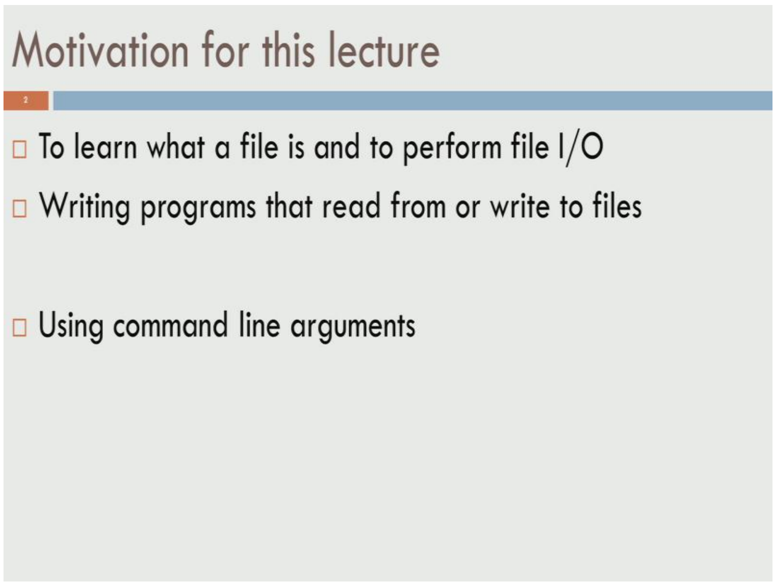
Programming, Data Structures and Algorithm
Prof. Shankar Balachandran
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 60
FILE Input / Output

Welcome to this last module for this course. We are going to do two things in this week's videos, one is about file input and output that will be the first part and let we also learn something about structure programming. So, let me give you a little bit of motivation for the lecture itself on file layout. So, it is not reasonable to expect the user to always give inputs from the key board or always just print things on the screen.

(Refer Slide Time: 00:42)

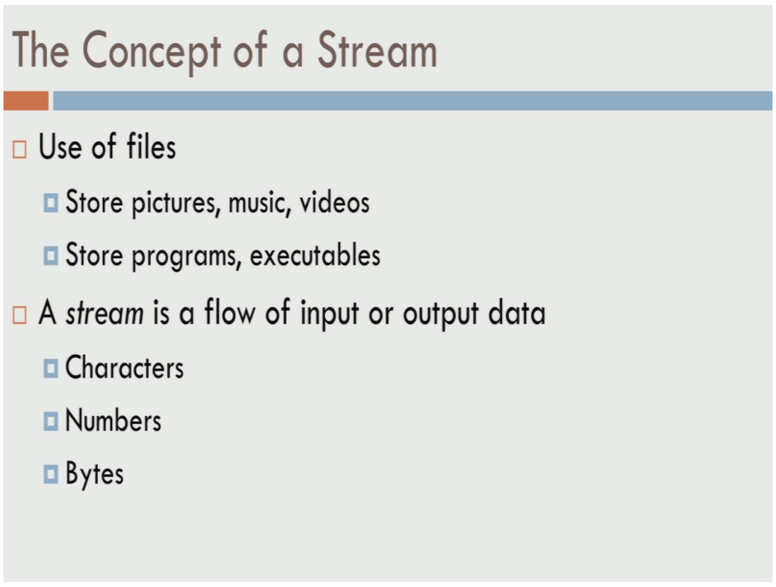


Motivation for this lecture

- To learn what a file is and to perform file I/O
- Writing programs that read from or write to files
- Using command line arguments

There are several places and instances where you want something recorded and you wanted to be reused later and this is a place where files come in to pictures. It is not the notion of files it is probably not earlient to you. So, what we will do in this lecture is we will try and learn what a file is and how to perform input and output operations on a file, we will also see how to write programs that we will read from or write to files and we learn one another topic about command line arguments. So, that will be the first video for this week. So, let us look at the notion of a stream.

(Refer Slide Time: 01:19)



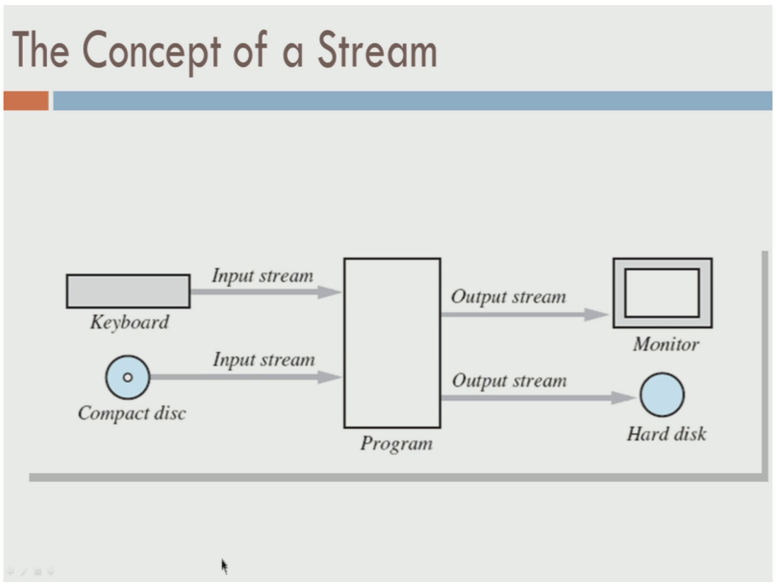
The Concept of a Stream

- Use of files
  - Store pictures, music, videos
  - Store programs, executables
- A *stream* is a flow of input or output data
  - Characters
  - Numbers
  - Bytes

When we look at files, things like mp3 files, video files, document files or even html pages that you download are all files, they have all been created somewhere, they enable you to transmit information from one place to the other, save it, retrieve it and so on and without files, we cannot do any of those. So, if you expect a user to always give inputs over a key board that is not always good, sometimes input is not even coming from the keyboard, it could be coming from some other device.

So, the notion of a file let you both store and retrieve information and in this context, we will use this notion called a stream. A stream is essentially a flow of input or output data, this could be anything. This could be characters, bytes, numbers anything that you send from one program to another program or from one device to another device and so on. So, let us look at this basic example.

(Refer Slide Time: 02:22)



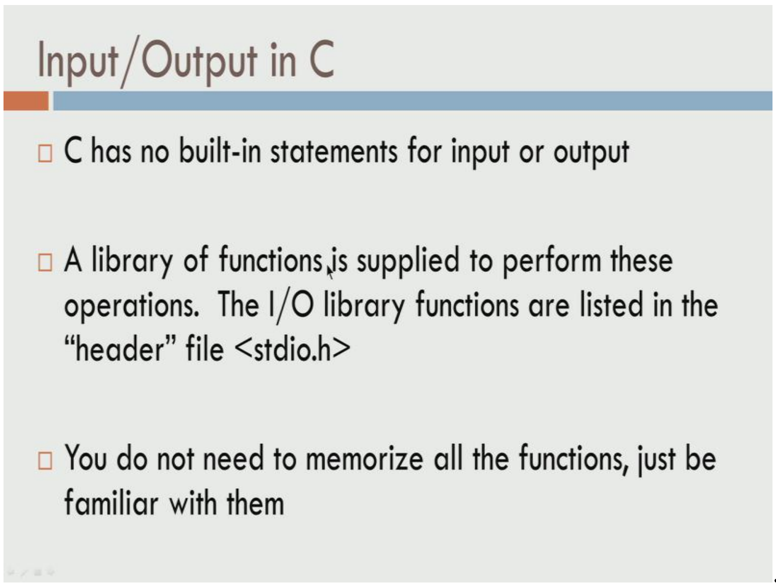
The Concept of a Stream

So, let us say I write a program and what we have been doing is we are always been taking inputs from the key board, whatever I do demonstration of, I take inputs from the key board and the output was always on the screen. And when you did your programming exercises online, this input itself was not from a keyboard, you had saved something and it got saved in to a file, but the output was still on your screen.

There are other places for example, let us say my program is actually an mp3 player or a CD player, the input is going to come from a compact disk and the output is probably a speaker or the input stream is coming from a compact disk and let us say, you take a wave file and you make a mp3 out of it. So, the input is from your CD, you are ripping a CD and you are writing to a disk which is the hard disk.

So, we need to be able to not just interact with keyboard and monitor alone, we need be able to interact with the other devices also, at the same time we do not want to be bogged down by all the details of these devices and so on and that is what the notion of 5 layers come in.

(Refer Slide Time: 03:26)



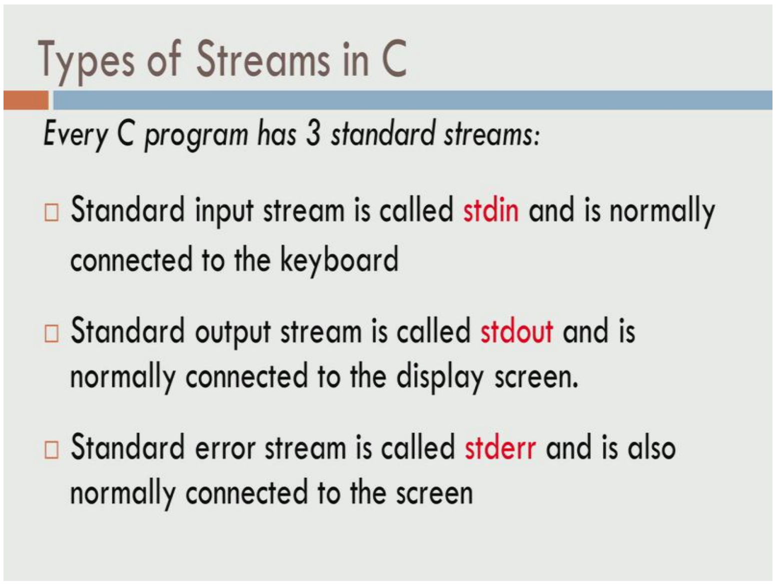
Input/Output in C

- C has no built-in statements for input or output
- A library of functions is supplied to perform these operations. The I/O library functions are listed in the "header" file <stdio.h>
- You do not need to memorize all the functions, just be familiar with them

.

So, C basically has no direct support for input and output and so instead it is supplied through a library and we have seen this so for, the header file that we have been including called stdio dot h is essentially that. So, it has several functions that will enable you to do input and output. You do not have to really remember all of them, but it is good to know that all the functions that you need for input and output are actually in this file, are all in this header called stdio dot h.

(Refer Slide Time: 03:57)



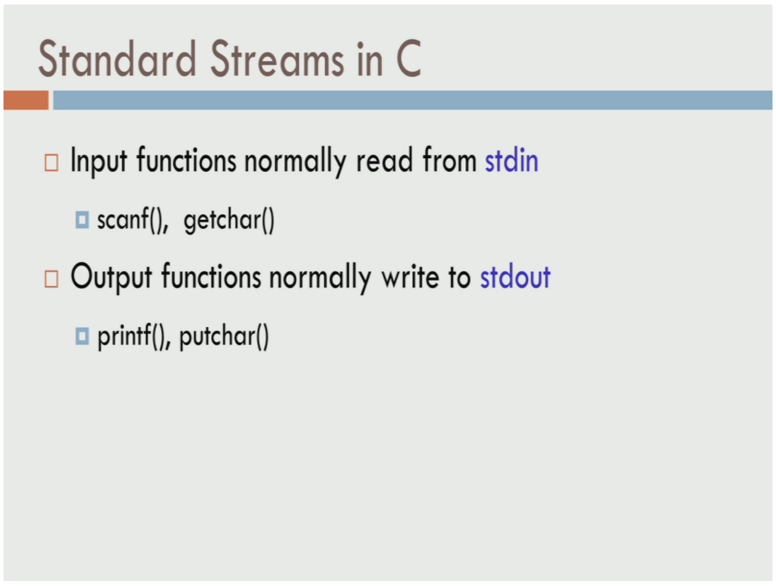
Types of Streams in C

Every C program has 3 standard streams:

- Standard input stream is called stdin and is normally connected to the keyboard
- Standard output stream is called stdout and is normally connected to the display screen.
- Standard error stream is called stderr and is also normally connected to the screen

So, let us now concentrate on what C gives us. So, every C program is automatically associated with three standard streams, so that is why you will see the prefix called stdin, std for it, so there are three standard streams stdin, stdout and stderr. So, stdin is for input stream, stdout is for output stream or anything that gets return from the program and stderr is for printing any errors and it is usually associated with the screen itself. So, stdin is usually associated with keyboard, stdout is usually associated with the screen and stderr is also usually associated with the screen.

(Refer Slide Time: 04:46)

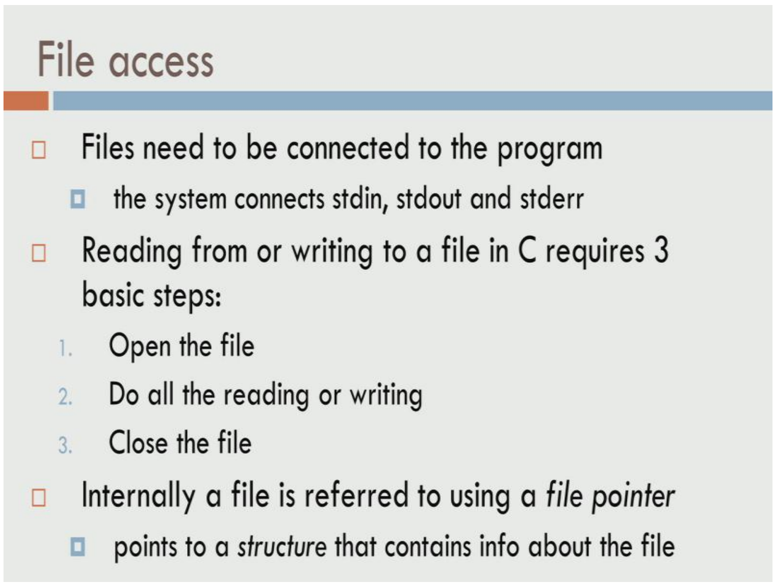


Standard Streams in C

- Input functions normally read from stdin
  - scanf(), getchar()
- Output functions normally write to stdout
  - printf(), putchar()

And whenever we did things like scanf and getchar and so on, we were actually reading from the key boards, so that say standard stream, you do not have to do anything specially and output was always return to the screen also. So, with printf we did not bother what to do with it, because we know that the default stream is anyway stdout which is connected to the monitor.

(Refer Slide Time: 05:04)



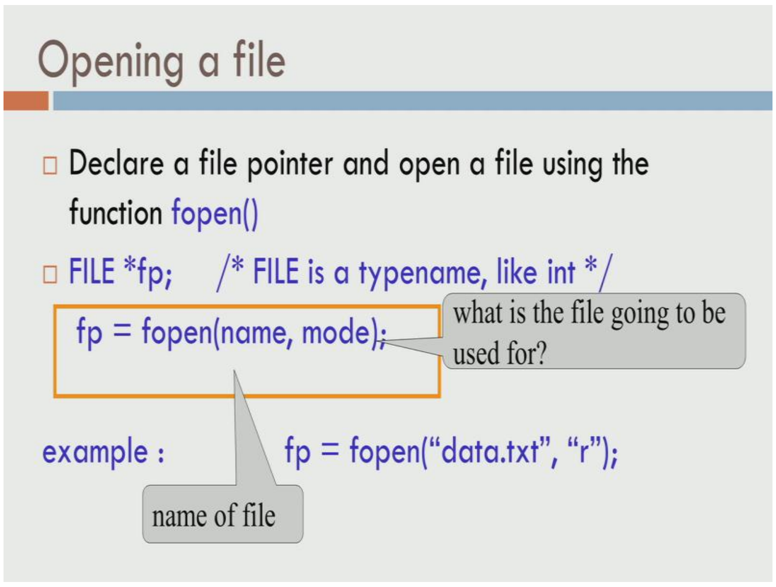
File access

- Files need to be connected to the program
  - the system connects stdin, stdout and stderr
- Reading from or writing to a file in C requires 3 basic steps:
  1. Open the file
  2. Do all the reading or writing
  3. Close the file
- Internally a file is referred to using a *file pointer*
  - points to a *structure* that contains info about the file

But, when you have files that you want to deal with on your own, it is not the case anymore. So, even though stdin, out and err are automatically connected, if you want to do anything more with other files, if you want stream from other places you need to do something more. So, reading and writing a file in C requires three basic steps, you need to be able to open a file, you need to be able to do all the reading or writing that goes with it and finally you should close the file.

And instead of dealing with file names explicitly in our programs, you will deal with what are called file pointers. So, file pointer is essentially a pointer to a structure that contains a lot of information about the file. Again, we do not have to know a lot of details about what is in the structure, but just remember that we will deal with file pointers and not file names directly.

(Refer Slide Time: 05:54)



Opening a file

- Declare a file pointer and open a file using the function fopen()
- FILE *fp; /* FILE is a typename, like int */

```
fp = fopen(name, mode);
```

example : fp = fopen("data.txt", "r");

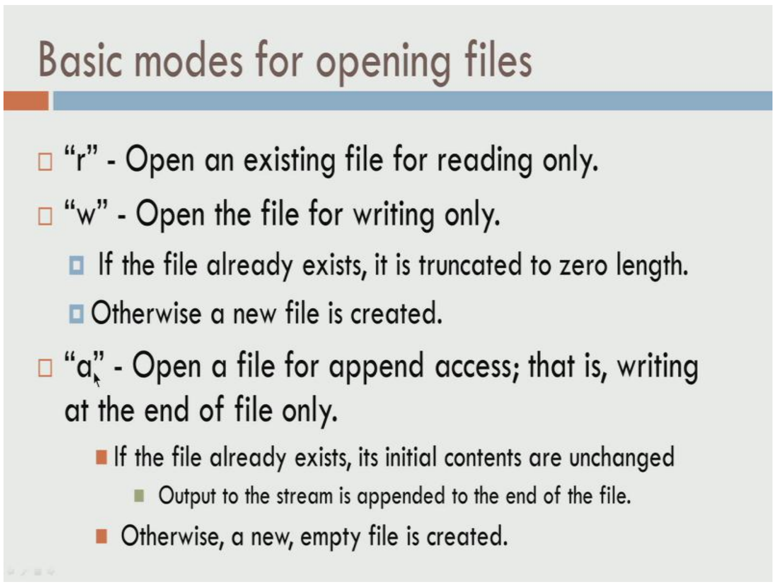
what is the file going to be used for?
name of file

So, let us start with the first step namely opening a file. So, if I want a programmatic word to open a file, so it is not about clicking on a screen and opening a file. So, I want my program to be able to open a file and do some operations on it, so to do that you declare a file pointer and you open a file using this function called fopen. So, file pointer is declared with as you see in the second bullet here, FILE star fp.

So, fp star fp is indicates that it is a pointer and to what is it pointing, it is pointing to a structure called file. So, fp is a pointer to the data type structure and many times we will drop this thing called structure and we will just say that fp is a file pointer. So, we have a file pointer now and we can do fp equals fopen of name comma mode. So, the name should be the name of the file that you are trying to open and mode can be one of the three things, so let us see this example fp is fopen of data dot text comma r.

So, if we say r it is supposed to be in read only mode, we are trying to open the file data dot text in read only mode which means you do not want to write anything from your program, you want only read contents of the file data dot text. So, one can notice that both the file name and the mode are actually put within double quotes. So, fopen expects two strings, one string which is the name of the file and another string which is the mode itself.

(Refer Slide Time: 07:26)



Basic modes for opening files

- "r" - Open an existing file for reading only.
- "w" - Open the file for writing only.
  - If the file already exists, it is truncated to zero length.
  - Otherwise a new file is created.
- "a" - Open a file for append access; that is, writing at the end of file only.
  - If the file already exists, its initial contents are unchanged
    - Output to the stream is appended to the end of the file.
  - Otherwise, a new, empty file is created.

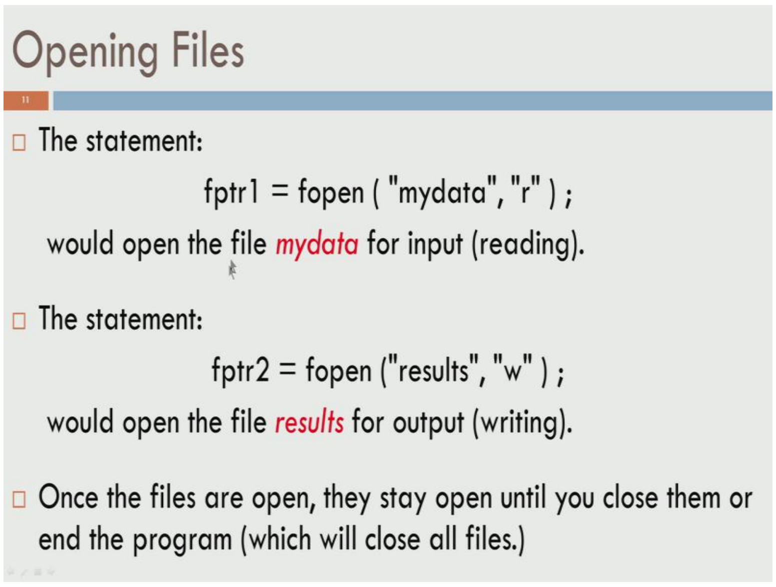
So, let us look at the other modes of opening a file, so r I already mentioned in the previous slide, w is the other mode in which you open the file, but only for writing which means from your program you can only write to it, you cannot go and read what is in the file. So, that is called write mode and a, which is called the append mode. So, let us look at the distinction between I mean among these three modes, r means it will try and open an exciting file.

So, clearly what if I try to open a file that is not even existent, we need to be able to capture this file. So, let us say I am going to open a file, but there is no file by that name, if that happens we need a mechanism which says there is no file by that name. So, that is an issue that we have to deal with for r. For w, you are trying to write to a file, you are not going to read it from there at least in this program as of now.

So, if the file already exists and since you are only writing, the previous contents of the files does not matter, we can as well remove all the contents and start with zero length. So, since you are only writing from the current program, if you by mistake open an existing file, the previous contents will already deleted. But, normally you would expect your program to create a new file, so if it is a new file and since you are writing, it will automatically create a new file with the name that you have suggested.

A stands for append, append means add at the end and this is as the name indicates, it is useful for writing only at the end of the file. So, clearly this is similar to write, it has some similarity with the write, if the file already exists, the initial contents are unchanged. However, if the file is not there, append is as good as actually opening with the w mode. Only that all the rights will always happen to the end of the file, so these are the three modes.

(Refer Slide Time: 09:27)



Opening Files

- The statement:

  fptr1 = fopen ("mydata", "r") ;

  would open the file *mydata* for input (reading).

- The statement:

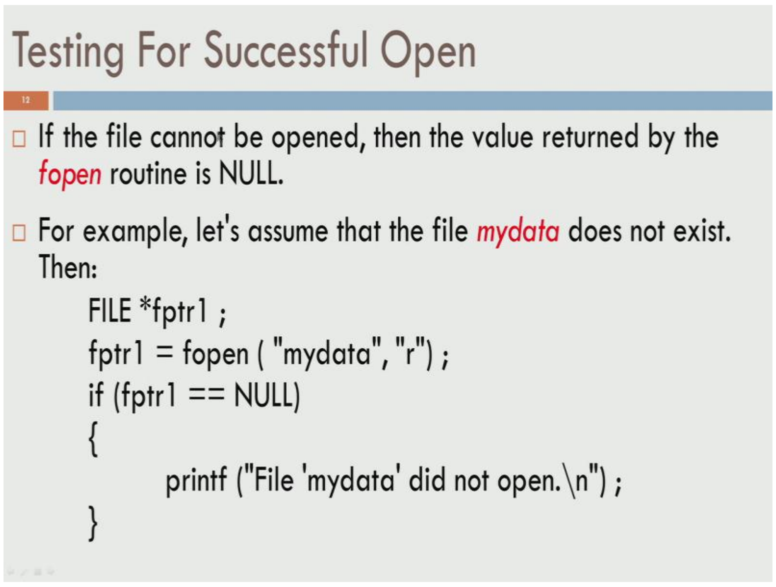
  fptr2 = fopen ("results", "w") ;

  would open the file *results* for output (writing).

- Once the files are open, they stay open until you close them or end the program (which will close all files.)

Let us now look at how to open a file, so we have fptr1 which says fopen mydata comma r, you are trying to open a file called mydata in read only mode and fptr1 should be file star fptr1. In this example, fptr2 is fopen results comma w, we are trying to write something into results file and so both of these once they are opened, you are essentially it is like opening a gate through which a stream can flow. So, once you open a file you can now start reading and writing, so you can either read from the stream or you can write to the stream and these files will remain open until you explicitly close them. So, if you do not explicitly close them, the files will remain open till the end of the program.

(Refer Slide Time: 10:15)



Testing For Successful Open

- If the file cannot be opened, then the value returned by the fopen routine is NULL.
- For example, let's assume that the file mydata does not exist. Then:

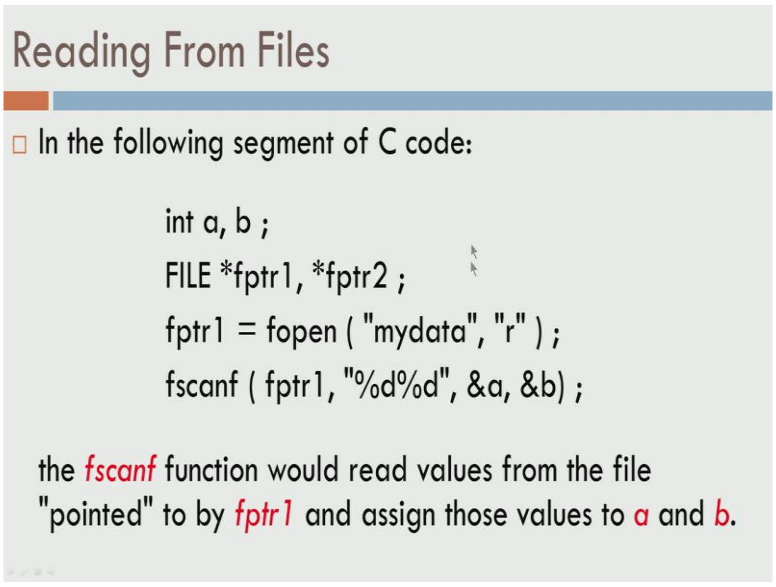
```
FILE *fptr1 ;
fptr1 = fopen ( "mydata", "r") ;
if (fptr1 == NULL)
{
        printf ("File 'mydata' did not open.\n") ;
}
```

So, let us look at how do you test for successful open, I mention this as a problem. So, I try to read a file and the file is not existent, what do I do now. So, let us see this example, file star fptr1 and you try to open mydata. So, it will try and open it from the directory in which the program is running from and if the file is does not exists, fptr1 will be supplied with a value called NULL. So, if fopen fails it will return NULL, however if fopen succeeds it will be a valid pointer.

So, you can use this information and go and check if fptr1 is null or not, if it is null you know that the file open fail for whatever reason, may be the file was not there, that is the most common reason in failed and you can say that file my mydata did not open and you have to tell the user to input another file name or some such thing. So, unsuccessful open for read is a problem, because you cannot proceed from there, if the file is not even there.

Whereas for write, fopen may not work, because you do not have enough space in your disk and so on, so there could be other reasons why fopen may fail, if you open it in write mode.

(Refer Slide Time: 11:29)



Reading From Files

- In the following segment of C code:

```
int a, b ;
FILE *fptr1, *fptr2 ;
fptr1 = fopen ( "mydata", "r" ) ;
fscanf ( fptr1, "%d%d", &a, &b) ;
```

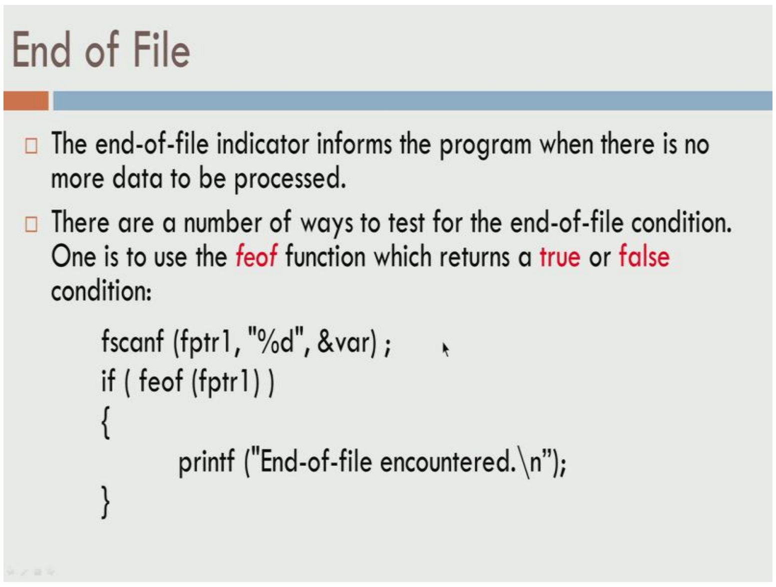
the *fscanf* function would read values from the file "pointed" to by *fptr1* and assign those values to *a* and *b*.

So, now let us get look at this second step, so we said we will open the file now, we know how to check the error if the file cannot open, let us look at the second step reading from the files. So, second step was either reading or writing, let us start with reading. So, int a comma b these are two integers and we have a pointer fptr1 which is of type file star, let us look at this line here. We are open the file, in this example there is no check on whether fptr1 is null or not, but let us see how to read two integers from the file.

Let us assume that fptr1 has two integers or the file mydata has two integers saved in it, you can, so this line here is trying to do a scan and you can see some similarity with scanf that you have seen before. We have been using this all along, so let us look at this. So, it is similar to scanf, except that there seems to be three difference sets of parameters, so this is the format that you give to fscanf.

So, this is the format in which you are going to read, this is the set of pointers that we are passing, so this is very similar to scanf, except that you also pass the file pointer fptr1. So, fscanf takes three parameters, one which is the file pointer, two which is the format, three which is the set of variables in which you have to read. So, in scanf all we have done is we got rid of the fptr1 because scanf by default always reads from stdin. So, fscanf once this is over, it would have read two integers from the file mydata.

(Refer Slide Time: 13:05)



End of File

- The end-of-file indicator informs the program when there is no more data to be processed.
- There are a number of ways to test for the end-of-file condition. One is to use the feof function which returns a true or false condition:

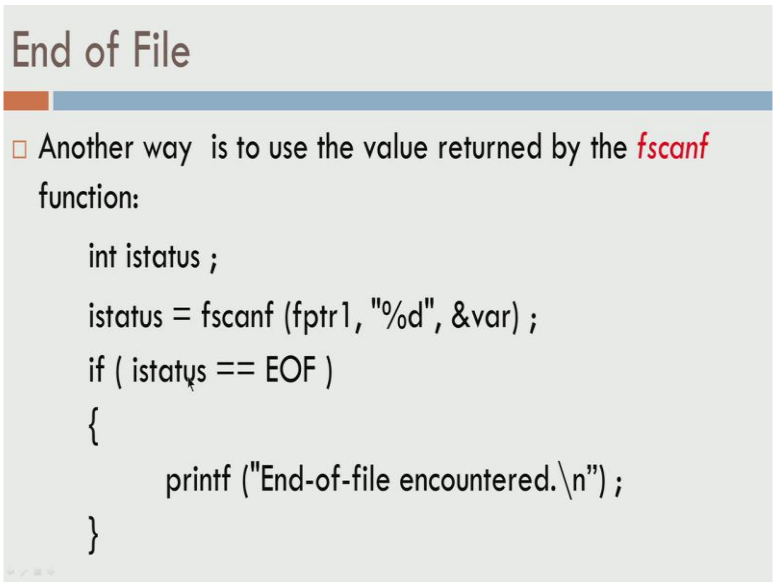
```
fscanf (fptr1, "%d", &var) ;
if ( feof (fptr1) )
{
        printf ("End-of-file encountered.\n");
}
```

So, when you read a file you could be writing a loop to read one line after the other, let us say I have a file in which I have ten lines of two integers each. I keep reading, at some point I have to find out that I have reached the end of the file, because once I reached the end of the file, for the program it is probably no processing to do any more. So, it has process the file and may be all the work that needs to be done is already done.

So, there could be some other file in which there are thirty lines, another file in which there are hundred lines and so on, you need a mechanism by which you do not have to know the number of lines in the file for your program to operate. And so you keep processing one line after the other and at some point you realize that you have to find out that you have reached the end of file and that is done by C using this function called feof.

So, if you do a scanf and let us say you are expecting something from the input stream and if I already reached the end of the file, then if you check on this thing, feof of fptr1, so this eof stands for end of file. So, in your file stream if you have reached the end of file of the pointer fptr1, it means that whatever that file that you are reading using fptr1 you have reached the end of the file, there is no more processing to do. So, this is one way to find out if you have reached the end of the file.

(Refer Slide Time: 14:32)



End of File

- Another way is to use the value returned by the *fscanf* function:

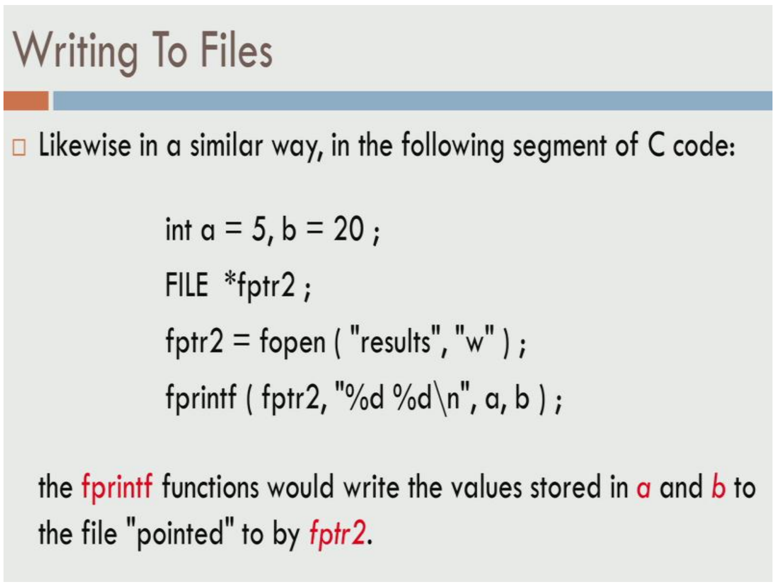
```
int istatus ;
istatus = fscanf (fptr1, "%d", &var) ;
if ( istatus == EOF )
{
        printf ("End-of-file encountered.\n”) ;
}
```

There is also another way in which you can check whether you have reached the end of file. You keep reading one line at a time, at some point if you reached the end of the file. Let us say you did this fscanf fptr1 percentage d ampersand var ((Refer Time: 14:45)) that is exactly what we had earlier, we do a scanf. Interestingly, scanf also returns a return value, so we never bothered checking the return of scanf, but fscanf returns a value which can be used to check whether you have reached the end of input stream or not.

So, there is a special character called a special value called EOF, so it is capital E capital O capital F. So, that is a special value, if fscanf returns this special value, then it means there was some problem in scanning the… So, it actually indicates that you have reached the end of the file. Otherwise, fscanf would return a value which is not EOF. So, if fscanf returns EOF, you know that you have reached the end of the file.

So, either you call feof of fptr1 or you call, you copy the value return value from fscanf and you check against EOF and it will tell you that the end of file has been encounter.

(Refer Slide Time: 15:42)



Writing To Files

- Likewise in a similar way, in the following segment of C code:

```
int a = 5, b = 20 ;
FILE  *fptr2 ;
fptr2 = fopen ( "results", "w" ) ;
fprintf ( fptr2, "%d %d\n", a, b ) ;
```

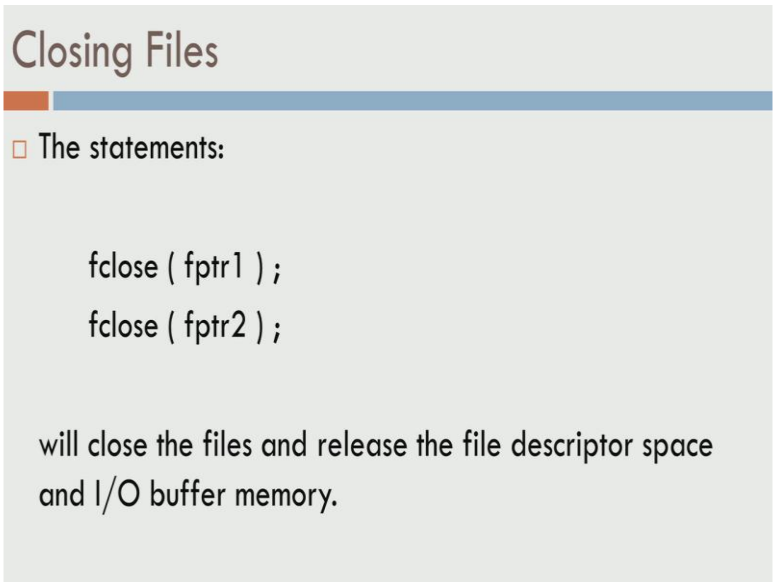
the fprintf functions would write the values stored in *a* and *b* to the file "pointed" to by *fptr2*.

So, that is one of the things that you can do in the second step, the other thing that you can do is you can actually go and write to files. So, for example if you look at this code as you can expect, it is a variation of printf. So, as before we have two integers and we have a file pointer called fptr2, we open the results file in write mode which means if the file exists, you will over write it, if it does not exists it opens a new file with the name results.

And now this fprintf is similar to printf, so you have the format specifier, you have the parameters that are supposed to be printed, but you also pass the file pointer as the first parameter. So, this line will actually write a comma b, the values a comma b into the file pointed by fptr2 and you keep writing and you will keep writing one line after the other till the end, till then you can, so you can stop writing.

So, for example if you have another fprintf fptr2 percentage d percentage d back slash n, let us say 5 comma 10, so that we will write one line of 5 comma 20 and another line would be 5 comma 10, you will see two lines in your file.

(Refer Slide Time: 16:50)



Closing Files

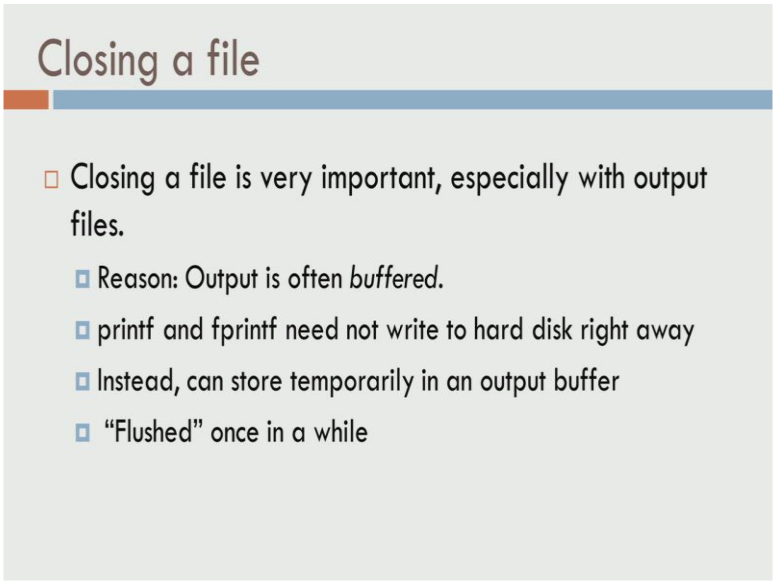
- The statements:

```
fclose ( fptr1 ) ;
fclose ( fptr2 ) ;
```

will close the files and release the file descriptor space and I/O buffer memory.

The third and final step in having file I/O done is closing the file, so fclose of fptr1 and fclose of fptr2 will close the file associated with the stream fptr1 and fptr2. So, and whatever extra work that it has been doing, it will release all of that back to the system. So, it is important that you actually close the files, because if you actually open a reading stream maybe it is not that become an issue, but if you have been writing to a file and if you did not close, you can run in to problems. So, closing a file is important because, usually output is buffered.

(Refer Slide Time: 17:29)



Closing a file

- Closing a file is very important, especially with output files.
  - Reason: Output is often *buffered.*
  - printf and fprintf need not write to hard disk right away
  - Instead, can store temporarily in an output buffer
  - “Flushed” once in a while

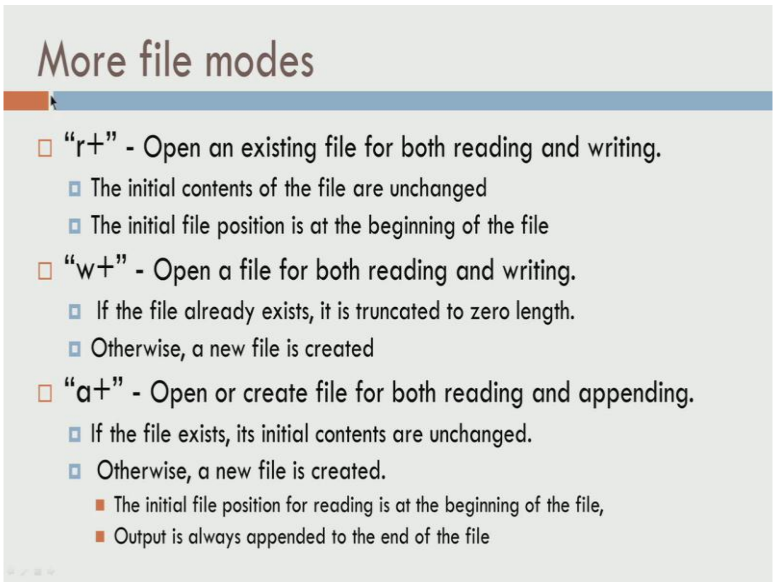
So, what we mean by that is let us say I want to printf to a screen, I wanted to do printf it may not appear in the screen immediately. So, what can happen is whatever I supposed to be return, it could get return to the buffer and eventually the buffer gets flushed to the actual device. So, the flushing could be to the hard disk or the flushing could be to your monitor and so on. So, the best example that I can give is if you use USB sticks, many times it tell you to remove the USB sticks, especially if I have written something on a USB memory drive, they ask you to eject it carefully.

So, you would go and click on something and you would say safely remove the device and so on. If we have wondered what you have been doing, what it actually does is when you remove, if you written anything to a device it is possible that the actual entries have not been written, the actual bits and bytes have not been return, they are still in a buffer sitting in the RAM and only when you ask the system to be flushed, then there will be a commit to a device.

So, this happens for various reasons which we cannot get in to now, but if that buffering happened and if you plugged our USB memory stick, your USB memory stick is not guarantee to have all the data. However, if you go through your operating system and say safely eject the device, it will flush all the buffers to your device and then you can remove it.

So, fclose essentially forces all the buffers to be flushed and all the contents that you wrote from the program is guaranteed to be written into the file. So, there are several other modes that you can use.

(Refer Slide Time: 19:20)



More file modes

- "r+" - Open an existing file for both reading and writing.
  - The initial contents of the file are unchanged
  - The initial file position is at the beginning of the file
- "w+" - Open a file for both reading and writing.
  - If the file already exists, it is truncated to zero length.
  - Otherwise, a new file is created
- "a+" - Open or create file for both reading and appending.
  - If the file exists, its initial contents are unchanged.
  - Otherwise, a new file is created.
    - The initial file position for reading is at the beginning of the file,
    - Output is always appended to the end of the file

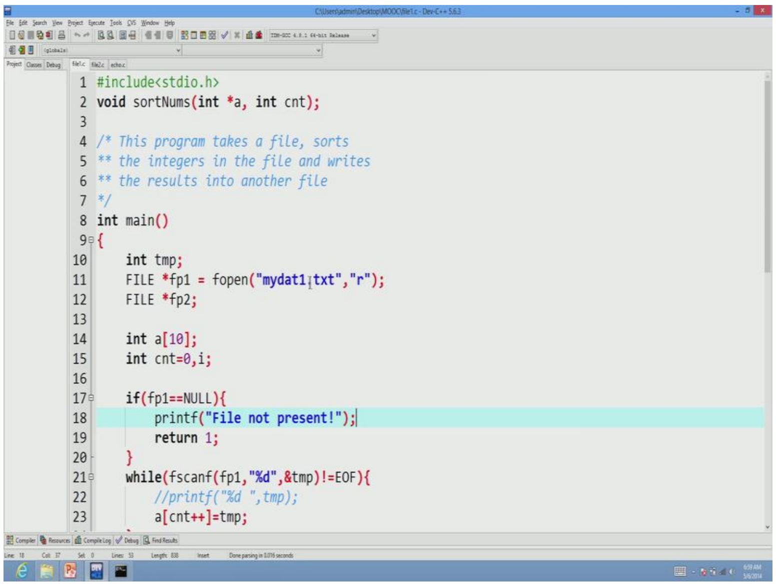
So, there is a mode called r plus, there is a mode called w plus and another called a plus. So, the plus indicate something, so this is the reason why we have been passing it as a string. It is not just one character, at times may have to pass more than one character, so in this case r plus w plus are a plus. So, r plus is it indicates that you want to be able to read, but you want something more from at this mode. So, you open an existing file for both reading and writing, because this is also something that you may need once in a while.

So, what happens is because you are opening in r, it will not delete the file it will not delete the contents. The initial contents of the file are unchanged and the initial position at which you can start doing your operations will be at the beginning of the file. So, if we want to over write only part of the file, you can use r plus, you opened with r and r plus and you can start write, over writing from the top and you can stop at some point instead of deleting all the contents.

W plus is also in some sense similar to what you are doing in r plus, but there is a key difference. So, both of them open a file for both reading and writing, but w plus being this notion that you destroy the file when you open, something in w mode it does exactly that. If the file already exists, it is truncated to zero length otherwise a new file is created and on this you will be able to do both reading and writing.

Finally, a plus is distinct from w plus, this also opens the file for both reading and writing, but if the file exists, it is initial contents are unchanged, in that senses it is similar to r plus. Otherwise, it will create a new file, in that sense it is similar to w plus. So, these are three other modes in which you can open a file. So, what we will do now is we will do a quick, I will show you a quick demo of file I/O, I have written some small programs to show you file I/O.

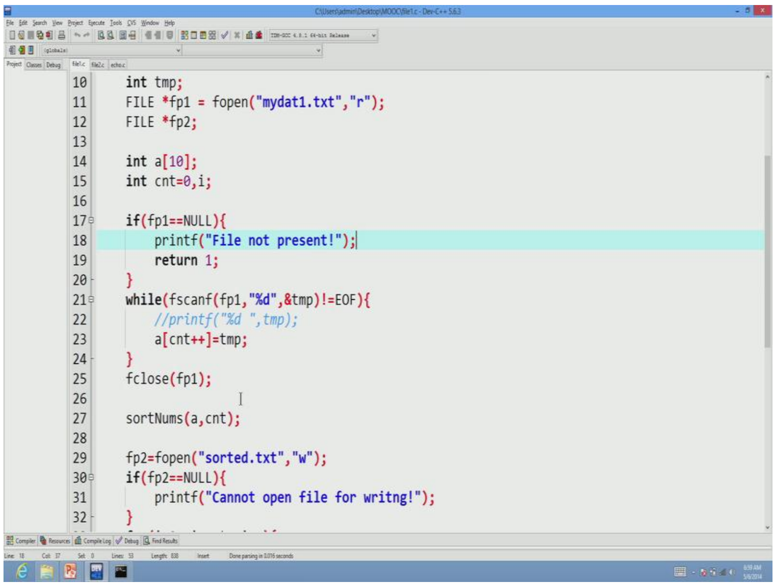
(Refer Slide Time: 21:13)



```
#include<stdio.h>
void sortNums(int *a, int cnt);

/* This program takes a file, sorts
** the integers in the file and writes
** the results into another file
*/
int main()
{
    int tmp;
    FILE *fp1 = fopen("mydat1.txt","r");
    FILE *fp2;

    int a[10];
    int cnt=0,i;

    if(fp1==NULL){
        printf("File not present!");
        return 1;
    }
    while(fscanf(fp1,"%d",&tmp)!=EOF){
        //printf("%d ",tmp);
        a[cnt++]=tmp;
```

So, there is this file called file1 dot c in which I have done something. So, I am going to open a file called mydat1 dot text in read only mode and this file is expected to have a bunch of integers and what I am going to do is I am going to sort the numbers which are in the file and I am going to write it to this file called sorted dot text.

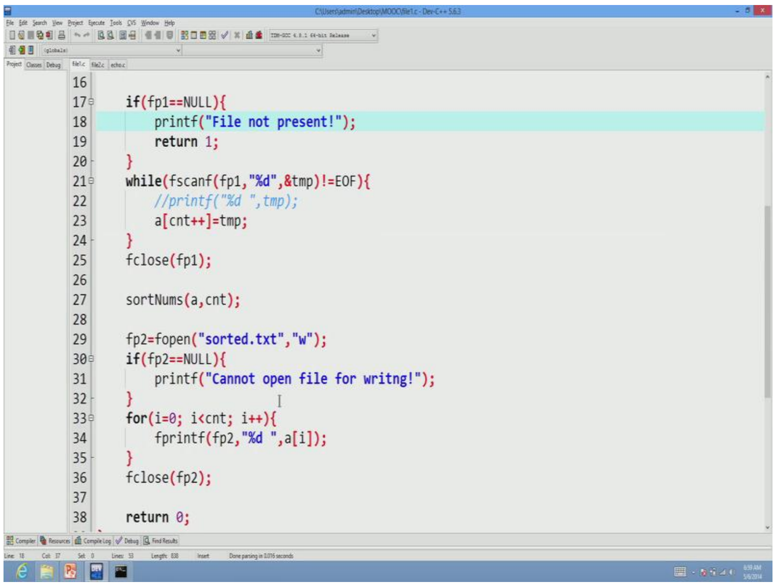
(Refer Slide Time: 21:28)



```c
    int tmp;
    FILE *fp1 = fopen("mydat1.txt","r");
    FILE *fp2;

    int a[10];
    int cnt=0,i;

    if(fp1==NULL){
        printf("File not present!");
        return 1;
    }
    while(fscanf(fp1,"%d",&tmp)!=EOF){
        //printf("%d ",tmp);
        a[cnt++]=tmp;
    }
    fclose(fp1);

    sortNums(a,cnt);

    fp2=fopen("sorted.txt","w");
    if(fp2==NULL){
        printf("Cannot open file for writng!");
    }
```

So, I am going to do three things, open a file called mydat1 dot text into an array, sort the array in your program and write the sorted results to your another file called sorted dot text that was this program achieves.

(Refer Slide Time: 21:49)



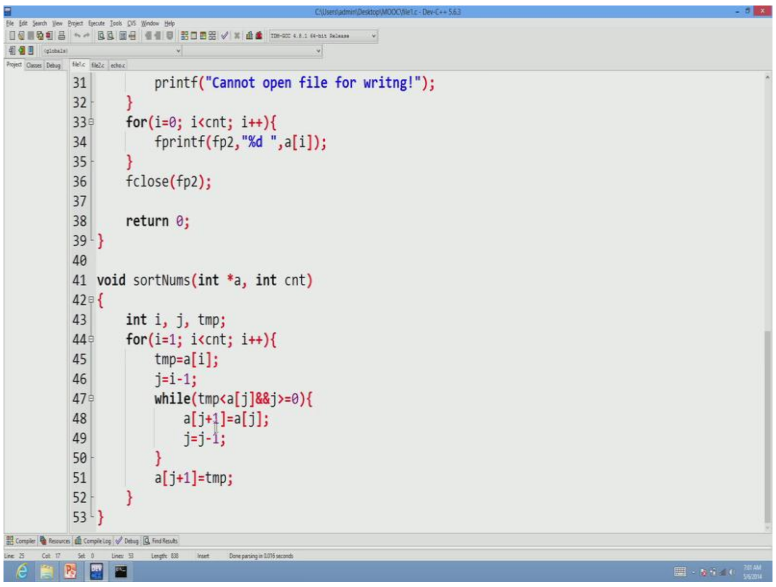
```c

    if(fp1==NULL){
        printf("File not present!");
        return 1;
    }
    while(fscanf(fp1,"%d",&tmp)!=EOF){
        //printf("%d ",tmp);
        a[cnt++]=tmp;
    }
    fclose(fp1);

    sortNums(a,cnt);

    fp2=fopen("sorted.txt","w");
    if(fp2==NULL){
        printf("Cannot open file for writng!");
    }
    for(i=0; i<cnt; i++){
        fprintf(fp2,"%d ",a[i]);
    }
    fclose(fp2);

    return 0;
```

Let us see how this is done. So, there are two file pointers fp1 and fp2, fp1 opens mydat1 dot text in read only mode and I also assume that the number of entries in the file is not more than 10. So, I assume that utmost ten integers was stored and nothing more. So, what I have done is I declare an array called a of 10 in size which means it can store up to ten integers and look at this line, line 21 through 24.

If you look at it, while fscanf for fp1 percentage d ampersand temp, what does fscanf do. Given a file pointer, it will try and use the format and read from the input stream, in this case from mydat1 into this variable called temp. So, when you read, it is possible that you have ten integers, but at some point you will hit the end of the file. So, fscanf you go and check, it will do this work fscanf fp1 percentage d ampersand temp.

So, this line actually reads, tries and reads from a file, but if it is unsuccessful and if it returns eof, the while loop will terminate. Otherwise, it keeps reading the contents into a and as long as there is 10 or less lines, a will not be over run, it will have ten entries. So, up to ten integers from this file mydat1 is, then I am calling this functions sortnums a comma cnt. I will not worry about sortnums now, so I have written a small program, a small piece of code which does insertion sort, so I will not worry about that.

(Refer Slide Time: 23:31)

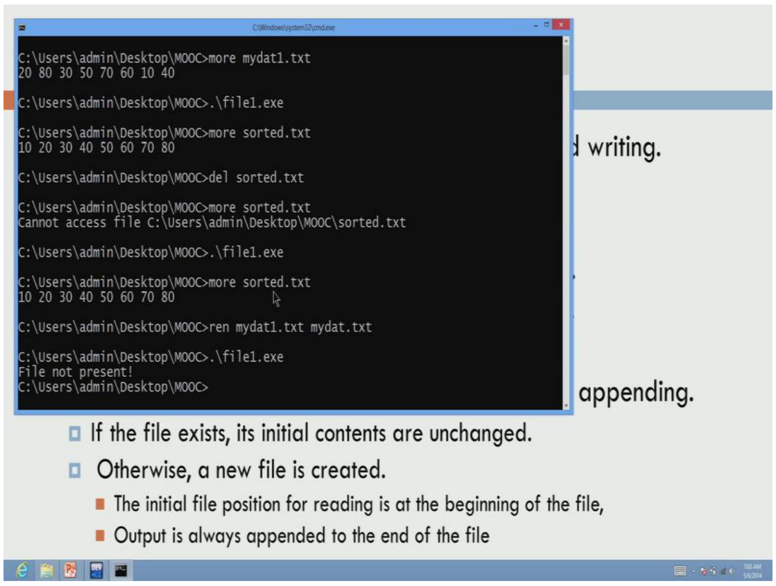


```
31          printf("Cannot open file for writng!");
32      }
33      for(i=0; i<cnt; i++){
34          fprintf(fp2,"%d ",a[i]);
35      }
36      fclose(fp2);
37
38      return 0;
39  }
40
41  void sortNums(int *a, int cnt)
42  {
43      int i, j, tmp;
44      for(i=1; i<cnt; i++){
45          tmp=a[i];
46          j=i-1;
47          while(tmp<a[j]&&j>=0){
48              a[j+1]=a[j];
49              j=j-1;
50          }
51          a[j+1]=tmp;
52      }
53  }
```

So, I will assume that a gets sorted and I am going to write things to a file and how do I do it, I open fp2 which is opening this sorted dot text in w mode. If fp2 is null, it cannot file the open for writing, otherwise you do fprintf fp2 percentage d a of i, it takes one a at a time and it write it is fp2 in the format integer. And one thing you can notice is line number 25 and line number 36 closes the two files.

So, this file you have done with reading at line number 24, you do not have anything more to read, so you can close the first file mydat1 dot text at line number 25. And since you have done with all the writing of all the variables that you, of all the values we can close this file at line number 36. So, I will compile this and I will show how it is going to run. So, I have compiled it. Let me show what the file has.

(Refer Slide Time: 24:35)



```
C:\Users\admin\Desktop\MOOC>more mydat1.txt
20 80 30 50 70 60 10 40

C:\Users\admin\Desktop\MOOC>.\file1.exe

C:\Users\admin\Desktop\MOOC>more sorted.txt
10 20 30 40 50 60 70 80

C:\Users\admin\Desktop\MOOC>del sorted.txt

C:\Users\admin\Desktop\MOOC>more sorted.txt
Cannot access file C:\Users\admin\Desktop\MOOC\sorted.txt

C:\Users\admin\Desktop\MOOC>.\file1.exe

C:\Users\admin\Desktop\MOOC>more sorted.txt
10 20 30 40 50 60 70 80

C:\Users\admin\Desktop\MOOC>ren mydat1.txt mydat.txt

C:\Users\admin\Desktop\MOOC>.\file1.exe
File not present!
C:\Users\admin\Desktop\MOOC>
```

d writing.

appending.

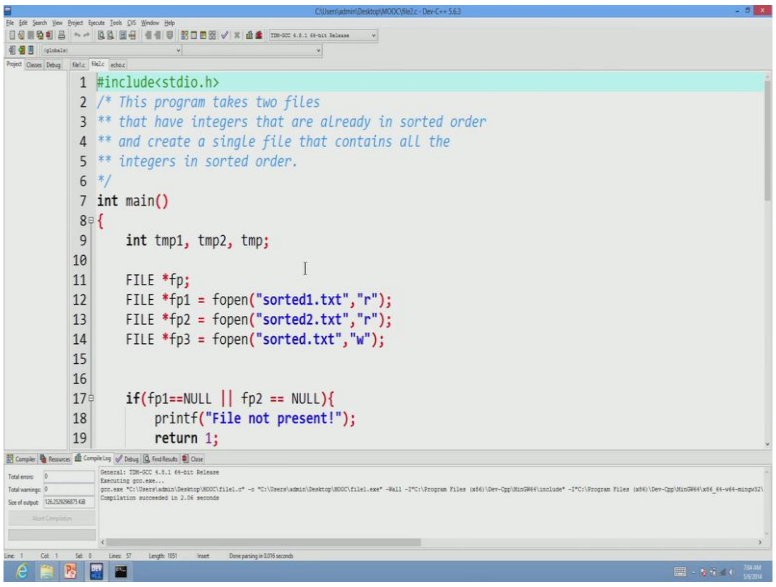
- If the file exists, its initial contents are unchanged.
- Otherwise, a new file is created.
  - The initial file position for reading is at the beginning of the file,
  - Output is always appended to the end of the file

So, I am going to show you what mydat1 dot text has, so it has a bunch of integers, so 20, 80, 30, 50, 70, 60, 10, 40. There are 80 integers clearly less than 10, so I am good. So, I have already compiled it earlier, so the executable is called file1 dot exe that is what I call the file as file1 dot c. When I compiles and gets file1 dot exe and when I run it by now what I should have done is it should have return something called sorted dot text.

You can see that the numbers 10 to 80 where all here in an unsorted order and they are all in sorted order now. So, just to show that this was actually created by the program, let me delete the sorted dot text, because it is a file that you are supposed to write every time the program is run. You can see that sorted dot text, there is no file by that name and if I run the program again and see what is in sorted dot text, you can see that the file got new entries now.

So, I deleted the file and when I ran the program it created the file called sorted dot text. So, for a while let us assume that this mydat1 dot text did not exist, I will instead call it mydat dot text, this was expected by the program mydat1, I renamed it. Now, if I run file1, so I printed this error called file not present, this file which is not present is mydat1 dot text, it would not change the sorted dot text, because I exited from the program. So, this is a small piece of code that I wanted to show. I have another piece of code which is also interesting.

(Refer Slide Time: 26:19)

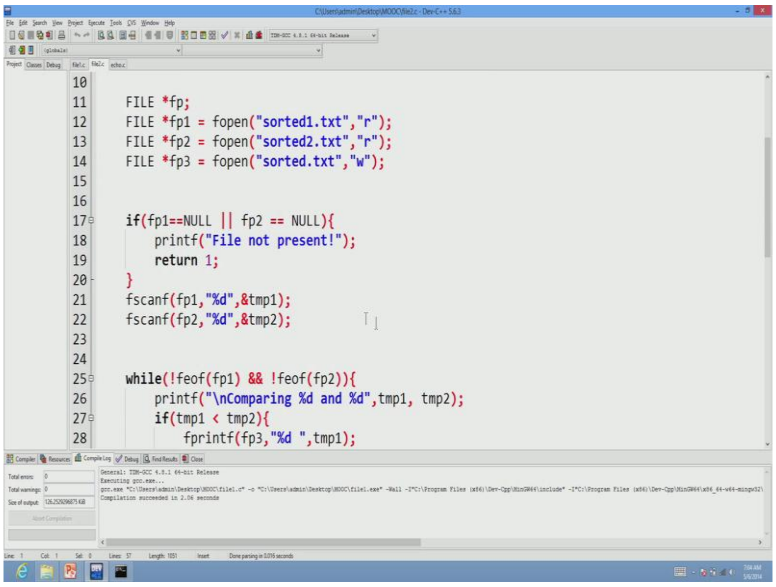


```c
#include<stdio.h>
/* This program takes two files
** that have integers that are already in sorted order
** and create a single file that contains all the
** integers in sorted order.
*/
int main()
{
    int tmp1, tmp2, tmp;

    FILE *fp;
    FILE *fp1 = fopen("sorted1.txt","r");
    FILE *fp2 = fopen("sorted2.txt","r");
    FILE *fp3 = fopen("sorted.txt","w");


    if(fp1==NULL || fp2 == NULL){
        printf("File not present!");
        return 1;
```

```
General: TDM-GCC 4.8.1 64-bit Release
Executing gcc.exe...
gcc.exe "C:\Users\admin\Desktop\MOOC\file1.c" -o "C:\Users\admin\Desktop\MOOC\file1.exe" -Wall -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\include" -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\x86_64-w64-mingw32\
Compilation succeeded in 2.06 seconds
```

So, it is possible that I have two files which are both sorted entries, now I want to merge these files. So, in the previous case one thing that happened was I actually knew that the contents of the file had only ten entries, but I may not know that up front. I need a mechanism by which I do not want to depend on any count. So, in this program what it does is it takes two files, it assumes that both these files are sorted and it will try and merge these two into one single sorted file. So, we are going to take two inputs sorted1 dot text and 2 dot text and we are going to write a file called sorted dot text. So, you can see that sorted dot text gets opened in write only mode.

(Refer Slide Time: 27:01)



```
10
11    FILE *fp;
12    FILE *fp1 = fopen("sorted1.txt","r");
13    FILE *fp2 = fopen("sorted2.txt","r");
14    FILE *fp3 = fopen("sorted.txt","w");
15
16
17    if(fp1==NULL || fp2 == NULL){
18        printf("File not present!");
19        return 1;
20    }
21    fscanf(fp1,"%d",&tmp1);
22    fscanf(fp2,"%d",&tmp2);
23
24
25    while(!feof(fp1) && !feof(fp2)){
26        printf("\nComparing %d and %d",tmp1, tmp2);
27        if(tmp1 < tmp2){
28            fprintf(fp3,"%d ",tmp1);
```

```
General: TDM-GCC 4.8.1 64-bit Release
Executing gcc.exe...
gcc.exe "C:\Users\admin\Desktop\MOOC\file1.c" -o "C:\Users\admin\Desktop\MOOC\file1.exe" -Wall -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\include" -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\x86_64-w64-mingw32\
Compilation succeeded in 2.06 seconds
```

Total errors: 0
Total warnings: 0
Size of output: 126.2529296875 KiB

And if either fp1 or fp2 file failed, you know that you cannot read, so it is not present. Otherwise, you scan one integer from each one of these streams, so you have one integer in stream one and one integer in stream two, you read that and then I have written a piece of program from line number 25 to 37 is a piece of program which runs on a while loop. So, what does it do, so I have two files which are of different lengths, it is possible that I will end up consuming everything from one stream before I even finish the other stream.

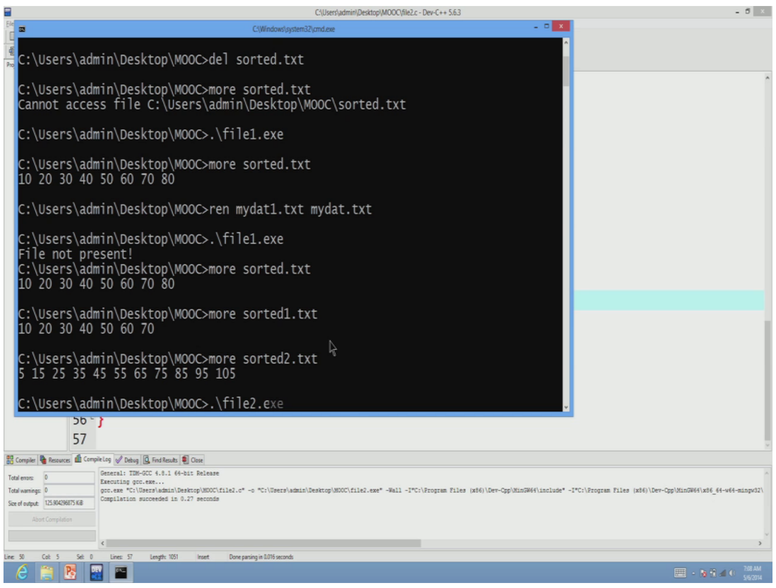
So, what I am going to check is have I not reached the end of stream one or stream two. If I have not reached the end of stream one and not of stream two either. As long as I have not reached the end of any of the streams, then I can always read two integers and I will be able to compare them and do something with it. So, at this point I have two integers that I have read, I have not reached the end of the stream which means I should be able to compare and write the results back.

So, the first stream you read into temp1, the second stream is reading into temp2, so you are comparing temp1 and temp2. If temp1 is less than temp2, then write temp1 into the output file and read one more integer from file1. Otherwise, if temp1 is greater than temp2 or equal to temp2, you write temp2 into the file and you read one more integer from file2, so you can see what is happening here. So, either the entry in file1 or the entry in file2 will be, one of them will be the larger than the other.

Here write that and move one step in that steam and what could happen is when you exists this loop, you may not have exhausted both the streams. One steam could end prematurely and other one has more numbers, but remember that the numbers are sorted. So, I can do something like what you did not merge sort, that is what I have been trying to do here. Line number 25 to 37 is merging two streams instead of merging two arrays, but if the arrays are of different size, then you have to just concatenate whatever is left out at the end of the resultant array and that is what this loop does.

So, line number 47 to 50, what it does is it, so this 39 to 46 finds out which stream got over. So, if fp1 got over fp2 should be copied, if fp2 got over fp1 and to the end should be copied and what line number 47 to 50 does is, it just copies whatever is left out in whichever file it is and it keeps scanning and keeps coping till the, that file is also exhausted. And once that is over, we can close all the streams fp1, 2 and 3 and you can return this 0. So, let me again compile it and there are no compilation errors.

(Refer Slide Time: 30:00)



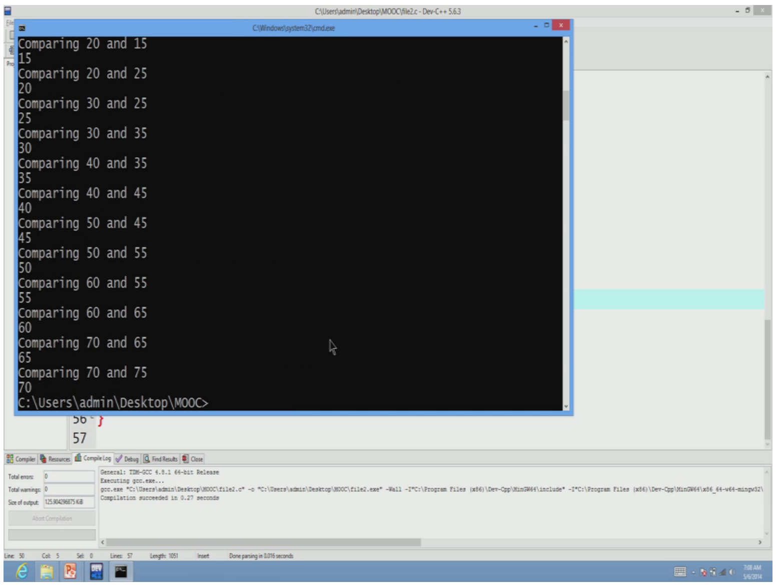
```
C:\Users\admin\Desktop\MOOC>del sorted.txt

C:\Users\admin\Desktop\MOOC>more sorted.txt
Cannot access file C:\Users\admin\Desktop\MOOC\sorted.txt

C:\Users\admin\Desktop\MOOC>.\file1.exe

C:\Users\admin\Desktop\MOOC>more sorted.txt
10 20 30 40 50 60 70 80

C:\Users\admin\Desktop\MOOC>ren mydat1.txt mydat.txt

C:\Users\admin\Desktop\MOOC>.\file1.exe
File not present!
C:\Users\admin\Desktop\MOOC>more sorted.txt
10 20 30 40 50 60 70 80

C:\Users\admin\Desktop\MOOC>more sorted1.txt
10 20 30 40 50 60 70

C:\Users\admin\Desktop\MOOC>more sorted2.txt
5 15 25 35 45 55 65 75 85 95 105

C:\Users\admin\Desktop\MOOC>.\file2.exe
```

```
General: TDM-GCC 4.8.1 64-bit Release
Executing gcc.exe...
gcc.exe "C:\Users\admin\Desktop\MOOC\file2.c" -o "C:\Users\admin\Desktop\MOOC\file2.exe" -Wall -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\include" -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\x86_64-w64-mingw32\
Compilation succeeded in 0.27 seconds
```

Total errors: 0
Total warnings: 0
Size of output: 125.904296875 KiB

So, let us look at sorted dot text, because that is the output we are going to write. So, it has 10 to 80 now, but let us look at sorted1 dot text which is one of the input files, it has 10 to 70 and I am sorted2 dot text I created with more values than sorted1 dot text, it has 5, 15, 25 all the way up to 1 naught 5. So, there are seven values here, but there are eleven values here. So, your program will finished this stream before it finishes this stream. So, we will have more work to do after this stream is processed, so I will run file2 now.

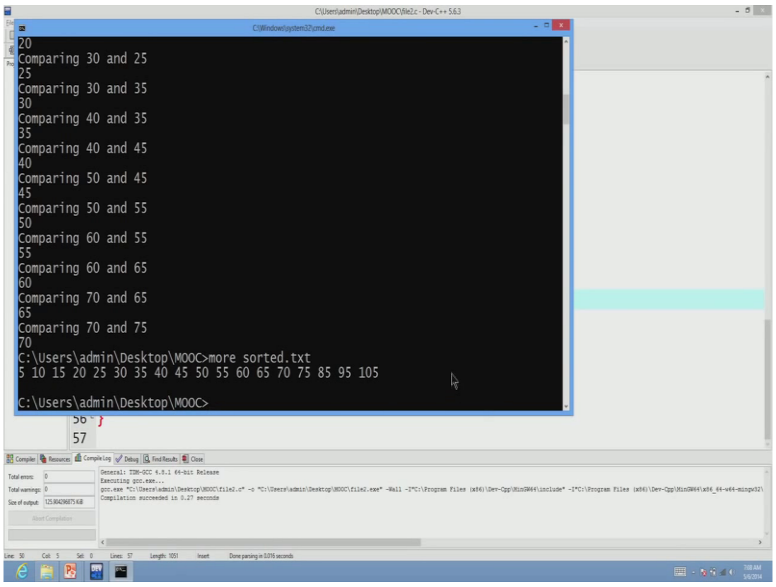
(Refer Slide Time: 30:38)



```
Comparing 20 and 15
15
Comparing 20 and 25
20
Comparing 30 and 25
25
Comparing 30 and 35
30
Comparing 40 and 35
35
Comparing 40 and 45
40
Comparing 50 and 45
45
Comparing 50 and 55
50
Comparing 60 and 55
55
Comparing 60 and 65
60
Comparing 70 and 65
65
Comparing 70 and 75
70
C:\Users\admin\Desktop\MOOC>
```

So, I run file2 and it was printing various things on the screen.

(Refer Slide Time: 30:42)



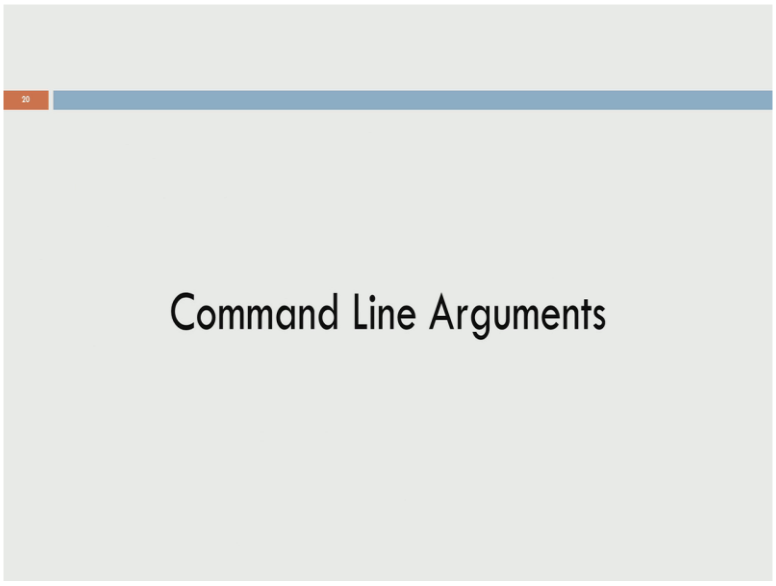
```
20
Comparing 30 and 25
25
Comparing 30 and 35
30
Comparing 40 and 35
35
Comparing 40 and 45
40
Comparing 50 and 45
45
Comparing 50 and 55
50
Comparing 60 and 55
55
Comparing 60 and 65
60
Comparing 70 and 65
65
Comparing 70 and 75
70
C:\Users\admin\Desktop\MOOC>more sorted.txt
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 85 95 105

C:\Users\admin\Desktop\MOOC>
```

Now, let us go and look at sorted dot text. So, it got over written first, sorted dot text was earlier written by file1 dot c. Now, I have file2 dot c, this is the second program and it over wrote the sorted dot text. Now, you can see that the entries of the resultant file are in sorted order. So, we read two files, the key thing I want you to see is ((Refer Time:

31:07)) this feof is another way to look at end of file and in this case, I am simultaneously forwarding both the file pointers by…

So, I am trying to see if either one of them reach the end, but one of the file pointers will move inside the loop, either this or this will move depending on which one was lower. Let us get back to the presentation, so, we looked at two examples in which we read and wrote two files.

(Refer Slide Time: 31:33)

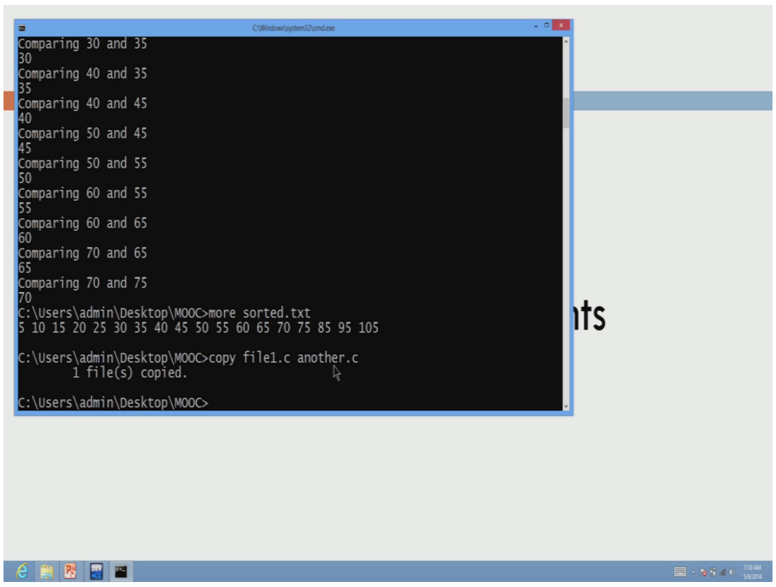


20

Command Line Arguments

The other key thing that I want you to learn is the notion of what are called command line arguments.

(Refer Slide Time: 31:43)

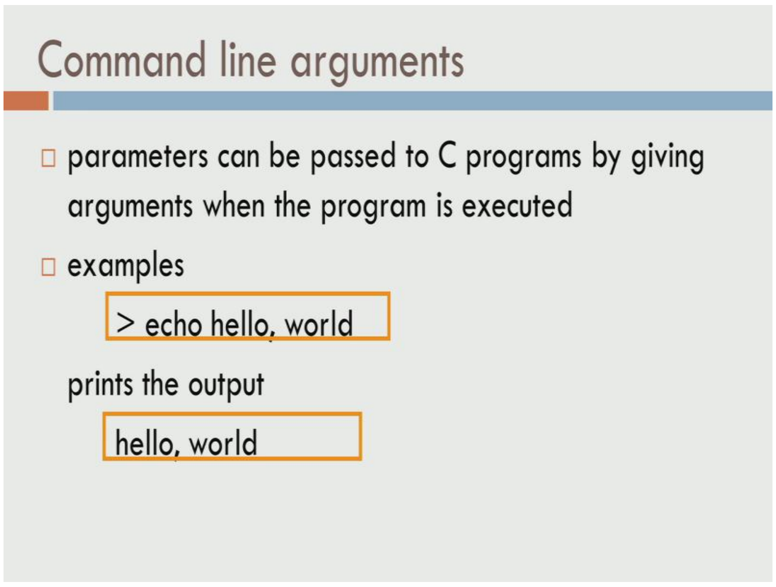


```
Comparing 30 and 35
30
Comparing 40 and 35
35
Comparing 40 and 45
40
Comparing 50 and 45
45
Comparing 50 and 55
50
Comparing 60 and 55
55
Comparing 60 and 65
60
Comparing 70 and 65
65
Comparing 70 and 75
70
C:\Users\admin\Desktop\MOOC>more sorted.txt
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 85 95 105

C:\Users\admin\Desktop\MOOC>copy file1.c another.c
        1 file(s) copied.

C:\Users\admin\Desktop\MOOC>
```

So, if you notice the things that I have been doing here, so let us say I want to copy file1 dot c to another dot c. So, I wrote this thing called, so should it is called copy, so copy is a windows command to copy one file to another, but, copy itself is a program. Somehow, I am able to pass values file1 dot c and another dot c here and I am able to get copy to do some work. I did not say copy and then I typed file1 dot c in keyboard and another dot c in the keyboard and so on.

When I launch the program itself I was able to give the parameters that it is needed and the copying got done. So, this is called command line parameter. So, we are in a command line, we are calling an executable and we have given two parameters file1 dot c and another dot c, we want to be able to process command line parameters. That is your, that is the next part of this video.

(Refer Slide Time: 32:41)



Command line arguments

- parameters can be passed to C programs by giving arguments when the program is executed
- examples

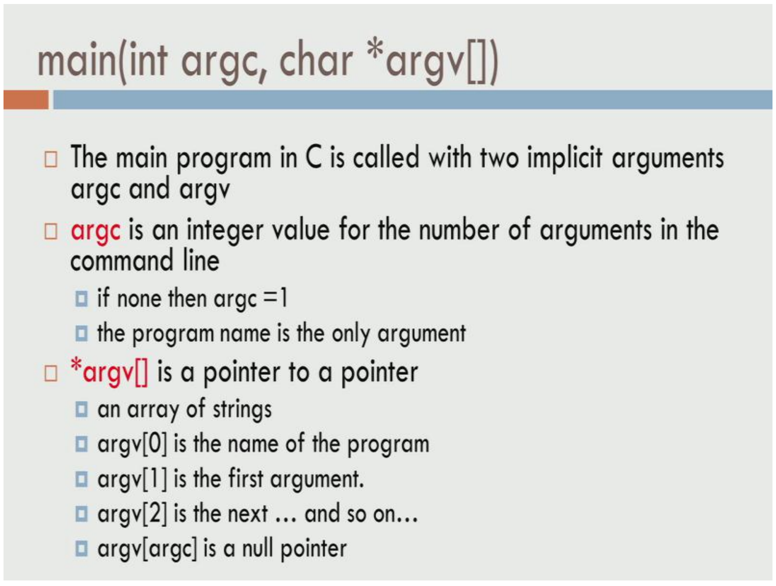
```
> echo hello, world
```

prints the output

```
hello, world
```

So, command line arguments work like this. You can pass arguments to C programs by passing them right when the program is executed. So, you can sit in the command line and type along with the parameters that you want. You do not have to type them in later when the program is running. So, for example if you typed echo, hello comma world, then you would see hello world printed on the screen. So, echo is actually taking command line parameters hello comma and world and it is processing it and it actually is printing on the screen.

(Refer Slide Time: 33:16)



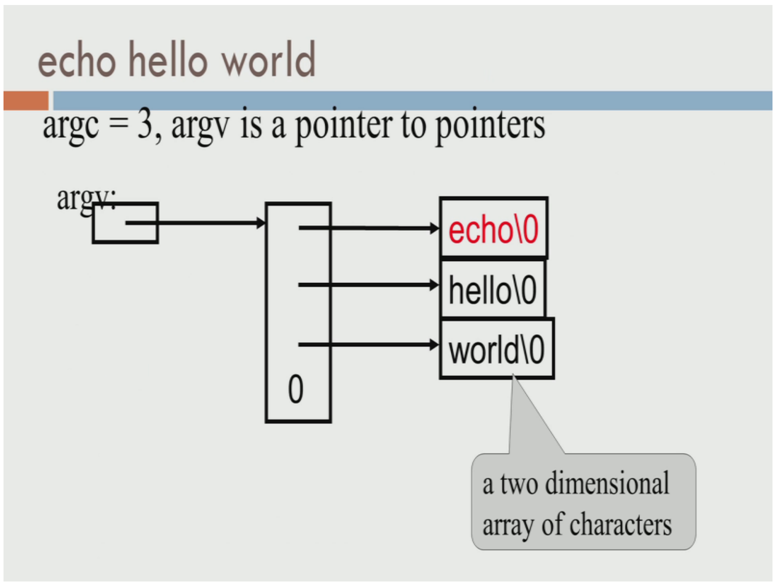
main(int argc, char *argv[])

- The main program in C is called with two implicit arguments argc and argv
- argc is an integer value for the number of arguments in the command line
  - if none then argc =1
  - the program name is the only argument
- *argv[] is a pointer to a pointer
  - an array of strings
  - argv[0] is the name of the program
  - argv[1] is the first argument.
  - argv[2] is the next … and so on…
  - argv[argc] is a null pointer

So, this is done using a very sophisticated mechanism. So, you remember the main program does not take arguments by default, so we never used... So, I even though I said main is a function we never passed any arguments to it, but you can actually pass two arguments int argc and characters star argv of array. So, argc is an integer whereas argv if we look at it, it is a pointer to a pointer. So, it is a set of pointers to character pointers.

So, argv this square bracket means this is a set of pointers and what does it pointing to it is character star. So, it is a pointer to pointer, so you can also think of it, it is an array of strings. So, argv of 0 is usually the name of the program, argv of 1 is the first argument that you have passed, argv of 2 is a second and so on. So, argc is an integer value that takes the number of arguments including the file name itself. So, for example ((Refer Time: 34:17)) if I did copy file1 dot c, another dot c this is, there are three arguments copy file1 dot c and another dot c. So, argc would be 3, argv of 0 would be copy, argv of 1 would be file1 dot c and argv of 2 would be another dot c.

(Refer Slide Time: 34:35)



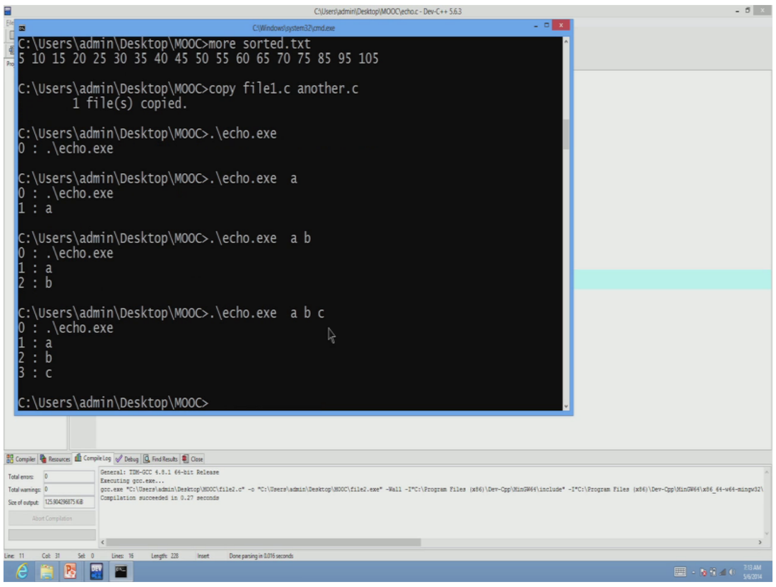
echo hello world

argc = 3, argv is a pointer to pointers

So, what I have, what really happens is you get argc equal to 3 and argv is a pointer to a pointer and what do you have, you have three pointers which are pointing to echo, hello and world, remember these are strings. So, you have a backslash 0 at the end and the fourth pointer actually points to null, so you have in some sense a two dimensional array of characters. Again as before, I have written a small program which does something very simple ((Refer Time: 34:59)).

So, it takes argc and character star argv, all it does is it iterates over all the arguments and prints them on the screen. So, it prints what is the argument number and what is the value of the argument. So, i runs from 0 to less than argc, so it goes 0, 1, 2, 3 up to argc minus 1 and it prints i comma argv of i. So, I have already compiled this program, let me run it now.

(Refer Slide Time: 35:23)

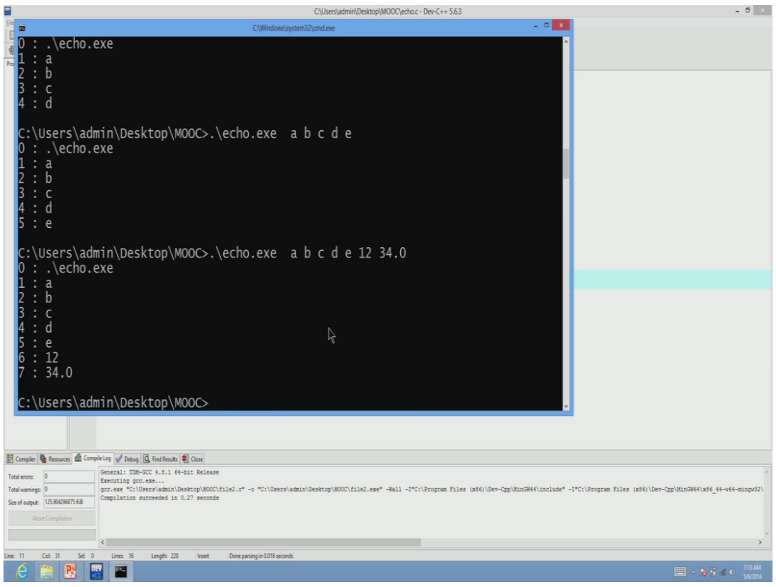


```
C:\Users\admin\Desktop\MOOC>more sorted.txt
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 85 95 105

C:\Users\admin\Desktop\MOOC>copy file1.c another.c
        1 file(s) copied.

C:\Users\admin\Desktop\MOOC>.\echo.exe
0 : .\echo.exe

C:\Users\admin\Desktop\MOOC>.\echo.exe  a
0 : .\echo.exe
1 : a

C:\Users\admin\Desktop\MOOC>.\echo.exe  a b
0 : .\echo.exe
1 : a
2 : b

C:\Users\admin\Desktop\MOOC>.\echo.exe  a b c
0 : .\echo.exe
1 : a
2 : b
3 : c

C:\Users\admin\Desktop\MOOC>
```

```
General: TDM-GCC 4.8.1 64-bit Release
Executing gcc.exe...
gcc.exe "C:\Users\admin\Desktop\MOOC\file2.c" -o "C:\Users\admin\Desktop\MOOC\file2.exe" -Wall -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\include" -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\x86_64-w64-mingw32\
Compilation succeeded in 0.27 seconds
```

So, I call this program echo, let us say I typed only echo dot exe, I did not pass any parameters. It just gives me the 0th argv which is echo dot c, echo dot exe itself. Let us say now I give echo a, a is one of the parameters. So, this is the 0th argv and this is the 1th argv, so you get those two, I can pass an arbitrary number of inputs, see this, so you have 0, 1, 2. So, you have three arguments echo dot exe a and b, I can do this and so on and it is not necessary that you pass only one variable type, I have passed an integer here, then I can pass the real value here, you can do any of those.

(Refer Slide time: 36:10)



```
0 : .\echo.exe
1 : a
2 : b
3 : c
4 : d

C:\Users\admin\Desktop\MOOC>.\echo.exe  a b c d e
0 : .\echo.exe
1 : a
2 : b
3 : c
4 : d
5 : e

C:\Users\admin\Desktop\MOOC>.\echo.exe  a b c d e 12 34.0
0 : .\echo.exe
1 : a
2 : b
3 : c
4 : d
5 : e
6 : 12
7 : 34.0

C:\Users\admin\Desktop\MOOC>
```

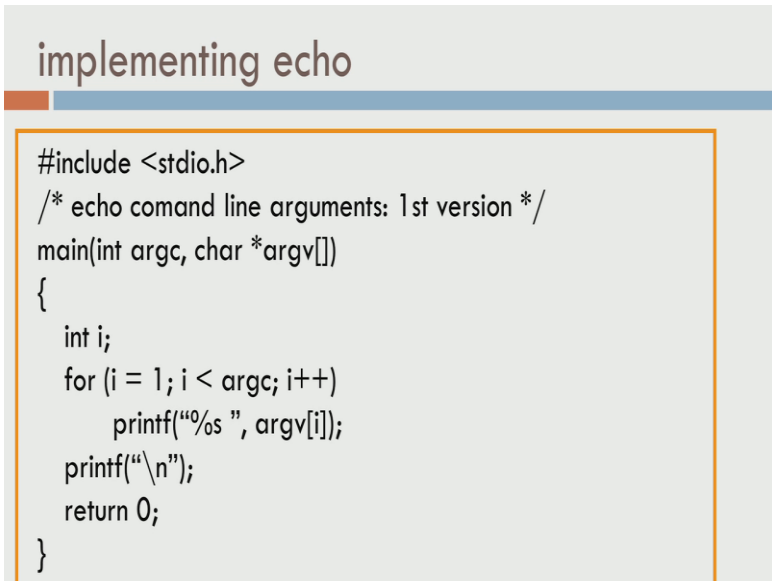
```
Total errors: 0
Total warnings: 0
Size of output: 125.904296875 KiB
```

```
General: TDM-GCC 4.8.1 64-bit Release
Executing gcc.exe...
gcc.exe "C:\Users\admin\Desktop\MOOC\file2.c" -o "C:\Users\admin\Desktop\MOOC\file2.exe" -Wall -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\include" -I"C:\Program Files (x86)\Dev-Cpp\MinGW64\x86_64-w64-mingw32\
Compilation succeeded in 0.27 seconds
```

So, anything that you pass in the command line will be passed to the program as argv. So, they are treated as strings inside and I am printing one string after the other. So, this comes in handy, whenever you do not want the keyboard input at all from the program. Even to read a file name, let us say I want to write a program like this, copy file1 dot c to file2 dot c, so I do not want the user to type it later.

Instead, I want that to be given in a command line itself and automatically this file, so you will treat that as a string, you have to open the file by that name, read it, open the other file in write mode, write it and so on, all of that is there. But, you have passed the file names right in the command line itself. So, these are two things that is very useful, the notion of being able to do file I/O and the notion of being able to take command line arguments.

(Refer Slide Time: 37:02)



implementing echo

```
#include <stdio.h>
/* echo comand line arguments: 1st version */
main(int argc, char *argv[])
{
   int i;
   for (i = 1; i < argc; i++)
         printf("%s ", argv[i]);
   printf("\n");
   return 0;
}
```

So, that is a same program that is here and this brings us to the end of this module.