

Lecture 4: Streams

CS 106L, Winter '21

Today's Agenda

- Recap: **References**
- Streams
- File Streams
- String Streams
- Buffering
- State Bits
- Chaining

Recap: Something

Uniform initialization

```
int x{3};           // One syntax to initialize all variables

// Both of these are vectors with elements {3, 5, 7}
std::vector<int> a_vector{3, 5, 7};
std::vector<int> another_vector = {3, 5, 7};

Student s = {"Ethan", "CA", 20};
```

References

```
int b = 5;  
int& a = b;  
a = 2; // now b = 2 as well
```

```
void switch(int& c) { c = 3; }  
switch(a); // now b = 3 as well
```

```
const int& d = a;  
d = 5; // fails because const reference
```

You can return references

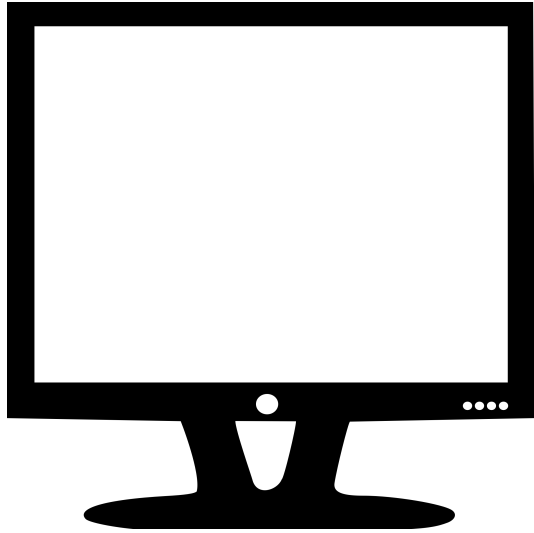
Common idiom: return an element inside our class.

```
// Note that the parameter must be a non-const reference to return
// a non-const reference to one of its elements!
int& front(std::vector<int>& vec) {
    // assuming vec.size() > 0
    return vec[0];
}

int main() {
    std::vector<int> numbers{1, 2, 3};
    front(numbers) = 4; // vec = {4, 2, 3}
    return 0;
}
```

Streams

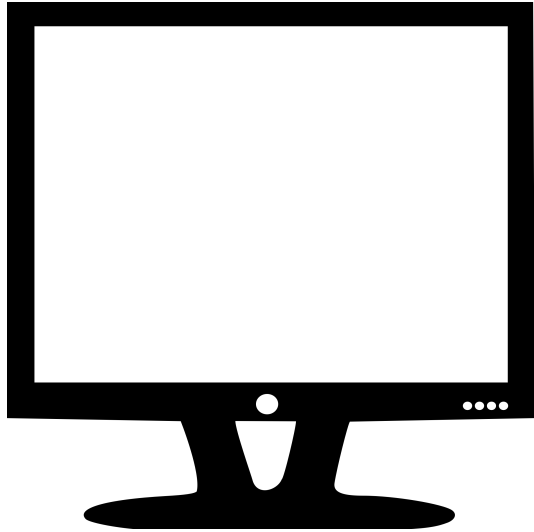
A stream is an abstraction for input/output



`std::cout`



`<< "Hello World"`



`std::cout`

`“Hello World”`



`std::cout`



Streams can take different types of input!

```
cout << "Strings work!" << endl;  
cout << 1729 << endl;  
cout << 3.14 << endl;  
cout << "Mixed types: " << 1123 << endl;
```

Any primitive type can be inserted;
for other types, you need to explicitly tell C++ how to do this

Streams convert between the *string* representation of data and the data itself.

Idea: both input and output are strings;
need to do computation on object representation

Types of Streams

Output Streams

- Of type `std::ostream`
- Can only **receive** data with the `<<` operator
 - Converts data to string and **sends** it to stream

```
std::cout << 5 << std::endl;           // prints 5 to the console
```

```
std::ofstream out("out.txt", std::ofstream::out);  
out << 5 << std::endl;                 // out.txt contains 5
```


Live Code Demo:

Ostreams.cpp

Input Streams

What does this do?

```
int x;  
std::cin >> x;
```

// what happens if input is 5 ?

// how about **51375** ?

Input Streams

- Of type `std::istream`
- Can only **give you** data with the `>>` operator
 - Receives string from stream and converts it to data

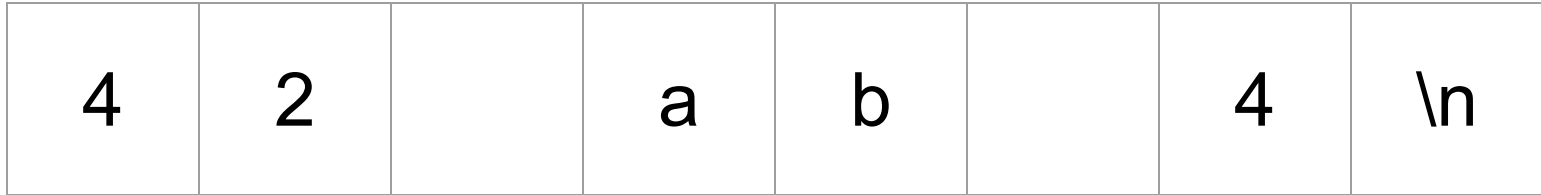
```
// input: 5 abc 7
int x, z; string y;
std::cin >> x > y > z;           // x=5, y="xyz", z=7

// std::ifstream does the same for files
std::ifstream in("out.txt", std::ifstream::in);
in >> x >> y >> z;           // out.txt contains 5
```

 **Questions?** 

Why does this work?

Think of a `std::istream` as a **sequence** of characters



position

```
int x; string y; int z;  
input >> x;  
input >> y;  
input >> z;
```

Extracting an int reads **as many characters**
as possible until whitespace

4	2		a	b		4	\n
---	---	--	---	---	--	---	----



position

```
int x; string y; int z;  
input >> x; // 42 put into x  
input >> y;  
input >> z;
```

Next time, first **skip over any whitespace**

4	2		a	b		4	\n
---	---	--	---	---	--	---	----



position

```
int x; string y; int z;  
input >> x;  
input >> y; // ab put into y  
input >> z;
```


When no more data is left, **fail bit set to true**



↑
position

```
int x; string y; int z;  
input >> x;  
input >> y;  
input >> z; // 4 put into z
```

Input Streams

Given what we just learned, what does this do? Answer in the chat.

```
int x;  
std::cin >> x;  
std::cout << x * 5 << std::endl;  
  
// what happens if input is blah ?
```

Reading using >> extracts a single “word”
including for strings

To read a whole line, use
`getline(istream& stream, string& line);`

Don't mix >> with getline!

- >> reads up to the next whitespace character and *does not* go past that whitespace character.
- **getline** reads up to the next delimiter (by default, “\n”), and *does* go past that delimiter.
- Don't mix the two or bad things will happen!



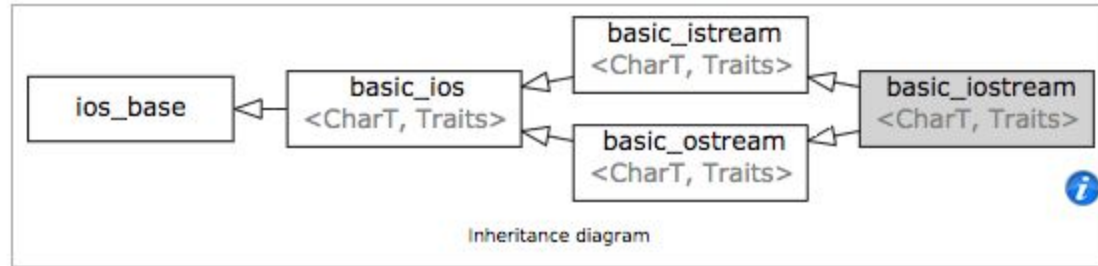
Note for 106B/X: Don't use >> with Stanford libraries, which use getline.

Additional Stream Methods

```
input.get(ch);           // reads a single char
input.clear();           // resets the fail bit
input.open("filename");  // opens stream on a file
input.seekg(0);          // rewinds stream to start

input.close();           // closes stream
                        // done automatically for you,
                        // so not necessary
```

std::iostream is both a istream & ostream



 **Questions?** 

Stringstreams

Work with a **string** as if it were a stream

```
std::string input = "5 seventy 2";
```

```
std::istringstream i(input);
```

```
int x; std::string y; int z;
```

```
i >> x >> y >> z;
```

```
std::cout << z << std::endl;
```

Live Code Demo:

String Streams

Stream Internals

Stream Internals

- Buffering
- State Bits
- Chaining (a.k.a. why `<< << <<` works)

Buffering

Writing to console/file is slow.

If we had to write each character separately, slow runtime.

Accumulate characters in a temporary **buffer**.

```
input << "hel";  
input << "lo ";  
input << "world";
```

h	e	l	l	o		w	o	r	l	d
---	---	---	---	---	--	---	---	---	---	---

When full, write **entire buffer** to output

```
Output: hello world
```

Empty the buffer early by flushing:

```
stream << std::flush;           // flush what we have so far
stream << std::endl;            // flush with newline
// this is equivalent to: stream << "\n" << std::flush;
```


Buffer Takeaways

- The internal sequence of data stored in a stream is called a **buffer**.
- Istreams use buffers to store data we haven't used yet.
- Ostreams use buffers to store data that hasn't been outputted yet.

🤔 There's actually a third standard stream, `std::cerr`, which is not buffered. Why?

State Bits

Streams have four **state bits**



Good bit: whether ready for read/write



Fail bit: previous operation failed, future operations frozen



EOF bit: previous operation reached end of file



Bad bit: external integrity error

Using State Bits

```
// here's a very common read loop:
```

```
while (true) {  
    stream >> temp;           // read data  
    if (stream.fail()) break; // checks for fail bit OR bad bit  
    doSomething(temp);  
}
```

Streams can be converted to bool

```
stream >> temp;  
if (stream.fail()) break;           // checks for fail bit OR bad bit  
doSomething(temp);
```



```
stream >> temp;  
if (!stream) break;                // same thing  
doSomething(temp);
```

Aside: Chaining

Chaining >> and <<

>> and << are actually functions!

```
std::ostream& operator<<(std::ostream& out, const std::string& s);
```

```
std::ostream& operator<<(std::ostream& out, const std::string& s);
```

should be `std::cin >> temp;`

```
std::cout << "hello";
```



```
operator<<(std::cout, "hello");
```

```
std::cin << temp;
```



should be `operator >> (std::cin, variable);`

```
operator>>(std::cin, temp);
```

This is how the magic std::cout mixing types works!

```
std::ostream& operator<<(std::ostream& out, const std::string& s);  
std::ostream& operator<<(std::ostream& out, const int& i);
```

```
cout << "test" << 5;           // (cout << "test") << 5;
```

```
operator<<(operator<<(cout, "test"), 5);
```



```
operator<<(cout, 5);
```



```
cout
```


Using State Bits – Part 2

// here's a very common read loop:

```
while (true) {  
    stream >> temp;  
    if (!stream) break;  
    doSomething(temp);  
}
```

This returns the stream itself!

// read data

// checks for fail bit OR bad bit

Using State Bits – Part 2

```
// here's a very common read loop:
```

```
while (stream >> temp) {  
    doSomething(temp);  
}
```