

Initialization & References

...

And streams and structs ... :)

Today



- **Streams recap**
- Initialization
- References

Definition

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

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**
- 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
```

Input Streams

- Have type `std::istream`
- Can only **convert** strings using the `>>` operator
 - **Converts** 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;  
    //std::stoi("5")  
    // 283648 983.01  
//reads exactly one int then 1 string from console
```

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

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

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!

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)`!**

Recap: 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.

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);
```

} “=” automatically makes
a copy! Must use & to
avoid this.

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

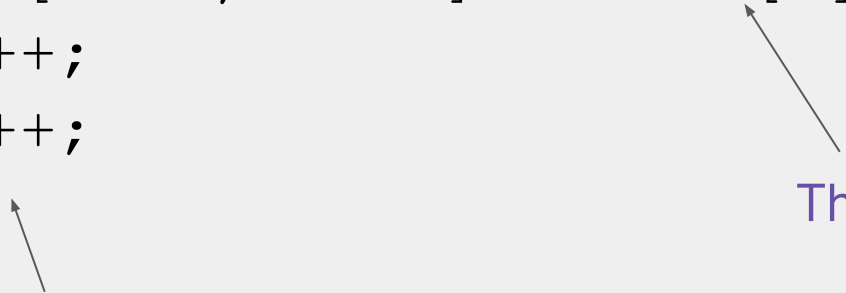
Code demo: References bugs

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 is updating that same
copy!

This creates a copy of the
course


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.

Questions?

Code demo: `r_spaghetti`

Recap:

- Use input streams to get information
- Use structs to bundle information
- Use uniform initialization wherever possible
- Use references to have multiple aliases to the same thing
- Use const references to avoid making copies whenever possible