Templates
Warning: we will not use slides for this lecture. This is all meant as a reference.
Summary of Feedback

Likes
• engaging lectures
• slides look nice
• we’re “nice”
• class is really chill
• amount of questions

Dislikes
• too fast/too much content
• too many slides
• want more practice
• want interactive lectures
• amount of questions
Key Changes

- we’ll post 1-2 practice exercises on Piazza after lecture.
- content/slides will be scaled back by ~30%.
- practice time: can choose between asking questions, or working on the practice exercises.
Game Plan

- template functions
- varadic templates
- concept lifting
- implicit interfaces & concepts
Write a minmax function which returns a pair \{\text{min}, \text{max}\} of parameters.

```cpp
int main() {
    auto [min, max] = my_minmax(4, 7);
    cout << min << endl; // 4
    cout << max << endl; // 7
}
```
Write a minmax function which returns a pair {min, max} of parameters.

```cpp
pair<int, int> my_minmax(int a, int b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```
What if we wanted to handle different types?

```cpp
int main() {
    auto [min, max] = my_minmax(4.2, -7.9);
    cout << min << endl; // -7.9
    cout << max << endl; // 4.2
}
```
What if we wanted to handle different types?

```cpp
int main() {
    auto [min, max] = my_minmax("Anna", "Avery");
    cout << min << endl; // Avery
    cout << max << endl; // Anna
}
```
template functions
Classic C-solution: write separate functions.

```c
pair<int, int> my_minmax_int(int a, int b) {
    if (a < b) return {a, b};
    else return {b, a};
}

pair<double, double> my_minmax_double(double a, double b) {
    if (a < b) return {a, b};
    else return {b, a};
}

pair<string, string> my_minmax_string(string a, string b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```

Problem: each function has a different name!
Slightly better: overloaded functions.

```cpp
pair<int, int> my_minmax(int a, int b) {
    if (a < b) return {a, b};
    else return {b, a};
}

pair<double, double> my_minmax(double a, double b) {
    if (a < b) return {a, b};
    else return {b, a};
}

pair<string, string> my_minmax(string a, string b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```

Problem: you have to write a function for every single type.
Slightly better: overloaded functions.

```cpp
pair<int, int> my_minmax(int a, int b) {
    if (a < b) return {a, b};
    else return {b, a};
}

pair<double, double> my_minmax(double a, double b) {
    if (a < b) return {a, b};
    else return {b, a};
}

pair<string, string> my_minmax(string a, string b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```

Bigger problem: how do you handle user defined types?
An observation: the highlighted parts are identical.

```cpp
pair<int, int> my_minmax(int a, int b) {
    if (a < b) return {a, b};
    else return {b, a};
}

pair<double, double> my_minmax(double a, double b) {
    if (a < b) return {a, b};
    else return {b, a};
}

pair<string, string> my_minmax(string a, string b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```
Only the types are different.

```cpp
pair<int, int> my_minmax(int a, int b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```

```cpp
pair<double, double> my_minmax(double a, double b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```

```cpp
pair<string, string> my_minmax(string a, string b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```
Let’s write a general form in terms of a type T.

```cpp
pair<T, T> my_minmax(T a, T b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```
Let’s write a general form in terms of a type T.

```cpp
pair<T, T> my_minmax(T a, T b) {
    if (a < b) return {a, b};
    else return {b, a};
}
```
We have a generic function!

```cpp
def my_minmax(T a, T b):
    if a < b:
        return (a, b)
    else:
        return (b, a)
```

pair<T, T> my_minmax(T a, T b) {
    if (a < b) return {a, b};
    else return {b, a};
}
template <typename T>
pair<T, T> my_minmax(T a, T b) {
    if (a < b) return {a, b};
    else return {b, a};
}
Explicit instantiation: specify the type T.

```cpp
int main() {
    auto [min1, max1] = my_minmax<double>(4.2, -7.9);
    auto [min2, max2] = my_minmax<string>("Avery", "Anna");
    auto [min3, max3] = my_minmax<int>(3, 3);
    auto [min4, max4] = my_minmax<double>(2, 2.3);
    auto [min5, max5] = my_minmax<vector<int>>({1, 2}, {3, 1});
}
```
Let’s walk through what the compiler does!

```cpp
int main() {
    auto [min1, max1] = my_minmax<double>(4.2, -7.9);
    auto [min2, max2] = my_minmax<string>("Anna", "Avery");
    auto [min3, max3] = my_minmax<int>(3, 3);
    auto [min4, max4] = my_minmax<double>(2, -6.2);
    auto [min5, max5] = my_minmax<vector<int>>({1, 2}, {3, 1});
}
```

We have a template that looks like that!
Let's walk through what the compiler does!

```cpp
int main() {
    auto [min1, max1] = my_minmax<double>(4.2, -7.9);
    auto [min2, max2] = my_minmax<string>("Anna", "Avery");
    auto [min3, max3] = my_minmax<int>(3, 3);
    auto [min4, max4] = my_minmax<double>(2, -6.2);
    auto [min5, max5] = my_minmax<vector<int>>({1, 2}, {3, 1});
}
```

Let's replace the T's.
We have our function!
And just in case the type is a large collection.

template <typename T>
pair<T, T> my_minmax(T a, T b) {
    if (a < b) return {a, b};
    else return {b, a};
}
And just in case the type is a large collection.

template<typename T>
pair<T, T> my_minmax(const T& a, const T& b) {
    if (a < b) return {a, b};
    else return {b, a};
}
Your turn: make this function generic!

```cpp
int getInteger(const string& prompt, const string& reprompt) {
    while (true) {
        cout << prompt;
        string line; int result; char extra;
        if (!getline(cin, line))
            throw domain_error("[shortened]");
        istringstream iss(line);
        if (iss >> result && !(iss >> extra)) return result;
        cout << reprompt << endl;
    }
}
```
Your turn: make this function generic!

template <typename T>
T getInteger(const string& prompt, const string& reprompt) {
    while (true) {
        cout << prompt;
        string line; T result; char extra;
        if (!getline(cin, line))
            throw domain_error("[shortened]");
        istringstream iss(line);
        if (iss >> result && !(iss >> extra)) return result;
        cout << reprompt << endl;
    }
}
(optional)

varadic templates

skipped during lecture, but really cool.

C++11
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?
Why write generic functions?

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.
How many times does the integer [val] appear in a vector of integers?

```cpp
template <>
int countOccurrences(const vector<int>& vec,
                     int val) {

    int count = 0;
    for (size_t i = 0; i < vec.size(); ++i) {
        if (vec[i] == val) ++count;
    }
    return count;
}
```

What unnecessary assumption does the function make?
How many times does the integer [val] appear in a vector of integers?

```cpp
template <>
int countOccurrences(const vector<int>& vec, int val) {
    int count = 0;
    for (size_t i = 0; i < vec.size(); ++i) {
        if (vec[i] == val) ++count;
    }
    return count;
}
```

What unnecessary assumption does the function make?
How many times does the [type] [val] appear in a vector of [type]?

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

What unnecessary assumption does the function make?
How many times does the [type] [val] appear in a vector of [type]?

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

What unnecessary assumption does the function make?
How many times does the [type] [val] appear in a [collection] of [type]?

```cpp
template <typename Collection, typename DataType>
int countOccurrences(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;
}
```

This code does not work. Why?
How many times does the [type] [val] appear in a [collection] of [type]?

```cpp
list<int> list = {1.1, 3.14, 3.14, 3.14, 1.1};
int count = countOccurrences(list, 3.14);
```

Sample code calling our function that won’t work.
How many times does the [type] [val] appear in a [collection] of [type]?

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;
}

We are indexing through a potentially unindexable collection.
Recall: iterators offer a “standardized” way of traversing a container.

```cpp
for (auto iter = container.begin(); iter != container.end(); ++iter) {
    cout << *iter << '\n';
}
```

No matter what container is, this prints the elements of that container.
How many times does the [type] [val] appear in a [collection] of [type]?

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

Solved using iterators!
How many times does the [type] [val] appear in a [collection] of [type]?

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

This still makes one last assumption.
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;
}
Why write generic functions?

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<int>.
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
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<int>.
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
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<int>.
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
countOccurences(v.begin(), v.end(), 4.7);
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<int>.
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
countOccurences(v.begin(), v.end(), 4.7);
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<int>.
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
countOccurences(v.begin(), v.end(), 4.7);
countOccurences(s.begin(), s.end(), 'X');
Count how many times ‘X’ appears in a deque<char>.
Count how many times 5 appears in the second half of a list<int>.
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
countOccurences(v.begin(), v.end(), 4.7);
countOccurences(s.begin(), s.end(), ‘X’);
Count how many times ‘X’ appears in a deque<char>.
Count how many times 5 appears in the second half of a list<int>.
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
countOccurences(v.begin(), v.end(), 4.7);
countOccurences(s.begin(), s.end(), ‘X’);
countOccurences(d.begin(), d.end(), ‘X’);
Count how many times 5 appears in the second half of a list<int>.
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
countOccurences(v.begin(), v.end(), 4.7);
countOccurences(s.begin(), s.end(), ‘X’);
countOccurences(d.begin(), d.end(), ‘X’);

Count how many times 5 appears in the second half of a list<int>.
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
countOccurences(v.begin(), v.end(), 4.7);
countOccurences(s.begin(), s.end(), ‘X’);
countOccurences(d.begin(), d.end(), ‘X’);
countOccurences((l.begin()+l.end())/2, l.end(), 5);
Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
countOccurences(v.begin(), v.end(), 4.7);
countOccurences(s.begin(), s.end(), ‘X’);
countOccurences(d.begin(), d.end(), ‘X’);
countOccurences((l.begin()+l.end())/2, l.end(), 5);

Count how many elements in the second half of a list<int> are at most 5.
Why write generic functions?

countOccurences(v.begin(), v.end(), 3);
countOccurences(v.begin(), v.end(), 4.7);
countOccurences(s.begin(), s.end(), ‘X’);
countOccurences(d.begin(), d.end(), ‘X’);
countOccurences((l.begin()+l.end())/2, l.end(), 5);

Count how many elements in the second half of a list<int> are at most 5.

We’ll tackle this one next time!
Your turn:
lift this function to its most generic form.

```cpp
int main() {
    vector<int> v1{1, 2, 3, 4};
    vector<int> v2{1, 2, 4, 6};
    vector<int> v3{1, 2, 3, 4};
    vector<int> v4{1, 2, 3};

    auto [match, l1, l2] = mismatch(v1, v2); // {false, 3, 4}
    auto [match, r1, r3] = mismatch(r1, r3); // {true, 0, 0}
    auto [match, k1, k4] = mismatch(k1, k4); // undefined
}
```
Your turn:
lift this function to its most generic form.

tuple<bool, int, int> mismatch(const vector<int>& vec1, 
const vector<int>& vec2)
size_t i = 0;
while (i < vec1.size() && vec1[i] == vec2[i]){
    ++i;
}  
if (i == vec1.size()) return {true, 0, 0};
else return {false, vec1[i], vec2[i]};
Your turn:

lift this function to its most generic form.

```cpp
tuple<bool, int, int> mismatch(const vector<int>& vec1, const vector<int>& vec2)
{
    size_t i = 0;
    while (i < vec1.size() && vec1[i] == vec2[i]){
        ++i;
    }
    if (i == vec1.size()) return {true, 0, 0};
    else return {false, vec1[i], vec2[i]};
}
```

Can you get rid of the boolean?
Your turn:
lift this function to its most generic form.

```cpp
template <??>
pair<??, ??> mismatch(???)
```
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};
}
Implicit Interfaces and Concepts
The compiler literally replaces each template parameter with whatever you instantiate it with.

Suppose I write the code above.

```cpp
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.

```cpp
template <typename InputIterator, typename DataType>
int countOccurrences(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.

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.

```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 problem is here: *iter has type int, and can’t be compared to an iterator.
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;
}
```

What must be true of `InputIterator` and `DataType`?
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;
}

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

```c++
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;
}
```

`iter` must be equality comparable to `end`. 
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`. 

11 October 2019
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;
}
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.
More practice: what is the implicit interface?

template <typename Collection, typename DataType>
int countOccurrences(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;
}
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;
}
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;
}
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;
}
Example

When templates go wrong.
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.

```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);
```

The client can easily see the concepts `It` and `Type` must satisfy.
Further Reading on Concepts

Next time

Functions and Algorithms