module M41, Partha Pratim Das, Weekly Recap, Objectives & Outlines, Standard Library for I/O, Files and Streams, File Open / Close, Formatted I/O, Open, Read, Unformatted I/O, Direct IO, File Positioning, and Module Summary. The main content area includes the following text:

- Discussed formatted and unformatted I/O using C Standard Library
- Discussed I/O with file and string

At the bottom of the slide, it says "Programming in Modern C++", "Partha Pratim Das", and "M41.41".

So, these are the common functions which we will need for doing this `ftell`, `fseek` and `rewind`. So, this brings us to the end of this naturally input output standard io is a very very big topic. So, I just tried to give you the basic idea of association between file and stream and the buffered input or output that typically goes on through the system for performance. And going forward. We will see the same view in terms of C++. Thank you very much for your attention.