Using Streams

Cristian Cibils
(ccibils@stanford.edu)

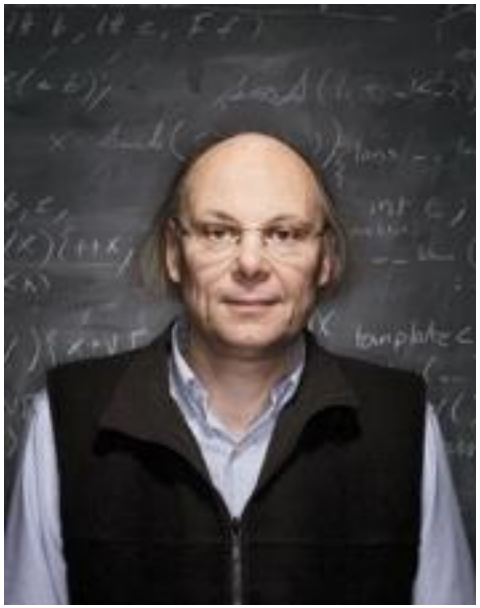
Announcements

Office Hours:

- Tue-Thu after class, right here!

Quick Recap

- We spent last class talking about the idea behind C++ streams
  - `istream`, `ostream`, `stringstream`
  - `>>`, `<<`, `getline`
- Today we will explore some of the difficulties that come with using them in practice.



C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows away your whole leg.

--Bjarne Stroustrup

Complicated Parts

We're now ready to talk about some of the messy parts of working with streams.

- Unread data sits on the stream
- A stream which has failed stays failed
- Mixing >> and getline
- Putting it all together
- Understanding the Stanford library functions

Complicated Parts

We're now ready to talk about some of the messy parts of working with streams.

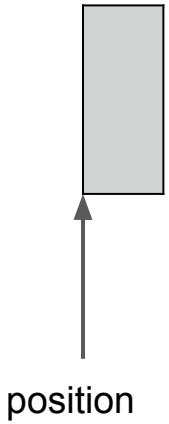
- Unread data sits on the stream
- A stream which has failed stays failed
- Mixing >> and getline
- Putting it all together
- Understanding the Stanford library functions

Unread Data

- If we read read an integer from `cin`, and the user types "42 foo", we will read the integer 42.
- However, the data " foo" will still be sitting on `cin`.
- So, if we try and read another integer from the stream, it will fail before the user even gets to enter any input.

Unread Data

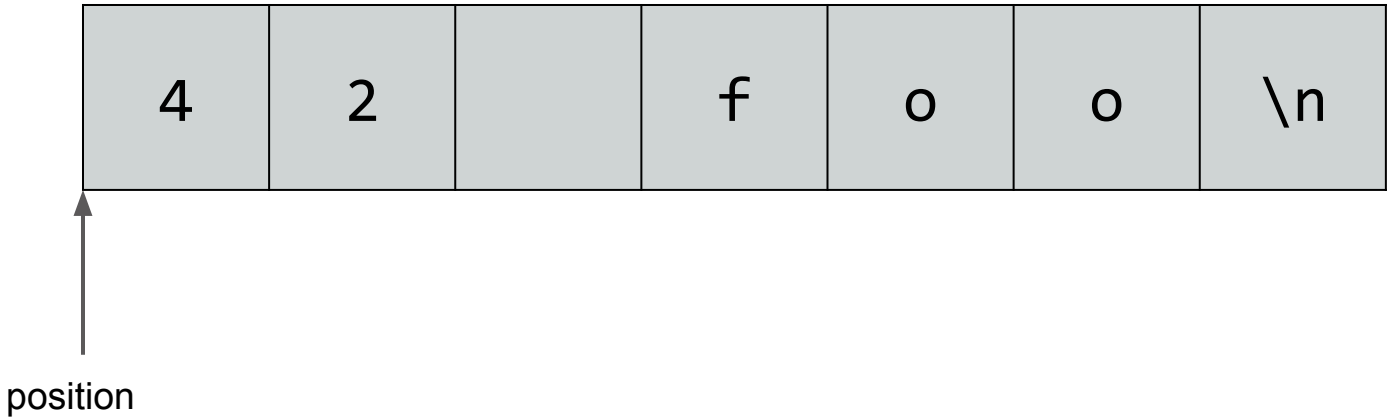
`cin` starts off with no input in it




```
int value;
cin >> value; // value == ?
```

Unread Data

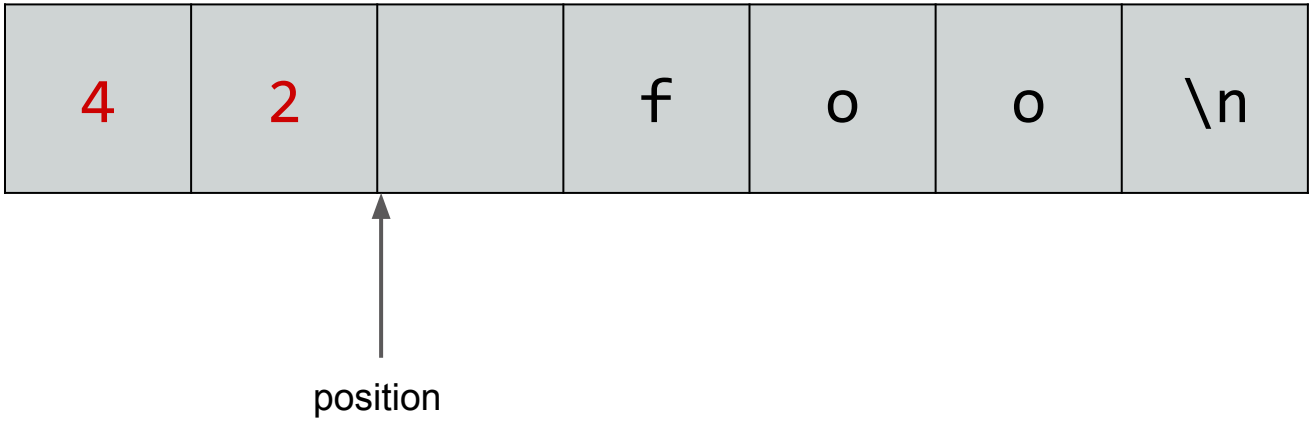
The user types in input, including some trailing garbage, in this case “foo”




```
int value;
cin >> value; // value == ?
```

Unread Data

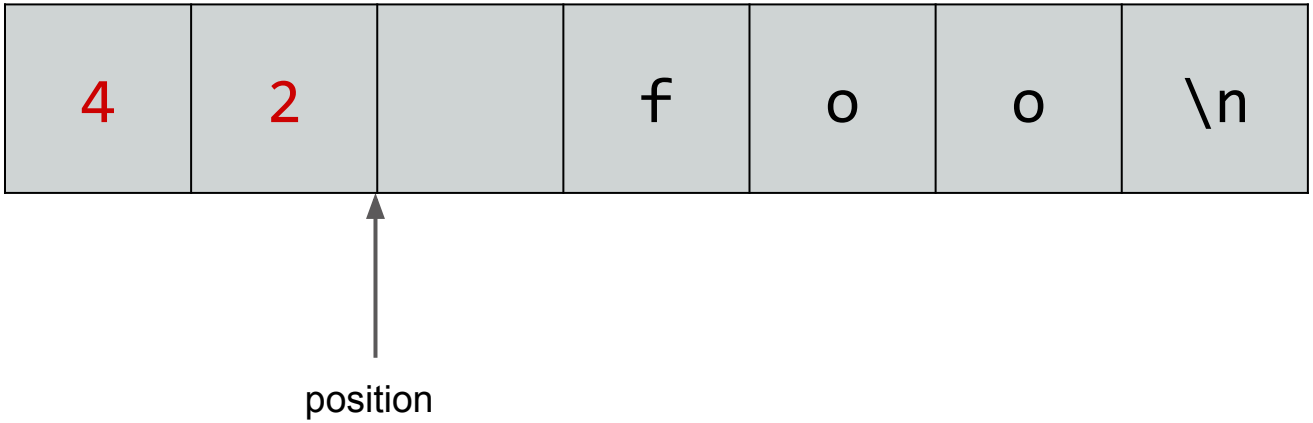
The first read of `cin` will successfully read ‘42’




```
int value;
cin >> value; // value == 42
```

Unread Data

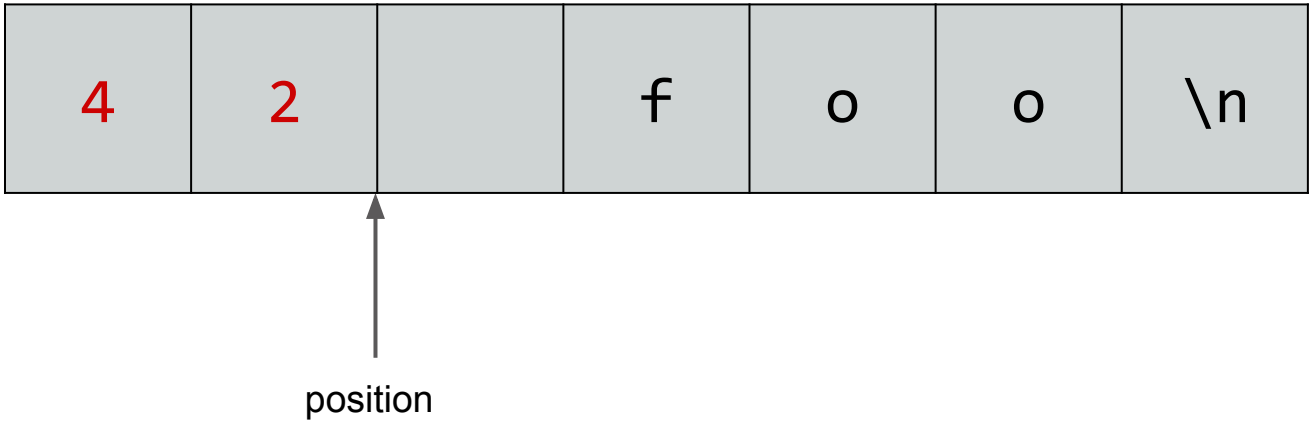
The second read of `cin` will occur, but the program will not ask for more input




```
int secondValue;
cin >> secondValue; // ???
```

Unread Data

The second read of `cin` will fail, since “foo” cannot be interpreted as an integer




```
int secondValue;
cin >> secondValue; // ???
```

Unread Data

Let's take a look at this problem in an example getInteger function

(StaysOnStream.pro)

Complicated Parts

We're now ready to talk about some of the messy parts of working with streams.

- Unread data sits on the stream
- A stream which has failed stays failed
- Mixing >> and getline
- Putting it all together
- Understanding the Stanford library functions

Once failed, stays failed

- If we try and read an int from a stream containing only the data "hello", the operation will fail, and the **fail bit** will be set.
  - The same applies if you try and read a double when the string contains only a string, etc.
- The fail bit will remain set until you explicitly call .clear().
- Let's take a look at a code snippet demonstrating that

Once failed, stays failed

Let's take a look at this problem
(StaysFailed.pro)

Complicated Parts

We're now ready to talk about some of the messy parts of working with streams.

- Unread data sits on the stream
- A stream which has failed stays failed
- Mixing >> and getline
- Putting it all together
- Understanding the Stanford library functions

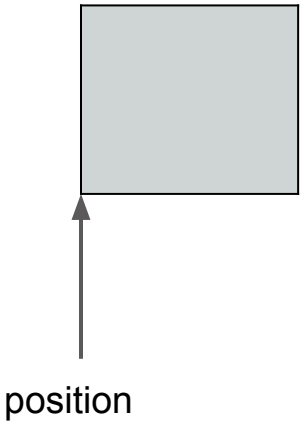
Mixing >> and getline

Let’s see what happens when mix >> and getline. See if you can guess what goes wrong

(MixingGetline.pro)

Mixing >> and getline

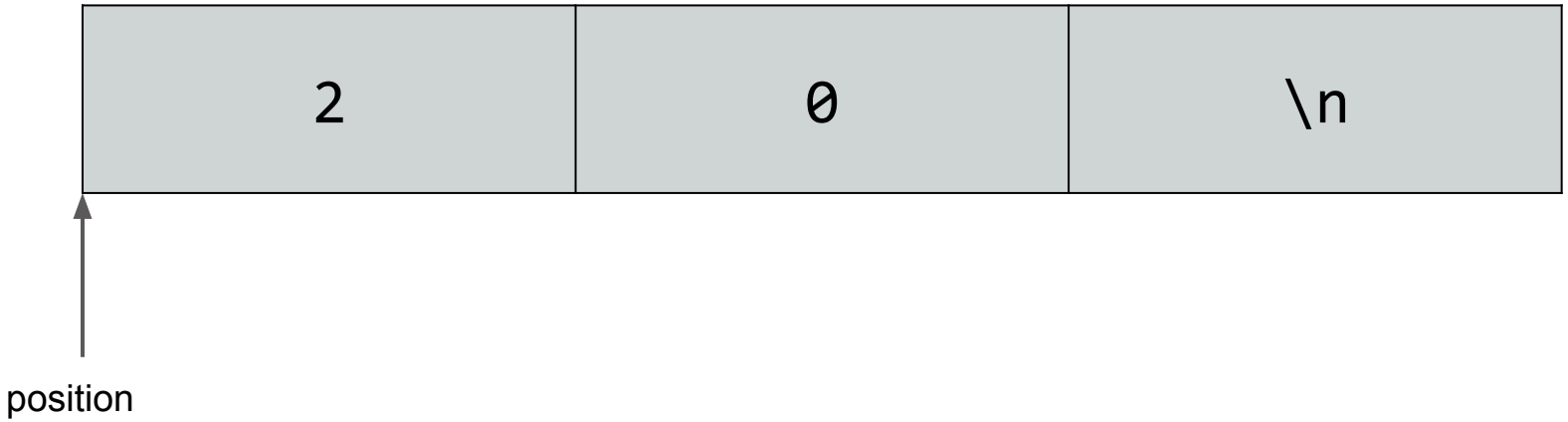
`cin` starts off with no input on it




```
int value;
cin >> value; // value == ?
```

Mixing >> and getline

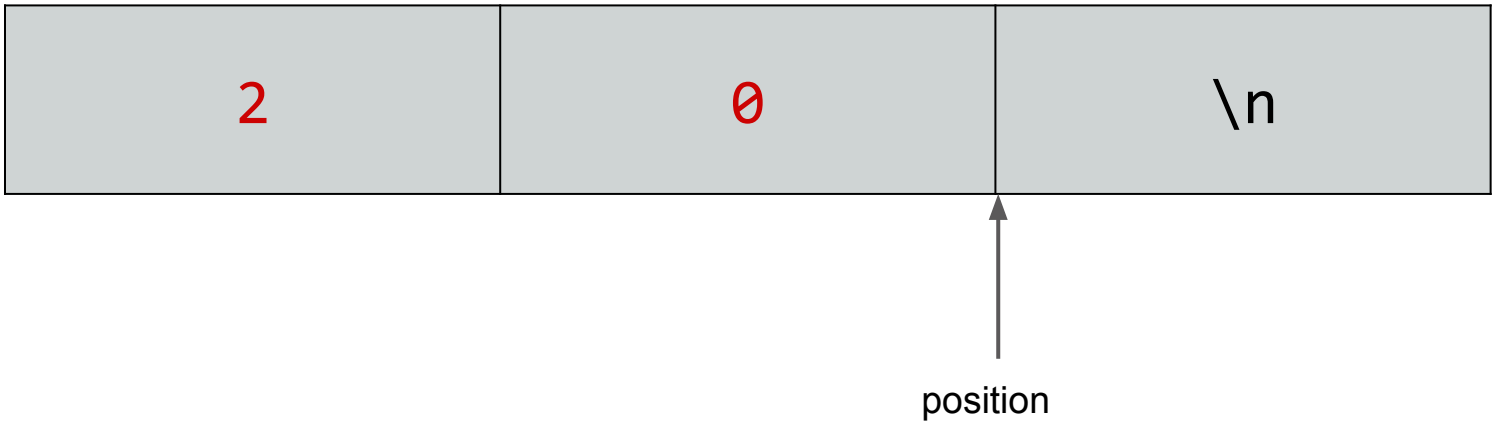
The user types in input, a newline at the end




```
int value;
cin >> value;
```

Mixing >> and getline

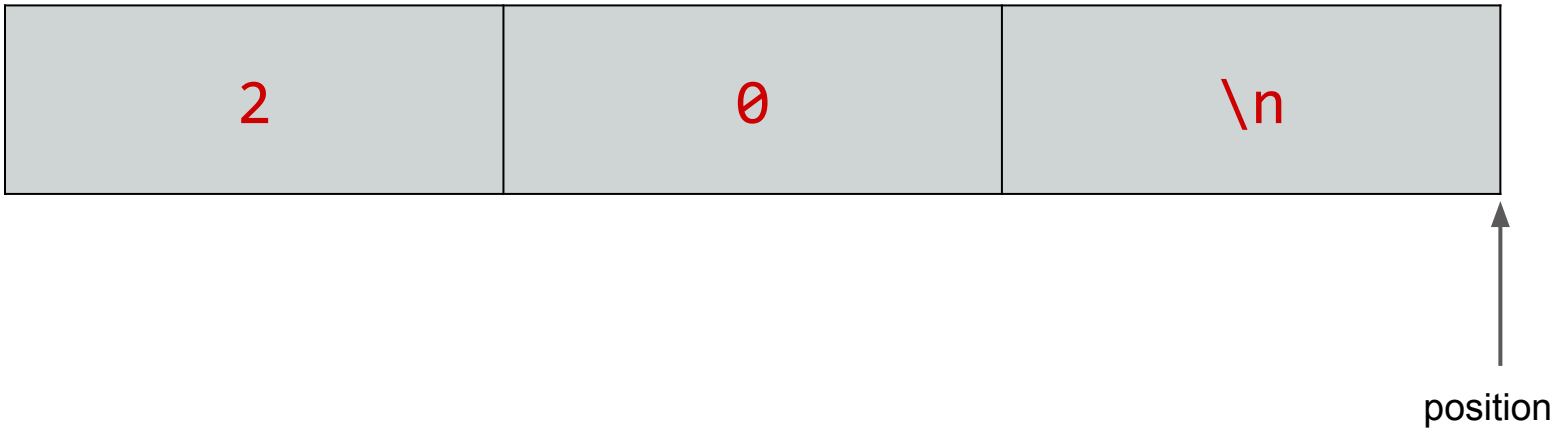
The first read of `cin` will successfully read ‘20’




```
int numUnits;
cin >> numUnits; // numUnits == 20
```

Mixing >> and getline

The call to `getline` will read the newline still sitting on the buffer!




```
string favoriteClass;
getline(cin, favoriteClass);
```

Complicated Parts

We're now ready to talk about some of the messy parts of working with streams.

- Unread data sits on the stream
- A stream which has failed stays failed
- Mixing >> and getline
- Putting it all together
- Understanding the Stanford library functions

Putting It Together

Let's try putting this together and writing a complete program.

- We will read a number from the user and do some manipulation on it
- We will only accept the user's input if it contains a valid integer, and nothing but a valid integer (no trailing junk)

Putting It Together

Let's take a look at this problem
(PutTogether.pro)

Complicated Parts

We're now ready to talk about some of the messy parts of working with streams.

- Unread data sits on the stream
- A stream which has failed stays failed
- Mixing >> and getline
- Putting it all together
- Understanding the Stanford library functions

Stanford Libraries

Now that we've had some exposure to streams, we can actually understand what the Stanford library functions are doing under the hood!

simpio.h	strlib.h
getInteger getReal getLine	intToString realToString stringToInt stringToReal

Stanford Libraries

```
// getLine is pretty easy to write
string getLine() {
   string line;
   getline(cin, line);
   return line;
}
```

Stanford Libraries

Okay, so getLine was pretty easy to write. How about the rest?

simpio.h	strlib.h
getInteger getReal ~~getLine~~	intToString realToString stringToInt stringToReal

Stanford Libraries

Let's take a look at getInteger (getReal is similar).

See code in SimpleIO.pro

simpio.h	strlib.h
~~getInteger~~ ~~getReal~~ ~~getLine~~	intToString realToString stringToInt stringToReal

Stanford Libraries

Let's take a look at the functions in "strlib.h" now

See code in StrLib.pro

simpio.h	strlib.h
~~getInteger~~ ~~getReal~~ ~~getLine~~	~~intToString~~ ~~realToString~~ ~~stringToInt~~ ~~stringToReal~~

Awesome Shortcut

Instead of doing:

```
while (true) {
   stream >> value;
   if (stream.fail()) break;
   //stuff
}
```

Awesome Shortcut

Do:

```
while (stream >> value) {
   //stuff
}
```

Why?

Awesome Shortcut

Stream extraction/insertion returns the stream

```
cout << “Hello World” << endl;
```

is equivalent to:

```
(cout << “Hello World”) << endl;
```

Awesome Shortcut

Streams return the negation of the **fail bit** when evaluated as a boolean

```
if (stream)
```

is equivalent to

```
if (!stream.fail())
```

Stream Manipulation

- The setw (set whitespace) method allows you to pad your output

```
cout << “[“ << setw(10) << “Hi!” <<
            “]” << endl;
```

Will output

```
[       Hi!]
```

Stream Manipulation

- To switch the padding from the left to the right of the input use the keyword `right`

```
cout << “[“ << right << setw(10) <<
        “Hi!” << “]” << endl;
```

Will output

```
[Hi!       ]
```

Stream Manipulation

- The padding is all spaces by default but you can change that by using the `setfill` manipulator

```
cout << “[“ << setfill(‘!’) << right
<< setw(10) << “Hi!” << “]” << endl;
```

Will output

```
[Hi!!!!!!!!]
```

Stream Manipulation

- Other manipulators include changing the base of a number and the format of the output

```
cout << hex << 10; //prints a
cout << oct << 10; //prints 8
```

Closing Thoughts

C++ streams are definitely not simple, but once you understand them, you'll get used to their quirks and understand their usefulness.

Next Class

- Assignment 1 goes out
- Start looking at standard C++ collection classes