Streams

How can we convert between string-represented data and the real thing?

masks strongly recommended

Attendance

bit.ly/3kqtXM

Announcements

Announcements

- Office hours times posted on class website!
  - Haven: Tuesday 4:30 - 5:30pm in 260-113
  - Sarah: Wednesday 3:15 - 4:15pm in 120-314

A note about feedback

- We welcome feedback! This class is meant for you.
  - Always welcome to send us an email, make an Ed post, or talk to us after class or in office hours
  - If you want to provide feedback anonymously, we created an anonymous feedback form (also posted on Ed)
- Thank you for the feedback we received already!

A note about feedback

- Thank you for the feedback we received already!
  - Lecture pace a little fast
  - Want more practice besides homework
- Response:
  - Intentionally going to go slower
  - Attempting to pilot optional and ungraded practice problems

Recap:

- **Uniform Initialization**
  - A “uniform” way to initialize variables of different types!

Recap:

- **Uniform Initialization**
  - A “uniform” way to initialize variables of different types!
- **References**
  - Allow us to alias variables

Recap:

- **Uniform Initialization**
  - A “uniform” way to initialize variables of different types!
- **References**
  - Allow us to alias variables
- **Const**
  - Allow us to specify that a variable can’t be modified

Today

- **What are streams?**
- Output streams
- Input streams
- String streams!

Definition

stream: an abstraction for input/output. Streams convert between *data* and the *string representation of data.*

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
```

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
```

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s << std::endl;
```

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s << std::endl;
```

ERROR!

A stream you've used: cout

```
// use a stream to print
std::cout << 5 << std::e
// and most from the STI
std::cout << "Sarah" <<
// Mix types!
std::cout << "Sarah is "
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s << std::endl;
```

Reminder: Our student struct

```
struct Student {
    string name;
    string state;
    int age;
};
```

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// structs?
Student s = {"Sarah", "CA", 21};
std::cout << s.name << s.age << std::endl;
```

Works ✓

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " << 21 << std::endl;
// Any primitive type + most from the STL work!
// For other types, you will have to write the
                << operator yourself!
```

A stream you've used: cout

```
// use a stream to print any primitive type!
std::cout << 5 << std::endl; // prints 5
// and most from the STL work!
std::cout << "Sarah" << std::endl;
// Mix types!
std::cout << "Sarah is " <<
// Any primitive type + most
// For other types, you will
        << operator yourself!
```

We'll talk about how to write the << operator for custom types during lecture 11 on Operators!

`std::cout` **is an *output stream*. It has type** `std::ostream`

Two ways to classify streams

By Direction:

By Source or Destination:

Two ways to classify streams

By Direction:

- **Input streams:** Used for reading data (ex. ‘std::istream’, ‘std::cin’)

By Source or Destination:

Two ways to classify streams

By Direction:

- **Input streams:** Used for reading data (ex. ‘std::istream’, ‘std::cin’)
- **Output streams:** Used for writing data (ex. ‘std::ostream’, ‘std::cout’)

By Source or Destination:

Two ways to classify streams

By Direction:

- **Input streams:** Used for reading data (ex. ‘std::istream’, ‘std::cin’)
- **Output streams:** Used for writing data (ex. ‘std::ostream’, ‘std::cout’)
- **Input/Output streams:** Used for both reading and writing data (ex. ‘std::iostream’, ‘std::stringstream’)

By Source or Destination:

Two ways to classify streams

By Direction:

- **Input streams:** Used for reading data (ex. ‘std::istream’, ‘std::cin’)
- **Output streams:** Used for writing data (ex. ‘std::ostream’, ‘std::cout’)
- **Input/Output streams:** Used for both reading and writing data (ex. ‘std::iostream’, ‘std::stringstream’)

By Source or Destination:

- **Console streams:** Read/write to console (ex. ‘std::cout’, ‘std::cin’)

Two ways to classify streams

By Direction:

- **Input streams:** Used for reading data (ex. ‘std::istream’, ‘std::cin’)
- **Output streams:** Used for writing data (ex. ‘std::ostream’, ‘std::cout’)
- **Input/Output streams:** Used for both reading and writing data (ex. ‘std::iostream’, ‘std::stringstream’)

By Source or Destination:

- **Console streams:** Read/write to console (ex. ‘std::cout’, ‘std::cin’)
- **File streams:** Read/write to files (ex. ‘std::fstream’, ‘std::ifstream’, ‘std::ofstream’)

Two ways to classify streams

By Direction:

- **Input streams:** Used for reading data (ex. ‘std::istream’, ‘std::cin’)
- **Output streams:** Used for writing data (ex. ‘std::ostream’, ‘std::cout’)
- **Input/Output streams:** Used for both reading and writing data (ex. ‘std::iostream’, ‘std::stringstream’)

By Source or Destination:

- **Console streams:** Read/write to console (ex. ‘std::cout’, ‘std::cin’)
- **File streams:** Read/write to files (ex. ‘std::fstream’, ‘std::ifstream’, ‘std::ofstream’)
- **String streams:** Read/write to strings (ex. ‘std::stringstream’, ‘std::istringstream’, ‘std::ostringstream’)

Today

- ~~What are streams?~~
- **Output streams**
- Input streams
- String streams!

Output Streams!

Output Streams

- Have type `std::ostream`
- Can only ***send*** data to the stream
  - Interact with the stream using the `<<` operator
  - Converts any type into string and ***sends*** it to the stream

Output Streams

- Have type `std::ostream`
- Can only ***send*** data to the stream
  - Interact with the stream using the `<<` operator
  - Converts any type into string and ***sends*** it to the stream
- `std::cout` is the output stream that goes to the console

```
std::cout << 5 << std::endl;
// converts int value 5 to string “5”
// sends “5” to the console output stream
```

Output File Streams

- Have type `std::ofstream`
- Only ***send*** data using the `<<` operator
  - Converts data of any type into a string and sends it to the **file stream**

Output File Streams

- Have type `std::ofstream`
- Only ***send*** data using the `<<` operator
  - Converts data of any type into a string and sends it to the **file stream**

- Must initialize your own `ofstream` object linked to your file

```
std::ofstream out("out.txt");
// out is now an ofstream that outputs to out.txt
out << 5 << std::endl; // out.txt contains 5
```

`std::cout` **is a *global constant object* that you get from**

`#include <iostream>`

`std::cout` is a *global constant object* that you get from `#include <iostream>`

To use any other output stream, you must first initialize it!

Questions?

Today

- ~~What are streams?~~
- ~~Output streams~~
- **Input streams**
- String streams!

Input Streams!

What does this code do?

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

What does this code do?

```
int x;
std::cin >> x;
// what happens if input is 5 ?
// how about 51375 ?
// how about 5 1 3 7 5?
```

Let's try it out!

A note about nomenclature

- “>>” is the **stream extraction operator** or simply extraction operator
  - Used to **extract data from a stream** and place it into a variable

A note about nomenclature

- “>>” is the **stream extraction operator** or simply extraction operator
  - Used to **extract data from a stream** and place it into a variable
- “<<” is the **stream insertion operator** or insertion operator
  - Used to **insert data into a stream** usually to output the data to a file, console, or string

`std::cin` **is an *input stream.* It has type** `std::istream`

Input Streams

- Have type `std::istream`
- Can only ***receive*** strings using the `>>` operator
  - ***Receives*** a string from the stream and converts it to data

Input Streams

- Have type `std::istream`
- Can only ***receive*** strings using the `>>` operator
  - ***Receives*** a string from the stream and converts it to data
- `std::cin` is the input stream that gets input from the console

```
int x;
string str;
std::cin >> x >> str;
//reads exactly one int then one string from console
```

Nitty Gritty Details: `std::cin`

- First call to `std::cin >>` creates a command line prompt that allows the user to type until they hit enter

Nitty Gritty Details: `std::cin`

- First call to `std::cin >>` creates a command line prompt that allows the user to type until they hit enter
- Each `>>` ONLY reads until the next *whitespace*
  - Whitespace = tab, space, newline

Nitty Gritty Details: `std::cin`

- First call to `std::cin >>` creates a command line prompt that allows the user to type until they hit enter
- Each `>>` ONLY reads until the next *whitespace*
  - Whitespace = tab, space, newline
- Everything after the first whitespace gets saved and used the next time `std::cin >>` is called
  - The place its saved is called a **buffer**!

Nitty Gritty Details: `std::cin`

- First call to `std::cin >>` creates a command line prompt that allows the user to type until they hit enter
- Each `>>` ONLY reads until the next *whitespace*
  - Whitespace = tab, space, newline
- Everything after the first whitespace gets saved and used the next time `std::cin >>` is called
  - The place its saved is called a **buffer**!
- If there is nothing waiting in the buffer, `std::cin >>` creates a new command line prompt

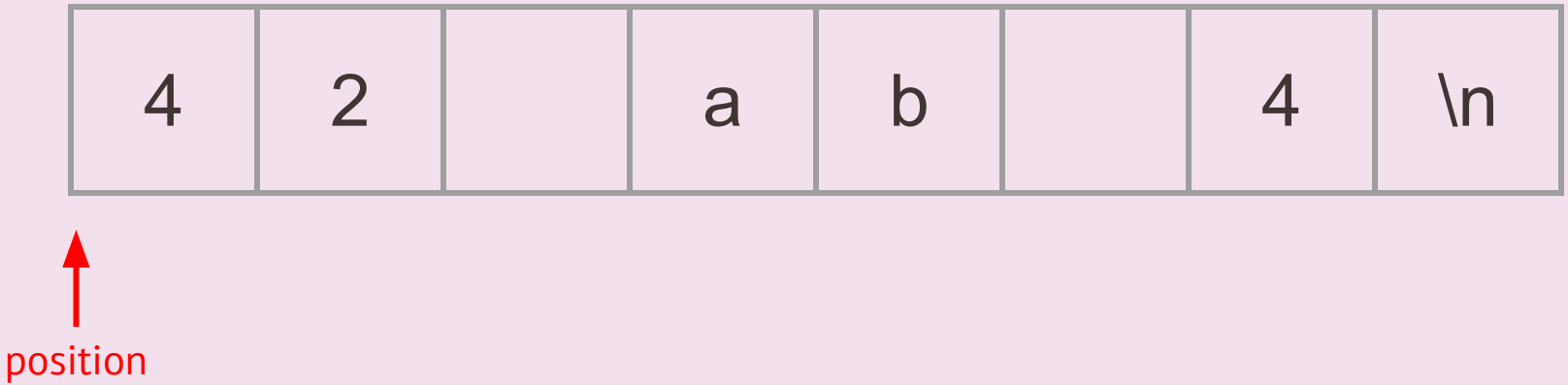
Nitty Gritty Details: `std::cin`

- First call to `std::cin >>` creates a command line prompt that allows the user to type until they hit enter
- Each `>>` ONLY reads until the next *whitespace*
  - Whitespace = tab, space, newline
- Everything after the first whitespace gets saved and used the next time `std::cin >>` is called
  - The place its saved is called a **buffer**!
- If there is nothing waiting in the buffer, `std::cin >>` creates a new command line prompt
- Whitespace is eaten; it won't show up in output

That was a lot of text. Let’s see what that looks like in practice

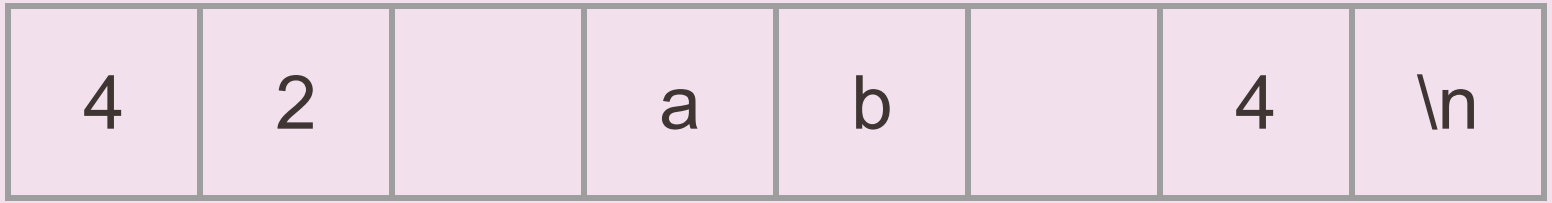
Code Demo

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




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

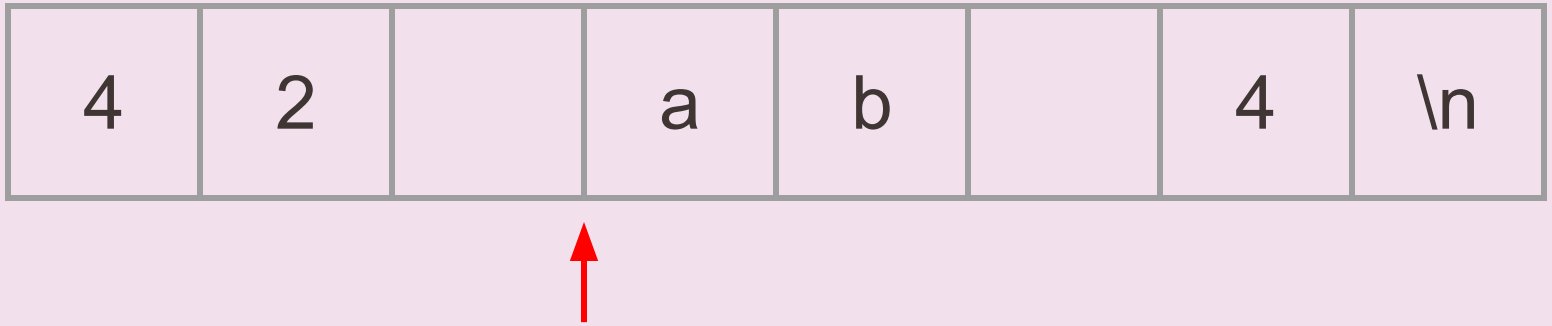
Think of a `std::istream` as a sequence of characters



position

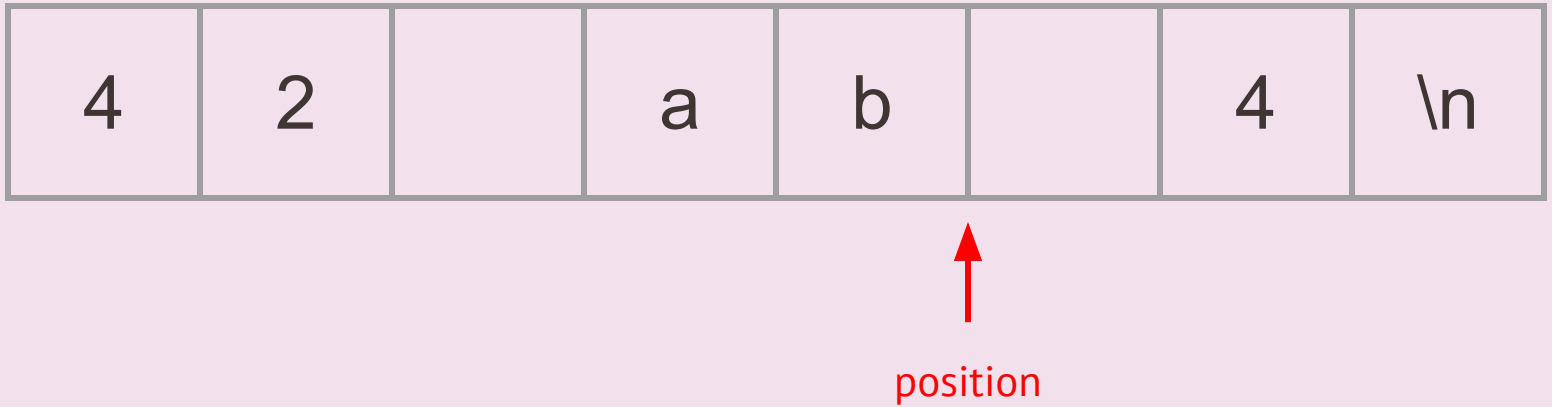
```
int x; string y; int z;
cin >> x; //42 put into x
cin >> y;
cin >> z;
```

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



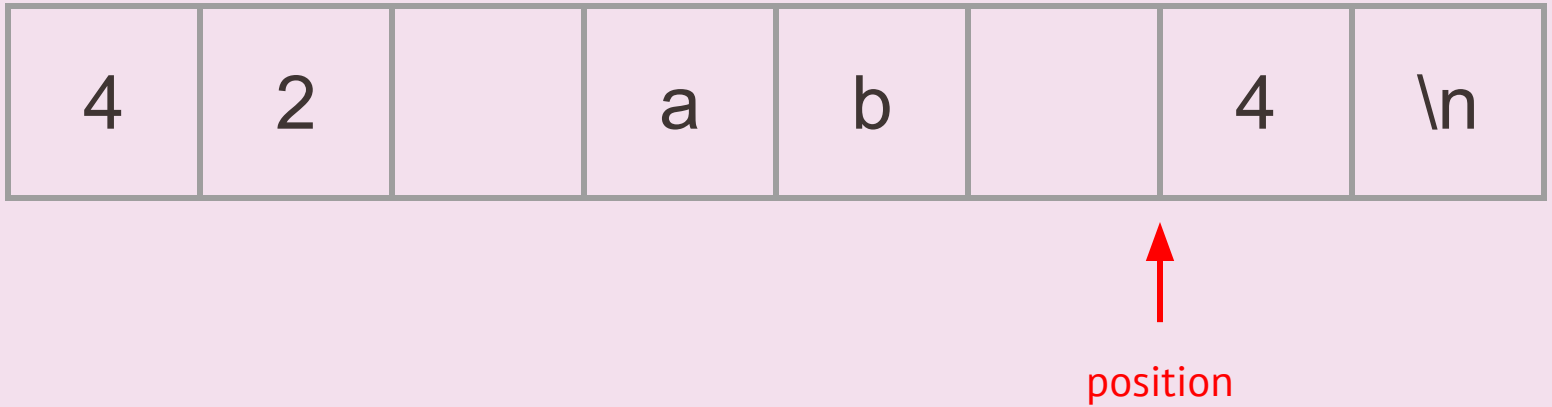

```
int x; string y; int z;
cin >> x; //42 put into x
cin >> y;
cin >> z;
```

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



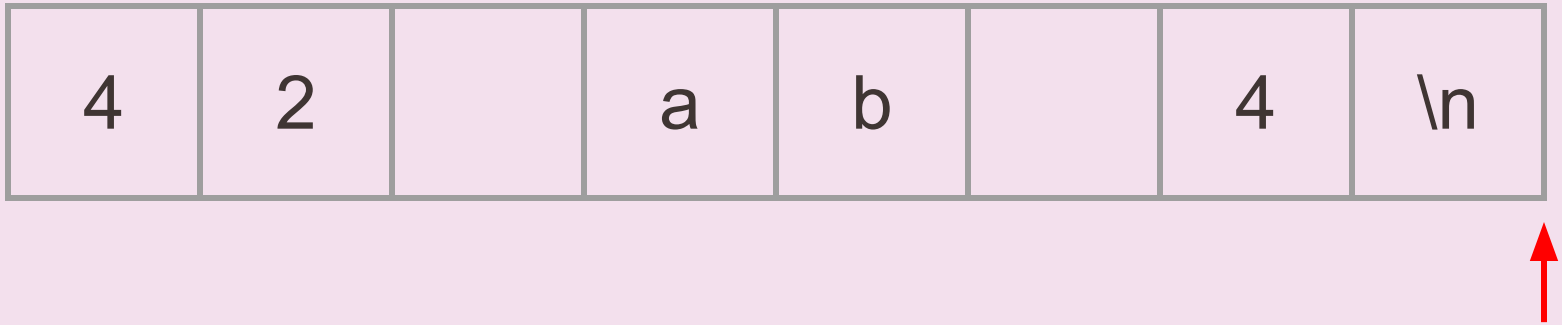

```
int x; string y; int z;
cin >> x;
cin >> y; //ab put into y
cin >> z;
```

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




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

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

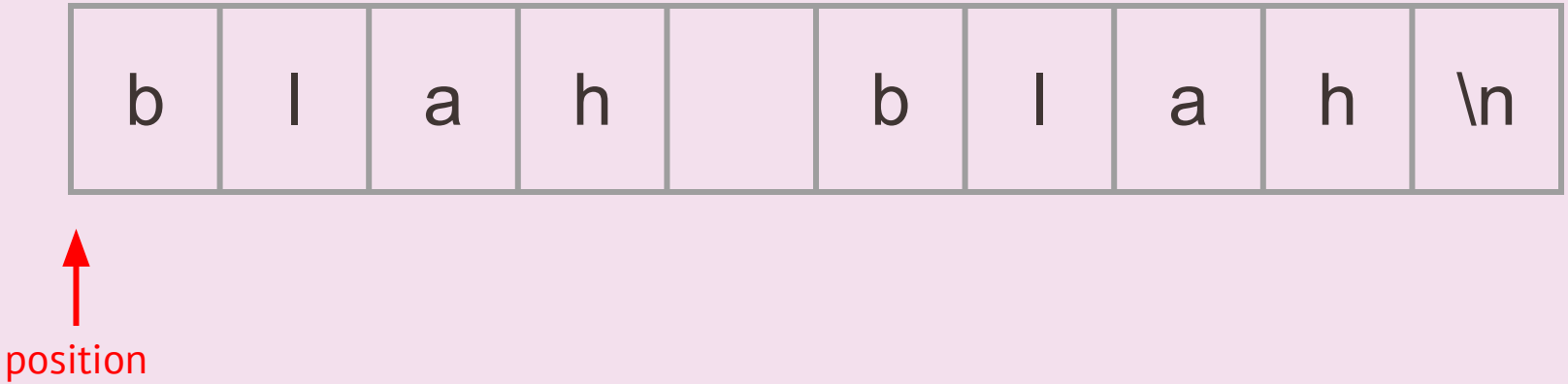



```
int x; string y; int z;
cin >> x;
cin >> y;
cin >> z; //4 put into z
```

Input Streams: When things go wrong

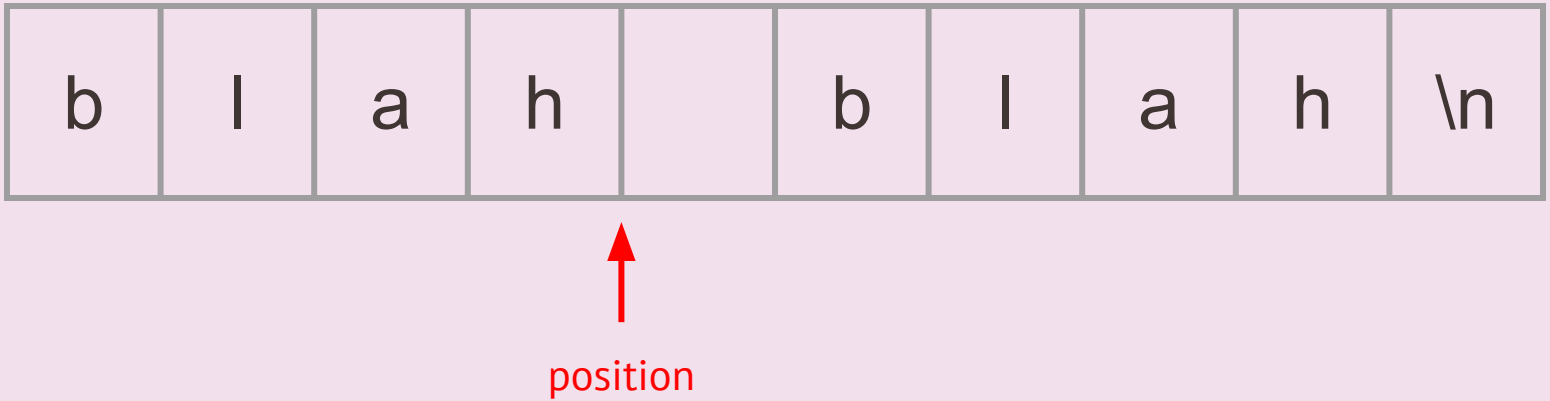
```
string str;
int x;
std::cin >> str >> x;
//what happens if input is blah blah?
std::cout << str << x;
```

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



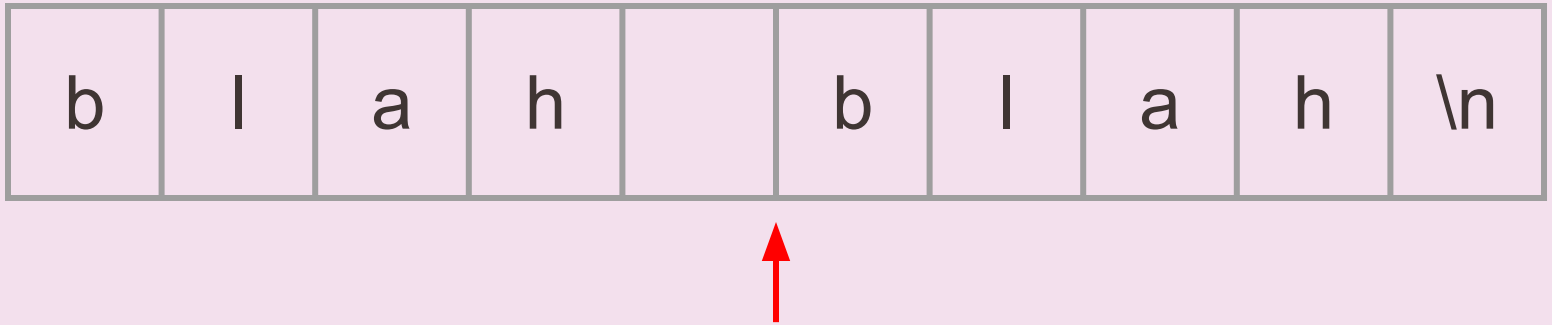

```
string str; int x;
std::cin >> str >> x;
```

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



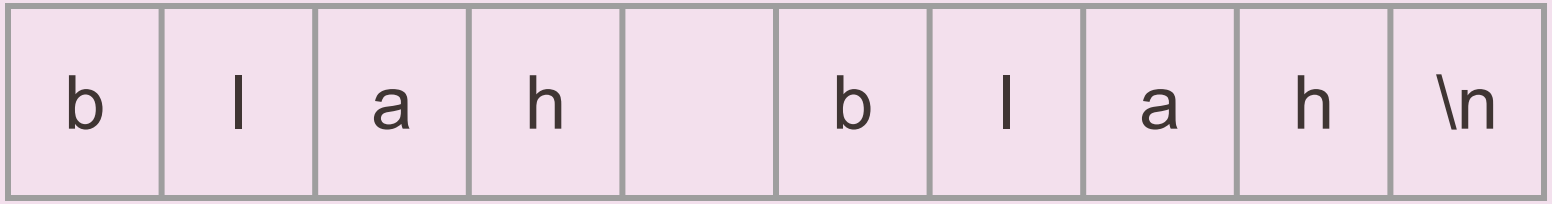

```
string str; int x;
std::cin >> str >> x;
```

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



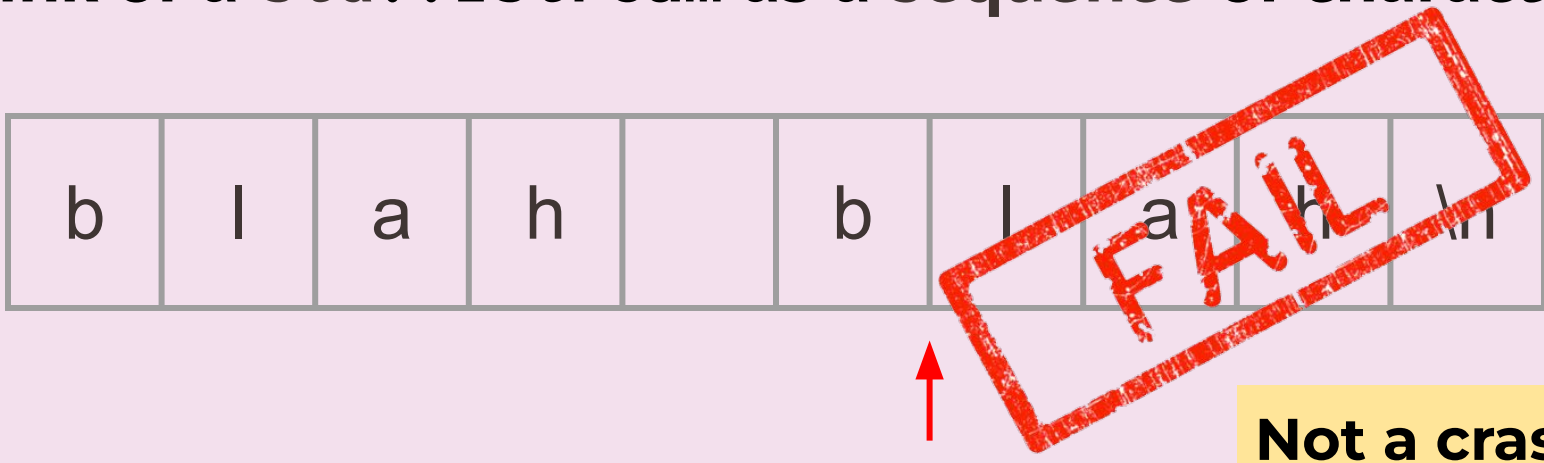

```
string str; int x;
std::cin >> str >> x;
```

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




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

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




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

Not a crash.
x stores 0 to indicate a fail.

Input Streams: When things go wrong

```
string str;
int x;
string otherStr;
std::cin >> str >> x >> otherStr;
//what happens if input is blah blah blah?
std::cout << str << x << otherStr;
```

Let's try it out!

Input Streams: When things go wrong

str → "blah"
x → 0
otherStr → NOTHING

```
string str;
int x;
string otherStr;
std::cin >> str >> x >> otherStr;
//what happens if input is blah blah blah?
std::cout << str << x << otherStr;
//once an error is detected, the input stream's
//fail bit is set, and it will no longer accept
//input
```

Input Streams: When things go wrong

```
int age; double hourlyWage;
cout << "Please enter your age: ";
cin >> age;
cout << "Please enter your hourly wage: ";
cin >> hourlyWage;
//what happens if first input is 2.17?
```

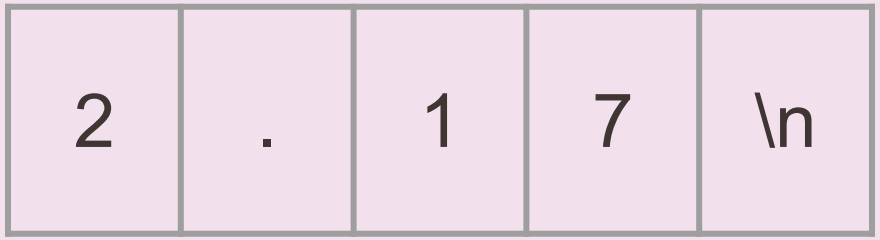
Think of a `std::istream` as a sequence of characters


```
cin >> age;
cout << "Wage: ";
cin >> hourlyWage;
```

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


```
cin >> age;
cout << "Wage: ";
cin >> hourlyWage;
```

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



position

Reads until it finds something that isn't an int!

```
cin >> age; // age = 2
cout << "Wage: ";
cin >> hourlyWage;
```

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


```
cin >> age;
cout << "Wage: ";
cin >> hourlyWage;// =.17
```

Questions?

std::getline()

```
// Used to read a line from an input stream
// Function Signature
istream& getline(istream& is, string& str, char delim);
```

std::getline()

```
// Used to read a line from an input stream
// Function Signature
istream& getline(istream& is, string& str, char delim);
```

getline reads from

std::getline()

```
// Used to read a line from an input stream
// Function Signature
istream& getline(istream& is, string& str, char delim);
```

getline reads from

& stores output in

std::getline()

```
// Used to read a line from an input stream
// Function Signature
istream& getline(istream& is, string& str, char delim);
```

getline reads from

& stores output in

Stops when read.
'\n' = default

```
// Designed to work with character sequences
```

std::getline(istream& is, string& str, char delim)

- **How it works:**
  - Clears contents in str

std::getline(istream& is, string& str, char delim)

- **How it works:**
  - Clears contents in str
  - Extracts chars from is and stores them in str until:

std::getline(istream& is, string& str, char delim)

- **How it works:**
  - Clears contents in str
  - Extracts chars from is and stores them in str until:
    - End of file reached, sets EOF bit (checked using is.eof())

std::getline(istream& is, string& str, char delim)

- **How it works:**
  - Clears contents in str
  - Extracts chars from is and stores them in str until:
    - End of file reached, sets EOF bit (checked using is.eof())
    - Next char in is is delim, extracts but does not store delim

std::getline(istream& is, string& str, char delim)

- **How it works:**
  - Clears contents in str
  - Extracts chars from is and stores them in str until:
    - End of file reached, sets EOF bit (checked using is.eof())
    - Next char in is is delim, extracts but does not store delim
    - str out of space, sets FAIL bit (checked using is.fail())

std::getline(istream& is, string& str, char delim)

- **How it works:**
  - Clears contents in str
  - Extracts chars from is and stores them in str until:
    - End of file reached, sets EOF bit (checked using is.eof())
    - Next char in is is delim, extracts but does not store delim
    - str out of space, sets FAIL bit (checked using is.fail())
  - If no chars extracted for any reason, FAIL bit set

std::getline(istream& is, string& str, char delim)

- **How it works:**
  - Clears contents in str
  - Extracts chars from is a
    - End of file reached,
    - Next char in is is delim, extracts but does not store delim
    - str out of space, sets FAIL bit (checked using is.fail())
  - If no chars extracted for any reason, FAIL bit set

In contrast:

- **“>>”** only **reads until it hits whitespace** (so can’t read a sentence in one go)

std::getline(istream& is, string& str, char delim)

- **How it works:**
  - Clears contents in str
  - Extracts chars from is a
    - End of file reached,
    - Next char in is is del
    - str out of space, sets FAIL bit (checked using is.fail())
  - If no chars extracted for any reason, FAIL bit set

In contrast:

- **“>>”** only **reads until it hits whitespace** (so can’t read a sentence in one go)
- BUT **“>>”** can **convert data to built-in types** (like ints) while getline can only produce strings.

std::getline(istream& is, string& str, char delim)

- **How it works:**
  - Clears contents in str
  - Extracts chars from is a
    - End of file reached,
    - Next char in is is del
    - str out of space, sets
  - If no chars extracted for

In contrast:

- **“>>”** only **reads until it hits whitespace** (so can’t read a sentence in one go)
- BUT **“>>”** can **convert data to built-in types** (like ints) while getline can only produce strings.
- AND **“>>”** only **stops reading at predefined whitespace** while getline can stop reading at any delimiter you define

Reading using >> extracts a single "word" or built-in type

including for strings

To read a whole line, use

```
std::getline(istream& stream, string& line);
```

How to use getline

- Notice `getline(istream& stream, string& line)` takes in both parameters by reference!

```
std::string line;
std::getline(cin, line); //line changed now!
//say the user entered “Hello World 42!”
std::cout << line << std::endl;
//should print out “Hello World 42!”
```

DEMO

Let’s see what happens when we mix >> and getline 👀

Playground

IMPORTANT: 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.
- TL;DR they don't play nicely

📝 **Note for 106B:** Don't use >> with Stanford libraries, they use getline.

Input File Streams

- Have type `std::ifstream`
- Only receives strings using the `>>` operator
  - Receives strings from a file and converts it to data of any type

Input File Streams

- Have type `std::ifstream`
- Only receives strings using the `>>` operator
  - Receives strings from a file and converts it to data of any type
- Must initialize your own `ifstream` object linked to your file

```
std::ifstream in("out.txt");
// in is now an ifstream that reads from out.txt
string str;
in >> str; // first word in out.txt goes into str
```

`std::cin` is a ***global constant object*** that you get from `#include <iostream>`

To use any other input stream, you must first initialize it!

Questions?

Today

- ~~What are streams?~~
- ~~Output streams~~
- ~~Input streams~~
- String streams!

Stringstreams

Stringstreams

- **What:** A stream that can read from or write to a string object
- **Purpose:** Allows you to perform input/output operations on a string as if it were a stream

```
std::string input = "123";

std::stringstream stream(input);

int number;

stream >> number;

std::cout << number << std::endl;  // Outputs "123"
```

If you only want to read OR write data:

- **Read only:** std::istringstream
  - Give any data type to the istringstream, it'll store it as a string!
- **Write only**: std::ostringstream
  - Make an ostringstream out of a string, read from it word/type by word/type!
- Follows same patterns as the other i/ostreams!

Let’s practice using these new streams!

CODE DEMO

Recap

- Streams convert between data of any type and the string representation of that data

Recap

- Streams convert between data of any type and the string representation of that data
- Streams have an endpoint: console for cin/cout, files for i/o fstreams, string variables for i/o stringstreams where they read in a string from or output a string to.

Recap

- Streams convert between data of any type and the string representation of that data
- Streams have an endpoint: console for cin/cout, files for i/o fstreams, string variables for i/o stringstreams where they read in a string from or output a string to.
- To send data (in string form) to a stream, use `stream_name << data`

Recap

- Streams convert between data of any type and the string representation of that data
- Streams have an endpoint: console for cin/cout, files for i/o fstreams, string variables for i/o stringstreams where they read in a string from or output a string to.
- To send data (in string form) to a stream, use `stream_name << data`
- To extract data from a stream, use `stream_name >> data`, and the stream will try to convert a string to whatever type `data` is

Thanks for coming!

Next time: Containers!