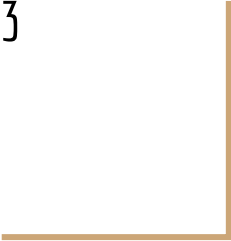




Chapter 9: Streams

CSE 2010 - Week 13



Background

- The programs that we have written up until now can take input from the user through the keyboard and processes it accordingly using `cin>>` or `getline()`.
- We have also been able to output messages to the user using `cout <<`
- To use `cout` and `cin`, we need to `#include <iostream>`, since `cout` and `cin` are output and input streams, respectively.
- Streams in C++ are an abstraction that represent a device on which input and output operations are performed.
- Streams are essential in programming because they allow for interaction between the program, users, and outside data.

Background

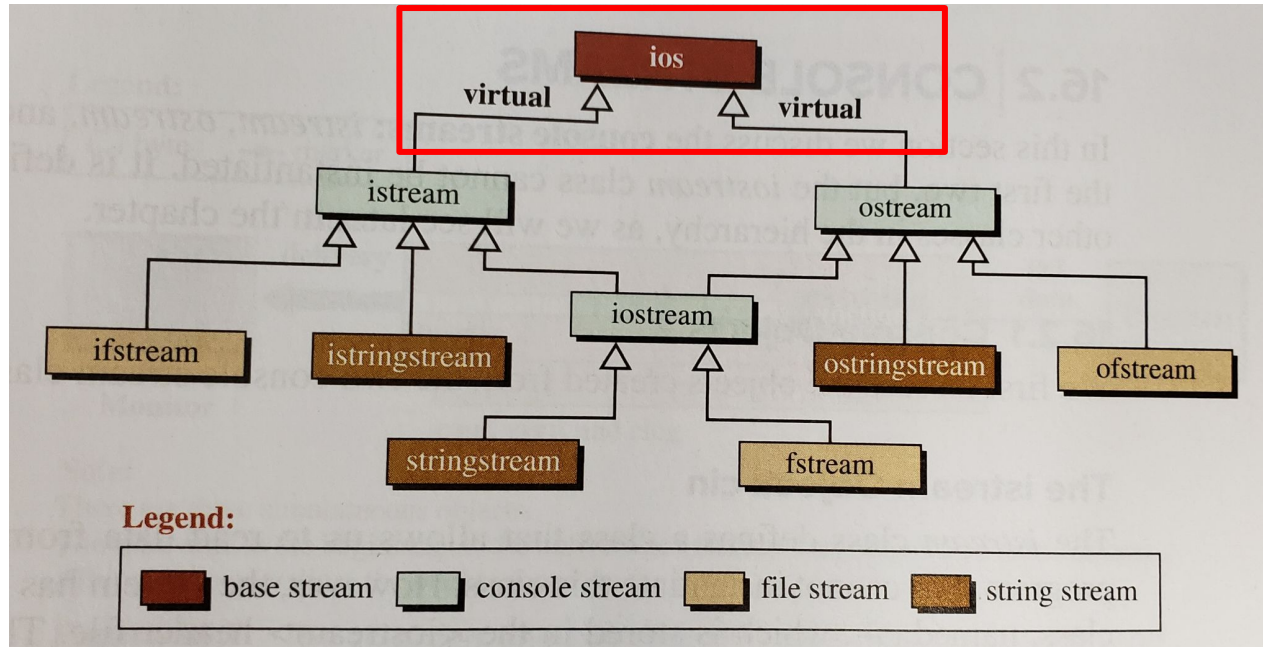
- We have already been working with streams (cin, cout), stream functions (cin.fail(), cin.clear()...etc).
- This chapter we are going to take a much closer look at streams
 - What are streams?
 - Different stream classes
 - Stream states
 - Reading and writing from/to files (file streams)
 - Formatting streams

What is a stream?

- When we run our programs, data must be stored in memory.
- The data stored in memory for processing comes from an external source and goes to an external sink.
- So far, we have used the keyboard (as the source) and the monitor (as the link) for data.
- A source and a sink cannot be directly connected to a program, so we have streams.
 - An input stream stands between a source and the program
 - An output stream stands between a program and a sink.
- Basic definition is that a stream is a sequence of bytes flowing in or out of a program.

Stream Classes

C++ defines a hierarchy of classes that allow us to use streams

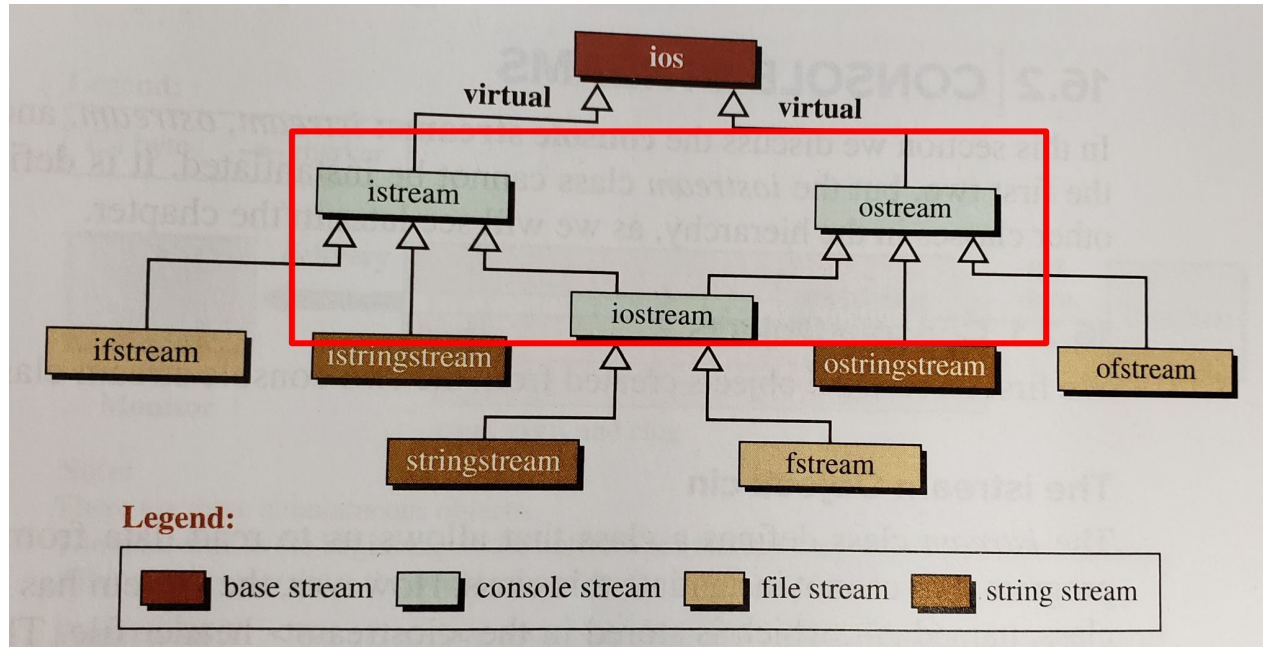


`ios` class: virtual base class

- The base class for all input/output streams.
- Defines data members and member functions that are inherited by all input/output stream classes.
- It is never instantiated, which means it never uses its own data members or functions.
- They are used by other stream classes.

Stream Classes

C++ defines a hierarchy of classes that allow us to use streams

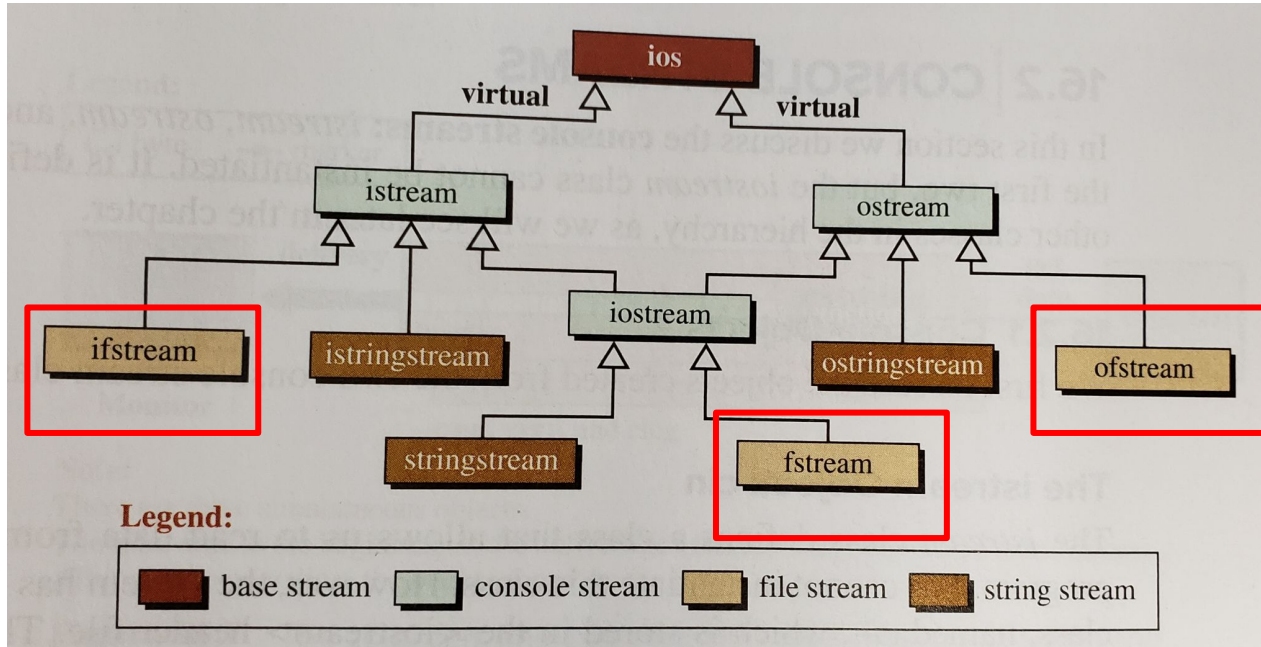


istream, ostream, iostream :

- Console classes.
- istream defines a class that allows us to read data from the keyboard.
- ostream defines a class that allows us to write data to the monitor.
- iostream inherits both

Stream Classes

C++ defines a hierarchy of classes that allow us to use streams

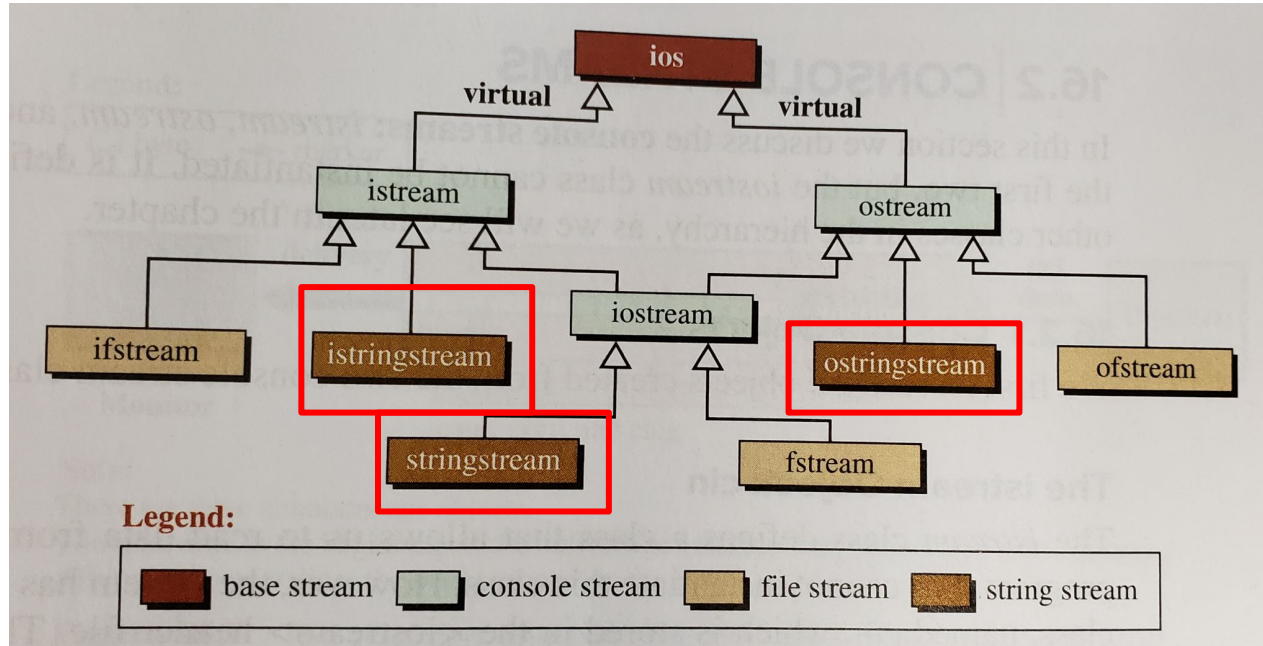


`ifstream`, `ofstream`, `fstream` :

- File streams that connects our programs to files.
- `ifstream` allows for input from files (inherits `istream`)
- `ofstream` allows for output to files (inherits `ostream`)
- `fstream` allows for file input/output (inherits `iostream`)

Stream Classes

C++ defines a hierarchy of classes that allow us to use streams

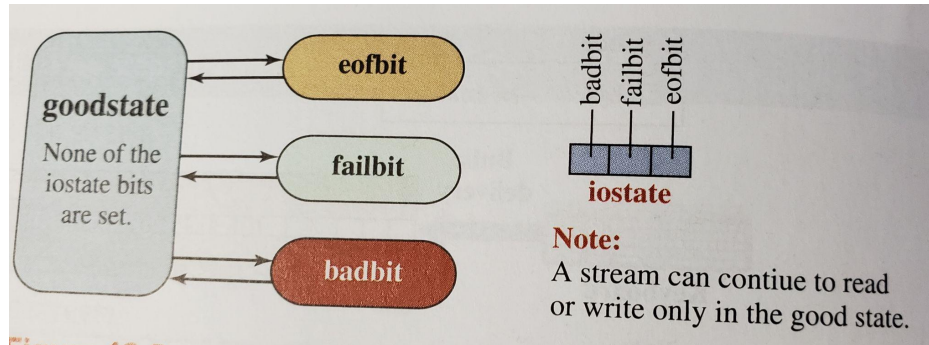


```
istringstream,  
ostringstream, stringstream :
```

- String streams used to input data from or output data to strings.
- `istringstream` allows for input from strings (inherits `istream`)
- `ostringstream` allows for output to strings (inherits `ostream`)
- `fstream` allows for string input/output (inherits `iostream`)

Stream States

- When we are inputting our outputting data, it is important for a stream to keep track of its state.
- The stream state is stored in a data member called `iostate` that shows the status of the stream object (included in `ios` class).
- `iostate` is a type made up of 3 constants
 - `eofbit` - For input streams, means no more characters to extract (end of buffer)
 - `failbit` - For input streams, invalid read operation. For output streams, invalid write operation.
 - `badbit` - For input/output streams, stream integrity is lost.
 - If all of the above bits are not set, then the stream is in a `goodstate` and can be used.



Stream States

The `ios` class also provides member functions to check the condition of the state.

- `bool eof()` - returns true if `eofbit` is set, false otherwise.
- `bool fail()` - returns true if `failbit` is set, false otherwise.
- `bool bad()` - returns true if `badbit` OR `failbit` is set, false otherwise.
- `bool good()` - returns true if `iostate` is in a good state (all bits not set), returns false otherwise.
- `clear()` - It clears all bits (sets all to zero).

We've used `cin.fail()` to check if an input is in a failed state.

We've used `while(cin>>x)` which continues until one of the 3 bits is set.

Two more things before we get into using file streams.....

- When we use any streams, there is typically 5 things we need to make sure we do:
 - Construct the stream
 - Connect the stream to the source or sink
 - Read or write from the stream
 - Disconnect the source or the sink
 - Destroy the stream
- When we work with streams, it's important to note that none of the classes in the hierarchy have provided a copy constructor or assignment operators.
 - We can never pass an object of any stream class to a function by value, so we need to pass there by reference.
 - We can never return an object of any stream class from a function by value, so we need to do so by reference.
 - We cannot pass or return an object of any stream class using a constant modifier, since these objects are constantly changing.

Reading and Writing From/To Files

- So far, our programs use the keyboard to take user input and the monitor to output data. Once the program ends, this data no longer exists.
- We will now learn how we can read from files and write to files.
- Need to `#include <fstream>`
 - Since it inherits `iostream`, this will allow for both input and output.

Reading From Files

- Make sure to `#include <fstream>`
- Step 1: Constructing a stream to read files.

```
ifstream streamName;
```

- Step 2: Connect the stream to the source. Always double check that it opened properly

```
streamName.open(filename);  
if(streamName.fail())  
    Cout << "Error opening file with name.\n";
```

- Step 3: Read from the stream;

```
datatype var;//declare variable for what you want to read  
streamName >> var;//will read in data, and store into var until  
//white space or newline is reached
```

- Step 4: Disconnect the source

```
streamName.close();
```

- Step 5: Destroy stream....is done automatically :)

```

1  /*
2  * Filename: read_file.cpp
3  * Example program to see how to open a file and read from it
4  */
5  #include <iostream>
6  #include <fstream>//needed to read from file
7
8  using namespace std;
9
10 int main()
11 {
12     //create input file stream
13     ifstream input_data;
14     //connect to a file
15     input_data.open("input1.txt");
16     //check that it opened properly
17     if(input_data.fail())
18         cout << "Error opening \"input1.txt\"\n";
19     else{
20         //prepare to read from the stream
21         //since I will be reading in ints, need an int
22         int n;
23         //now I can start reading
24         //I want to keep reading to the end of the file, so I will
25         //use a while loop
26         while(input_data >> n){
27             cout << "Just read: " << n << "\n";
28         }
29         cout << "All done reading file.\n";
30     }
31     //disconnect file
32     input_data.close();
33
34     return 0;
35 }

```

input1.txt

```

1 2
2 -1
3 10
4 26
5 1005
6 99
7 -45
8 0

```

Output:

```

Just read: 2
Just read: -1
Just read: 10
Just read: 26
Just read: 1005
Just read: 99
Just read: -45
Just read: 0
All done reading file.

```

`ifstream/istream` as parameters

- It is good practice to designate a function to read in files, especially if you need to store multiple values into a vector or array.
- You can either declare a file object in the function if you don't need it elsewhere, or pass it as a parameter (by reference)
- If you pass it as a parameter, use `istream` as the parameter type
 - This will allow you to send not only `ifstream`, but also `cin`

```
Example: void read_data(istream& in);
```

- The `istream` object needs to be passed by reference for two reasons:
 - The object is modified with every input, so we need to allow it to change.
 - Passing by reference will ensure that we do not "slice" away any info if we send an `ifstream` object as the argument.
 - Remember that `ifstream` inherits `istream`, so it is valid to send an `ifstream` object to an `istream` parameter.



Let's look at some examples!
max_file.cpp & file_vec.cpp on Canvas



Writing to Files

- Make sure to `#include <fstream>`
- Step 1: Constructing a stream to write to files.

```
ofstream streamName;
```

- Step 2: Connect the stream to the sink. Always double check that it opened properly
- If the file doesn't exist, the program will simply create a new file with the name you provide.
- Output files will fail to open if it already exists and is in use, or if the file is marked as read-only.

```
streamName.open(filename);  
if(streamName.fail())  
    Cout << "Error opening file with name.\n";
```

- Step 3: Write to the stream;

```
datatype var;//declare variable for what you want to write  
//give variable some value  
streamName << var;//will write the value to the stream/file
```

- Step 4: Disconnect the sink

```
streamName.close();
```

- Step 5: Destroy stream....is done automatically :)



Let's look at some examples!

`write_random.cpp` & `file_vec2.cpp` on Canvas



Important note for writing to files...

- If you open a file that already exists and has data in it, your program will **OVERWRITE** the file.
- There are ways to prevent this by selecting different opening modes.

Opening Modes

- The default opening mode for file streams puts the stream marker at the beginning of the file and will read from or write to starting there.
- We have other opening models available to us when we are reading/writing.
- ios = iostream class
 - ios::in – open for input
 - ios::out – open for output; contents are destroyed
 - ios::app – open for output and write at the end (append)
 - ios::ate – move to the end immediately after opening (at end)
 - ios::trunc – truncate the file to zero length
 - ios:: binary – read and write in binary mode
- Syntax for opening modes:
 - streamName.open(filename,open mode);
output_data.open("out.txt", ios::app);//will append to file without overwriting

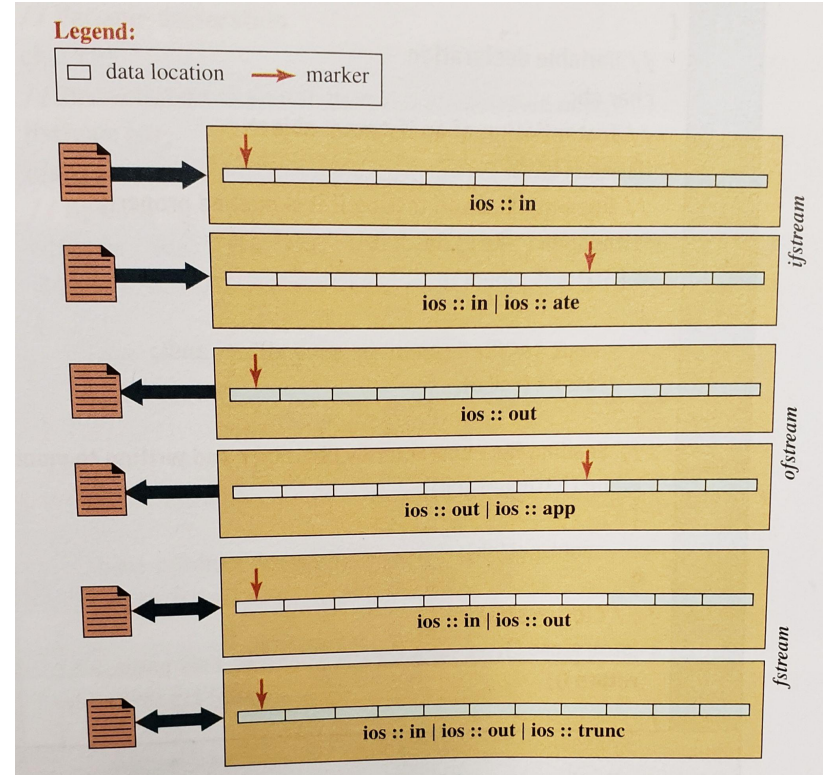
Opening Modes

Opening modes for input (reading)

- `ios::in` - open for input
 - Will open the file for input and puts the marker at the first byte of the buffer.
 - This allows us to start reading the files from the first byte.
 - With each read, the marker moves to the next byte, until we reach an empty position.
 - At the end of reading, the eof bit is set.
- `(ios::in | ios::ate)` - open for input at the end
 - The file is open for reading, but the marker goes to the position after the last byte.
 - We can use this if we want to read the bytes in reverse or we can check the size of the file.

Opening modes for output (writing)

- `ios::out` - open for output
 - Will open the file for output.
 - If the file contains any data, they are deleted.
- `(ios::out | ios::app)` - open for output | append
 - The file is open for writing, but the marker goes to the position after the last byte and will append to it.
 - The existing data is preserved.



Reading from and writing to the same file, or different files

- Make sure to `#include <fstream>`
- Step 1: Constructing a stream to read & write files.

```
fstream streamName;
```

- Step 2: Connect the stream to the source. Always double check that it opened properly

```
streamName.open(filename, ios::in|ios::out);  
if(streamName.fail())
```

```
    Cout << "Error opening file with name.\n";
```

- Step 3: Depending on what you want to do, use the appropriate operator;

```
    streamName >> var; //will read  
    streamName << var; // will write  
    //white space or newline is reached
```

- Step 3.1: If you expect to write after reading, you will have to also use the clear function to clear the stream and set the eof bit to false, allowing for input at the end of the file. (`streamName.clear()`)
- Step 4: Disconnect the source

```
streamName.close();
```

- Step 5: Destroy stream....is done automatically :)

Let's look at an example! (read_write.cpp, read_write2.cpp)