Types and Streams

...

Types make things better...and sometimes harder...but still better >:(

Recap

C++: Basic Syntax + the STL

Basic syntax

- Semicolons at EOL
- Primitive types (ints, doubles etc)
- Basic grammar rules

The STL

- Tons of general functionality
- Built in classes like maps, sets, vectors
- Accessed through the namespace std::

Standard C++: Basic Syntax + std library

Basic s

- Sem
- Prim
  doub
- Basi

The STL

- Tons of general functionality
- Built in classes like maps, sets, vectors
- Accessed through the namespace std::
- Extremely powerful and well-maintained

Namespaces

- MANY things are in the `std::` namespace
  - e.g. **std::cout**, **std::cin**, **std::lower_bound**
- CS 106B always uses the `using namespace std;` declaration, which automatically adds **std::** for you
- We won't (most of the time)
  - it's not good style!

Today

- **Types**
- Brief intro to structs
- Streams

C++ Fundamental Types

```
int val = 5; //32 bits
char ch = 'F'; //8 bits (usually)
float decimalVal1 = 5.0;  //32 bits (usually)
double decimalVal2 = 5.0;  //64 bits (usually)
bool bVal = true;  //1 bit
```

C++ Fundamental Types++

```
#include <string>
int val = 5; //32 bits
char ch = 'F'; //8 bits (usually)
float decimalVal1 = 5.0;  //32 bits (usually)
double decimalVal2 = 5.0;  //64 bits (usually)
bool bVal = true;  //1 bit
std::string str = "Frankie";
```

🧠 Fill in the types!

```
_____ a = "test";
_____ b = 3.2 * 5 - 1;
_____ c = 5 / 2;

_____ d(int foo) { return foo / 2; }
_____ e(double foo) { return foo / 2; }
_____ f(double foo) { return int(foo / 2); }

_____ g(double c) {
    std::cout << c << std::endl;
}
```

🧠 Fill in the types!

```
string a = "test";
double b = 3.2 * 5 - 1;
int    c = 5 / 2;          // int/int → int, what's the value?

_____ d(int foo) { return foo / 2; }
_____ e(double foo) { return foo / 2; }
_____ f(double foo) { return int(foo / 2); }

_____ g(double c) {
    std::cout << c << std::endl;
}
```

🧠 Fill in the types!

```
string a = "test";
double b = 3.2 * 5 - 1;
int    c = 5 / 2;          // int/int → int, what's the value?

int d(int foo) { return foo / 2; }
double e(double foo) { return foo / 2; }
int f(double foo) { return int(foo / 2); }

_____ g(double c) {
    std::cout << c << std::endl;
}
```

🧠 Fill in the types!

```
string a = “test”;
double b = 3.2 * 5 - 1;
int    c = 5 / 2;          // int/int → int, what’s the value?

int d(int foo) { return foo / 2; }
double e(double foo) { return foo / 2; }
int f(double foo) { return int(foo / 2); }

void g(double c) {
    std::cout << c << std::endl;
}
```

C++ is a statically typed language

Definition

statically typed: everything with a name (variables, functions, etc) is given a type before runtime

C++ Types in Action

```
int a = 3;
string b = "test";

char func(string c) {
    // do something
}

b = "test two";

func(b);

// don't need to declare type after initialization
```

Dynamic vs Static typing: Python vs C++

Python

```python
a = 3
b = "test”

def func(c):
    # do something
```

C++

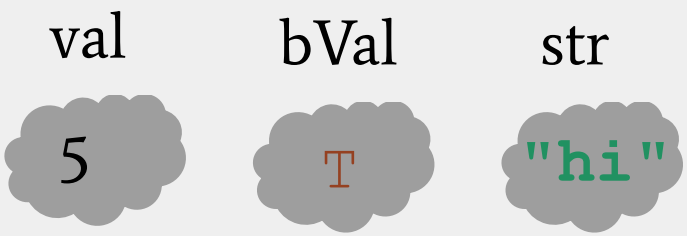
```cpp
int a = 3;
string b = "test”;

char func(string c) {
    // do something
}
```

Dynamic vs Static typing: Python vs C++

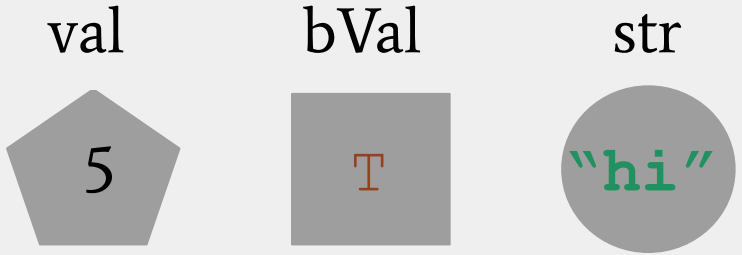
Python

```
val = 5;
bVal = true;
str = "hi";
```



C++

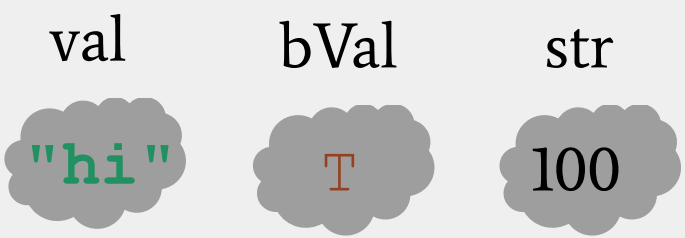
```
int val = 5;
bool bVal = true;
string str = "hi";
```



Dynamic vs Static typing: Python vs C++

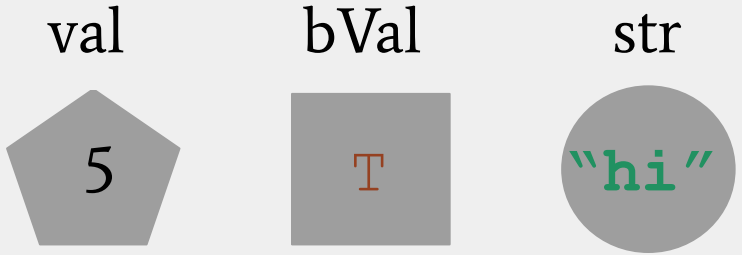
Python

```
val = 5
bVal = true
str = "hi"
val = "hi"
str = 100
```



C++

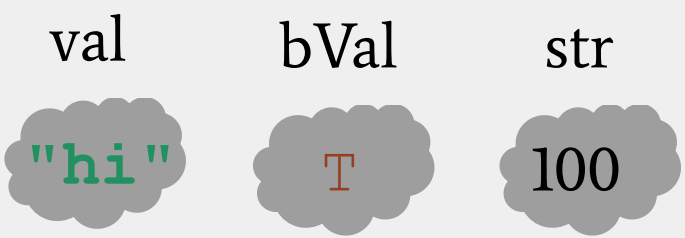
```
int val = 5;
bool bVal = true;
string str = "hi";
```



Dynamic vs Static typing: Python vs C++

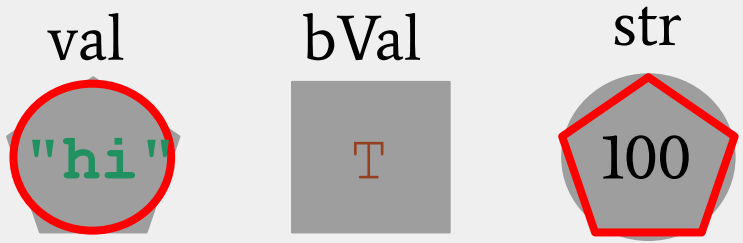
Python

```
val = 5
bVal = true
str = "hi"
val = "hi"
str = 100
```



C++

```
int val = 5;
bool bVal = true;
string str = "hi";
val = "hi";
str = 100;
```



Dynamic vs Static typing: Python vs C++

Python

```
def div_3(x):
    return x / 3

div_3("hello")
```

C++

```
int div_3(int x){
    return x / 3;
}

div_3("hello")
```

Dynamic vs Static typing: Python vs C++

Python

```
def div_3(x):
    return x / 3

div_3("hello")
```

//CRASH during runtime, can't divide a string

C++

```
int div_3(int x){
    return x / 3;
}

div_3("hello")
```

Dynamic vs Static typing: Python vs C++

Python

```
def div_3(x):
    return x / 3

div_3("hello")
```

//CRASH during runtime, can't divide a string

C++

```
int div_3(int x){
    return x / 3;
}

div_3("hello")
```

//Compile error: this code will never run

Dynamic vs Static typing: Python vs C++

Python

```python
def add_3(x):
    return x + 3

add_3("10")
```

C++

```cpp
int add_3(int x){
    return x + 3;
}

add_3("10");
```

Dynamic vs Static typing: Python vs C++

Python

```
def add_3(x):
    return x + 3
add_3("10")

//returns “103”
```

C++

```
int add_3(int x){
    return x + 3;
}
add_3("10");
```

Dynamic vs Static typing: Python vs C++

Python

```
def add_3(x):
    return x + 3

add_3("10")


//returns “103”
```

C++

```
int add_3(int x){
    return x + 3;
}

add_3("10");
//Compile error: “10” is a string! This code wont run
```

static typing helps us to prevent errors **before our code runs**

C++ to Python, probably

Static Types + Functions

Python

```
def div_3(x)
```

```
div_3: __ -> ??
```

C++

```
int div_3(int x)
```

```
div_3: int -> int
```

Static Types + Functions

What are the types of the following functions?

```
int add(int a, int b);
   int, int -> int
string helloworld();

_____________________
string echo(string phrase);

_____________________
double divide(int a, int b);

_____________________
```

Static Types + Functions

What are the types of the following functions?

```
int add(int a, int b);
   int, int -> int
string helloworld();
   void -> string
string echo(string phrase);
   string -> string
double divide(int a, int b);
   int, int -> double
```

Overloading

- What if we want two versions of a function for two different types?
- Example: int division vs double division

Overloading

Define two functions with the same name but different types

```
int half(int x) {          // (1)
    return x / 2; // typecast: int → double
}

double half(double x) {    // (2)
    return x / 2;
}
half(3)           // uses version (1), returns ?
half(3.0)     // uses version (2), returns ?
```

Overloading

Define two functions with the same name but different parameters

```
int half(int x) {          // (1)
    return x / 2; // typecast: int → double
}

double half(double x) {    // (2)
    return x / 2;
}
func(3)           // uses version (1), returns 1
func(3.0)     // uses version (2), returns 1.5
```

Today

- ~~Types~~
- **Brief intro to structs**
- Streams

Definition

struct: a group of named variables *each with their own type.* A way to bundle different types together

Structs in Code

```
struct Student {
   string name; // these are called fields
   string state; // separate these by semicolons
   int age;
};

Student s;
s.name = "Frankie";
s.state = "MN";
s.age = 21; // use . to access fields
```

Use structs to pass around grouped information

```
Student s;
s.name = "Frankie";
s.state = "MN";
s.age = 21; // use . to access fields

void printStudentInfo(Student student) {
  cout << s.name << " from " << s.state;
  cout << " (" << s.age ")" << endl;
}
```

Use structs to return grouped information

```
Student randomStudentFrom(std::string state) {
  Student s;
  s.name = "Frankie";//random = always Frankie
  s.state = state;
  s.age = std::randint(0, 100);
  return s;
}

Student foundStudent = randomStudentFrom("MN");
cout << foundStudent.name << endl; // Frankie
```

Abbreviated Syntax to Initialize a struct

```
Student s;
s.name = "Frankie";
s.state = "MN";
s.age = 21;

//is the same as ...
```

Abbreviated Syntax to Initialize a struct

```
Student s;
s.name = "Frankie";
s.state = "MN";
s.age = 21;

//is the same as ...
Student s = {"Frankie", "MN", 21};
```

Questions?

Today

- ~~Types~~
- ~~Brief intro to structs~~
- **Streams**

Definition

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

A stream you've used: cout

```
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Frankie" << std::endl;
```

A stream you've used: cout

```
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Frankie" << std::endl;
// Mix types!
std::cout << "Frankie is " << 21 << std::endl;
```

A stream you've used: cout

```
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Frankie" << std::endl;
// Mix types!
std::cout << "Frankie is " << 21 << std::endl;
// structs?
Student s = {"Frankie", "MN", 21};
std::cout << s << std::endl;
```

A stream you've used: cout

```
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Frankie" << std::endl;
// Mix types!
std::cout << "Frankie is " << 21 << std::endl;
// structs?
Student s = {"Frankie", "MN", 21};
std::cout << s << std::endl;
```

A stream you've used: cout

```
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Frankie" << std::endl;
// Mix types!
std::cout << "Frankie is " << 21 << std::endl;
// structs?
Student s = {"Frankie", "MN", 21};
std::cout << s.name << s.age << std::endl;
```

A stream you've used: cout

```
std::cout << 5 << std::endl; // prints 5
// use a stream to print any primitive type!
std::cout << "Frankie" << std::endl;
// Mix types!
std::cout << "Frankie 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 receive 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 receive 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", std::ofstream::out);
// 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!

Code Demo: ostreams

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?
```

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

Intput Streams

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

Intput Streams

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

```
int x;
string str;
std::cin >> x >> str;
//reads exactly one int then 1 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

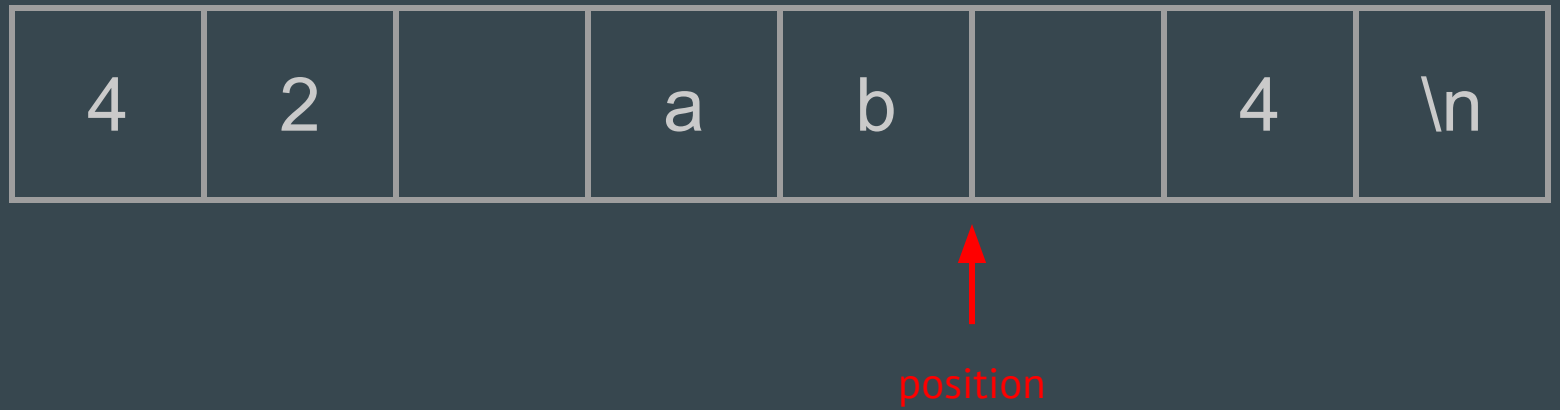
position

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

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

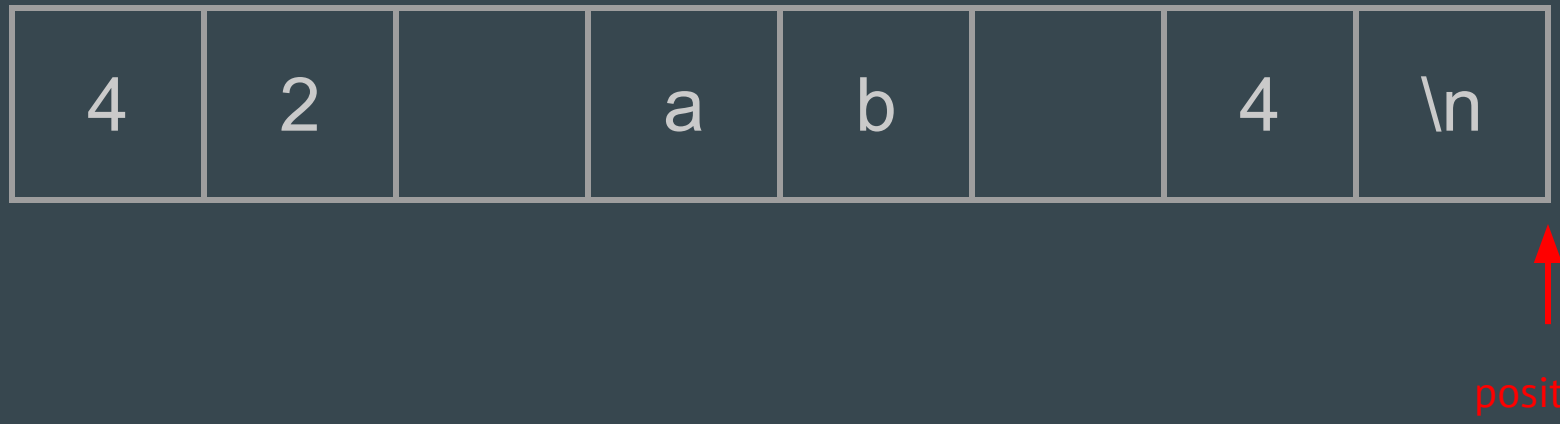

```
int x; string y; int z;
cin >> x; //42 put into x
cin >> y;
cin >> z;
```

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;
cin >> z; //4 put into z
```

Output Streams: When things go wrong

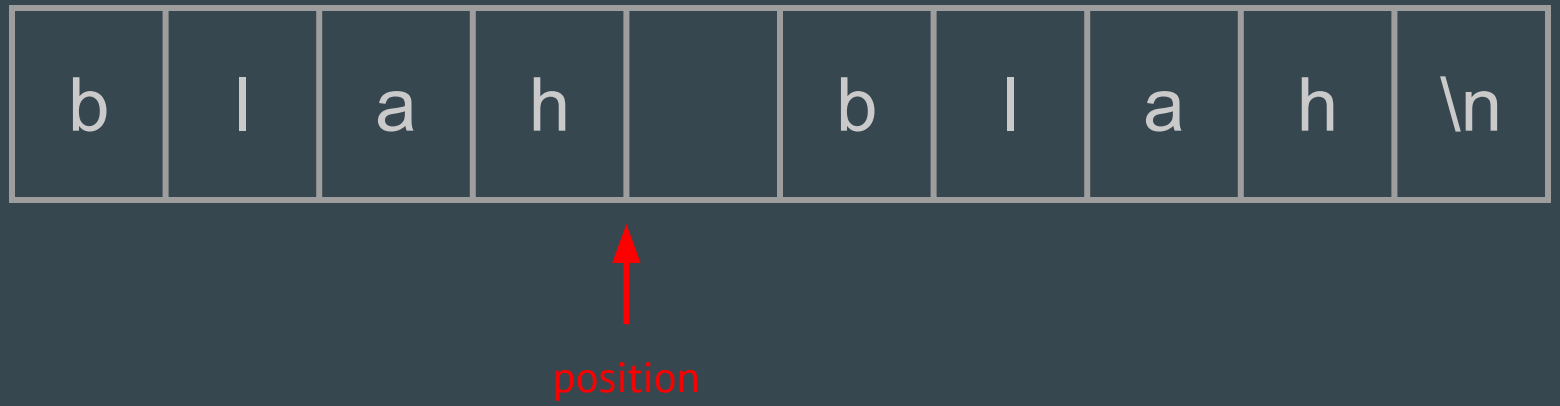
```
string str;
int x;
std::cin >> str >> x;
//what happens if input is blah blah?
std::cout << str << x;
```

Playground (istreams.cpp)

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

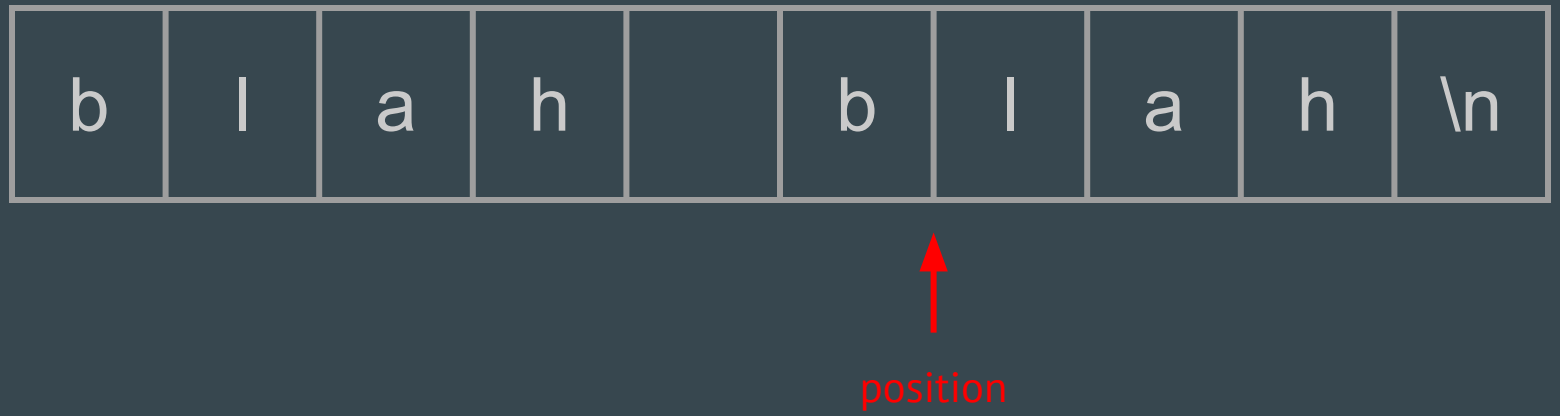

```
string str; int x;
std::cin >> str >> x;
```

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



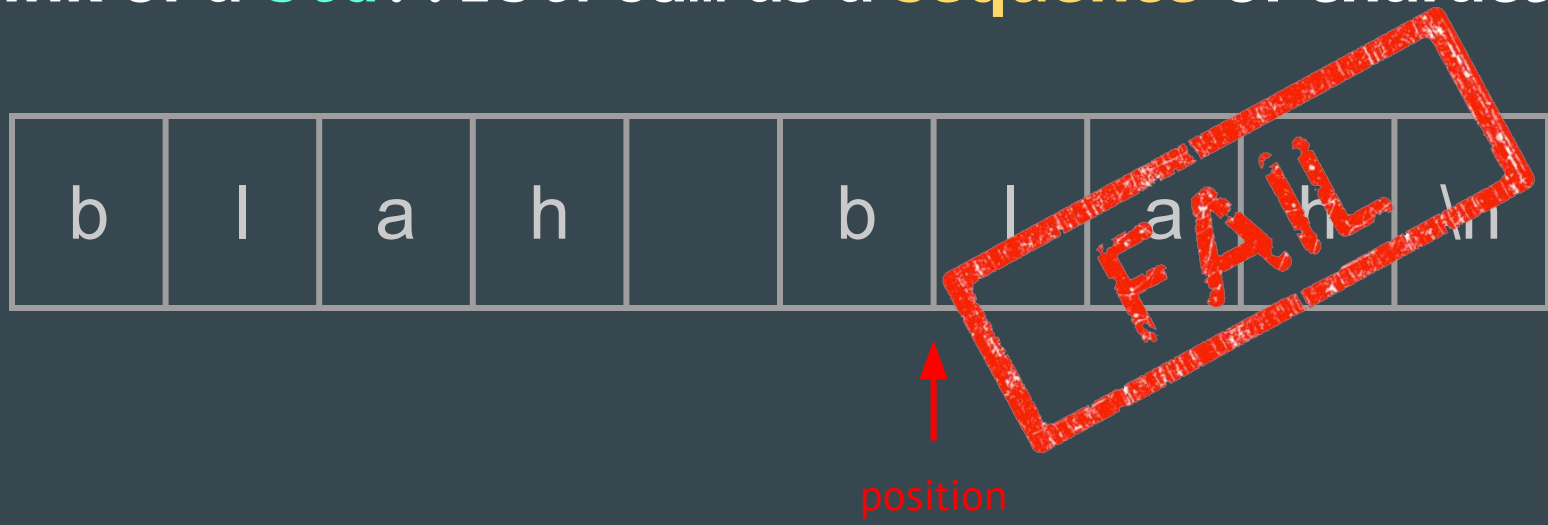

```
string str; int x;
std::cin >> str >> x;
```

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;
```

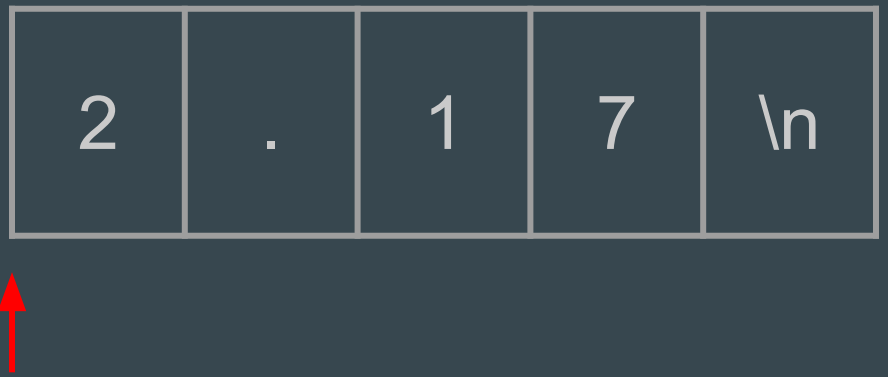
Output Streams: When things go wrong

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

Output 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
```

`std::cin` is dangerous to use on its own!

Reading using >> extracts a single "word" or type

including for strings

To read a whole line, use

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

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 send data using the `>>` operator
  - Receives data of any type into and converts it into a string to send to the **file stream**

Input File Streams

- Have type `std::ifstream`
- Only send data using the `>>` operator
  - Receives data of any type into and converts it into a string to send to the **file stream**

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

```
std::ifstream in("out.txt", std::ifstream::in);
// 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

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/ostreams you've seen!

ostringstreams

```
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();
}
```

istringstreams

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

Lets write getInteger!