Initialization & References

And streams and structs … :)

Today

- **Recap: Streams**
- Structs pt 2
- Initialization
- References

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;
// 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!
```

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

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?
```

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>`

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

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(judgement);
    string fluff; int age; bool lovesCpp; string name;
    converter >> name; //reads “Frankie”
    converter >> fluff >> fluff; //reads “,” then “age”
    converter >> age; //reads 21
    converter >> fluff >> fluff; //reads ”,” then “rocks”
    if(fluff == "rocks") return Student{name, age, "bliss"};
    else return Student{name, age, "misery"};
}
```

Questions?

Today

- ~~Recap: Streams~~
- **Structs pt 2**
- Initialization
- References

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
```

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
```

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”);
```

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?

Code Demo: quadratic.cpp

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.

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

- ~~Recap: Streams~~
- ~~Structs pt 2~~
- **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
```

Aside: Stanford Vector vs std::vector...

Vector

```
Vector<int> v;
Vector<int> v(n, k);
v.add(k);
v[i] = k;

v.isEmpty();
v.size();
v.clear();
v.insert(i, k);
v.remove(i);
```

std::vector

```
std::vector<int> v;
std::vector<int> v(n, k);
v.push_back(k);
v[i] = k;

v.empty();
v.size();
v.clear();
// stay tuned...
```

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)!

Today

- ~~Recap: Streams~~
- ~~Structs pt 2~~
- ~~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); //x becomes a reference to x
keepX(b);   //x becomes a copy of x

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

References in 106L

```
//some super exciting and complicated piece of code...
```

Jk, just a regular code demo

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++;
   }
}
```

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++;
    }
}
```

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++;
    }
}
```

Code demo: References errors

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);
```

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

(if time): implement getInteger!

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