CSCI251/CSCI851 Autumn-2020
Advanced Programming (**S2f**)

C++ Foundations VI:
Handling files

Outline

- Text file streams.
  - Errors in opening files.
  - Errors in reading files.
- Character input.
- Buffering.
- Binary I/O.

Text File Streams

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

int main()
{
    ifstream inData;                    //declare an input file stream
    ofstream outData;                   //declare an output file stream
    string firstName, lastName;

    inData.open("names.txt");           // open input file
    outData.open("marks.txt");          // open output file


    inData >> firstName >> lastName;    // read from a file stream
    outData << 85.6;                    // write into a file stream

    inData.close();                     // close the input file
    outData.close();                    // close the output file
    return 0;
}
```

Unbounded file streams …

- On the previous slide we opened an input stream, and we opened an output stream.
- We could have used `fstream`, an unbounded file stream allowing both reading and writing.

```
fstream fstrm;
fstrm.open(s, mode);
```

- Modes, for this and the i-o versions, partially:

```
in, out, app, ate, trunc, binary
```

- Multiple modes can be set.
- They are part of `fstream::` …

- `app`: Always write at the end.
- `ate`: On opening go the end.
- `trunc`: Wipes out existing content.
- `binary`: Doesn't have the text conversion layer, you read/write in binary.

- There are constraints on these flags.
- Some examples:
  - `out` is for `fstream` or `ofstream`.
  - `trunc` can only be set if `out` is.
  - `app` and `trunc` are mutually exclusive.

Errors in opening files …

- Don’t assume a file stream has been opened successfully.
  - Incorrect file name:

```
inFile.open("names.txl");
```

  - Incorrect file opening mode:

```
ifstream inFile;
inFile.open("names.txt", ios::trunc);
```

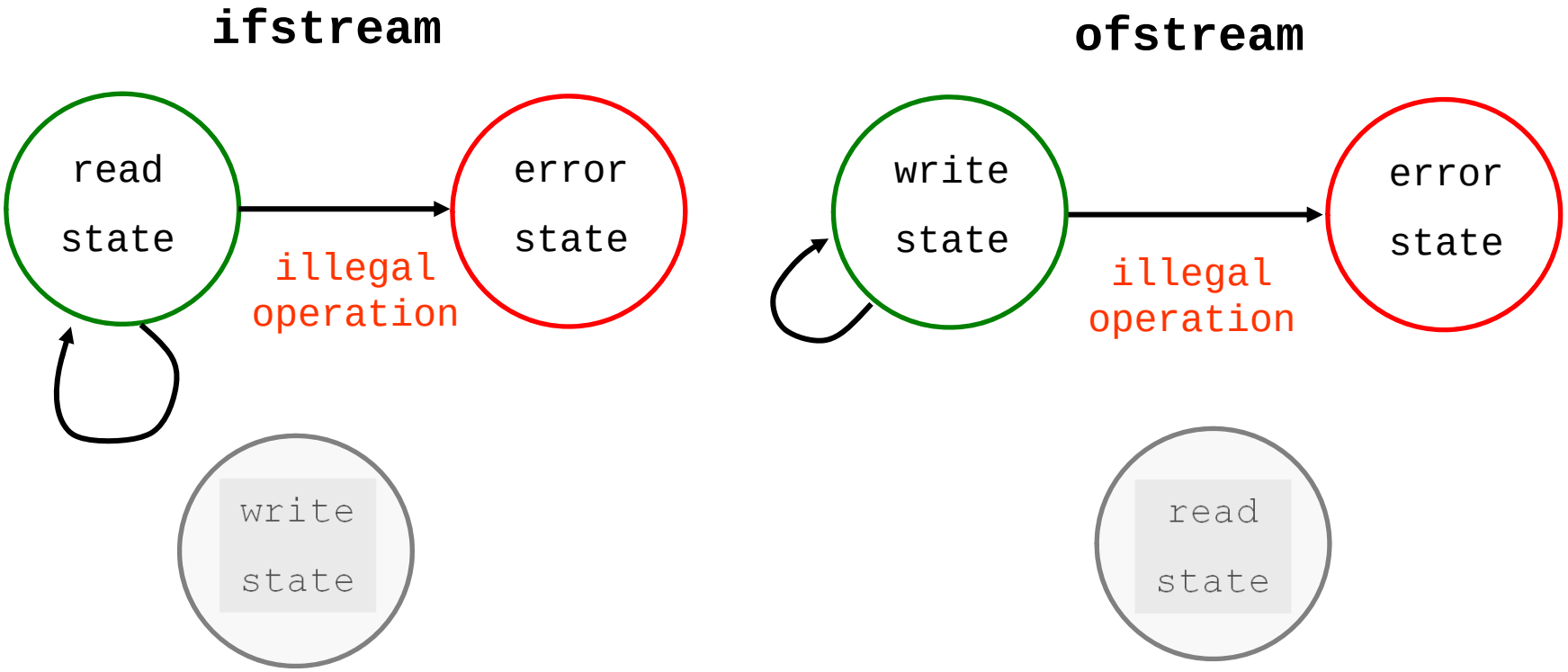
  - Not enough room on the hard drive.
  - Hardware failure.
- Always check the status of a stream after open.

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

int main()
{
    ifstream inData;       // declare an input file stream
    char fileName[] = "exams.txt";
    string lastName, mark;

    inData.open( fileName );       // open input file
    if (!inData)           // check if opened successfully
    {
      cerr << "Error opening : " << filename << endl;
      return -1;           // exit with an error code
    }
    inData >> lastName >> mark;

    inData.close();        // Close the input file
    return 0;
}
```

Errors in reading files …

- So the file seemed to open okay but …
- … being pessimistic, what goes wrong next.
  - The program may not have data to read as it hits the end of file.
  - The data may be invalid: … an alphabetic character instead of a digit character; a control character instead of an alphabetic one; etc.
  - The data may not be physically accessed from the disk due to its damage or network failure.

Error States



Any illegal operation with a stream switches it to the Error State !

- Comprehensive error checking is needed, and appropriate error recovery.
  - C++ provides three status flags and four functions to detect possible errors.

1. The flag `eof` indicates that the end of file is reached.

```
if( inData.eof() )  {  Error recovery action }
```

2. The flag `fail` indicates a failure due to invalid data.

```
if( inData.fail() ) {  Error recovery action }
```

3. The flag `bad` indicates a hardware problem.

```
if( inData.bad() ) {  Error recovery action }
```

4. The function `good( )` returns true if no any error has been detected.

- Note that files are viewed here as sequences of bytes with an end-of-file character at the end.

What happens?

- A text file has content:

```
1 2 3
```

- Assuming appropriate headers etc …

```
while( ! inData.eof() )
  {
      inData >> number;
      cout << number << " ";
  }
```

- Produces output …

```
1 2 3 3
```

Caution: `eof()` doesn't test for the end-of-file. It simply returns a value of the corresponding indicator. The indicator is changed by >> attempts.

Error recovery …

- Once the stream is in the error state, it will stay that way until you take specific action:
  - All subsequent operations will do nothing, or loop forever no matter what they are or what is in the input.
  - You have to clear the stream by calling `clear()` to recover the stream from the fail state.

```
inFile >> newNumber;
if( inFile.fail() )
{
    inFile.clear();
    inFile.ignore(100, '\n');
}
```

Recovers the file stream from the error state. However, further reading of data may be useless as the wrong characters are still in the stream buffer.

Discard 100 characters (or until the end-of-line indicator) from the stream buffer.

Example (fixed format text file)

```
int readData(ifstream& inFile, InfoType& student, string myId)
{
   string nameFirst;
   string nameLast;
   string Id;

   do inFile >> nameFirst >> nameLast >> Id;
   while( inFile.good() && Id != myId );

   if(inFile.fail())  return -1;          // invalid character
   if(inFile.bad())   return -2;          // hardware failure
   if(inFile.eof() && Id=="") return -3; // myId not found

   student.firstName = nameFirst;
   student.lastName = nameLast;
   student.Id = Id;

   return 0;
}
```

Character input …

- How do we read **all** characters from the input stream; including blanks, tabs, and new-lines?
  - This could be a input file stream or something else, like standard in.
  - The extraction operator doesn't read white space or characters.
- You can use `get` functions to read a character:

```
ifstream inFile;
char nextChar;
nextChar = inFile.get();
```

- You can use `getline` to read a line of characters from a text file.

```
char lineBuffer[bufSize];
infile.getline( lineBuffer, bufSize );
```

- Be careful.

test.txt

```
34.99
Motor Oil
```

```
float price;
char productName[20] ;
char fileName[] = "test.txt";
ifstream inData;

inData.open( fileName );

inData >> price;

inData.getline(productName, 20);

cout << price << endl;
cout << productName << endl;
```

- The newline character is still in the stream buffer, and that is interpreted as an empty string.

- So, we need to clear the buffer …

```
inData >> price;
inData.ignore( 20, ‘\n’ );
inData.getline(productName, 20);
```

A different form and a delimiter

- It's often unreasonable to forecast the input length.
- So we can use this form …

```
getline(cin,input);
```

- This has a default delimiter or endpoint of \n, that's a new line.
- But you can change this …

```
getline(cin,input,'t');
```

- Be careful with this, you can capture a lot more than you expect.

Character output
Output formatting

- Use `put()` in a similar way to `get()`.

- You can use output manipulators if you want to format your output:

```
cout << setiosflags(ios::fixed);
cout << setiosflags(ios::showpoint);
cout << setprecision(2);
```

Common Escape Sequences

Table 2-4 Commonly Used Escape Sequences

	Escape Sequence	Description
\n	Newline	Cursor moves to the beginning of the next line
\t	Tab	Cursor moves to the next tab stop
\ b	Backspace	Cursor moves one space to the left
\r	Return	Cursor moves to the beginning of the current line (not the next line)
\\	Backslash	Backslash is printed
\'	Single quotation	Single quotation mark is printed
\"	Double quotation	Double quotation mark is printed

Buffering

- When you direct data into a file, it is not sent there immediately.
  - It is physically written to the device only when the buffer is full.
- When you read data from a file, you input it from the input buffer.
  - When the buffer becomes empty it is refilled with a new block of data.
- **Why buffer?**
  - It reduces the number of accesses to the external device.
- Only `cerr`, standard error, is never buffered.
  - Errors are critical.

Text and Binary Files

- Text files:
  - composed of characters
  - data types have to be formatted/converted into a sequence of characters
  - variable number of characters (bytes) for the same data types
  - usually sequential access
- Binary files
  - binary numbers (bits)
  - fixed size of the same data types
  - random access
  - more compact, less portable than text files

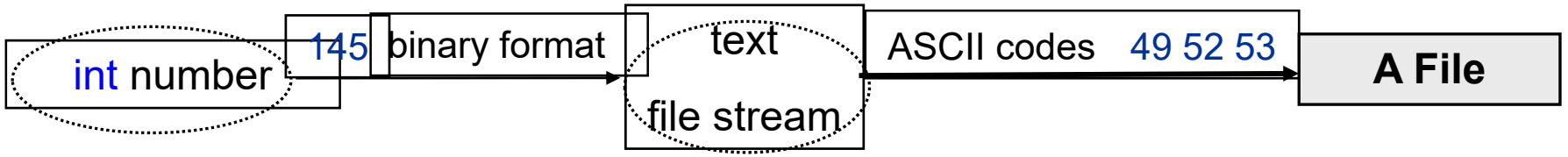
Text Files Input/Output

- Text File Input/Output functions also carry out data type conversion:

Example:

```
int number;
outFile << number ;
```

The **<<** operator converts `int` into a sequence of ASCII codes and writes them into a file.

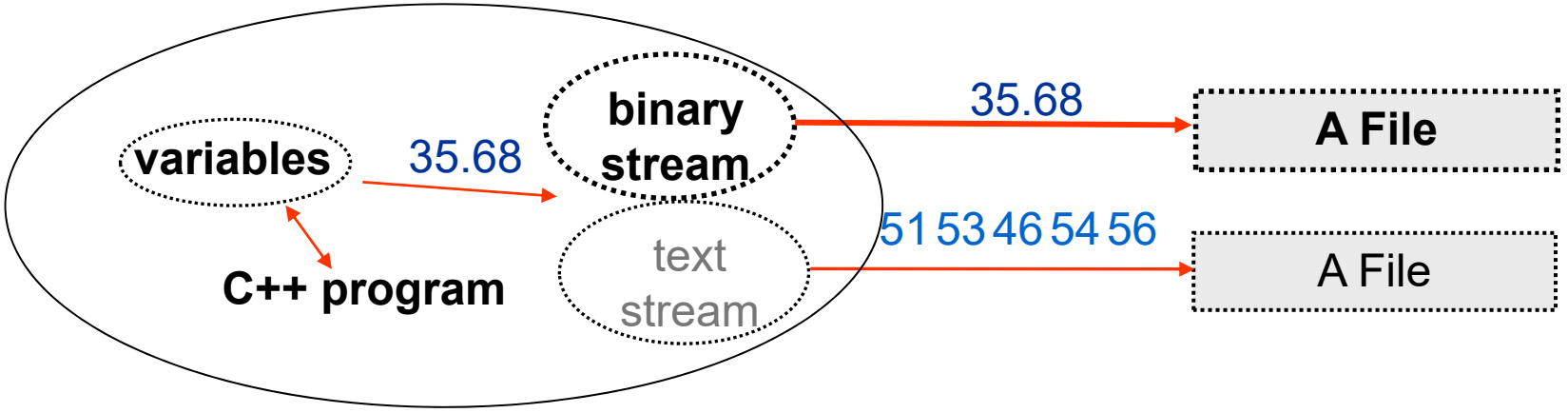



```
inFile >> &number;
```

The **>>** operator converts a sequence of ASCII characters into an integer number and stores it into a variable.

Binary File Stream Concept

- A Binary File Stream is an interface between a program and a physical file that **does not perform any type conversion .**



- File Streams can be Text or Binary, but physical files do not have any special marker to indicate their type.
- A file name or its extension does not affect the file type.