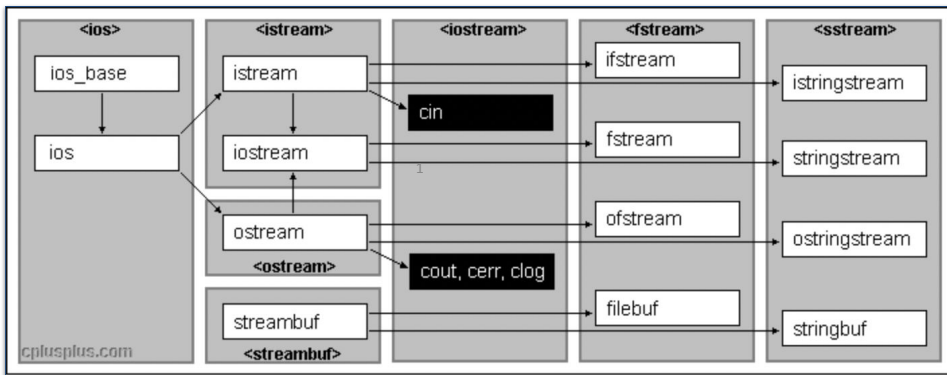


Lecture 4: Streams



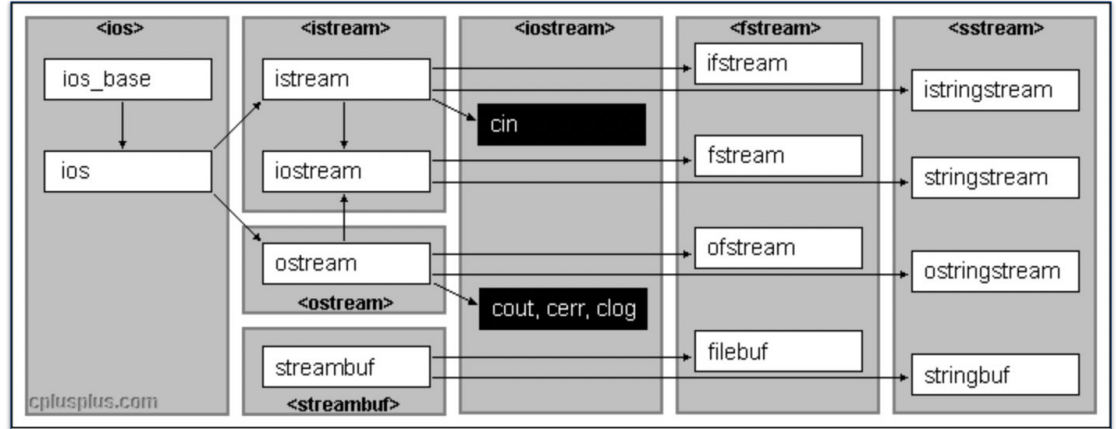
Stanford CS106L, Spring 2026

Preston Seay & Emily Ai



Plan

1. Quick recap
2. What are streams??!!
3. Output streams
4. Input streams
5. stringstream



Slides are available at...

cs106l.stanford.edu

Recap

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's Agenda

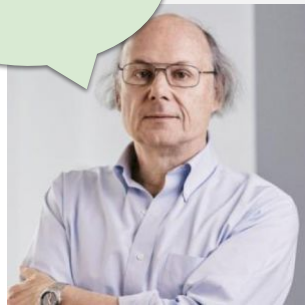
- What are streams?
- Output streams
- Input streams
- `stringstreams`

Why streams?

"Designing and implementing a general **input/output** facility for a programming language is notoriously difficult"

- *Bjarne Stroustrup*

So I did it



Streams

~~"Designing and implementing a general input/output facility for a programming language is notoriously difficult C++"~~

- *a stream* :)



Streams: a general input/output facility for C++

Abstractions:

- hide unnecessary details and expose what is only relevant

Abstractions:

- hide unnecessary details and expose what is only relevant
- provides a **consistent interface**, and in the case of streams the interface is for reading and writing data!

THE BIG IDEA

Streams help us read and write data

What questions do you have?



bjarne_about_to_raise_hand

But what is a **stream**?

You may not know what
a stream is, but chances
are **you probably use
them all the time!**



A familiar stream!

```
std::cout << "Hello, World" << std::endl;
```

A familiar stream!

```
std::cout << "Hello, World" << std::endl;
```



This is a stream

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 << "Emily" << 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 << "Emily" << std::endl;  
// Mix types!  
std::cout << "Emily is " << 19 << 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 << "Emily" << std::endl;  
// Mix types!  
std::cout << "Emily is " << 19 << std::endl;  
// structs?  
Student s = {"Emily", "CA", 19};  
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 << "Emily" << std::endl;  
// Mix types!  
std::cout << "Emily is " << 19 << std::endl;  
// structs?  
Student s = {"Emily", "CA", 19};  
std::cout << s << std::endl;
```

A stream you've used: `cout`

```
// use a stream to print
std::cout << 5 << std::endl;
// and most from the STL
std::cout << "Emily" << std::endl;
// Mix types!
std::cout << "Emily is 5 years old" << std::endl;
// structs?
Student s = {"Emily", "CA", 19};
std::cout << s << std::endl;
```

```
struct Student {
    string name;
    string school;
    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 << "Emily" << std::endl;  
// Mix types!  
std::cout << "Emily is " << 19 << std::endl;  
// structs?  
Student s = {"Emily", "CA", 19};  
std::cout << s << std::endl;
```

24

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 << "Emily" << std::endl;  
// Mix types!  
std::cout << "Emily is " << 19 << std::endl;  
// structs?  
Student s = {"Emily", "CA", 19};
```

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 << "Emily" << std::endl;  
// Mix types!  
std::cout << "Emily is " << 19 << std::endl;  
// structs?  
Student s = {"Emily", "CA", 19};  
std::cout << s.name << s.age << 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 << "Emily" << std::endl;  
// Mix types!  
std::cout << "Emily is " << 19 << std::endl;  
// structs?  
Student s = {"Emily", "CA", 19};  
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 << "Emily" << std::endl;  
// Mix types!  
std::cout << "Emily is " << 19 << 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 << "Emily" << std::endl;  
// Mix types!  
std::cout << "Emily is " << 10 << std::endl;  
// Any primitive type + most STL types work!  
// For other types, you will need to define the  
<< operator yourself
```

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

More examples

```
//create a file called "data.txt"
std::ofstream fout("data.txt");
fout << "I'm writing to this file";

std::ifstream fin("data.txt");
std::string first_word;
//store the first word from the file into fin
fin >> student_input;
```

```
std::cout << "hello CS106L!"

std::string student_input;
std::cin >> student_input;
```

notice the << and >>? That's abstraction at work! We can use a **consistent interface** to work with **input** and **output**



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

Let's practice!

What questions do you have?



bjarne_about_to_raise_hand

By Direction:

By Source or Destination:

By Direction:

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

By Source or Destination:

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:

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:

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')

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')

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')

What questions do you have?



bjarne_about_to_raise_hand

Today's Agenda

- ~~What are streams?~~
- Output streams
- Input streams
- `stringstreams`

Output Streams!

Output Streams

- Have type `std::ostream`
- You 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`
- You 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`
- You can only ***send*** data to file 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`
- You can only **send** data to file 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
```

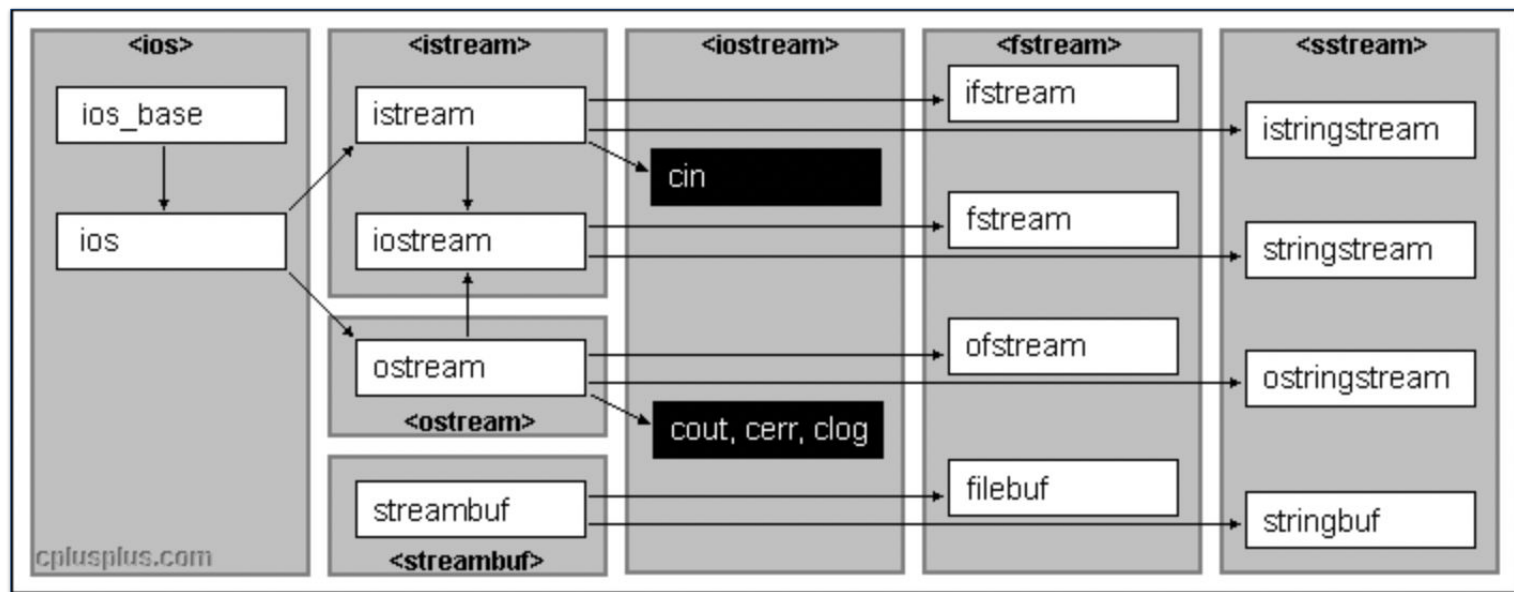
```
out << 5 << std::endl; // out.txt contains 5
```

Let's practice!

`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!**



What questions do you have?



bjarne_about_to_raise_hand

Today's Agenda

- ~~What are streams?~~
- ~~Output streams~~
- Input streams
- `stringstreams`

Input Streams!

Input Streams

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

Input Streams

- Have type `std::istream`
- You 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!**
- **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 File Streams

- Have type `std::ifstream`
- You can only receive strings using the `>>` operator
 - Receives strings from a file and converts it to data of any built-in type

Input File Streams

- Have type `std::ifstream`
- You can only receive strings using the `>>` operator
 - Receives strings from a file and converts it to data of any built-in 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!**

Let's practice!

What questions do you have?



bjarne_about_to_raise_hand

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



position

```
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;
```

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

Streams as conditions

```
string str;  
int x;  
string otherStr;  
//extraction fails  
std::cin >> str >> x >> otherStr;  
if (std::cin) {  
    std::cout << "Success!" << std::endl;  
}
```

Reading until the end of the stream

```
std::ifstream input("numbers.txt");  
  
int value;  
while (input >> value) {  
    input >> value;  
    std::cout << value << std::endl;  
}
```

Important note

```
std::ifstream input("numbers.txt");
```

```
int value;
```

```
while (!input.eof()) {
```

```
    input >> value;
```

```
    std::cout << value << std::endl;
```

```
}
```



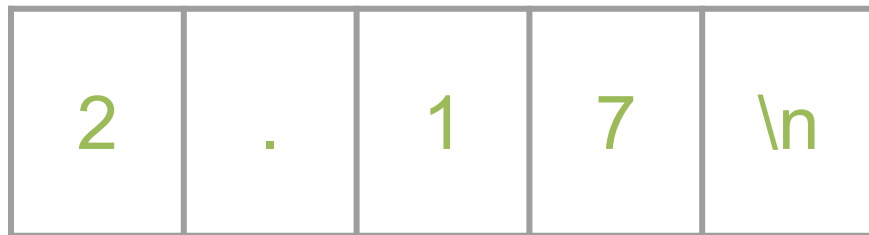
Don't do this!

**Let's look at a quick
demo!**

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?
```

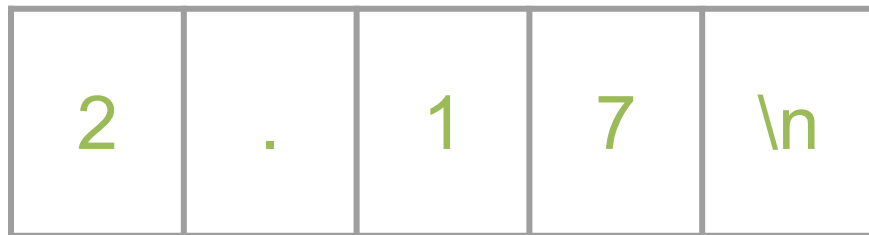
Think of a `std::istream` as a sequence of characters



position

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

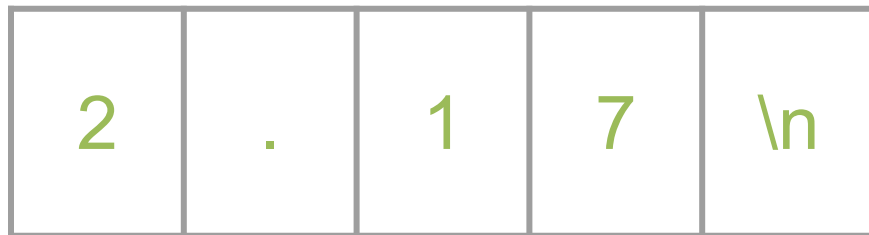
Think of a `std::istream` as a **sequence** of characters



position

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

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



position

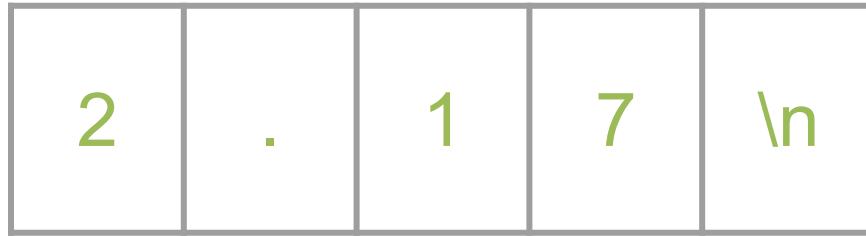
Reads until it finds
something that isn't an int!

```
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
```

What questions do you have?



bjarne_about_to_raise_hand

Introducing... `getline()`!



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`
 - 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`
 - End of file reached
 - Next char in `is` is `delim`
 - `str` out of space, so

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.

- 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**
 - End of file reached
 - Next char in **is** is **delim**
 - **str** out of space, so stop
- If no chars extracted for **str**, return false

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

```
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 practice!

IMPORTANT: Don't mix >> with getline!

- >> reads one value at a time, stopping when it reaches whitespace. The whitespace is not consumed as part of that read.
- **getline** reads all characters until it reaches a delimiter (by default '\n'). It consumes the delimiter but does not store it in the resulting string.
- TL;DR they don't play nicely



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

What questions do you have?



bjarne_about_to_raise_hand

Today's Agenda

- ~~What are streams?~~
- ~~Output streams~~
- ~~Input streams~~
- `stringstreams`

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!

Beauty of abstraction!

- Stringstreams follow the same patterns of console and file streams!

```
std::istringstream input("10 20 30");  
  
int value;  
while (input >> value) {  
    std::cout << value << std::endl;  
}
```

Let's practice!

What questions do you have?



bjarne_about_to_raise_hand

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 stringstream where they read from a string or output to a string

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 stringstream where they read from a string or output to a string
- 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 stringstream where they read from a string or output to a string
- 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

See you all on Tuesday!! :)

Have a great weekend :D