Game Plan

• implicit interfaces
• concepts (C++20)
• functions and lambdas
• algorithms library
• adaptors
• ranges (C++20)
Review: we wrote this generic minmax function.

template <typename T>
pair<T, T> my_minmax(T a, T b) {
    if (a < b) return {a, b};
    else return {b, a};
}
How many of you did the homework?

template <typename InputIt1 typename InputIt2>
pair< int, int > mismatch( InputIt1 first1, InputIt1 last1, InputIt2 first2)

while (first1 != last1 && *first1 == *first2){
    ++first1; ++first2;
}

return {first1, first2};
Questions
We will focus on solving the last task today!

Count how many times 3 appears in a vector<int>.
Count how many times 4.7 appears in a vector<double>.
Count how many times ‘X’ appears in a string.
Count how many times ‘X’ appears in a deque<char>.
Count how many times 5 appears in the second half of a list<string>.
Count how many elements in the second half of a list<string> are at most 5.
Generic Programming and Lifting
Looking at the assumptions you place on the parameters, and questioning if they are really necessary.

Can you solve a more general problem by relaxing the constraints?
Implicit Interfaces and Concepts
The compiler literally replaces each template parameter with whatever you instantiate it with.

vector<int> v1{1, 2, 3, 1, 2, 3};
vector<int> v2{1, 2, 3};
countOccurences(v1.begin(), v1.end(), v2.begin());
The compiler literally replaces each template parameter with whatever you instantiate it with.

template <typename InputIterator, typename DataType>
int countOccurences(InputIterator begin, InputIterator end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
The compiler literally replaces each template parameter with whatever you instantiate it with.

```cpp
template <typename InputIterator, typename DataType>
int countOccurences(vector<int>::input_iterator begin, vector<int>::input_iterator end, vector<int>::input_iterator val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
```
The compiler literally replaces each template parameter with whatever you instantiate it with.

template <typename InputIterator, typename DataType>
int countOccurences(vector<int>::input_iterator begin, 
vector<int>::input_iterator end, vector<int>::input_iterator val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
A template function defines an implicit interface that each template parameter must satisfy.

```cpp
template <typename InputIterator, typename DataType>
int countOccurences(InputIterator begin, InputIterator end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
```

What must be true of `InputIterator` and `DataType`?
A template function defines an implicit interface that each template parameter must satisfy.

```cpp
template <typename InputIterator, typename DataType>
int countOccurences(InputIterator begin, InputIterator end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
```

begin must be copyable.
A template function defines an implicit interface that each template parameter must satisfy.

template <typename InputIterator, typename DataType>
int countOccurences(InputIterator begin, InputIterator end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
A template function defines an implicit interface that each template parameter must satisfy.

```cpp
template <typename InputIterator, typename DataType>
    int countOccurences(InputIterator begin, InputIterator end, DataType val) {
        int count = 0;
        for (auto iter = begin; iter != end; ++iter) {
            if (*iter == val) ++count;
        }
        return count;
    }
```

You must be able to increment `iter`. 

A template function defines an implicit interface that each template parameter must satisfy.

template <typename InputIterator, typename DataType>
int countOccurences(InputIterator begin, InputIterator end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}

You must be able to dereference iter and equality compare it to val.
Each template parameter must have the operations the function assumes it has.

**InputIterator must support**
- copy assignment (iter = begin)
- prefix operator (++iter)
- comparable to end (begin != end)
- dereference operator (*iter)

**DataType must support**
- comparable to *iter

Nasty compile errors if instantiated type does not support these.
Example

When templates go wrong.
Questions
More practice: what is the implicit interface?

template <typename Collection, typename DataType>
int countOccurences(const Collection<DataType>& list, DataType val) {
    int count = 0;
    for (size_t i = 0; i < list.size(); ++i) {
        if (list[i] == val) ++count;
    }
    return count;
}
More practice: what is the implicit interface?

template <typename Collection, typename DataType>
int countOccurences(const Collection<DataType>& list, 
                      DataType val) {
    int count = 0;
    for (size_t i = 0; i < list.size(); ++i) {
        if (list[i] == val) ++count;
    }
    return count;
}

Collection must have a method size() that returns an integer.
More practice: what is the implicit interface?

template <typename Collection, typename DataType>
int countOccurences(const Collection<DataType>& list, DataType val) {
    int count = 0;
    for (size_t i = 0; i < list.size(); ++i) {
        if (list[i] == val) ++count;
    }
    return count;
}

Collection must support the subscript operator ([ ])
More practice: what is the implicit interface?

template <typename Collection, typename DataType>
int countOccurences(const Collection<DataType>& list, DataType val) {
  int count = 0;
  for (size_t i = 0; i < list.size(); ++i) {
    if (list[i] == val) ++count;
  }
  return count;
}

Furthermore, that return value must be equality comparable to DataType.
Questions
C++20 Concepts: named requirements on the template arguments

```cpp
template <typename It, typename Type>
    requires Input_Iterator<It> && Iterator_of<It> &&
    Equality_comparable<Value_type<It>, Type>
int countOccurences(It begin, It end, Type val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
```

The Standard library has a concepts you can use, or you can write your own!
A concept is a predicate, evaluated at compile-time, that is a part of the interface.

template<typename It, typename Type>
requires Input_Iterator<It> && Iterator_of<It> && Equality_comparable<
Value_type<It>, Type>
int countOccurences(It begin, It end, Type val);

The client can easily see the concepts It and Type must satisfy.
functions and lambdas
Concept Lifting

Looking at the assumptions you place on the parameters, and questioning if they are really necessary.

Can you solve a more general problem by relaxing the constraints?
How many times does the [type] [val] appear in [a range of elements]?

template <typename InputIterator, typename DataType>
int countOccurences(InputIterator begin, InputIterator end, DataType val) {

    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }

    return count;
}
How many times does the \[\text{type}] \[\text{val}] \text{ appear} \text{ in} \ [\text{a range of elements}]? 

```cpp
template <typename InputIterator, typename DataType>
int countOccurences(InputIterator begin, InputIterator end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
```

Let’s look at this part.
How many times does the element satisfy "equal [val]" in [a range of elements]?

```cpp
template <typename InputIterator, typename DataType>
int countOccurences(InputIterator begin, InputIterator end, DataType val) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (*iter == val) ++count;
    }
    return count;
}
```

This is another way to phrase what we are counting.
A predicate is a function which takes in some number of arguments and returns a boolean.

// Unary Predicate (one argument)
bool isEqualTo3(int val) {
    return val == 3;
}

// Binary Predicate (two arguments)
bool isDivisibleBy(int dividend, int divisor) {
    return dividend % divisor == 0;
}
How many times does the element satisfy [predicate] in [a range of elements]?

template <typename InputIterator, typename UnaryPredicate>
int countOccurences(InputIterator begin, InputIterator end, UnaryPredicate predicate) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (predicate(*iter)) ++count;
    }
    return count;
}

“equals [val]” is essentially a predicate function. Let’s further generalize the function.
We can then call this function with a predicate.

```c++
bool isLessThan5(int val) {
    return val < 5;
}

int main() {
    vector<int> vec{1, 3, 5, 7, 9};
    countOccurences(vec.begin(), vec.end(), isLessThan5);
    // returns 2

    return 0;
}
```
We can even make our predicate function a template!

template <typename DataType>
inline bool lessThanTwo(DataType val) { return val < 2; }

template <typename DataType>
inline bool isEven(DataType val) { return val % 2 == 0; }

template <typename DataType>
inline bool isPastMyBedTime(DataType val) {
    return val.hour >= 4;
}
We can even make our predicate function a template!

```cpp
int main() {
    vector<double> vec{1.0, 2.5, 3.0, 2.5, 2.5};
    deque<Time> week4{{{3, 40}, {3, 40}, {1, 20}, {8, 2}}};

    count0ccurences(vec.begin(), vec.end(), lessThan2);
    // returns 2
    count0ccurences(week4.begin(), week4.end(), isPastMyBedTime);
    // returns 1
}
```
Two major problems with this approach (“function pointers”)

```c
int main() {
    vector<double> vec{1.0, 2.5, 3.0, 2.5, 2.5};
    deque<Time> week4{{3, 40}, {3, 40}, {1, 20}, {8, 2}};

    countOccurences(vec.begin(), vec.end(), lessThan2);
    // returns 2
    countOccurences(week4.begin(), week4.end(), isPastMyBedTime);
    // returns 1
}
```
First, it’s kinda annoying that we have to write a separate functions for similar tasks.

```cpp
template <typename DataType>
inline bool lessThanTwo(DataType val) { return val < 2; }

template <typename DataType>
inline bool lessThanThree(DataType val) { return val < 3; }

template <typename DataType>
inline bool lessThanFour(DataType val) { return val < 4; }

template <typename DataType>
inline bool lessThanFive(DataType val) { return val < 5; }
```
Second, what if the function needs information not available at compile time (eg. user input)?

```cpp
template <typename DataType>
inline bool greaterThan(DataType val) {
    return val >= limit;
}

int main() {
    int limit = getInteger(“Minimum for A?”);
    vector<int> grades = readStudentGrades();
    cout << countOccurences(pi.begin(), pi.end(), greaterThan);
}
```

We don’t know limit until the user types it in.
Adding an extra parameter doesn’t work, since the function must be a unary predicate.

template<typename DataType>
inline bool greaterThan(DataType val, DataType limit) {
    return val >= limit;
}

int main() {
    int limit = getInteger("Minimum for A?");
    vector<int> grades = readStudentGrades();
    cout << countOccurences(pi.begin(), pi.end(), greaterThan);
}

Compiler error: greaterThan not a unary predicate function.
Pre-C++11 solution: function objects ("functors")

class GreaterThan {
    public:
        GreaterThan(int limit) : limit(limit) {}
        bool operator()(int val) {return val >= limit};
    private:
        int limit;
}

int main() {
    int limit = getInteger("Minimum for A?");
    vector<int> grades = readStudentGrades();
    GreaterThan func(limit);
    countOccurences(grades.begin(), grades.end(), func);
}
int main() {
    int limit = getInteger("Minimum for A?");
    vector<int> grades = readStudentGrades();
    auto func = [limit](auto val) { return val >= limit;};
    countOccurences(grades.begin(), grades.end(), func);
}
C++11 solution: lambda functions

```cpp
int main() {
    int limit = getInteger("Minimum for A?");
    vector<int> grades = readStudentGrades();
    auto func = [limit](auto val) { return val >= limit;};
    countOccurences(grades.begin(), grades.end(), func);
}
```
Anatomy of a Lambda function.

```cpp
auto func = [capture-clause](parameters) -> return-value {
    // body
};
```
Anatomy of a Lambda function.

auto func = [capture-clause](parameters) -> return-value {
    // body
};

The return-value is optional, if omitted it’s just like an auto return value.
Anatomy of a Lambda function.

auto func = [capture-clause](parameters) {
    // body
};

For lambdas the return value is usually obvious.
Anatomy of a Lambda function.

auto func = [capture-clause](parameters) {
    // body
};

Parameters of the function: you can use auto to templatize the lambda.
Anatomy of a Lambda function.

auto func = [capture-clause](auto val) {
    // body
};

Parameters of the function: you can use auto to templatize the lambda.
Anatomy of a Lambda function.

```cpp
auto func = [capture-clause](auto val) {
    // body
};
```

The body of the function.
Anatomy of a Lambda function.

```
auto func = [capture-clause](auto val) {
    return val >= limit;
};
```

The body of the function.
Anatomy of a Lambda function.

```cpp
int limit = getInteger(“Minimum for A?”);
vector<int> grades = readStudentGrades();

auto func = [capture-clause](auto val) {
    return val >= limit;
};

countOccurences(grades.begin(), grades.end(), func);
```

This however, does not compile, because limit is out of scope.
Anatomy of a Lambda function.

int limit = getInteger(“Minimum for A?”);
vector<int> grades = readStudentGrades();

auto func = [capture-clause](auto val) {
    return val >= limit;
};

countOccurences(grades.begin(), grades.end(), func);

Lambdas are sometimes called closures, because they do not “capture” local variables.
Anatomy of a Lambda function.

```cpp
int limit = getInteger(“Minimum for A?”);
vector<int> grades = readStudentGrades();

auto func = [capture-clause](auto val) {
    return val >= limit;
};

countOccurences(grades.begin(), grades.end(), func);
```

Unless, of course, you ask it to.
Anatomy of a Lambda function.

```cpp
int limit = getInteger("Minimum for A?");
vector<int> grades = readStudentGrades();

auto func = [limit](auto val) {
  return val >= limit;
};

countOccurences(grades.begin(), grades.end(), func);
```

Unless, of course, you ask it to.
int limit = getInteger(“Minimum for A?”);
vector<int> grades = readStudentGrades();

auto func = [limit](auto val) {
    return val >= limit;
};

countOCCurences(grades.begin(), grades.end(), func);

This code works perfectly!
Anatomy of a Lambda function.

```cpp
int limit = getInteger("Minimum for A?");
vector<int> grades = readStudentGrades();

auto func = [limit](auto val) {
    return val >= limit;
};

countOccurences(grades.begin(), grades.end(), func);
```

Sidenote: lambdas generate the class on slide 17 and create an instance of it.
Anatomy of a Lambda function.

```cpp
int limit = getInteger("Minimum for A?");
vector<int> grades = readStudentGrades();

auto func = [limit](auto val) {
    return val >= limit;
};

countOccurences(grades.begin(), grades.end(), func);
```

The problem is that you don’t know the name of the class the compiler created.
Anatomy of a Lambda function.

```cpp
int limit = getInteger("Minimum for A?");
vector<int> grades = readStudentGrades();

auto func = [limit](auto val) {
    return val >= limit;
};

countOccurences(grades.begin(), grades.end(), func);
```

This is where you must* use auto to have the compiler figure out the type.
Questions
You can also capture by reference.

```cpp
set<string> teas{“black”, “green”, “oolong”};
string banned = “boba”; // pls ... this is not a tea
auto likedByAvery = [&teas, banned](auto type) {
    return teas.count(type) && type != banned;
};
```
Not recommended: lazy way of capturing variables.

// capture all by value, except teas is by reference
auto func1 = [=, &teas](parameters) -> return-value {
  // body
};

// capture all by reference, except banned is by value
auto func2 = [&banned](parameters) -> return-value {
  // body
};
algorithms library
You learned how to write a generic function!

template<typename InputIterator, typename UniaryPredicate>
int countOccurences(InputIterator begin, InputIterator end, UniaryPredicate predicate) {
    int count = 0;
    for (auto iter = begin; iter != end; ++iter) {
        if (predicate(*iter)) ++count;
    }
    return count;
}
The STL algorithms library has a highly optimized version of what we wrote!

```
std::count, std::count_if
```

Defined in header `<algorithm>`

```
template< class InputIt, class T >
typename iterator_traits<InputIt>::difference_type
  count( InputIt first, InputIt last, const T &value );
```

(used until C++20)

```
template< class InputIt, class T >
constexpr typename iterator_traits<InputIt>::difference_type
  count( InputIt first, InputIt last, const T &value );
```

(since C++20)

```
template< class ExecutionPolicy, class ForwardIt, class T >
typename iterator_traits<ForwardIt>::difference_type
  count( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, const T &value );
```

(since C++20)

```
template< class InputIt, class UnaryPredicate >
typename iterator_traits<InputIt>::difference_type
  count_if( InputIt first, InputIt last, UnaryPredicate p );
```

(used until C++20)

```
template< class InputIt, class UnaryPredicate >
constexpr typename iterator_traits<InputIt>::difference_type
  count_if( InputIt first, InputIt last, UnaryPredicate p );
```

(since C++20)

```
template< class ExecutionPolicy, class ForwardIt, class UnaryPredicate >
typename iterator_traits<ForwardIt>::difference_type
  count_if( ExecutionPolicy&& policy, ForwardIt first, ForwardIt last, UnaryPredicate p );
```

(since C++17)
Let’s perform some tasks on a list of students!

```c++
struct Student {
    string name;
    Year year;
    double average;
};
```
Generic tasks: given a list of students...

- Find the highest/lowest scoring student.
- Calculate the average class grade.
- Find the median scoring student.
- Did anyone fail?
- Create separate vectors of freshmen and non-freshmen.
- Print all students (comma separated), sorted by year then by name.
- Find all students in 106L who’s already taken 106B.
Example

Solving common problems using algorithms.
Operating over containers without a single loop.
Find the highest/lowest scoring student

auto compAvg = [](const Student& s1, const Student& s2) {
    return s1.average < s2.average;
};

auto [min, max] =
    std::minmax_element(begin(students), end(students), compAvg);

Student minStudent = *min;
Student maxStudent = *max;
Calculate the average class grade.

```cpp
auto sumAvg = [](const auto& val, const auto& sum) {
    return val.average + sum;
};

double average = std::accumulate(begin(students), end(students), sumAvg)/students.size();
```
Calculate the median class grade.
O(Nlog N)

```cpp
auto compAvg = [](const Student& s1, const Student& s2) {
    return s1.average < s2.average;
};

int size = students.size();

// O(N log N) sort
std::sort(begin(students), end(students), compAvg);

Student medianStudent = students[size/2];
```
auto failed = [](const Student& s) { return s.average < 60; };

bool someoneFailed = 
    std::any_of(begin(students), end(students), failed);
Which student who failed?

```cpp
auto failed = [](const Student& s) { return s.average < 60; }
;

bool someoneFailed = 
    std::any_of(begin(students), end(students), failed);

Student student = 
    *std::find(begin(students), end(students), failed);
```
Let’s curve everyone’s grades!

// max and median are students from the previous slide
auto jerryCurve = [max, median] (const Student& s) {
  double slope = (max.average - median.average)/0.2;
  double grade = slope * (s.average - 0.8) + 0.8;
  return Student{s.name, s.year, grade};
};

std::transform(begin(students), end(students),
               begin(students), jerryCurve);
Sort everyone by year then name.

```cpp
auto comp = [] (const Student& s1, const Student& s2) {
  return s1.year < s2.year ||
  (s1.year == s2.year && s1.name < s2.name);
};

std::sort(begin(students), end(students), comp);
```
Then print everyone in a comma-separated list.

```cpp
auto comp = [] (const Student& s1, const Student& s2) {
  return s1.year < s2.year ||
      (s1.year == s2.year && s1.name < s2.name);
};

std::sort(begin(students), end(students), comp);

auto joinCommas = [] (const Student& s, const string& current) {
  return current + "", " + s.name;
}

cout << std::accumulate(begin(students)+1, end(students),
                         students[0], joinCommas);
```
Create separate vectors of freshmen and non-freshmen.

```cpp
auto isFreshman = [] (const Student& s) {
    return s.year == Year::FRESHMEN;
};

auto partition_point = 
    stable_partition(begin(students), end(students), isFreshmen);

vector<Student> freshmen(begin(students), partition_point);
vector<Student> notFreshmen(partition_point, end(students));
```
Questions
What does stable partition do?

FROSH 1
NOT FROSH 2
NOT FROSH 3
FROSH 4
NOT FROSH 5
NOT FROSH 6
FROSH 7
After a call to `stable_partition`:

- **FROSH 1**
- **FROSH 4**
- **FROSH 7**
- **NOT FROSH 2**
- **NOT FROSH 3**
- **NOT FROSH 5**
- **NOT FROSH 6**

**Return value:** partition point

**Stable:** order preserved within “frosh” and “not-frosh” group.
Why use algorithms?

- Abstraction: perform algorithms without looking at elements.
- Generic: operations are based on ranges, not containers.
- Correct: heavily tested, most definitely correct.
- Heavily optimized: performs optimizations using features we haven’t/won’t even learn about.
Questions
Iterator Adaptors
Copy all students of a certain year into a new vector.

```cpp
Year yearToFind = /* something */;
vector<Student> studentsInYear;
auto isInYear = [yearToFind](const Student& s) {
    return s.year == yearToFind;
}

// this doesn't work!
copy_if(begin(students), end(students),
    begin(studentsInYear), isInYear);
```
Copy all students of a certain year into a new vector.

begin
FROSH 1  NOT FROSH 2  FROSH 3  FROSH 4  FROSH 5  NOT FROSH 6  FROSH 7
end

begin
uninitialized memory

Note: sometimes vector’s have more space than required.
Copy all students of a certain year into a new vector.

Let’s run copy_if!
Copy all students of a certain year into a new vector.

begin
FROSH 1
NOT FROSH 2
FROSH 3
FROSH 4
FROSH 5
NOT FROSH 6
FROSH 7
end

Let’s run copy_if!
uninitialized memory
Copy all students of a certain year into a new vector.

begin

FROSH 1
NOT FROSH 2
FROSH 3
FROSH 4
FROSH 5
NOT FROSH 6
FROSH 7

end

uninitialized memory

Let’s run copy_if!
Copy all students of a certain year into a new vector.

begin

FROSH 1
NOT FROSH 2
FROSH 3
FROSH 4
FROSH 5
NOT FROSH 6
FROSH 7

end

begin

FROSH 1
FROSH 3

uninitialized memory

Let’s run copy_if!
Copy all students of a certain year into a new vector.

Whoops, we wrote into uninitialized memory!
Copy all students of a certain year into a new vector.

begin
FROSH 1  NOT FROSH 2  FROSH 3  FROSH 4  FROSH 5  NOT FROSH 6  FROSH 7
end

begin
FROSH 1  FROSH 3  FROSH 4  FROSH 5

Whoops, we wrote into uninitialized memory!

13 October 2019
We need a special iterator which extends the container.

```cpp
Year yearToFind = /* something */;
vector<Student> studentsInYear;
auto isInYear = [yearToFind] (const Student& s) {
    return s.year == yearToFind;
}

// this now works
copy_if(begin(students), end(students),
    back_inserter(studentsInYear), isInYear);
```
We need a special iterator which extends the container.

Let's run the fixed version
We need a special iterator which extends the container.

Let’s run the fixed version
We need a special iterator which extends the container.

Let’s run the fixed version
Another example: set operations

```cpp
vector<Student> LStudents = readStudents("cs106l.txt");
vector<Student> BStudents = readStudents("cs106b.txt");
std::sort(begin(LStudents), end(LStudents), compAvg);
std::sort(begin(BStudents), end(BStudents), compAvg);
vector<Student> studentsWithoutMidterms;

std::set_difference(begin(LStudents), end(LStudents), begin(BStudents), end(BStudents),
back_inserter(studentsWithoutMidterms));
```
Example

Stream iterators with algorithms. Unnecessary censorship and *Hamilton*.
Questions
Ranges (C++20)

Kinda overflow.
Create a sorted range without highest and lowest two value.

vector<int> v{21, 1, 3, 8, 13, 1, 5, 2};
vector<int> v2 = v;
sort(begin(v2), end(v2));
auto first = begin(v2);
advance(first, 2);
auto last = first;
advance(last, size(v2) - 4);
v2.erase(last, end(v2));
v2.erase(begin(v2), first);

Let’s run the fixed version
Raising abstraction: we are

```cpp
vector<Student> students = readStudents("cs106l.txt");

for (auto& s : students) {
  // do something with s
}
```
What if we wanted to ignore the last student?

```cpp
vector<Student> students = readStudents("cs106l.txt");

for (auto iter = begin(students); iter != end(students) - 1; ++iter) {
   // do something with *iter
}
```
What if we wanted to ignore the last student?

```cpp
vector<Student> students = readStudents("cs106l.txt");

for (auto& s : students) {
    // do something with s
}
```

That’s a problem!
What if we wanted to ignore the last student?

```cpp
vector<Student> students = readStudents("cs106l.txt");

for (auto& s : students) {
    // do something with s
}
```

That’s a problem!
Next time

STL Summary