

# Streams

...

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



**masks required**

# Announcements

# Announcements

- Office hours times posted on class website!
  - Haven: Tuesday 4:30 - 5:30pm in 380-380F
  - Sarah: Thursday 4:30 - 5:30pm in 380-380F
  - Note: Sarah's OH are Friday 3 - 4pm on Zoom this week only
- Don't forget to do Assignment Setup!
- Assignment 1 due Sunday, Oct 23rd @ 11:59pm

# Recap

- Everything with a name in your program has a **type**
- **Strong type systems** prevent errors before your code runs!
- **Structs** are a way to bundle a bunch of variables of many types
- **std::pair** is a type of struct that had been defined for you and is in the STL
- So you access it through the **std:: namespace** (std::pair)
- **auto** is a keyword that tells the compiler to deduce the type of a variable, it should be used when the type is obvious or very cumbersome to write out

# A note about STL naming conventions

- **STL** = Standard Template Library
  - Contains TONS of functionality (algorithms, containers, functions, iterators) some of which we will explore in this class
- The **namespace** for the STL is **std**
  - std is the abbreviation for standard
  - IDK why they didn't name the namespace stl
- So to **access elements** from the STL use **std::**

# Today



- Streams!
  - Output streams
  - Input streams
  - File streams and 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 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!
```

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

# Output Streams

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

# Output Streams

- Have type `std::ostream`
- Can only ***send*** data 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?

# Code Demo: ostream

# 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!

`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
- 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

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



↑  
position

```
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



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



position

```
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



position

```
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



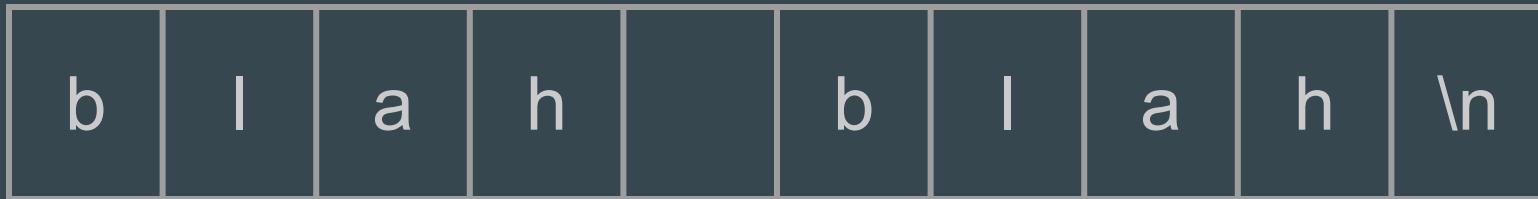
position

```
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;
```

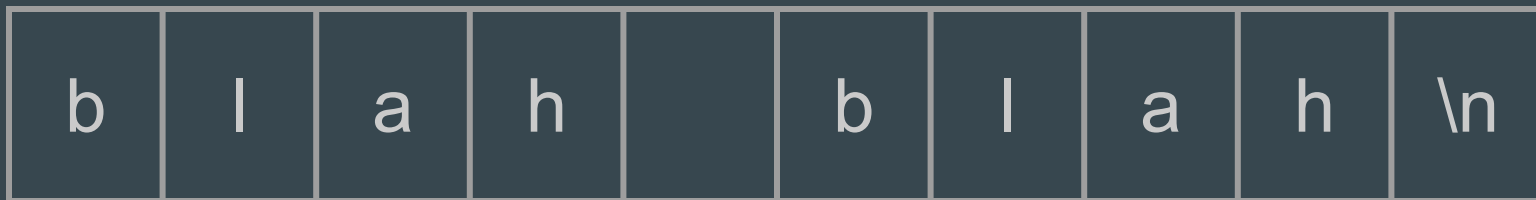
Think of a `std::istream` as a **sequence** of characters



↑  
position

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

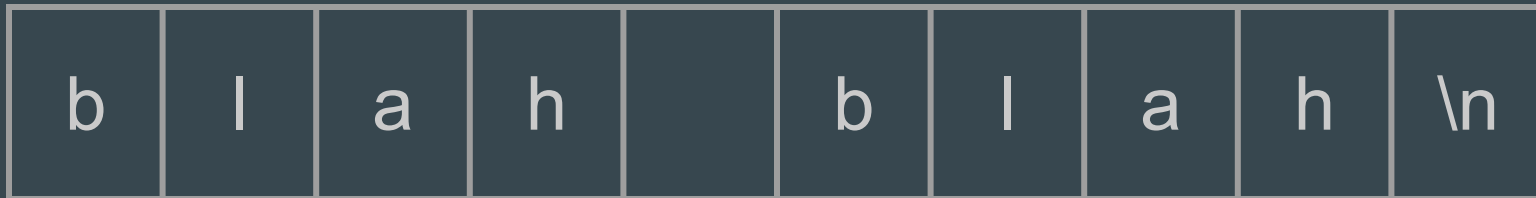
Think of a `std::istream` as a **sequence** of characters



position

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

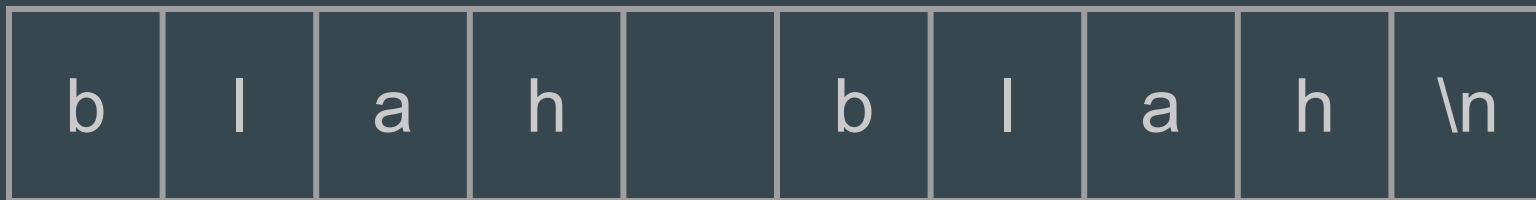
Think of a `std::istream` as a **sequence** of characters



position

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

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

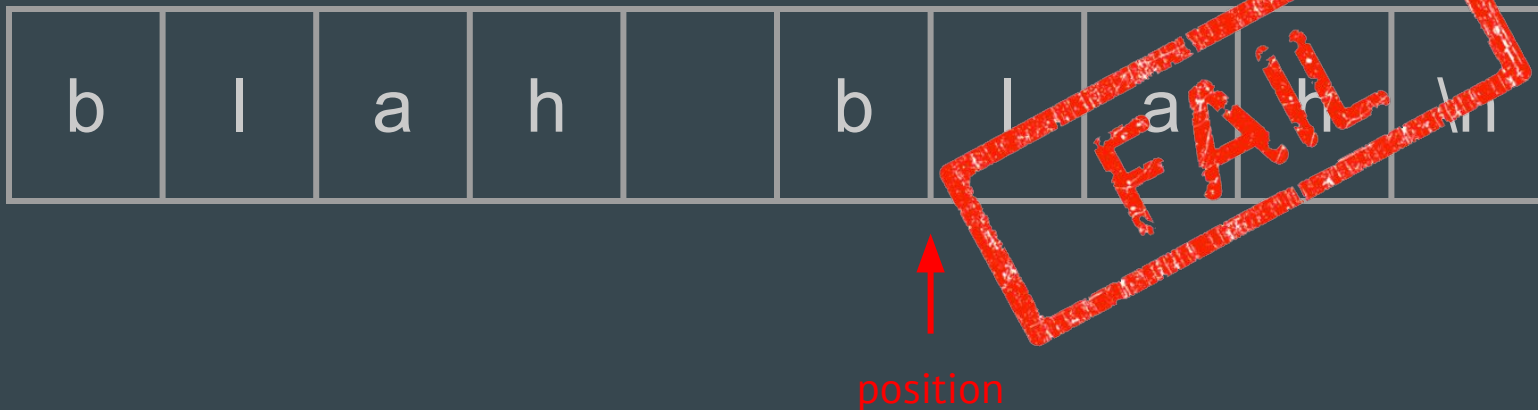


position

```
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;
```

# 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

```
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
```

**str** → **blah**

**x** → **0**

**otherStr** → **NOTHING**

# 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



position

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

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



position

```
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



position

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



Questions?

## std::getline()

- **Signature:** `istream& getline(istream& is, string& str, char delim);`
  - `is` = Stream to read from, `str` = Place where input from stream is stored, `delim` = When to stop reading (`'\n'` if not specified)
- Used to read a string or a line from an input stream

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

1 = true  
0 = false

## - How it works:

- Clears contents in **str**
- Extracts chars from **is** and stores them in **str** until:
  - End of file condition on **is**, sets EOF bit (can be checked using `is.eof()`)
  - Next char in **is** is **delim**, extracts but does not store **delim**
  - **str** max size is reached, sets FAIL bit (can be checked using `is.fail()`)
- If no chars extracted for any reason, FAIL bit set

Reading using >> extracts a single “word” or 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); //now line has changed!  
//say the user entered "Hello World 42!"  
std::cout << line << std::endl;  
//should print out "Hello World 42!"
```

Playground

## 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:** 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>
```

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

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

# Code Demo: istreams

Questions?

# Stringstreams

# Stringstreams

- Input stream: `std::istringstream`
  - Give any data type to the `istringstream`, it'll store it as a string!
- Output stream: `std::ostringstream`
  - Make an `ostringstream` out of a string, read from it word/type by word/type!
- The same as the other `i/o`streams you've seen!

# ostreams

```
string judgementCall(int age, string name,  
                    bool lovesCpp)  
{  
    std::ostringstream formatter;  
    formatter << name << ", age " << age;  
    if(lovesCpp) formatter << ", rocks.";  
    else formatter << " could be better";  
    return formatter.str();  
}
```



# istreamstreams

```
Student reverseJudgementCall(string judgement)
{ //input: "Sarah age 21, rocks"
    std::istream converter;
    string fluff; int age; bool lovesCpp; string name;
    converter >> name;
    converter >> fluff;
    converter >> age;
    converter >> fluff;
    string cool;
    converter >> cool;
    if(cool == "rocks") return Student{name, age, "bliss"};
    else return Student{name, age, "misery"};
} // returns:
```

# istreams

```
Student reverseJudgementCall(string judgement)
{ //input: "Sarah age 21, rocks"
    std::istream converter;
    string fluff; int age; bool lovesCpp; string name;
    converter >> name;
    converter >> fluff;
    converter >> age;
    converter >> fluff;
    string cool;
    converter >> cool;
    if(cool == "rocks") return Student{name, age, "bliss"};
    else return Student{name, age, "misery"};
} // returns: {"Sarah", 21, "bliss"}
```

**Lets write getInteger!**

# 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 streams 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