Using Streams

Reid Watson
(rawatson@stanford.edu)

Complicated Parts

We spent last class talking about the idea behind C++ streams, now lets explore some of the difficulties that come with using them in practice.

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.

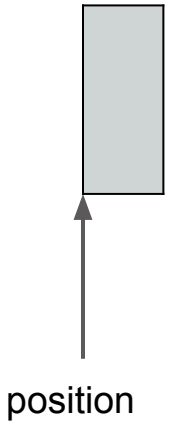
- Unread data sits on the stream
- A stream which has failed stays failed
- Mixing >> and getline
- Putting it all together
- Understanding the Stanford library functions

Complicated Parts

- If we read read an integer from `cin`, and the user types "30 foo", we will read the integer 30.
- 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.

Complicated parts

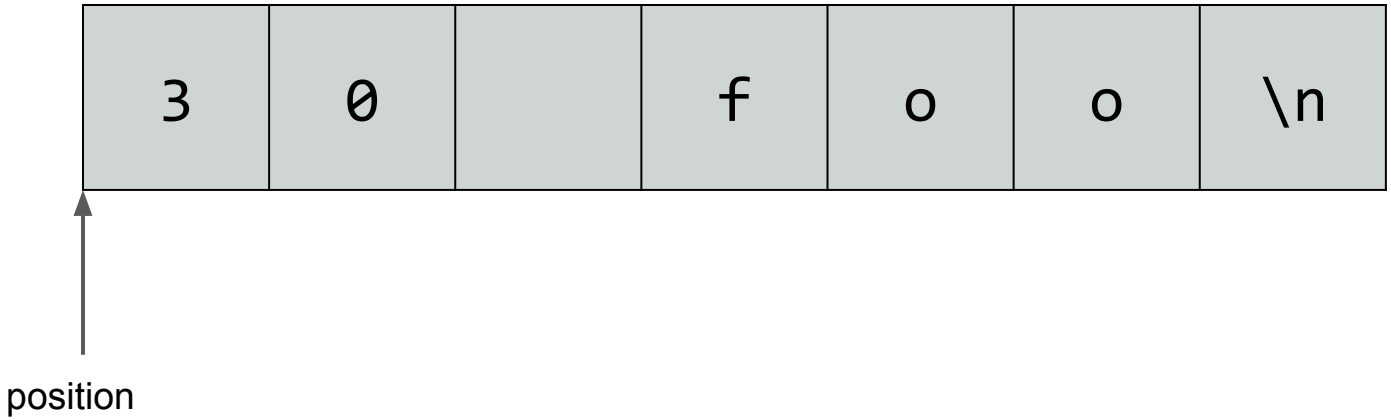
`cin` starts off with no input on it




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

Complicated parts

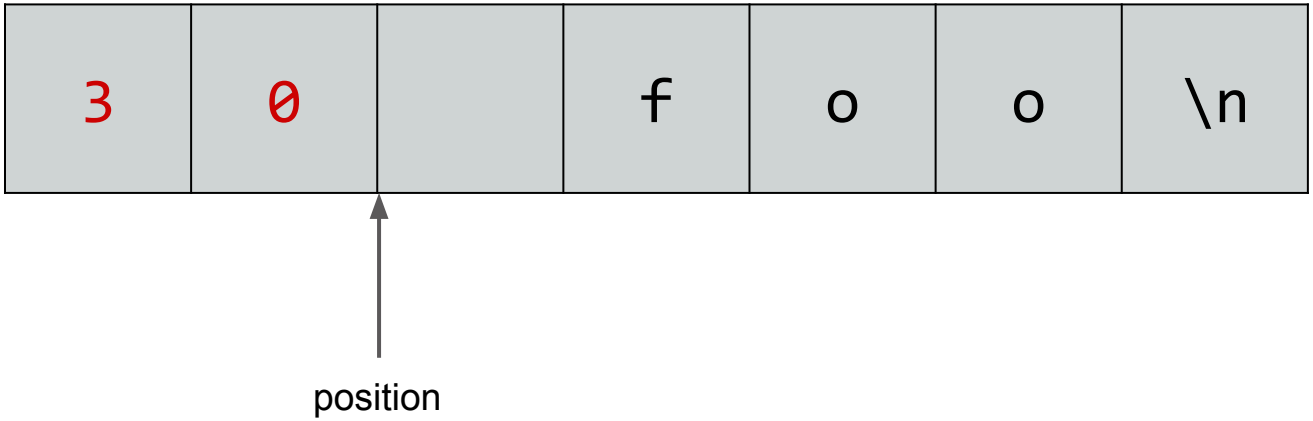
The user types in input, including some trailing garbage (“foo”)




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

Complicated parts

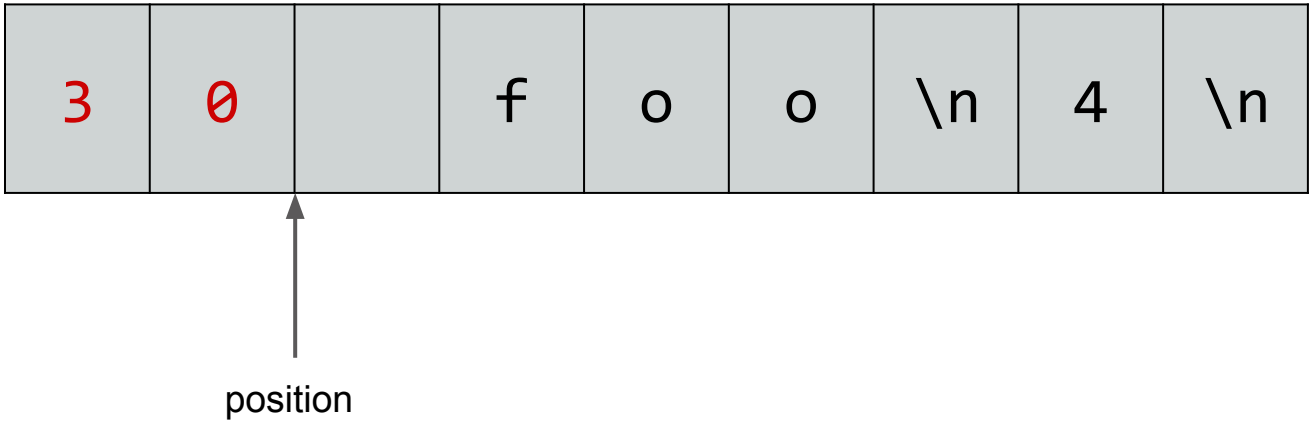
The first read of `cin` will successfully read ‘30’




```
int value;
cin >> value; // value == 30
```

Complicated parts

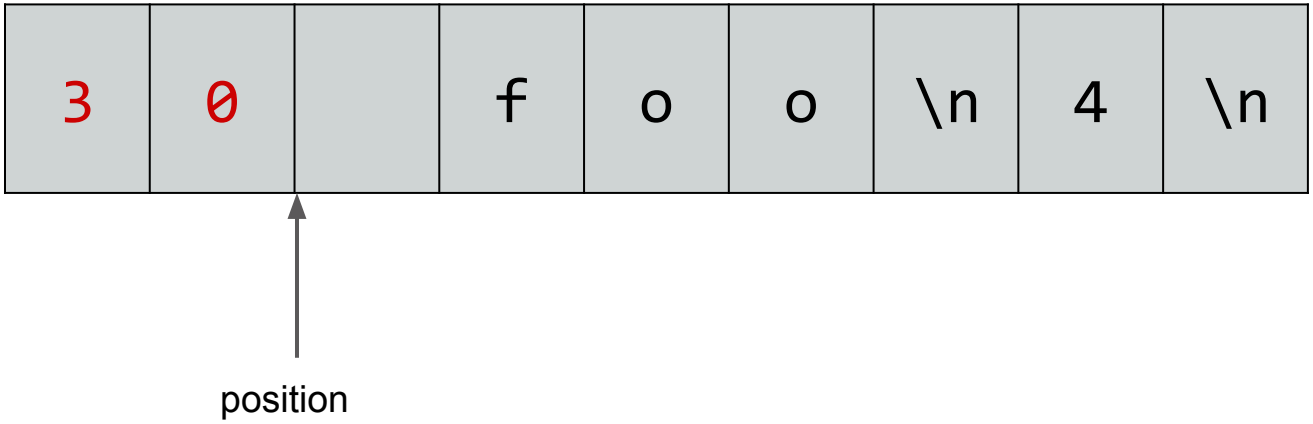
The second read of `cin` will occur, and the user will type in ‘4’




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

Complicated parts

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




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

Complicated Parts

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

See code in 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

Complicated Parts

- 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

Complicated Parts

Let's take a look at this problem

See code in 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

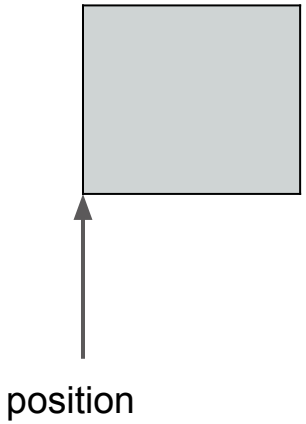
Complicated Parts

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

See code in MixingGetline.pro

Complicated parts

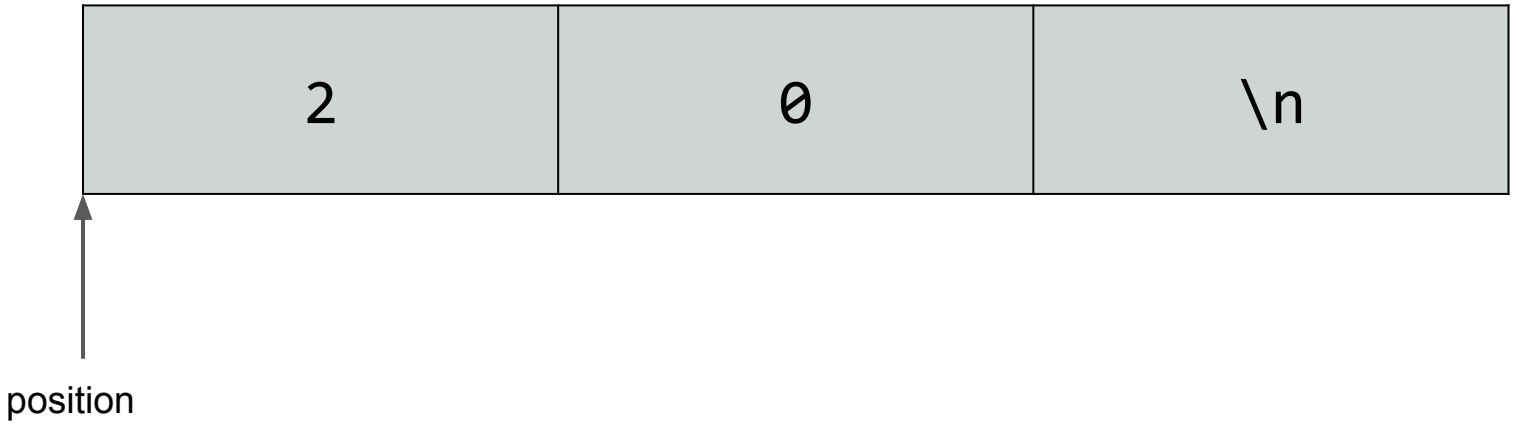
`cin` starts off with no input on it




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

Complicated parts

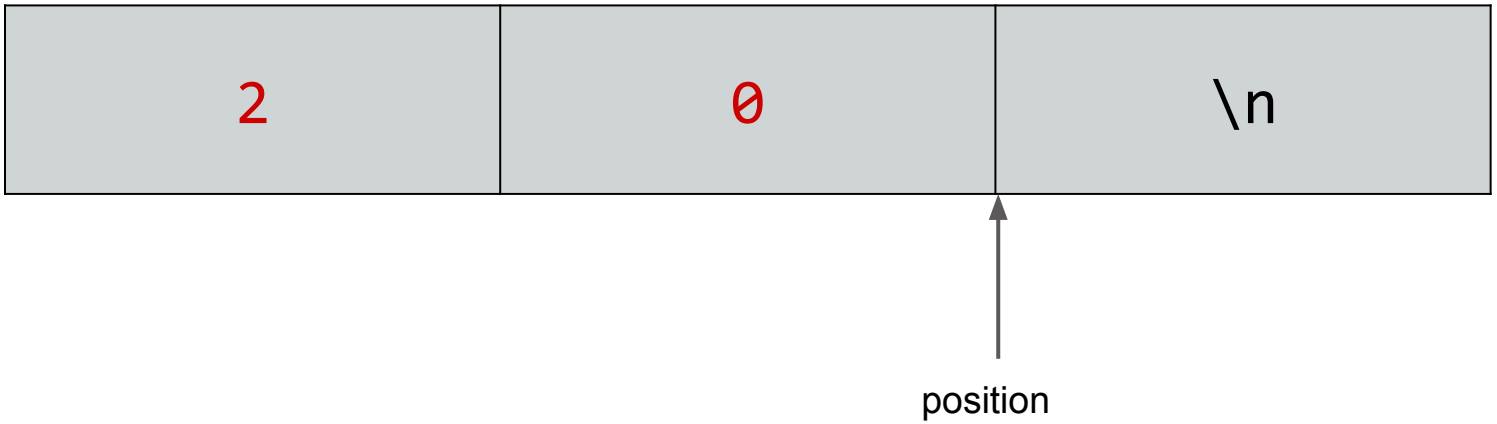
The user types in input, a newline at the end




```
int numberOfClasses;
cin >> numberOfClasses;
```

Complicated parts

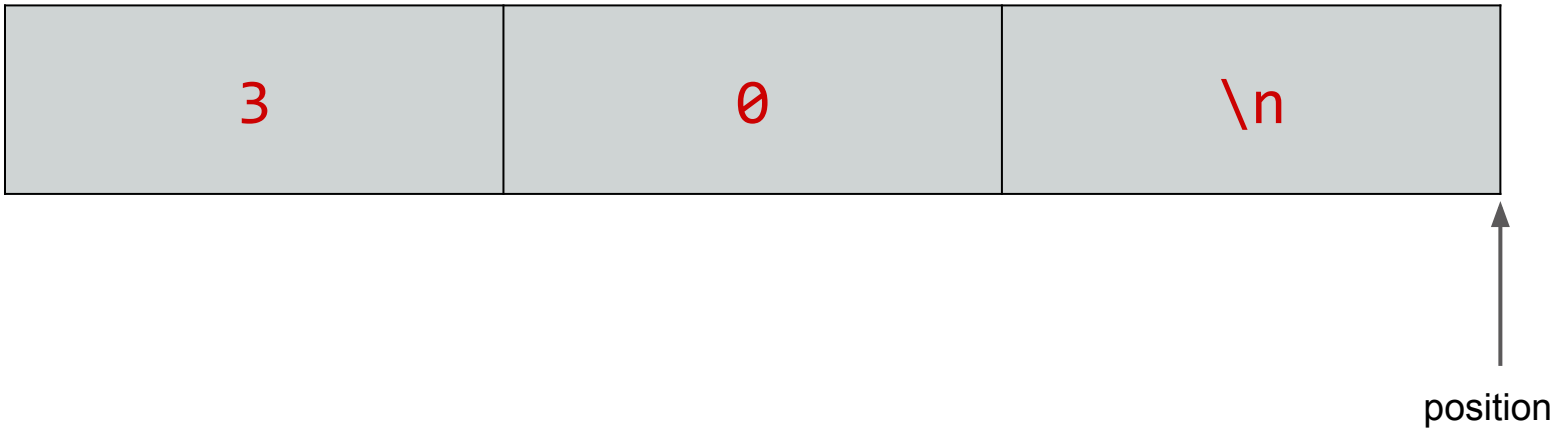
The first read of `cin` will successfully read ‘20’




```
int value;
cin >> value; // value == 30
```

Complicated parts

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




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

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

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

- We will read a number from the user and do some very basic 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)

Complicated Parts

Let's take a look at this problem

See code in put-together.cpp

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

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

Complicated Parts

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

Complicated Parts

OK, so getLine was pretty easy to write.

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

Complicated Parts

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

Complicated Parts

Let’s take a look at the strlib.h functions now

See code in StrLib.pro

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

Closing Thoughts

C++ streams are not the simplest things to work with, but once you understand them, you'll get used to their quirks and their usefulness.