

# Lecture 3: Initialization and References

CS 106L, Winter '20

# Today's Agenda

- Recap: **Structures**
- Standard C++ Vector (Intro)
- Uniform Initialization
- Announcements
- References
- Const and Const References

# Recap: Structs and Types


**A struct is a group of named variables**  
*each with their own type*

# Struct Example

```
struct Student {  
    string name;           // these are called fields  
    string state;         // separate these by semicolons  
    int age;  
};  
  
Student s;  
s.name = "Ethan";         // use the . operator to access fields  
s.state = "CA";  
s.age = 20;
```

# A **pair** is a struct with two fields.

```
int main() {  
    std::pair<bool, Student> query_result;  
    query_result.first = true;  
    Report current = query_result.second;  
}
```

 **std::pair** is a **template**. You can use any type inside it; type goes in the <>.  
(We'll learn more about templates in a future lecture.)

# 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.

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

## Before

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

## After

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



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



# Standard C++ vector (intro)

# Stanford Vector review

```
Vector<int> v;  
Vector<int> v(n, k);  
v.add(k);  
v[i] = k;  
auto k = v[i];  
  
v.isEmpty();  
v.size();  
v.clear();  
v.insert(i, k);  
v.remove(i);
```

# Stanford Vector vs. Standard `std::vector`

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

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

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

```
v.empty();  
v.size();  
v.clear();  
// stay tuned for a future lecture  
// stay tuned for a future lecture
```

 **Questions?** 

# Uniform Initialization

# **What's initialization?**

Initialization is how we provide initial values to variables

# Initialization examples

```
int x = 5; // initializing while we declare
```

```
int y;
```

```
y = 6; // initializing after we declare
```

**In C++, initialization used to be tricky, and varied substantially depending upon the type**



# Examples of complicated initialization

```
std::pair<bool, int> some_pair = std::make_pair(false, 6);  
  
// Student is a struct from last time  
// with name, state, and age fields  
Student s;  
s.name = "Ethan";  
s.state = "CA";  
s.age = 20;
```

# That's why uniform initialization is useful

Uniform initialization provides a way for us to use **brackets** to initialize anything succinctly

# Uniform Initialization to the Rescue!

## Before

```
std::pair<bool, int> some_pair =  
    std::make_pair(false, 6);
```

```
Student s;  
s.name = "Ethan";  
s.state = "CA";  
s.age = 20;
```

## After

```
std::pair<bool, int>  
    some_pair{false, 6};  
  
Student s{"Ethan", "CA", 20};
```

# We can also set our variable equal to the curly brackets chunk

## Before

```
std::pair<bool, int> some_pair =  
    std::make_pair(false, 6);
```

```
Student s;  
s.name = "Ethan";  
s.state = "CA";  
s.age = 20;
```

## After

```
std::pair<bool, int>  
    some_pair = {false, 6};  
  
Student s = {"Ethan", "CA", 20};
```

# Uniform initialization examples

```
int x{3}; // Uniform initialization not super needed here
int y = {3};
```

```
// Both of these are vectors with elements {3, 5, 7}
std::vector<int> a_vector{3, 5, 7};
std::vector<int> another_vector = {3, 5, 7};
```

```
// Later, when we get to classes, we can use uniform
// initialization to invoke our constructors, which dictate
// how we initialize our variable
```

# A “gotcha” with uniform initialization

Remember this way to initialize a `std::vector`?

```
int n = 3;
```

```
int k = 5;
```

```
std::vector<int> v(n, k); // {5, 5, 5}
```

## A “gotcha” with uniform initialization (cont.)

This initialization uses a **constructor** (which 106B will talk more about soon)

```
int n = 3;
```

```
int k = 5;
```

```
std::vector<int> v(n, k); // {5, 5, 5}
```

## A “gotcha” with uniform initialization (cont.)

Normally, we can replace the () with {} to use uniform initialization--not here!

```
int n = 3;
```

```
int k = 5;
```

```
std::vector<int> v(n, k); // {5, 5, 5}
```

```
std::vector<int> v2{n, k}; // {3, 5} -- not the same!!
```



# Using {} for vector creates an initializer\_list

When we create a `std::initializer_list`, we actually end up invoking a **different constructor**!

```
auto list_init{3, 5}; // type is std::initializer_list
```

## **Moral of the story:**

Make sure you're completely clear what constructor you're invoking when using uniform initialization

# Complicated uniform initialization example

Say we wanted to represent Stanford courses as structs

```
struct Course {  
    std::string code;  
    std::pair<Time, Time> time;  
    std::vector<std::string> instructors;  
};  
  
struct Time {  
    int hour, minute;  
};  
  
...  
  
Course now{"CS106L", { {14, 30}, {15, 50} }, {"Raghuraman", "Chi" };
```

 **Questions?** 

## Summary of Uniform Initialization

- You can use uniform initialization to initialize anything
  - Be careful of ambiguities with `std::initializer_list`

# References

# A coding problem

Write a function to shift all courses in a `std::vector` over by 1 hour

```
struct Course {
    std::string code;
    std::pair<Time, Time> time;
    std::vector<std::string> instructors;
};

struct Time {
    int hour, minute;
};

...

void shift(std::vector<Course>& courses) {
    // TODO
}
```

# **Live Code Demo:** Courses



# This is buggy!

```
void shift(vector<Course>& courses) {  
    for (size_t i = 0; i < courses.size(); ++i) {  
        auto [code, time, instructors] = courses[i];  
  
        time.first.hour++;  
        time.second.hour++;  
    }  
}
```

This creates a copy of the  
course

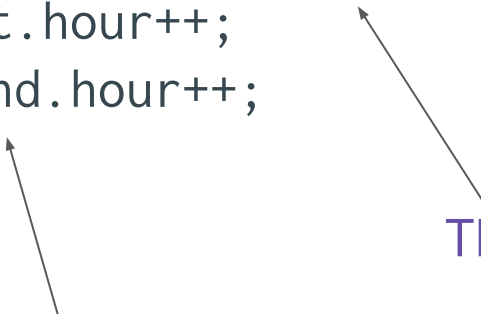


This is updating that same  
copy!



## This also doesn't work!

```
void shift(vector<Course>& courses) {  
    for (auto [code, time, instructors] : courses) {  
        time.first.hour++;  
        time.second.hour++;  
    }  
}
```



This is updating that same  
copy!

This creates a copy of the  
course

**We need more information on reference  
parameters!**

# What we've learned in 106B

```
int doubleValue(int& x) {  
    x *= 2;  
    return x;  
}  
  
int main() {  
    int myValue = 5;  
    int result = doubleValue(myValue);  
    cout << myValue << endl; // 10  
    cout << result << endl;  // 10  
}
```

The variable myValue is passed by **reference** into doubleValue.

x inside doubleValue is an **alias** for myValue in main. This means it's another name for the same variable!

A change to x is a change to myValue.

# References in variable assignment

Notice the ampersand -- ref is an alias for original!

```
std::vector<int> original{1, 2};  
std::vector<int> copy = original;  
std::vector<int>& ref = original;  
  
original.push_back(3);  
copy.push_back(4);  
ref.push_back(5);  
  
// original (and also ref!) = {1, 2, 3, 5}  
// copy = {1, 2, 4}
```

# Practice Problem (try afterward if you'd like)

Read through the code below and think through the questions! (answers on next slide)

```
std::vector<int> original{1, 2};  
std::vector<int> copy = original;  
std::vector<int>& ref = original;  
  
original.push_back(3);  
copy.push_back(4);  
ref.push_back(5);  
  
// original (and also ref!) = {1, 2, 3, 5}  
// copy = {1, 2, 4}  
ref = copy;  
copy.push_back(6);  
ref.push_back(7);  
// Q1: what are contents of original?  
// Q2: what are contents of copy?
```

# A reference is always an alias to the same variable!

This means setting `ref` equal to a new value is exactly the same as setting `original` equal to that value! We can't change what variable `ref` aliases

```
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);  
  
// original (and also ref!) = {1, 2, 3, 5}  
// copy = {1, 2, 4}  
ref = copy;  
copy.push_back(6);  
ref.push_back(7);  
// original = {1, 2, 4, 7}  
// copy = {1, 2, 4, 6}
```

# **Live Code Demo:**

## References pitfall



## Note: You can only create references to variables

This has to do with something called l-values (as we saw in our error message), which we'll discuss later in the course!

```
int& thisWontWork = 5; // This doesn't work!
```

 **Questions?** 

# 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);    // 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!

The below example isn't permitted!

```
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 a non-const ref to a const ref!

The below is also not permitted!

```
std::vector<int> vec{1, 2, 3};  
  
const std::vector<int>& c_ref = vec;  // a const reference  
  
// BAD - Can't declare a non-const reference as equal  
// to a const reference!  
std::vector<int>& ref = c_ref;
```

# If you don't write &, C++ will make a copy by default!

```
std::vector<int> vec{1, 2, 3};  
const std::vector<int>& c_ref = vec;           // a const reference  
  
// This is a non-const copy of vec, even though we're setting  
// it equal to a const reference! Remember that ref is just an  
// alias (aka another name) for vec  
std::vector<int> copy = c_ref;  
  
copy.push_back(4);  
// vec = {1, 2, 3}  
// copy = {1, 2, 3, 4}
```

# Need to explicitly specify const and & with auto!

```
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?** 

# More about References

# You can return references as well!

This is something that the C++ Standard Library frequently makes use of.

```
// Note that the parameter must be a non-const reference to return  
// a non-const reference to one of its elements!
```

```
int& front(std::vector<int>& vec) {  
    // assuming vec.size() > 0  
    return vec[0];  
}  
  
int main() {  
    std::vector<int> numbers{1, 2, 3};  
    front(numbers) = 4; // vec = {4, 2, 3}  
    return 0;  
}
```

## Can also return const references

```
const int& front(std::vector<int>& vec) {  
    // assuming vec.size() > 0  
    return vec[0];  
}
```

# Dangling references: references to out-of-scope vars

Never return a reference to a local variable! They'll go out of scope.

```
int& front(const std::string& file) {  
    std::vector<int> vec = readFile(file);  
    return vec[0];  
}  
  
int main() {  
    front("text.txt") = 4; // undefined behavior  
    return 0;  
}
```

## Return to example: How could we fix this code? (chat)

```
void shift(vector<Course>& courses) {  
    for (size_t i = 0; i < courses.size(); ++i) {  
        auto [code, time, instructors] = courses[i];  
  
        time.first.hour++;  
        time.second.hour++;  
    }  
}
```

# Return to example: How could we fix this code?

This is one option.

```
void shift(vector<Course>& courses) {  
    for (size_t i = 0; i < courses.size(); ++i) {  
        auto& [code, time, instructors] = courses[i];  
  
        time.first.hour++;  
        time.second.hour++;  
    }  
}
```

This creates a reference to  
the course

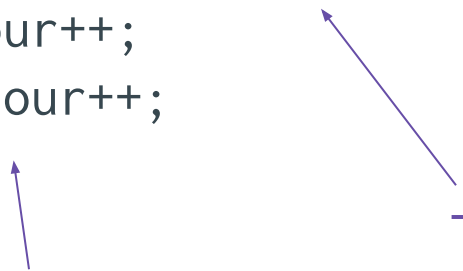
This updates that reference!



## Return to example: How could we fix this code?

This is another option.

```
void shift(vector<Course>& courses) {  
    for (auto& [code, time, instructors] : courses) {  
        time.first.hour++;  
        time.second.hour++;  
    }  
}
```



This updates that reference!

This creates a reference to  
the course

# When do we use references/const references?

- If we're working with a variable that takes up little space in memory (e.g. `int`, `double`), we don't need to use a reference and can just copy the variable
- If we need to **alias** the variable to modify it, we can use references
- If we don't need to modify the variable, but it's a big variable (e.g. `std::vector`), we can use const references

# Recap

- **Uniform initialization**
  - A “uniform” way to initialize variables of different types!
- **References**
  - Allow us to alias variables
- **Const**
  - Allow us to specify that a variable can't be modified

**Next time:**

Streams