Performing Read/Write Operations in a text file

(Unit-5)

- Now let us see how to perform r/w operations on a text file. in C++.
- C++ provides some built in classes to perform r/w operations on a file.
- The class hierarchy is as follows.

$$\begin{tikzcd}
 & ios \arrow[dl] \arrow[dr] \arrow[drr] & & \\
istream \arrow[dd] \arrow[dr] & & ostream \arrow[dl] \arrow[dd] & Streambuf \arrow[dd] \\
 & iostream \arrow[d] & & \\
ifstream & fstream & ofstream. & filebuf.
\end{tikzcd}$$

- ios the base class.
- istream is for i/p operations
- ostream " " o/p ".
- iostream is " i/o ".

~~• ios is the~~

• for operations on a file

- • ifstream → for i/p oprⁿ on a text file
- • ofstream → " o/p " " " " "
- • fstream → for i/o " " " " "

• istream, ostream & iostream classes r declared in <iostream> header file.

• ifstream, ofstream, fstream r declared in <fstream> header file

• These r used for operⁿ on a txt file.

• streambuf & filebuf r b/v for low level implementation of I/o oprⁿ.

what r the steps to perform R/W operations on a file

1) Create a stream
2) Open the file using stream & specify modes
3) Check if file open success or failed
4) Perform R/W on file
5) Close the stream.

1) Creating a stream

- A file stream can be created by creating an obj of ifstream, ofstream, fstream class.
- stream can be of 3 types based on type of i/o operation
- i/p stream :- It is used to perform i/p oprn on the file i.e read oprn on file.
- o/p stream :- for write oprn on file
- i/o " :- for r/w oprn on file.

- To create an ~~ifstream~~ i/p stream, create an obj of ifstream class

$$\underbrace{\text{ifstream}}_{\text{class name}} \quad \underbrace{\text{myin}}_{\text{obj name}};$$

eg ifstream myin;

here myin is an i/p stream.

- Similarly create objects of ofstream & fstream classes.

eg ofstream $\underbrace{\text{myout}}_{\text{o/p stream}}$; fstream $\underbrace{\text{myinout}}_{\text{i/o stream}}$;

2) link a file to a stream

- To link a file to a stream use open function.

eg to link a file to an i/p stream

```
ifstream myin; // create an i/p stream
myin.open("file.txt", mode);
```

↓

mode can have these values
- ios :: in
- ios :: out
- ios :: binary
- ios :: trunc
- ios :: app
- ios :: ate.

- open fun takes 2 parameters first is filename, second is mode.
- mode specifies the property of a file when it is opened.
- Modes available :-

1) ios:: in → use this mode when file is opened for read operation

2) ios :: out → use this when file opened for write opr^n.

3) ios :: binary → file is opened in binary mode i.e stream will behave as binary stream.

~~4) ios::~~ • by def file is opened in text / char stream.

4) ios :: ate

- ate stands for 'at end'.
- The file cursor will be moved to the end of file.
- However, file cursor can be moved to any other pos as well.
- Valid for both R/W operations.

5) ios :: app

- app stands for append
- Valid only for write operations
- by def, all the previous data is ~~overwritten~~ overwritten in write mode.
- Using app mode allows you to write to the end of file.
- So previous data is not deleted.

ate	app
• Valid for both R/W	• Valid for write opr^n only.
• file cursor can be moved to any other pos	• Can't be moved.

6) ios::trunc

- trunc stands for truncate
- trunc means to delete.
- It will delete all the data in the file.
- By def., o/p stream is opened in trunc mode.

Points :-

1) ios:: in mode is only for ~~read opx~~ i/p stream or i/o stream

2) ios::out " " " o/p stream or i/o "

3) ios:: binary valid for both i/p & o/p.

4) ios:: app can be used ~~valid~~ for o/p stream. only

Default modes for diff streams :-

If no mode is specified, then

1) ios:: in is def. for i/p stream
2) ios::out | ios::trunc " o/p "
3) ios:: in | ios::out " " " " " " "

opening a file thru constructor

- Each stream obj provides a constructor wch. can be used to directly open text files.
- ∴ open fun need not be used in this case

eg ifstream myin ("file.txt");

$$\begin{tikzcd} \text{ifstream myin} \arrow[d] & \text{("file.txt")} \arrow[d] \\ \text{Create i/p stream} & \text{pass value to the constructor} \end{tikzcd}$$

eg2 o/p stream :

```
ofstream myout ("file.txt");
```

eg3 i/o stream

```
fstream myinout ("file.txt");
```

Note:- No modes r specified, so def. modes will be used.

3) Checking if the file opened succesfully

- Next step is to check whether the file opened successfully.
- This can be done by checking the status of the stream.
- If open fails, then stream will be false o/wise true

eg
```
ifstream myin("file.txt"); // def mode
                              is ios::in
if (!myin) // Check stream state
{
    cout<<"Can't open";     } //file can't
    return 0;               }   be
}                               opened,
                                exit program.
```

5) Closing a file

- After performing R/W opr on a file the ~~stream~~ file shud be closed by using close fun

eg `myin.close();`

- When write opr r performed, the file will be saved when it is closed.

Summary

• The overall prog structure will be as follows :-

```
#include <fstream> // include this hdr file

~~mystream~~
ofstream myout ("file.txt"); //create an
                                  o/p stream.
if (! myout)
{     cout<<" Can't open";
      return 0;
}
   ≡  } Perform R/W opr

myout.close (); //close file
```

- Cout & Cin streams
- Buffering

Q) How does cout & cin perform R/W opr?

A) When a prog runs, the def streams r created
  → cout for o/p
  → cin " i/p
  → cerr " error (buffered)
  → Clog " " stream (unbuffered)

- Cout is an obj of ostream class
- Cin . . - - - - istream "

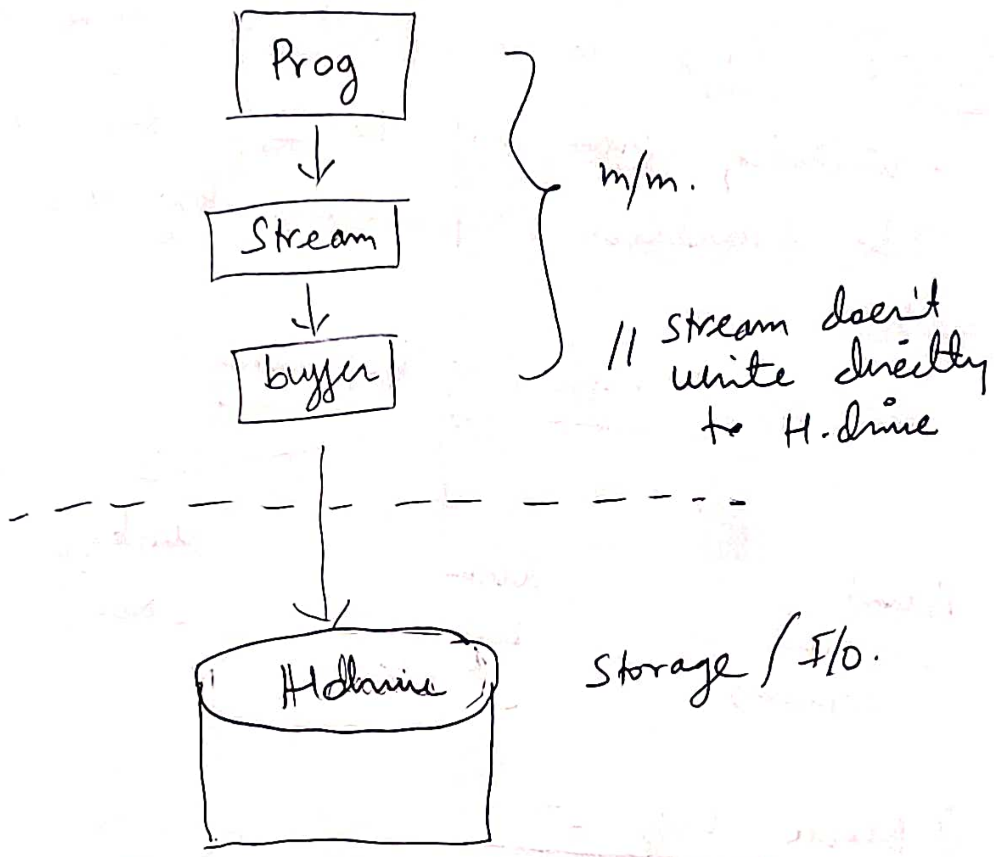
These classes r declared in <iostream>

- A user can W/R using these obj.

Q) What is the concept of stream buffering?

A. • When a prog performs o/p opr to H.drive, then it will not directly write to H.drive

- Instead, it writes data to a buffer & then data will be written to H.drive from the buffer.



Q) why is buffering done?

A • Prog r run in m/m & m/m opr r much faster than I/O opr

• So if a prog writes directly to I/O then it will take ~~a~~ a lot of time.

• So instead, the prog writes to a buffer & continues with the execution.

• Parallely, the data will be written to the I/O dev. from the buffer.

• So the prog. doesn't spend much time to perform the I/O.

• I/O opr will be performed b/w buffer & Hdrive separately.

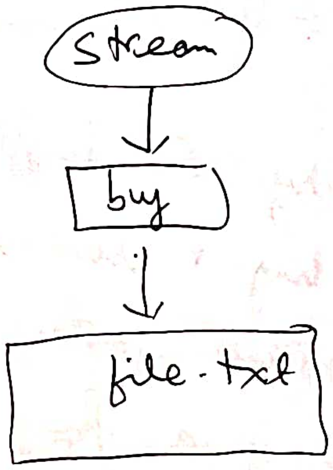
Note :- Buffer is a part of m/m (RAM)

- Writing from buf to I/o dev will be handled separately by os.

$$\begin{tikzcd}
 & \text{Types of buffering} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Block buffered streams} & \text{line buf S} & \text{unbuff strea.}
\end{tikzcd}$$

1) Block Buff Str..

- In this case, buff is done until a fixed ~~amt.~~ amount of data (say 500bytes) is filled in the buff.



- When the buff is full, data will be written to the file.
- Data will not be written to file until buf is full.

2) line buffered stream

- In this case, data will be kept in buffer until a new line char arrives
- Data will not be written to file until newline arrives in buf.

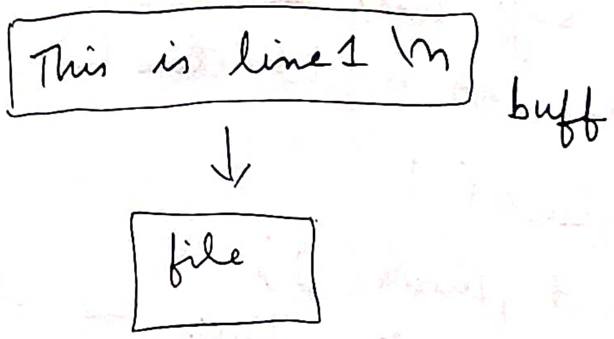
• Streams (cin, cout) r line buffered by def.

~~eg This is L1 \n~~

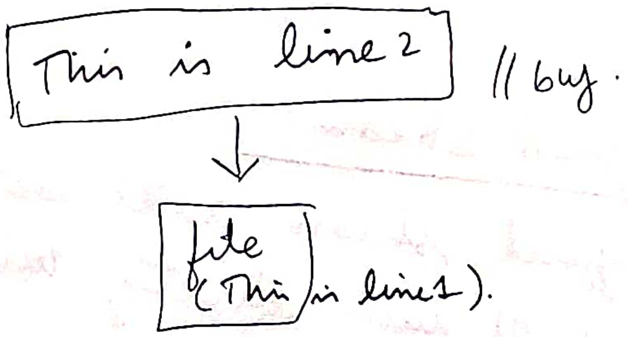
eg "This is line1 \n This is line 2"

• Suppose this text is written to a file.

• stream will write this to buffer.



//when line arrives it will be written to file.



//when line2 arrives, it will not be written to file bcoz there is no newline char in buf.

Q) How to force the ~~buf~~ buf to write its contents to the file?

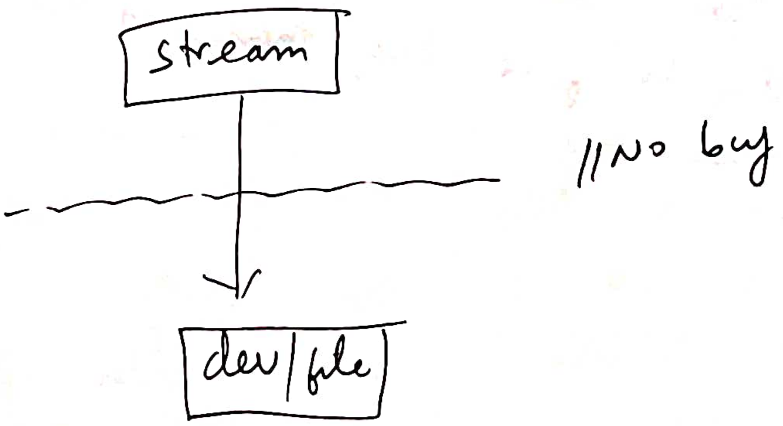
A There r two ways :-

1) when the file is closed or saved ~~using~~ using close() fun all the data in buff will be written to the file eg myout.close()

2) using fflush()
eg myout.fflush();
This fun forces the buffer to flush contents i.e to write remaining to the file

Unbuffered Stream

• An unbuffered stream will not buff the data. It will directly write to the dev.

•


. stderr is unbuff by def bcoz in case of err, data needs to be immediately written to the dev.

. O/wise the prog will terminate & ~~at~~ data will be lost.