"The Rubber Duck knows no frontiers, it doesn't discriminate people and doesn't have a political connotation."

"The friendly, floating Rubber Duck has healing properties: it can relieve mondial tensions as well as define them. The Rubber Duck is soft, friendly and suitable for all ages!"

Challenge Quiz: Which of the following are true?

Review Questions (from week 1 of CS 106B and CS 106L)

1. `auto` makes your program slower, so you should use it sparingly.

2. In "`auto[i, s] = make_pair(3, "hi")`" the compiler will deduce that s is a std::string.

3. A (Stanford) vector behaves like a tuple where all the members have the same type.

4. Structured binding unpacks the members of a struct in the order the members were declared in the struct declaration.

Challenge Questions (will require some critical thinking)

5. Structured binding can unpack individual elements of a Stanford Vector (or `std::vector`).

6. You should always pass parameters by reference, unless you actually want a copy created.

7. The line "`auto i;`" compiles.

Challenge Quiz: Which of the following are true?

Review Questions (from week 1 of CS 106B and CS 106L)

1. `auto` makes your program slower, so you should use it sparingly.

2. In "`auto[i, s] = make_pair(3, "hi")`" the compiler will deduce that s is a std::string.

3. A vector (eg. Stanford) behaves like a tuple where all the members have the same type.

4. Structured binding unpacks the members of a struct in the order the members were declared in the struct declaration.

Challenge Questions (will require some critical thinking)

5. Structured binding can unpack individual elements of a Stanford Vector (or `std::vector`).

6. You should always pass parameters by reference, unless you actually want a copy created.

7. The line "`auto i;`" compiles.

Challenge Quiz: Which of the following are true?

Review Questions (from week 1 of CS 106B and CS 106L)

1. `auto` makes your program slower, so you should use it sparingly.
FALSE: auto is resolved at compile-time, but your program at run-time is the same speed.
2. In "`auto[i, s] = make_pair(3, "hi")`" the compiler will deduce that s is a std::string.
FALSE: A string literal is a C-string (const char*), and auto will deduce s is a C-string.
3. A (Stanford) vector behaves like a tuple where all the members have the same type.
FALSE: the size of the vector can change, but a tuple is fixed size.
4. Structured binding unpacks the members of a struct in the order the members were declared in the struct declaration.
TRUE: we demonstrated this in lecture 3 as well in the find and shift functions!

Challenge Questions (will require some critical thinking)

5. Structured binding can unpack individual elements of a Stanford Vector (or `std::vector`).
FALSE: related to #3 - you can only know the number of elements of a vector at run-time.
6. You should always pass parameters by reference, unless you actually want a copy created.
FALSE: see the last section of this lecture.
7. The line "`auto i;`" compiles.
FALSE: auto does not allow uninitialized variables (can't figure out the type)

Asking questions!

- Ask your questions onto the chat on the webinar Q&A.
- Anna will monitor and answer questions.
- I'll stop at set points in the presentation and Anna will pick the most common/important questions for me to answer.
- Use Piazza to ask questions as well after class!
- Use non-verbal feedback, such as faster or slower and I'll adjust.

References

Game Plan

- uniform initialization
- references
- parameters and return
- (C++ library walkthrough)

Key questions we will answer today

- what is a reference?
- how do references help with safety and efficiency?
- where are references used in the STL?

- (random facts we'll also cover: size_t, auto, const, casts)

C++ details we will cover

Language

- uniform initialization
- references
- auto, const, and references
- (maybe) conversion/casts

Library

- std::vector
- std::string
- std::chrono (maybe)

Aside: size_t

Ever get this annoying warning message about unsigned integers?

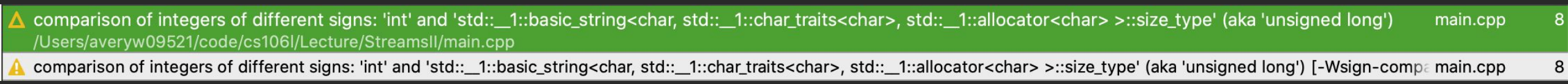
```
string str = “Hello World!”;
for (int i = 0; i < str.size(); ++i) {
   cout << str[i] << endl;
}
```

⚠ comparison of integers of different signs: 'int' and 'std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >::size_type' (aka 'unsigned long') main.cpp 8
/Users/averyw09521/code/cs106l/Lecture/StreamsII/main.cpp

⚠ comparison of integers of different signs: 'int' and 'std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >::size_type' (aka 'unsigned long') [-Wsign-compa main.cpp 8

Ever get this annoying warning message about unsigned integers?

```
string str = “Hello World!”;
for (int i = 0; i < str.size(); ++i) {
   cout << str[i] << endl;
}
```



This comparison is dangerous since it compares signed (i) with unsigned (str.size()).

Ever get this annoying warning message about unsigned integers?

```
string str = “Hello World!”;
for (size_t i = 0; i < str.size(); ++i) {
   cout << str[i] << endl;
}
```

Used mostly for
dealing with indices.

What’s the bug in this function?

```
string chopBothEnds(const string& str) {
   string result = “”;
   for (size_t i = 1; i < str.size()-1; ++i) {
      result += str[i];
   }
   return result;
}
```

std::vector vs. Stanford Vector - super basic intro

```
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();

// need iterators
// need iterators
```

Recap: auto and structures

auto directs the compiler figure the type for you

```
// return type: string, notice can't use auto for parameter
auto calculateSum(const vector<string>& v) {
  auto multiplier = 2.4;                          // double
  auto name = "Ito-En";                           // char* (c-string)
  auto betterName1 = string{"Ito-En"};            // string
  const auto& betterName2 = string{"Ito-En"};     // const string&
  auto copy = v;                                  // vector<string>
  auto& refMult = multiplier;                     // double&
  auto func = [](auto i) {return i*2;};           // ???

  return betterName1;
}
```

Careful: auto discards const and references!

pair/tuple functions

```
// make_pair/tuple (C++11) automatically deduces the type!
auto prices = make_pair(3.4, 5);        // pair<double, int>
auto values = make_tuple(3, 4, “hi”);   // tuple<int, int, char*>

// access via get/set
prices.first = prices.second;           // prices = {5.0, 5};
get<0>(values) = get<1>(values);        // values = {4, 4, “hi”};

// structured binding (C++17) – extract each component
auto [a, b] = prices;                   // a, b are copies of 5.0 and 5
const auto& [x, y, z] = values;         // x, y, z are const references
                                        // to the 4, 4, and “hi”.
```

struct functions

```
struct Discount {
  double discountFactor;
  int expirationDate;
  string nameOfDiscount;
}; // don't forget this semicolon :/

// Call to Discount's constructor or initializer list
auto coupon1 = Discount{0.9, 30, "New Years"};
Discount coupon2 {0.75, 7, "Valentine's Day"};

coupon1.discountFactor = 0.8;
coupon2.expirationDate = coupon1.expirationDate;

// structured binding (C++17) - extract each component
auto [factor, date, name] = coupon1;
```

Find the course with the desired course code.

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

```
struct Time {
  int hour, minute;
};
```

```
pair<Course, bool> find(vector<Course>& courses, string& target) {




}
```

Find the course with the desired course code.

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

```
struct Time {
  int hour, minute;
};
```

```
pair<Course, bool> find(vector<Course>& courses, string& target) {
   for (size_t i = 0; i < courses.size(); ++i) {
      auto [code, time, instructors] = courses[i];
      if (code == target) {
         return {courses[i], true};
      }
   }
   return { {} , false};
}
```

Even better: for-each loop!

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

```
struct Time {
  int hour, minute;
};
```

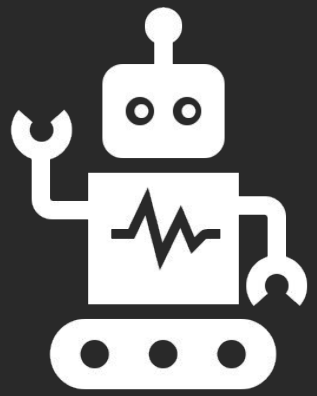
```
pair<Course, bool> find(vector<Course>& courses, string& target) {
   for (auto course : courses) {
      auto [code, time, instructors] = course;
      if (code == target) {
         return {course, true};
      }
   }
   return { {} , false};
}
```

Even even better: structured binding in a for-each loop!

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

```
struct Time {
  int hour, minute;
};
```

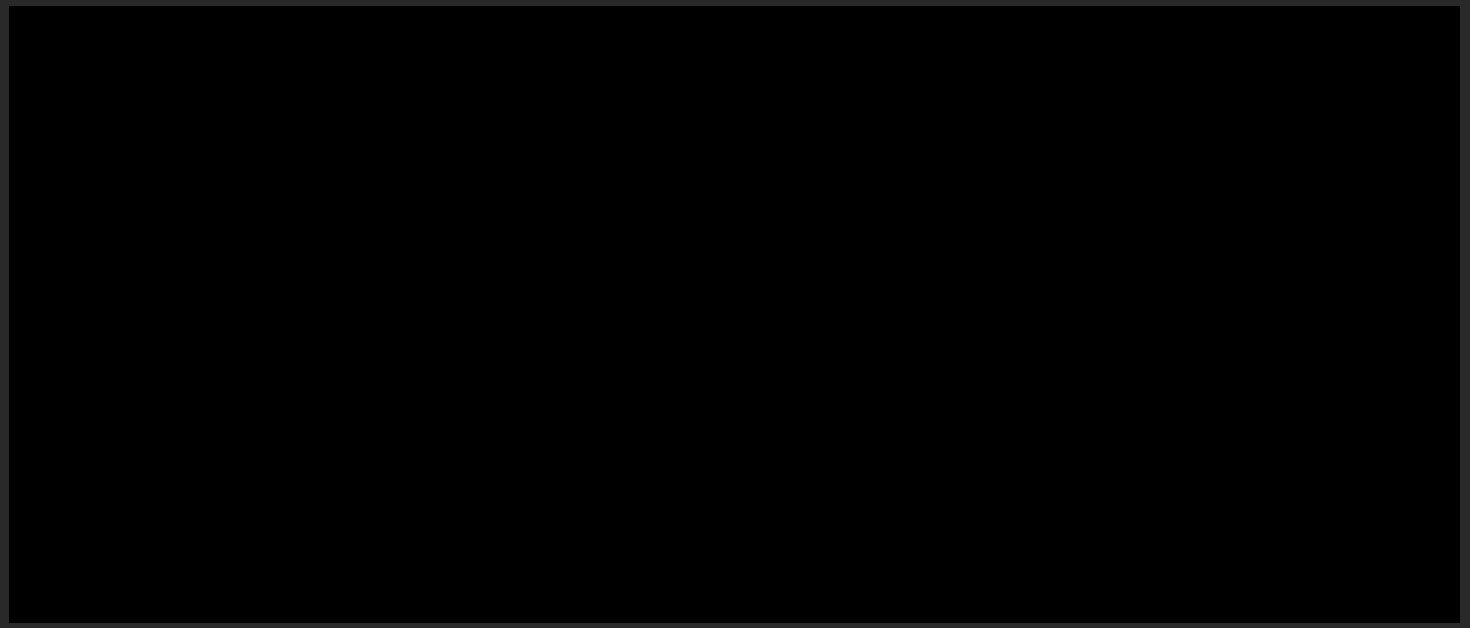
```
pair<Course, bool> find(vector<Course>& courses, string& target) {
   for (auto [code, time, instructors] : courses) {
      if (code == target) {
         // returning the course slightly more awkward :/
         return {{code, time, instructors}, true};
      }
   }
   return { {} , false};
}
```

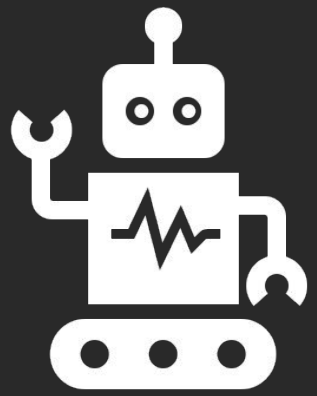


Questions

topic 1: uniform initialization

initialization in C++ is complicated





Switch to Live Coding

Uniform Initialization

uniform initialization is the way to initialize anything

```
1.  std::vector<string> default_init;
2.  std::vector<string> value_init{};
3.  std::vector<string> direct_init{3, "init"};
4.  std::vector<string> copy_init = {3, "init"};
5.  std::vector<string> list_init{"1", "2", "3"};
6.  std::vector<string> aggr_init = {"1", "2", "3"};
7.  // and many more
```

(5) and (6) use an std::initializer_list, so the vector is constructed with those elements.

Sidenote: some benefits of uniform initialization:

- no issues of uninitialized values
- no narrowing conversions possible

uniform initialization is the way to initialize anything

```
1.  std::vector<string> default_init;
2.  std::vector<string> value_init{};
3.  std::vector<string> direct_init{3, "init"};
4.  std::vector<string> copy_init = {3, "init"};
5.  std::vector<string> list_init{"1", "2", "3"};
6.  std::vector<string> aggr_init = {"1", "2", "3"};
7.  // and many more
```

We'll typically use these three constructors.

1. Default
3. Uniform Initialization
5. Initializer List

be careful with these initializer list gotcha's

Sometimes there are ambiguities with the initializer_list.

Use the parenthesis instead to avoid the initializer_list.

```
std::vector<int> list_init{3, 2}; // {3, 2}
std::vector<int> fill_ctor(3, 2); // {2, 2, 2}
```

auto + initializer_list = </3

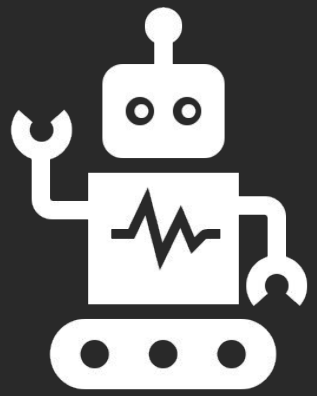
```
auto list_init{3, 2}; // type = initializer_list, not vector
```

Find the course with the desired course code.

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

```
struct Time {
  int hour, minute;
}
```

```
int main() {
  Course now {“CS106L”, { {13, 30}, {14, 30} }, {“Wang”, “Zeng”} };


}
```



Questions

Summary of Uniform Initialization

Use uniform initialization as your default way to call any constructor.

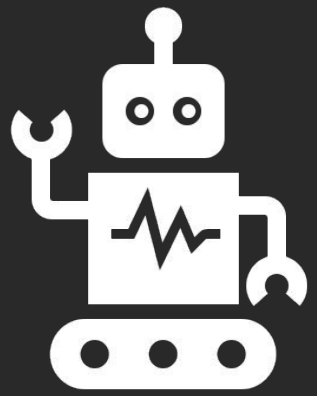
Be careful of ambiguities with initializer_list.

Write function that shifts all Courses forward by one hour.

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

```
struct Time {
  int hour, minute;
}
```

```
void shift(vector<Course>& courses) {



}
```



Switch to Live Coding

Uniform Initialization

This is a buggy implementation!

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

```
struct Time {
  int hour, minute;
}
```

```
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.

This is updating the copy.

This is a buggy implementation!

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

```
struct Time {
  int hour, minute;
}
```

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

This creates a copy.

This is updating the copy.

topic 2: references

Reference parameters: caller and callee share the same variable in memory.

```
1.  int doubleValue(int& x) {
2.    x *= 2;
3.    return x;
4.  }
5.
6.  int main() {
7.    int myValue = 5;
8.    int result = doubleValue(myValue);
9.    cout << myValue << endl;
10.   cout << result << endl;
11. }
```

The variable myValue is passed by reference into doubleValue.

x inside doubleValue is an alias for result in main.

A change to x is a change to result.

Aside: Chris's observation that you can't write an expression doubleValue(15) is really interesting.

The error message mentions something called an l-value. We'll get back to this!

a reference is an alias for another variable

```
1.  vector<int> original{1, 2};
2.  vector<int> copy = original;
3.  vector<int>& lref = original;
4.
5.  original.push_back(3);
6.  copy.push_back(4);
7.  lref.push_back(5);
8.
9.  // original (lref) = {1, 2, 3, 5}
10. // copy = {1, 2, 4}
```

More generally, you can use (l-value) references as another name for another variable.

Notice where we put the ampersand(&) - similar to how declared a reference parameter.

sidenote: why is it called an l-value reference? Is it related to the sidenote on the previous slide? Of course there is!

Poll Quiz Time!

```
1.  vector<int> original{1, 2};
2.  vector<int> copy = original;
3.  vector<int>& lref = original;
4.
5.  original.push_back(3);
6.  copy.push_back(4);
7.  lref.push_back(5);
8.
9.  // original (lref) = {1, 2, 3, 5}
10. // copy = {1, 2, 4}
11. lref = copy;
12. copy.push_back(6);
13. lref.push_back(7);
14. // Q1: what are contents of original?
15. // Q2: what are contents of copy?
```

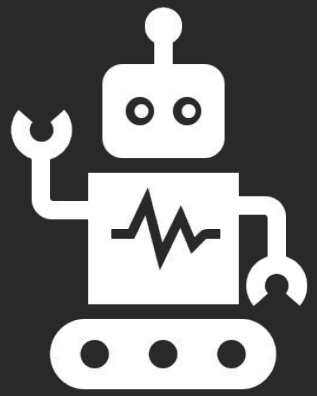
you cannot reassign a reference after construction

```
1.  vector<int> original{1, 2};
2.  vector<int> copy = original;
3.  vector<int>& lref = original;
4.
5.  original.push_back(3);
6.  copy.push_back(4);
7.  lref.push_back(5);
8.
9.  // original (lref) = {1, 2, 3, 5}
10. // copy = {1, 2, 4}
11. lref = copy;
12. copy.push_back(6);
13. lref.push_back(7);
14. // original = {1, 2, 4, 7}
15. // copy = {1, 2, 4, 6}
```

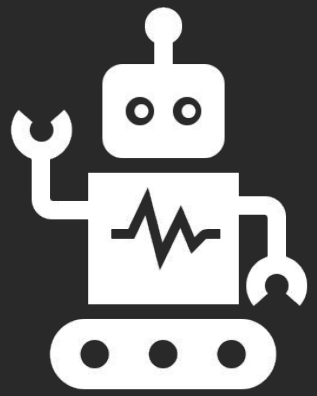
Continuing from previous slide…

What happens if you try to reassign a reference? As in (11), you are actually calling copy assignment.

A reference remains an alias to whatever variable it was bound to.



Questions



Switch to Live Coding

Const and Const Reference

a const variable cannot be modified after construction

```
1.  std::vector<int> vec{1, 2, 3};
2.  const std::vector<int> c_vec{7, 8};
3.  std::vector<int>& lref = vec;
4.  const std::vector<int>& c_lref = vec;
5.
6.
7.  vec.push_back(3);        // OKAY
8.  c_vec.push_back(3);      // BAD - const
9.  lref.push_back(3);       // OKAY
10. c_lref.push_back(3);     // BAD - const
```

How do we know if a certain operation for an object is const? Member functions are labelled "const" in their header.

We have a regular reference (3) and a const reference (4).

We can modify the non-const vector or non-const reference (7, 9).

We can't modify the const vector or const reference (8, 10).

You can't declare a non-const reference to a const vector.

auto drops const/reference unless explicitly specified

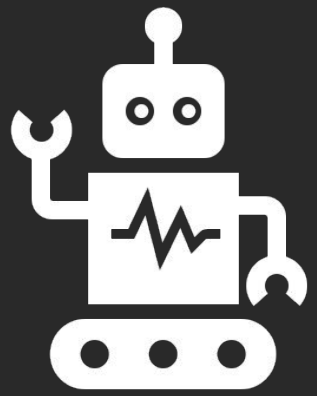
```
1.  std::vector<int> vec{1, 2, 3};
2.  const std::vector<int> c_vec{7, 8};
3.  std::vector<int>& lref = vec;
4.  const std::vector<int>& c_lref = vec;
5.
6.
7.  auto copy = c_lref;
8.  const auto copy = c_lref;
9.  auto& alref = lref;
10. const auto& c_alref = lref;
```

Without the const, (7) creates a non-const copy. (8) creates a const copy.

Without the reference, (7) and (8) create copies, while (9) and (10) create references.

c_alref is an alias to lref, which is an alias to vec. So c_alref is functionally an alias to vec.

Sidenote: auto also drops a volatile qualifier, which is important for embedded systems programming.



Questions

Announcements

Logistics

- Sign up for Piazza! Exercises are released on Piazza.
- GraphViz (optional project 1) has been released.

2-min stretch break!

returning references in the STL

Lots of STL classes have methods which return a reference to an element, allowing you to change that element.

```
1.  std::vector<int> vec{5, 6};    // {5, 6}
2.  vec[1] = 3;                    // {5, 3}
3.
4.  std::string str = "Ito-En";    // str = "Ito-En"
5.  str[0] = 'M';                  // str = "Mto-En"
```

returning references in the STL

Another example: streams with the << operator.

```
1.  std::cout << "First" << "Second";
```

This is secretly calling the following function, which you'll notice is returning a reference (in fact, to os itself).

```
2.  ostream& operator<<(ostream& os, const string& rhs);
```

The result of the first << returns cout itself, allowing the second << to occur. This is why you can chain <<'s!

```
3.  (std::cout << "First") << "Second";
4.  std::cout << "Second";
```

you can return references yourself as well.

```
1.  int& front(const std::vector<int>& vec) {
2.    // assuming vec.size() > 0
3.    return vec[0];
4.  }
5.
6.  int main() {
7.    std::vector<int> numbers{1, 2, 3};
8.    front(numbers) = 4; // vec = {4, 2, 3}
9.  }
```

vec (1) is a const reference for numbers, so front returns a reference to the first element of numbers.

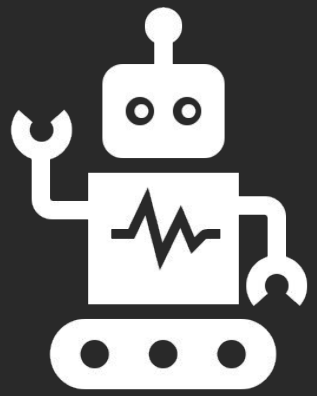
When we assign that reference to 4 (8), we are changing the first element of numbers.

Sidenote: I agree, it feels kinda strange to return references to stuff inside the parameters. However, in classes like std::vector, we're returning references to private members, giving the client a way to modify them.

dangling refs: references to variables out of scope

```
1.  int& front(const std::string& file) {
2.    std::vector<int> vec = readFile(file);
3.    return vec[0];
4.  }
5.
6.  int main() {
7.    front("text.txt") = 4; // undefined behavior
8.  }
```

Never return a reference to a local (automatic) variable (3), as they will go out of scope when you try to read/write to the reference (7).



Questions

This is the correct implementation of shift.

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

```
struct Time {
  int hour, minute;
}
```

```
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 the Time inside the Course.

This is the correct implementation of shift.

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

```
struct Time {
  int hour, minute;
}
```

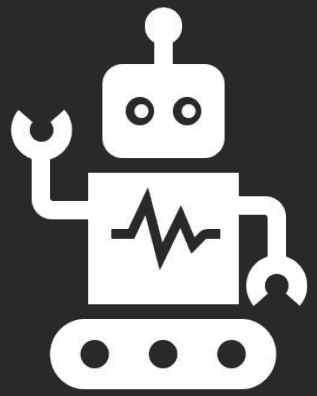
```
void shift(vector<Course>& courses) {
   for (auto& [code, time, instructors] : courses) {
      time.first.hour++;
      time.second.hour++;
   }
}
```

This creates a reference to the Course.

This updates the Time inside the Course.

This is the correct implementation of shift.

```
// non-const reference to each element in courses
for (auto& course : courses) {

}

for (auto& [code, time, instructors] : courses) {

}

// const reference to each element in courses
for (const auto& course : courses) {

}

for (const auto& [code, time, instructors] : courses) {

}
```



Questions

Summary of References

reference: an alias

const: can't modify

auto: drops const/reference

only return references to non-local vars

topic 3: parameters and return

C++ guidelines: parameter and return rules

	Cheap or impossible to copy (e.g., int, unique_ptr)	Cheap to move (e.g., vector<T>, string) or Moderate cost to move (e.g., array<vector>, BigPOD) or Don't know (e.g., unfamiliar type, template)	Expensive to move (e.g., BigPOD[], array<BigPOD>)
Out		X f() Pre-C++11 (eg. CS 106B) prefers f(X&) in this case.	
In/Out		f(X&)	
In	f(X)	f(const X&)	
In & retain "copy"			

Source: https://www.modernescpp.com/index.php/c-core-guidelines-how-to-pass-function-parameters

General guidelines from experience
Cheap to copy: size < 4*sizeof(int)
Not cheap to copy: size >= 4*sizeof(int)

Sidenote: the return statement creates a copy of a local, which is bad if expensive to copy, which explains the note.

In C++11, copy-elision causes the copy to be skipped.

Design Philosophy of C++

- Allow the programmer full control, responsibility, and choice if they want it.
- Express ideas and intent directly in code.
- Enforce safety at compile time whenever possible.
- Do not waste time or space.
- Compartmentalize messy constructs.

breakout discussion: designing headers

In your groups, discuss a header for a function that solves each of problem.

1. Count the number of occurrences of an element in a vector.
2. Read the lines of a filestream into a vector.

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain "copy"			

Breakout Discussion

Designing headers for functions.

breakout discussion: designing headers

In your groups, discuss a header for a function that solves each of problem.

```
size_t count_occurrences(const vector<int>&, int);
vector<string> read_lines(ifstream&);
```

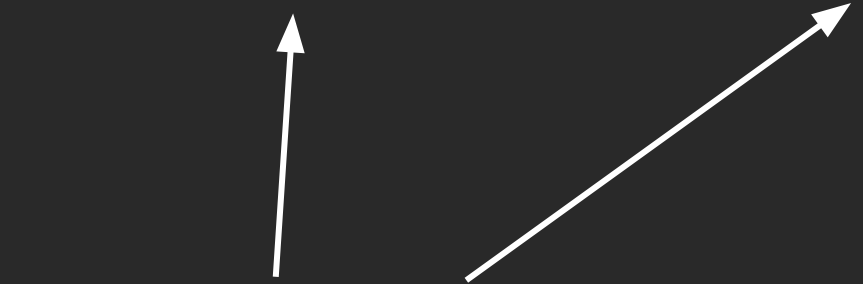
	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD[], array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain "copy"			

sneak peak: templates

```
size_t count_occurrences(const vector<int>&, int);
```

what if we wanted to write a template version of count_occurances?

```
template <typename Collection, typename DataType>
size_t count_occurrences(const Collection&, const DataType&);
```



unfamiliar type: assume expensive to copy.

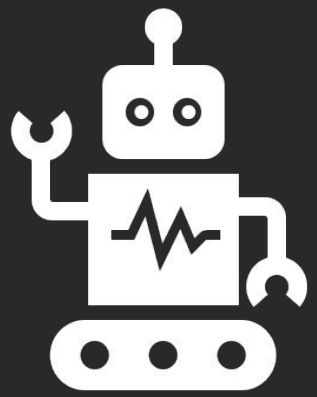
Preview of week 7: moving instead of copying

	Cheap or impossible to copy (e.g., int, unique_ptr)	**Cheap to move** (e.g., vector<T>, string) or **Moderate cost to move** (e.g., array<vector>, BigPOD) or **Don't know** (e.g., unfamiliar type, template)	**Expensive to move** (e.g., BigPOD**[]**, array<BigPOD>)
Out	X f()		
In/Out	f(X&)		
In	f(X)	f(const X&)	
In & retain copy		f(const X&) + f(X&&) & move **	
In & move from	f(X&&) **		

** or return unique_ptr<X>/make_shared_<X> at the cost of a dynamic allocation*

*** special cases can also use perfect forwarding (e.g., multiple in+copy params, conversions)*

General guidelines from experience
Cheap to copy: size < 4*sizeof(int)
Not cheap to copy: size >= 4*sizeof(int)

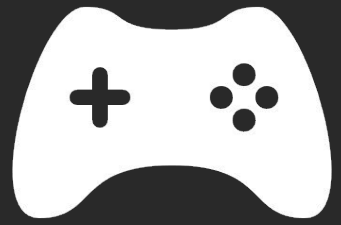


Deep Dive into Documentation

reference and const references as parameter/return
const member functions

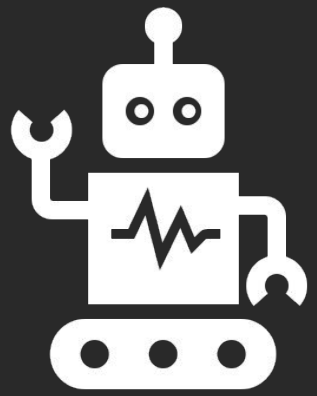
Where we've been today:

- size_t
- uniform initialization
- references (auto&, const, return, STL)
- parameters and return



Next time

Streams (i.e. C++ i/o)



Homework

GraphViz