Today

- ~~Types~~
- **Intro to structs**
- Sneak peek at 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

```cpp
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?

Definition

`std::pair`: An STL built-in struct with two fields *of any type*

`std::pair`

- **`std::pair`** is a ***template:*** You specify the types of the fields inside <> for each pair object you make
- The fields in **`std::pair`**s are named **`first`** and **`second`**

```
std::pair<int, string> numSuffix = {1,"st"};

cout << numSuffix.first << numSuffix.second;
//prints 1st
```

`std::pair`

- `std::pair` is a ***template:*** You specify the types of the fields inside <> for each pair object you make
- The fields in `std::pairs` are named **`first`** and **`second`**

```
struct Pair {
  fill_in_type first;
  fill_in_type second;
};
```

Use `std::pair` to return success + result

```
std::pair<bool, Student> lookupStudent(string name) {
 Student blank;
 if (found(name)) return std::make_pair(false, blank);
 Student result = getStudentWithName(name);
 return std::make_pair(true, result);
}
std::pair<bool, Student> output = lookupStudent("Keith");
```

Use `std::pair` to return success + result

```
std::pair<bool, Student> lookupStudent(string name) {
 Student blank;
 if (notFound(name)) return std::make_pair(false, blank);
 Student result = getStudentWithName(name);
 return std::make_pair(true, result);
}
 std::pair<bool, Student> output = lookupStudent("Keith");
```

To avoid specifying the types of a pair, use **`std::make_pair(field1, field2)`**

Questions?

Aside: Type Deduction with `auto`

Definition

`auto`: Keyword used in lieu of type when declaring a variable, tells the compiler to deduce the type.

Type Deduction using `auto`

```
// What types are these?
auto a = 3;
auto b = 4.3;
auto c = 'x';
auto d = "Hello";
auto e = std::make_pair(3, "Hello");
```

📝 `auto` **does not mean that the variable doesn't have a type.**

It means that the type is **deduced** by the compiler.

Type Deduction using auto

```
// What types are these?
auto a = 3;
auto b = 4.3;
auto c = 'x';
auto d = "Hello";
auto e = std::make_pair(3, "Hello");
```

Answers: int, double, char, char* (a C string), std::pair<int, char*>

📝 `auto` **does not mean that the variable doesn't have a type.**

It means that the type is **deduced** by the compiler.

‼️ `auto` **does not mean that the variable doesn’t have a type.**

It means that the type is **deduced** by the compiler.

Streams

•••

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

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 ***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!

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*** strings using the `>>` operator
  - ***Receives*** a string from the stream and converts it to data

Input Streams

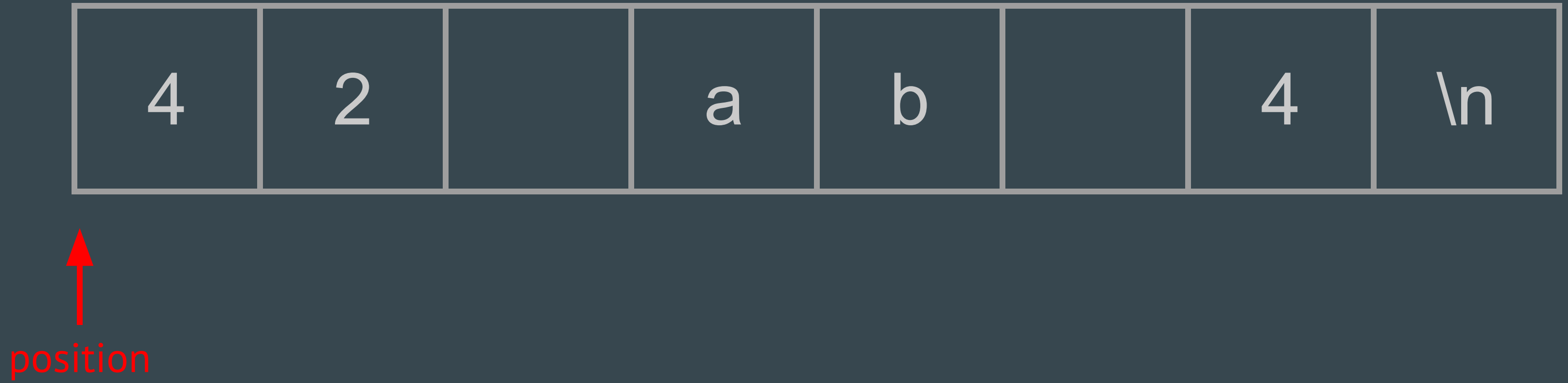
- Have type `std::istream`
- Can only ***receive*** strings using the `>>` operator
    - ***Receives*** a string from the stream and converts it to data
- `std::cin` is the input stream that gets input from the console

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

Nitty Gritty Details: `std::cin`

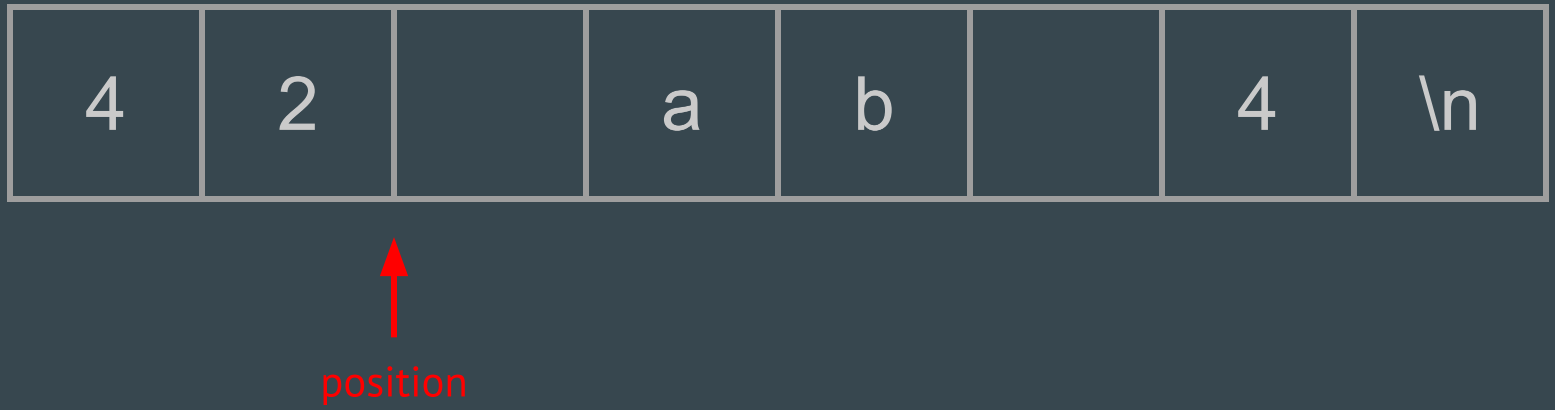
- First call to `std::cin >>` creates a command line prompt that allows the user to type until they hit enter
- Each `>>` ONLY reads until the next *whitespace*
  - Whitespace = tab, space, newline
- Everything after the first whitespace gets saved and used the next time `std::cin >>` is called
  - The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



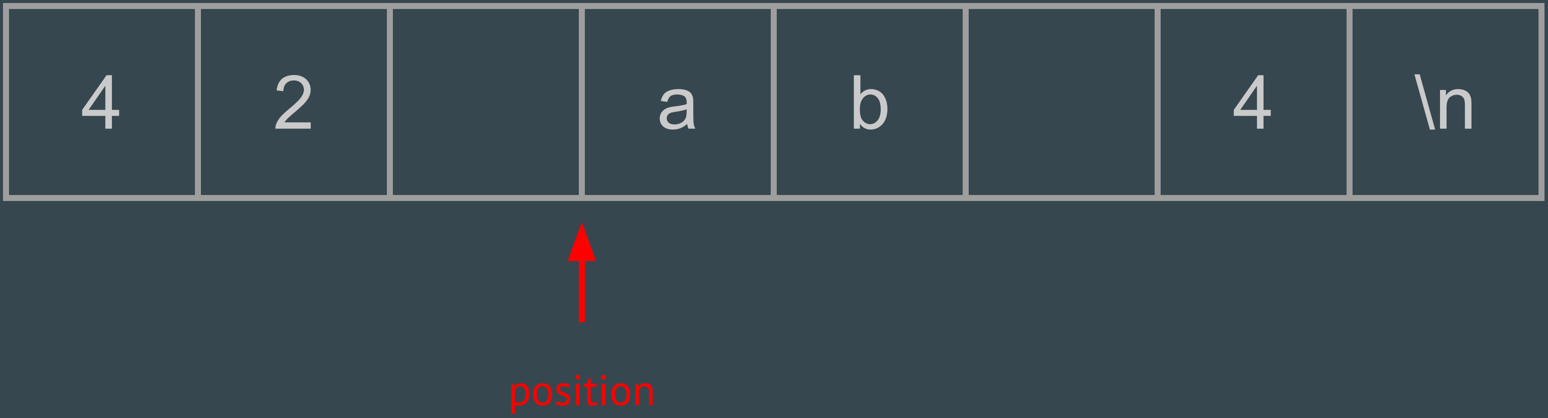

```
int x; string y; int z;
cin >> x;
cin >> y;
cin >> z;
```

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



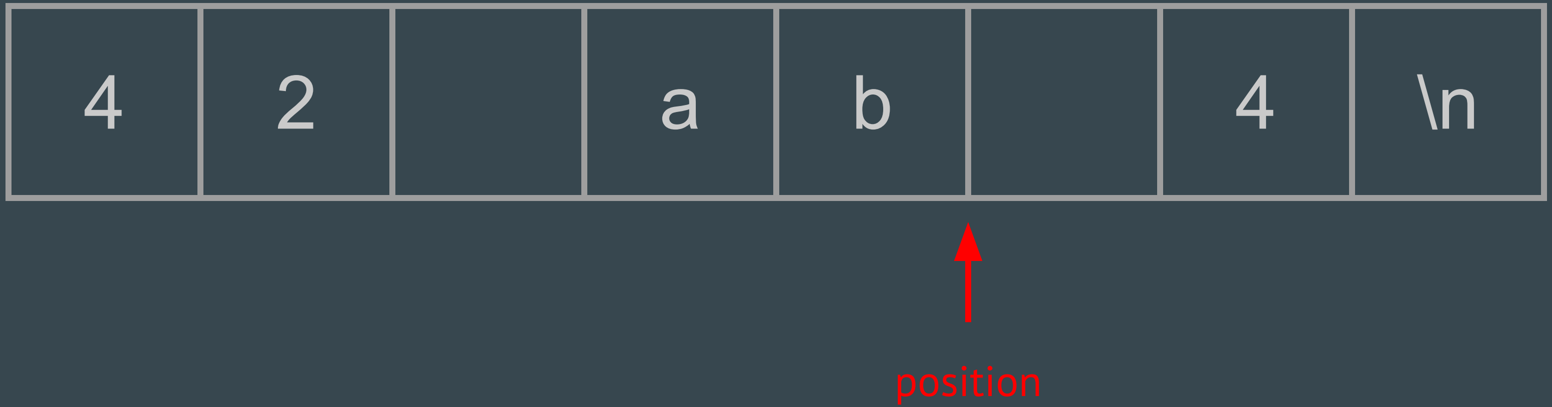

```
int x; string y; int z;
cin >> x; //42 put into x
cin >> y;
cin >> z;
```

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



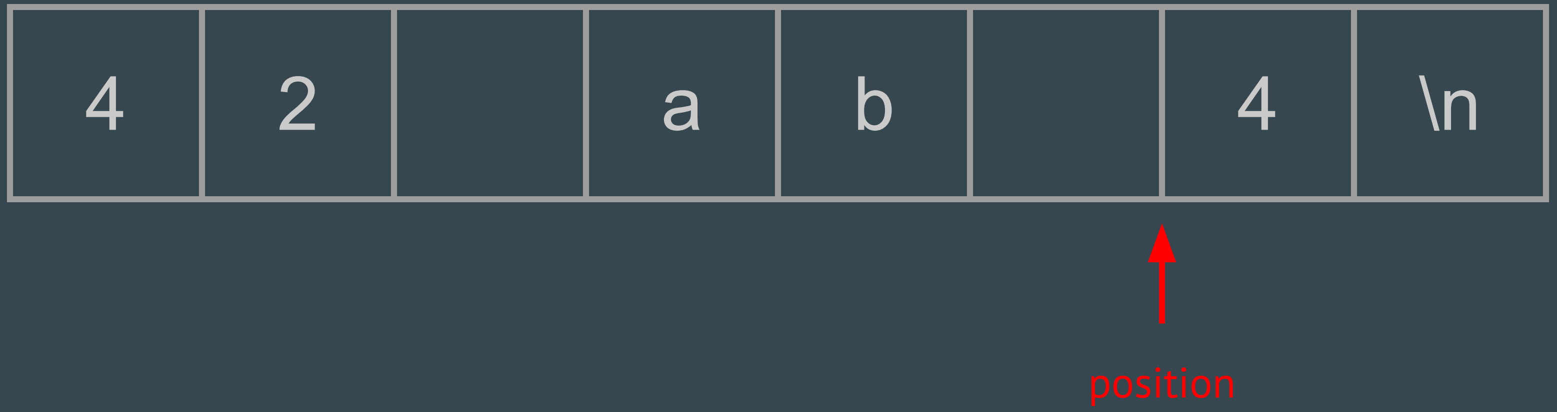

```
int x; string y; int z;
cin >> x; //42 put into x
cin >> y;
cin >> z;
```

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



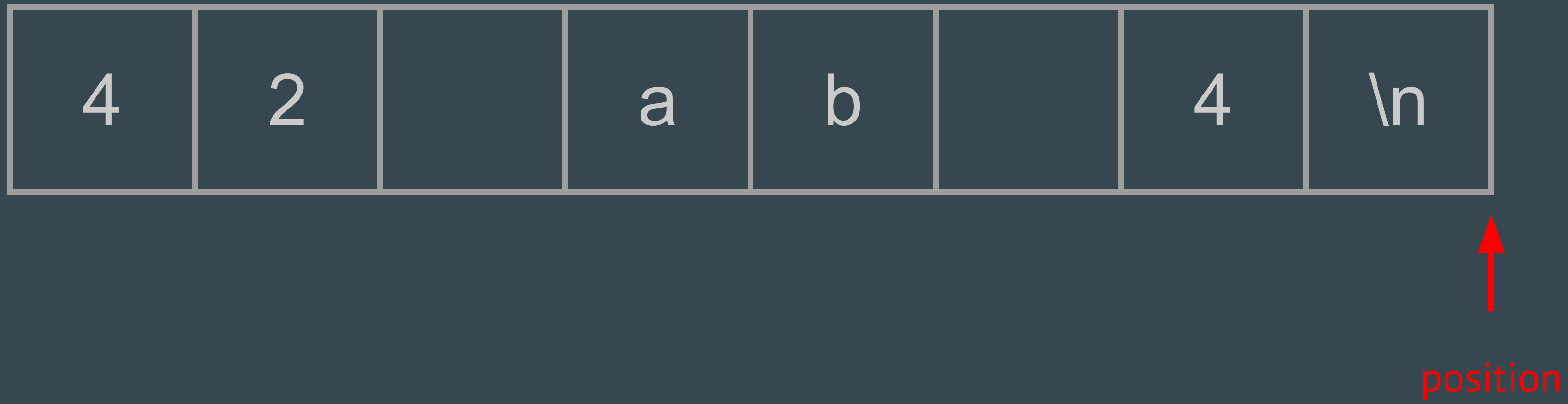

```
int x; string y; int z;
cin >> x;
cin >> y; //ab put into y
cin >> z;
```

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;
```

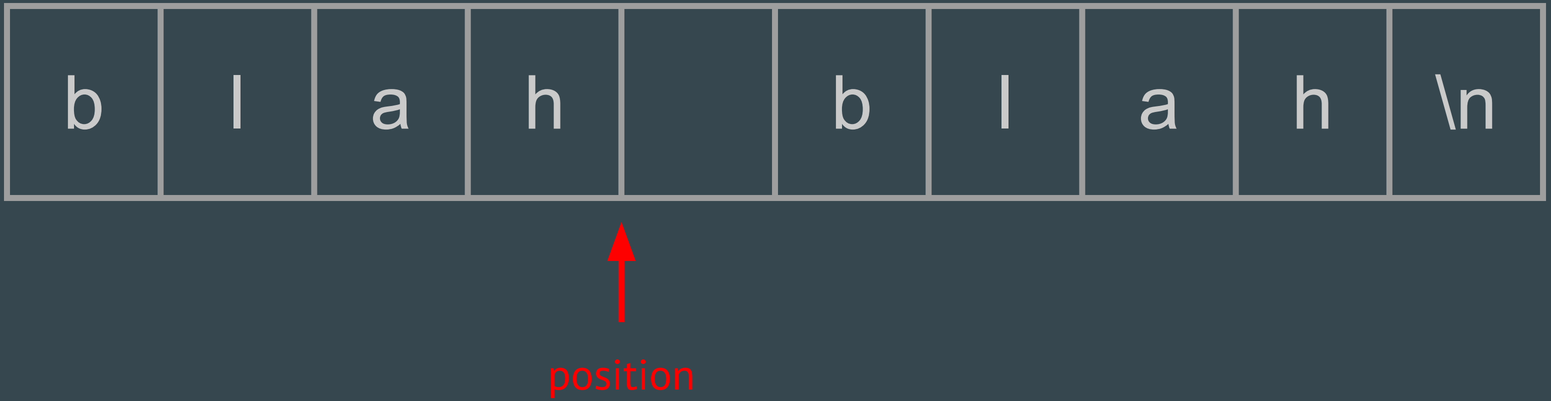
Think of a `std::istream` as a **sequence** of characters

b	l	a	h		b	l	a	h	\n

position

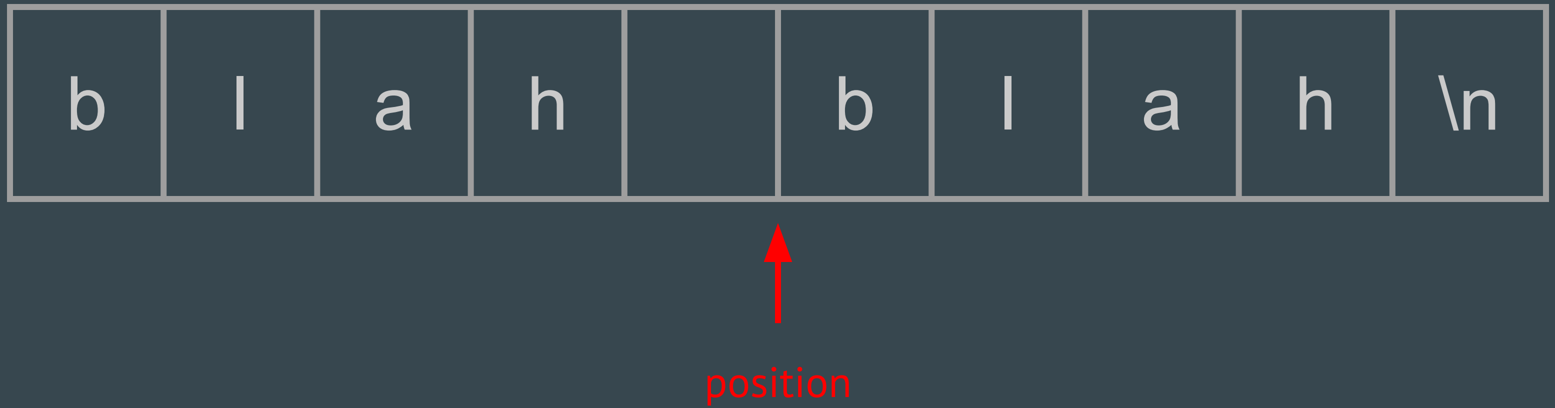
```
string str; int x;
std::cin >> str >> x;
```

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



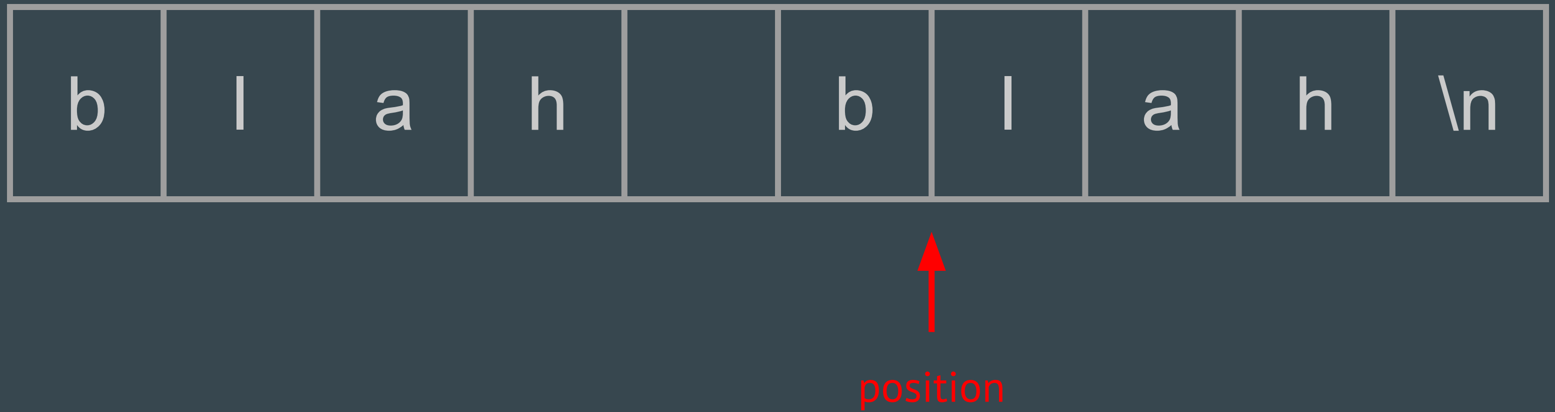

```
string str; int x;
std::cin >> str >> x;
```

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



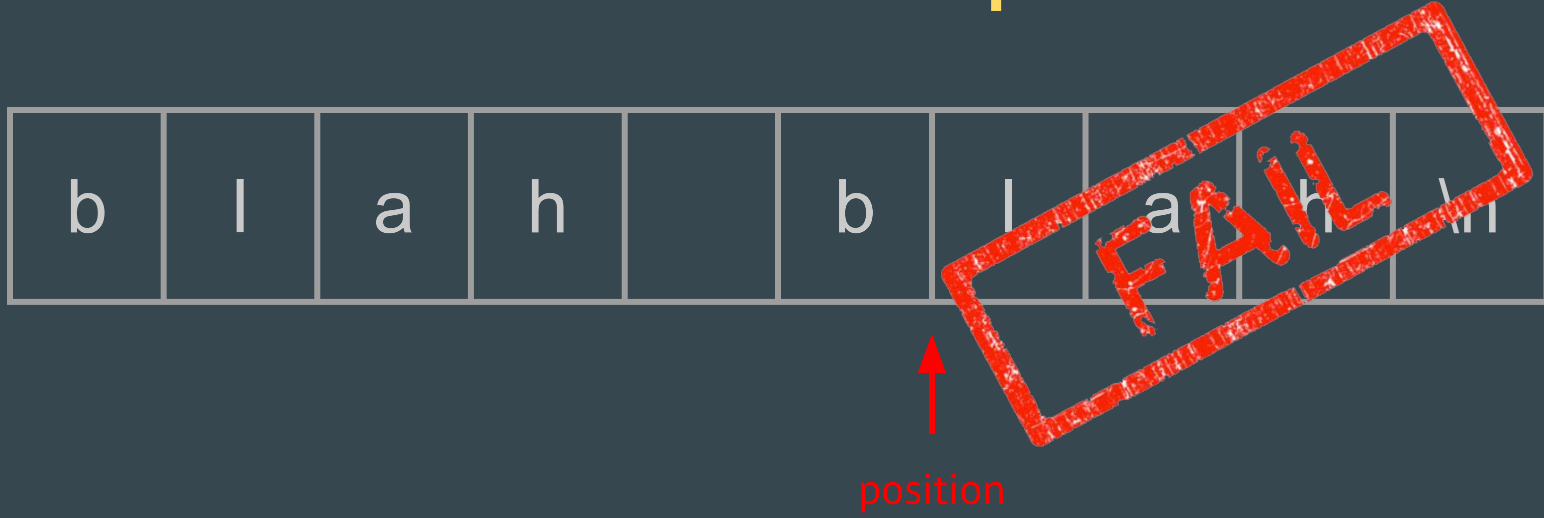

```
string str; int x;
std::cin >> str >> x;
```

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;
```

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;
//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

2	.	1	7	\n

position

```
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

2	.	1	7	\n

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
```

Playground (istreams.cpp)

`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);
```

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!"
```

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 `ofstream` 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

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)
{ //input: “Frankie age 22, rocks”
    std::istringstream 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:
```

istringstreams

```
Student reverseJudgementCall(string judgement)
{ //input: "Frankie age 22, rocks"
    std::istringstream 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: {"Frankie", 22, "bliss"}
```

Lets write getInteger!

Initialization & References

...

And streams and structs ... :)

Today

- ~~Streams recap~~
- Initialization
- References

Definition

Initialization: How we provide initial values to variables

Recall: Two ways to initialize a struct

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

Multiple ways to initialize a pair...

```
std::pair<int, string> numSuffix1 = {1,"st"};

std::pair<int, string> numSuffix2;

numSuffix2.first = 2;

numSuffix2.second = "nd";

std::pair<int, string> numSuffix2 =
                                        std::make_pair(3, "rd");
```

Initialization of vectors

```
std::vector<int> vec1(3,5);
// makes {5, 5, 5}, not {3, 5}!
std::vector<int> vec2;
vec2 = {3,5};
// initialize vec2 to {3, 5} after its declared
```

Definition

Uniform initialization: curly bracket initialization. Available for all types, immediate initialization on declaration!

Uniform Initialization

```
std::vector<int> vec{1,3,5};

std::pair<int, string> numSuffix1{1,"st"};

Student s{"Frankie", "MN", 21};

// less common/nice for primitive types, but possible!

int x{5};

string f{"Frankie"};
```

Careful with Vector initialization!

```
std::vector<int> vec1(3,5);
// makes {5, 5, 5}, not {3, 5}!
//uses a std::initializer_list (more later)
std::vector<int> vec2{3,5};
// makes {3, 5}
```

TLDR: use uniform initialization to initialize every field of your non-primitive typed variables - but be careful not to use vec(n, k)!

When should we use auto?

Quadratic: Typing these types out is a pain...

```
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    std::pair<bool, std::pair<double, double>> result =
                                               quadratic(a, b, c);
    bool found = result.first;
    if (found) {
        std::pair<double, double> solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

Quadratic: Typing these types out is a pain...

```
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    bool found = result.first;
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

Don’t overuse `auto`

Typing these types out is a pain...

```
int main() {
    auto a, b, c;
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    auto found = result.first;
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

Typing these types out is a pain...

```
int main() {
    auto a, b, c; //compile error!
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    auto found = result.first;
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << “No solutions found!” << endl;
    }
}
```

Typing these types out is a pain...

```
int main() {
    int a, b, c;
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    auto found = result.first; //code less clear :/
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

Don’t overuse `auto`

…but use it to reduce long type names

Structured Binding

Structured binding lets you initialize directly from the contents of a struct

Before

```
auto p =
    std::make_pair("s", 5);
string a = s.first;
int b = s.second;
```

After

```
auto p =
    std::make_pair("s", 5);
auto [a, b] = p;
// a is string, b is int
// auto [a, b] =
          std::make_pair(...);
```

📝 This works for regular structs, too. Also, no nested structured binding.

A better way to use `quadratic`

```
int main() {
    auto a, b, c;
    std::cin >> a >> b >> c;
    auto result = quadratic(a, b, c);
    auto found = result.first;
    if (found) {
        auto solutions = result.second;
        std::cout << solutions.first << solutions.second << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

A better way to use `quadratic`

```
int main() {
    auto a, b, c;
    std::cin >> a >> b >> c;
    auto [found, solutions] = quadratic(a, b, c);
    if (found) {
        auto [x1, x2] = solutions;
        std::cout << x1 << " " << x2 << endl;
    } else {
        std::cout << "No solutions found!" << endl;
    }
}
```

📝 This is better is because it's *semantically clearer*: variables have clear names.

Today

- ~~Streams recap~~
- ~~Initialization~~
- **References**

Definition

Reference: An alias (another name) for a named variable

References in 106B

```
void changeX(int& x){ //changes to x will persist
    x = 0;
}
void keepX(int x){
    x = 0;
}

int a = 100;
int b = 100;

changeX(a); //a becomes a reference to x
keepX(b);   //b becomes a copy of x

cout << a << endl; //0
cout << b << endl; //100
```

References in 106L: References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl;
cout << copy << endl;
cout << ref << endl;
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;

original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;
cout << ref << endl;
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;     // {1, 2, 4}
cout << ref << endl;
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;     // {1, 2, 4}
cout << ref << endl;      // {1, 2, 3, 5}
```

References to variables

```
vector<int> original{1, 2};
vector<int> copy = original;
vector<int>& ref = original;
original.push_back(3);
copy.push_back(4);
ref.push_back(5);

cout << original << endl; // {1, 2, 3, 5}
cout << copy << endl;     // {1, 2, 4}
cout << ref << endl;      // {1, 2, 3, 5}
```

"=" automatically makes a copy! Must use & to avoid this.

Code demo: References bugs

The classic reference-copy bug:

```
void shift(vector<std::pair<int, int>>& nums) {
    for (size_t i = 0; i < nums.size(); ++i) {
        auto [num1, num2] = nums[i];
        num1++;
        num2++;
    }
}
```

The classic reference-copy bug:

```
void shift(vector<std::pair<int, int>>& nums) {
    for (size_t i = 0; i < nums.size(); ++i) {
        auto [num1, num2] = nums[i];
        num1++;
        num2++;
    }
}
```

This creates a copy of the course

This is updating that same copy!

The classic reference-copy bug:

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto [num1, num2]: nums) {
        num1++;
        num2++;
    }
}
```

The classic reference-copy bug:

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto [num1, num2]: nums) {
        num1++;
        num2++;
    }
}
```

This is updating that same copy!

This creates a copy of the course

The classic reference-copy bug, fixed:

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}
```

The classic reference-rvalue error

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}

shift({{1, 1}});
```

The classic reference-rvalue error

```
void shift(vector<std::pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}

shift({{1, 1}});
// {{1, 1}} is an rvalue, it can’t be referenced
```

Definition: l-values vs r-values

- l-values can appear on the **left** or **right** of an =
- `x` is an l-value

```
int x = 3;
int y = x;
```

l-values have names

l-values are **not temporary**

Definition: l-values vs r-values

- **l-values** can appear on the **left** or **right** of an =
- `x` is an **l-value**

```
int x = 3;
int y = x;
```

l-values have names

l-values are **not temporary**

- **r-values** can ONLY appear on the **right** of an =
- `3` is an **r-value**

```
int x = 3;
int y = x;
```

r-values don't have names

r-values are **temporary**

The classic reference-rvalue error, fixed

```
void shift(vector<pair<int, int>>& nums) {
    for (auto& [num1, num2]: nums) {
        num1++;
        num2++;
    }
}
auto my_nums = {{1, 1}};
shift(my_nums);
```

Code demo: References errors

BONUS: Const and Const References

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);
c_vec.push_back(3);
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);     // OKAY
c_vec.push_back(3);
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);     // OKAY
c_vec.push_back(3);   // BAD - const
ref.push_back(3);
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);    // OKAY
c_vec.push_back(3);  // BAD - const
ref.push_back(3);   // OKAY
c_ref.push_back(3);
```

`const` indicates a variable can't be modified!

`const` variables can be references or not!

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};  // a const variable
std::vector<int>& ref = vec;         // a regular reference
const std::vector<int>& c_ref = vec;  // a const reference

vec.push_back(3);     // OKAY
c_vec.push_back(3);   // BAD - const
ref.push_back(3);    // OKAY
c_ref.push_back(3); // BAD - const
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// BAD - can't declare non-const ref to const vector
std::vector<int>& bad_ref = c_vec;
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// fixed
const std::vector<int>& bad_ref = c_vec;
```

Can’t declare non-const reference to const variable!

```
const std::vector<int> c_vec{7, 8};  // a const variable

// fixed
const std::vector<int>& bad_ref = c_vec;

// BAD - Can't declare a non-const reference as equal
// to a const reference!
std::vector<int>& ref = c_ref;
```

`const` & subtleties

```
std::vector<int> vec{1, 2, 3};
const std::vector<int> c_vec{7, 8};

std::vector<int>& ref = vec;
const std::vector<int>& c_ref = vec;

auto copy = c_ref;              // a non-const copy
const auto copy = c_ref;        // a const copy
auto& a_ref = ref;              // a non-const reference
const auto& c_aref = ref;       // a const reference
```

Remember: C++, by default, makes copies when we do variable assignment! We need to use & if we need references instead.

Classes

•••

How to make your own custom types!

Today

- ~~Recap: Containers + Iterators~~
- **Classes Introduction**
- Template Classes (intro)

CS 106B covers the barebones of C++ classes… we'll be covering the rest

•••

template classes • const-correctness • operator overloading • special member functions • move semantics • RAII

Definition

Class: A programmer-defined custom type. An abstraction of an object or data type.

But don’t structs do that?

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

Student s = {"Frankie", "MN", 21};
```

Issues with structs

- Public access to all internal state data by default

```
Student s = {"Frankie", "MN", 21};
s.age = -5;
//should guard against nonsensical values
```

Issues with structs

- Public access to all internal state data by default
- Users of struct need to explicitly initialize each data member.

```
Student s;
cout << s.name << endl; //s.name is garbage
s.name = "Frankie";
cout << s.name << endl; //now we're good!
```

“A struct simply feels like an open pile of bits with very little in the way of encapsulation or functionality. A class feels like a living and responsible member of society with intelligent services, a strong encapsulation barrier, and a well defined interface.”
- Bjarne Stroustrup

Turning `Student` into a class: Header File

```
//student.h
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

Private section:

- Usually contains all member variables
- Users can't access or modify anything in the private section

Turning `Student` into a class: Header File

```
//student.h
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
};
```

Public section:

- Users of the Student object can directly access anything here!
- Defines **interface** for interacting with the private member variables!

Private section:

- Usually contains all member variables
- Users can't access or modify anything in the private section

Turning `Student` into a class: Header File + .cpp File

```
//student.h
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
};
```

```
//student.cpp
#include student.h
std::string
Student::getName(){
//implementation here!
}
void Student::setName(){
}
int Student::getAge(){
}
 void Student::setAge(int
 age){
}
```

Recall: namespaces

- Put code into logical groups, to avoid name clashes
- Each class has its own namespace
- Syntax for calling/using something in a namespace:

```
namespace_name::name
```

Function definitions with namespaces!

- `namespace_name::name` in a function prototype means "this is the implementation for an interface function in `namespace_name`"

- Inside the `{...}` the private member variables for `namespace_name` will be in scope!

```
std::string Student::getName(){...}
```

//student.cpp

```
#include student.h
std::string Student::getName(){
    return name; //we can access name here!
}
void Student::setName(std::string name){

}
int Student::getAge(){

}
 void Student::setAge(int age){

}
```

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

//student.cpp

```
#include student.h
std::string Student::getName(){
    return name; //we can access name here!
}
void Student::setName(std::string name){
    name = name; //huh?
}
int Student::getAge(){

}
 void Student::setAge(int age){


}
```

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

The `this` keyword!

- Here, we mean "set the Student private member variable `name` equal to the parameter `name`"

```
void Student::setName(string name){
    name = name; //huh?
}
```

The `this` keyword!

- Here, we mean "set the Student private member variable `name` equal to the parameter `name`"
- `this->element_name` means "the item in this Student object with name *element_name*". Use `this` for naming conflicts!

```
void Student::setName(string name){
    this->name = name; //better!
}
```

//student.cpp

```
#include student.h
std::string Student::getName(){
    return name; //we can access name here!
}
void Student::setName(string name){
    this->name = name; //resolved!
}
int Student::getAge(){
    return age;
}
 void Student::setAge(int age){


}
```

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

//student.cpp

```
#include student.h
std::string Student::getName(){
    return name; //we can access name here!
}
void Student::setName(string name){
    this->name = name; //resolved!
}
int Student::getAge(){
    return age;
}
 void Student::setAge(int age){
    //We can define what "age" means!
    if(age >= 0){
        this -> age = age;
    }
    else error("Age cannot be negative!");
}
```

//student.h

```
class Student {
    public:
    std::string getName();
    void setName(string
    name);
    int getAge();
    void setAge(int age);

    private:
    std::string name;
    std::string state;
    int age;
 };
```

Constructors

- Define how the member variables of an object is initialized
- What gets called when you first create a Student object

//student.cpp

```
#include student.h
Student::Student(){
    age = 0;
    name = "";
    state = "";
}
```

Constructors

- Define how the member variables of an object is initialized
- What gets called when you first create a Student object
- Overloadable!

//student.cpp

```
#include student.h
Student::Student(){...}
Student::Student(string name, int age, string state){
    this->name = name;
    this->age = age;
    this->state = state;
}
```

Putting it all together: Using your shiny new class!

//main.cpp

```
#include student.h
int main(){
    Student frankie;
    frankie.setName("Frankie");
    frankie.setAge(21);
    frankie.setState("MN");
    cout << frankie.getName() << " is from " << frankie.getState() <<
endl;
}
```

Putting it all together: Using your shiny new class!

//main.cpp

```
#include student.h
int main(){
    Student frankie;
    frankie.setName("Frankie");
    frankie.setAge(21);
    frankie.setState("MN");
    cout << frankie.getName() << " is from " << frankie.getState();

    Student sathya("Sathya", 20, "New Jersey");
    cout << sathya.getName() << " is from " << sathya.getState();
}
```

One last thing... Arrays

- Arrays are a primitive type! They are the building blocks of all containers
- Think of them as lists of objects of **fixed size** that you can **index into**
- **Think of them as the struct version of vectors. You should not be using them in application code! Vectors are the STL interface for arrays!**

```
//int * is the type of an int array variable
int *my_int_array;

//this is how you initialize an array
my_int_array = new int[10];
//this is how you index into an array
int one_element = my_int_array[0];
```

One last thing... Arrays

```
//int * is the type of an int array variable
int *my_int_array;
//my_int_array is a pointer!

//this is how you initialize an array
my_int_array = new int[10];
                   +--+--+--+--+--+--+--+--+--+--+
//my_int_array ->  |  |  |  |  |  |  |  |  |  |  |
                   +--+--+--+--+--+--+--+--+--+--+
//this is how you index into an array
int one_element = my_int_array[0];
```

Destructors

- Arrays are memory **WE** allocate, so we need to give instructions for when to deallocate that memory!
- When we are done using our array, we need to `delete []` it!

```
//int * is the type of an array variable
int *my_int_array;

//this is how you initialize an array
my_int_array = new int[10];
//this is how you index into an array
int one_element = my_int_array[0];
delete [] my_int_array;
```

Destructors

- `delete`ing (almost) always happens in the **destructor** of a class!
- The destructor is defined using Class_name::~Class_name()
- No one ever explicitly calls it! Its called when Class_name object go out of scope!
- Just like all member functions, declare it in the .h and implement in the .cpp!

Code: strvector.cpp

•••

For real!

Today

- ~~Recap: Containers + Iterators~~
- ~~Classes Introduction~~
- **Template Classes (intro)**

Fundamental Theorem of Software Engineering: Any problem can be solved by adding enough layers of indirection.

The problem with IntVector

- Vectors should be able to contain any data type!

The problem with IntVector

- Vectors should be able to contain any data type!

Solution? Create StringVector, DoubleVector, BoolVector etc..

The problem with IntVector

- Vectors should be able to contain any data type!

Solution? Create StringVector, DoubleVector, BoolVector etc..

- What if we want to make a vector of `Student`s?
  - How are we supposed to know about every custom class?
- What if we don't want to write a class for every type we can think of?

The problem with IntVector

- Vectors should be able to contain any data type!

~~Solution? Create StringVector, DoubleVector, BoolVector etc..~~

- What if we want to make a vector of `Student`s?
  - How are we supposed to know about every custom class?
- What if we don't want to write a class for every type we can think of?

SOLUTION: Template classes!

Template Class: A class that is parametrized over some number of types. A class that is comprised of member variables of a general type/types.

Template Classes You've Used

- Vectors!

```
vector<int> numVec; vector<string> strVec;
```

Template Classes You've Used

- Vectors!

```
vector<int> numVec; vector<string> strVec;
```

- Maps!

```
map<int, string> int2Str; map<int, int> int2Int;
```

Template Classes You've Used

- Vectors!

```
vector<int> numVec; vector<string> strVec;
```

- Maps!

```
map<int, string> int2Str; map<int, int> int2Int;
```

- Sets!

```
set<int> someNums; set<Student> someStudents;
```

Template Classes You've Used

- Vectors!

```
vector<int> numVec; vector<string> strVec;
```

- Maps!

```
map<int, string> int2Str; map<int, int> int2Int;
```

- Sets!

```
set<int> someNums; set<Student> someStudents;
```

Pretty much all containers!

Writing a template: Syntax

```
//Example: Structs
template<typename First, typename Second> struct MyPair {
    First first;
    Second second;
};
```

```
//Exactly Functionally the same!
template<typename One, typename Two> struct MyPair {
    One first;
    Two second;
};
```

Writing a Template Class: Syntax

```
//mypair.h
template<typename First, typename Second> class MyPair {
    public:
        /*...*/
    private:
        First first;
        Second second;
};
```

Writing a Template Class: Syntax

```
//mypair.h
template<typename First, typename Second> class MyPair {
    public:
        /*...*/
    private:
        First first;
        Second second;
};
```

Writing a Template Class: Syntax

//mypair.h

```
template<typename First, typename Second> class MyPair {
    public:
        First getFirst();
        Second getSecond();

        void setFirst(First f);
        void setSecond(Second f);
    private:
        First first;
        Second second;
};
```

Writing a Template Class: Syntax

//mypair.h

```
template<typename First, typename Second> class MyPair {
    public:
        First getFirst();
        Second getSecond();

        void setFirst(First f);
        void setSecond(Second f);
    private:
        First first;
        Second second;
};
```

Use generic typenames as placeholders!

Implementing a Template Class: Syntax

//mypair.cpp

```
#include "mypair.h"

First MyPair::getFirst(){
    return first;
}
```

Implementing a Template Class: Syntax

//mypair.cpp

```
#include "mypair.h"

First MyPair::getFirst(){
    return first;
}
//Compile error! Must announce every member function is templated :/
```

Implementing a Template Class: Syntax

```
//mypair.cpp
#include "mypair.h"

template<typename First, typename Second>
First MyPair::getFirst(){
    return first;
}
```

Implementing a Template Class: Syntax

//mypair.cpp

```
#include “mypair.h”

template<typename First, typename Second>
First MyPair::getFirst(){
    return first;
}

template<typename Second, typename First>
Second MyPair::getSecond(){
    return second;
}
```

One final compile error....

```
// vector.h
template <typename T>
class vector<T> {
    T at(int i);
};
```

```
// vector.cpp
#include "vector.h"
template <typename T>
void vector<T>::at(int i) {
    // oops
}
```

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

One final compile error....

```
// vector.h
template <typename T>
class vector<T> {
    T at(int i);
};
```

```
// vector.cpp
#include "vector.h"
template <typename T>
void vector<T>::at(int i) {
    // oops
}
```

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

What the C++ compiler does with non-template classes

```
// main.cpp
#include "vectorint.h"
vectorInt a;
a.at(5);
```

1. `g++ -c vectorint.cpp main.cpp`: Compile and create all the code in vectorint.cpp and main.cpp. All the functions in vectorint.h have implementations that have been compiled now, and main can access them because it included vectorint.h
2. "Oh look she used vectorInt::at, sure glad I compiled all that code and can access vectorInt::at right now!"

What the C++ compiler does with template classes

```
// main.cpp
#include "vector.h"
vector a;
a.at(5);
```

1. `g++ -c vector.cpp main.cpp`: Compile and create all the code in main.cpp. Compile vector.cpp, but since it's a template, don't create any code yet.
2. "Oh look she made a vector<int>! Better go generate all the code for one of those!"
3. "Oh no! All I have access to is vector.h! There's no implementation for the interface in that file! And I can't go looking for vector<int>.cpp!"

The fix...

```cpp
// vector.h
template <typename T>
class vector<T> {
    T at(int i);
};
```

```cpp
// vector.cpp
#include "vector.h"
template <typename T>
void vector<T>::at(int i) {
    // oops
}
```

```cpp
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

Include vector.cpp in vector.h!

```
// vector.h
#include "vector.h"
template <typename T>
class vector<T> {
    T at(int i);
};
```

```
// vector.cpp

template <typename T>
void vector<T>::at(int i) {
    // oops
}
```

```
// main.cpp
#include "vector.h"
vector<int> a;
a.at(5);
```

```
g++ -c vector.cpp main.cpp
g++ vector.o main.o -o output
```

What the C++ compiler does with template classes

```
// main.cpp
#include "vector.h"
vector a;
a.at(5);
```

1. "Oh look she included vector.h! That's a template, **I'll wait to link the implementation until she instantiates a specific kind of vector**"
2. "Oh look she made a vector<int>! Better go generate all the code for one of those!"
3. "vector.h includes all the code in vector.cpp, which tells me how to create a vector<int>::at function :)"

Templates don’t emit code until instantiated, so include the .cpp in the .h instead of the other way around!