Streams II

Ali Malik

malikali@stanford.edu
Game Plan

- Recap
- Stream Miscellany
- Stream Internals
- Stream Manipulators
- Stringstream
- Tying it all Together
Announcements
Recap
Output Streams

Can only **receive** data.

- The `std::cout` stream is an example of an output stream.
- All output streams are of type `std::ostream`.

Send data using stream insertion operator: `<<`

Insertions converts data to string and **sends** to stream.
Input Streams

Can only **give** you data.

- The `std::cin` stream is an example of an input stream.
- All input streams are of type `std::istream`.

Pull out data using stream extraction operator: `>>`

Extraction **gets** data from stream as a string and converts it into the appropriate type.
There are some quirks with extracting a string from a stream.

Reading into a string using >> will only read a single word, not the whole line.

To read a whole line, use

```cpp
getline(istream& stream, string& line);
```
Stream Miscellany

Some additional methods for using streams:

```cpp
input.get(ch); // reads one char at a time
input.close(); // closes stream
input.clear(); // resets any fail bits
input.open("filename"); // open stream on file
input.seekg(0); // rewinds stream to start
```
Stream Internals
Writing to a console/file is a slow operation.

If the program had to write each character immediately, runtime would significantly slow down.

What can we do?
Idea:
Accumulate characters in a temporary buffer/array.
When buffer is full, write out all contents of the buffer to the output device at once.

This process is known as **flushing** the stream
Buffering

Let’s look at this in action:

Stream Buffering

(StreamBuffer.pro)
Buffering

The internal sequence of data stored in a stream is called a buffer.

Istreams use them to store data we haven’t used yet

Ostreams use them to store data they haven’t passed along yet.
If we want to force the contents of the buffer to their destination, we can flush the stream:

```cpp
stream.flush(ch);          // use by default
stream << std::flush;     // use if you are printing
flush(stream)             // no good reason to use this
stream << std::endl;      // use if you want a newline
```
If we want to force the contents of the buffer to their destination, we can flush the stream:

```cpp
stream.flush(ch); // use by default
stream << std::flush; // use if you are printing
flush(stream); // no good reason to use this
stream << std::endl; // use if you want a newline
```
If we want to force the contents of the buffer to their destination, we can flush the stream:

```cpp
stream.flush(ch); // use by default
stream << std::flush; // use if you are printing
stream << std::endl; // use if you want a newline
```
If we want to force the contents of the buffer to their destination, we can flush the stream:

```cpp
stream.flush(ch); // use by default
stream << std::flush; // use if you are printing
stream << std::endl; // use if you want a newline
```

This is equivalent to `stream << "\n" << std::flush;`
Let’s look at this in action:

Stream Buffering

(StreamBuffer.pro)
Buffering

Not all streams are buffered (`std::cerr` is an example).

We can get a very real sense of the speed difference:

Stream Buffering Speed

(BuffersSpeed.pro)
Stream bits

Streams have four bits to give us information about their state:

- **Good bit**  No errors, the stream is good to go
- **EOF bit**   End-of-file was reached during a previous operation
- **Fail bit**  Logical error on a previous operation
- **Bad bit**   Likely unrecoverable error on previous operation
Which bit to use?

1. Read data
2. Check if data is valid, if not break
3. Use data
4. Go back to step 1

```cpp
while(true) {
    stream >> temp;
    if(stream.fail()) break;
    do_something(temp);
}
```
Stream Shortcuts
The `<<` and `>>` operators are not magic, they are actually functions!

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

```cpp
operator<<(std::cout, "hello");
```
Chaining >> or <<

We know functions can return things.

The << and >> operators return the stream passed as their left argument.

This is why this works:

```cpp
std::cout << "hello" << 23 << "world";
```
We know functions can return things.

The `<<` and `>>` operators return the stream passed as their left argument.

This is why this works:

```cpp
(((std::cout << "hello") << 23) << "world");
```
We know functions can return things.

The $<<$ and $>>$ operators return the stream passed as their left argument.

This is why this works:

```cpp
((std::cout << "hello") << 23) << "world");
```
We know functions can return things.

The `<<` and `>>` operators return the stream passed as their left argument.

This is why this works:

```cpp
(((std::cout) << 23) << "world");`
```
Chaining >> or <<

We know functions can return things.

The << and >> operators return the stream passed as their left argument.

This is why this works:

```cpp
((std::cout << 23) << "world");
```
Chaining $\gg\gg$ or $\ll$

We know functions can return things.

The $\ll$ and $\gg\gg$ operators return the stream passed as their left argument.

This is why this works:

```cpp
((std::cout << 23) << "world");
```
Chaining >> or <<

We know functions can return things.

The << and >> operators return the stream passed as their left argument.

This is why this works:

```cpp
((std::cout) << "world");
```
Chaining $\gg\gg$ or $\ll$

We know functions can return things.

The $\ll$ and $\gg\gg$ operators return the stream passed as their left argument.

This is why this works:

```
(std::cout $\ll$ "world");
```
Which bit to use? - Part II

Let's look at this code again:

```cpp
while(true) {
    stream >> temp;
    if(stream.fail()) break;
    do_something(temp);
}
```
Let’s look at this code again:

```cpp
while (true) {
    stream >> temp;
    if (stream.fail()) break;
    do_something(temp);
}
```

Streams can be converted to bool
Which bit to use? - Part II

Let’s look at this code again:

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

Streams can be converted to bool.
Let’s look at this code again:

```cpp
while(true) {
    stream >> temp;
    if(!stream) break;  // checks fail or bad bit
    do_something(temp);
}
```
Let's look at this code again:

```cpp
while(true) {
    stream >> temp;
    if(!stream) break;  // checks fail or bad bit
    do_something(temp);
}
```

We know this returns the stream.
Which bit to use? - Part II

Let’s look at this code again:

```c
while(true) {
    if(!(stream >> temp)) break;
    do_something(temp);
}
```
Let’s look at this code again:

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

We can simplify the logic.
Which bit to use? - Part II

Let’s look at this code again:

```c
while(stream >> temp) {
    do_something(temp);
}
```
The same principle applies with getline

```cpp
while (stream >> temp) {
    do_something(temp);
}
```

```cpp
while (getline(stream, temp)) {
    do_something(temp);
}
```
Stream Manipulators
Stream Manipulator

There are some special keywords that change the behaviour of the stream when inserted.

```cpp
std::endl and std::flush are two examples.
```
Stream Manipulator

Common:
- `endl`: inserts a newline and flushes the stream
- `ws`: skips all whitespace until it finds another char
- `boolalpha`: prints “true” and “false” for bools

Numeric:
- `hex`: prints numbers in hex
- `setprecision`: adjusts the precision numbers print with

Padding:
- `setw`: pads output
- `setfill`: fills padding with character
Some examples - Padding

```cpp
#include <iomanip>

std::cout << "[" << std::setw(10) << "Hi" << "]"
   << std::endl;
```

Outputs:

```
[     Hi]
```
Some examples - Padding

```cpp
#include <iomanip>

std::cout << "[" << std::left
 << std::setw(10) << "Hi" << "]" << std::endl;
```

Outputs:

```
[Hi       ]
```
Some examples - Padding

```
#include <iomanip>

std::cout << "[" << std::left << std::setfill('-')
   << std::setw(10) << "Hi" << "]" << std::endl;
```

Outputs:

```
[Hi--------]
```

```
[Hi--------]
```
Some examples - Numeric

#include <iomanip>

std::cout << std::hex << 10;  // prints a
std::cout << std::oct << 10;  // prints 12
std::cout << std::dec << 10;  // prints 10
Stream manipulators can be passed into streams to change how they behave.

They have a variety of uses, and if you’d like to format something differently, there’s probably a manipulator for it.

You can find a list of the most common ones at http://www.cplusplus.com/reference/library/manipulators/
stringstream
stringstream

Sometimes we want to be able to treat a string like a stream.

Useful scenarios:

- Converting between data types
- Tokenizing a string
#include <sstream>
Std::string line = "137 2.718 Hello";
std::stringstream stream(line);

int myInt;
double myDouble;
std::string myString;
stream >> myInt >> myDouble >> myString;

std::cout << myInt << std::endl;
std::cout << myDouble << std::endl;
std::cout << myString << std::endl;
Tying it Together
Buffering

Let’s write the Stanford simpio library!

Simple IO

(OurSimpIO.pro)
Next Time

Sequential Containers