Template Functions

CS 106L, Fall ‘21

How do we write functions that are data type-agnostic?

Today's agenda

- Recap: iterators and template classes
- Generic programming
- Template functions
  - Type deduction
    - rvalues, lvalues, and references
  - Template metaprogramming (TMP)
    - Type traits
- Template function practice (motivating lambdas)

STL Iterators

- Iterators are objects that point to elements inside containers.
- Each STL container has its own iterator, but all of these iterators exhibit a similar behavior!
- Generally, STL iterators support the following operations:

```
std::set<type> s = {0, 1, 2, 3, 4};
std::set::iterator iter = s.begin();          // at 0
++iter;                                       // move iterator by 1 spot; at 1
*iter;                                        // get element at iterator; 1
(iter != s.end());                            // can compare iterator equality
auto second_iter = iter;                      // "copy construction"
```

Iterator documentation:
https://www.cplusplus.com/reference/iterator/

Looping over collections

How do we access and print elements in collections?

```
std::set<int> set{3, 1, 4, 1, 5, 9};
for (initialization; termination condition; increment) {
  const auto& elem = retrieve element;
  cout << elem << endl;
}

std::map<int> map{{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (initialization; termination condition; increment) {
  const auto& [key, value] = retrieve element at index;      // structured binding!
  cout << key << ":" << value << ", " << endl;
}
```

Looping over collections

How do we access and print elements in collections?

```
std::set<int> set{3, 1, 4, 1, 5, 9};
for (auto iter = set.begin(); iter != set.end(); ++iter) {
  const auto& elem = *iter;
  cout << elem << endl;
}

std::map<int> map{{1, 6}, {1, 8}, {0, 3}, {3, 9}};
for (auto iter = map.begin(); iter != map.end(); ++iter) {
  const auto& [key, value] = *iter;                    // structured binding!
  cout << key << ":" << value << ", " << endl;
}
```

Template Classes

A class that is parametrized over some number of types.

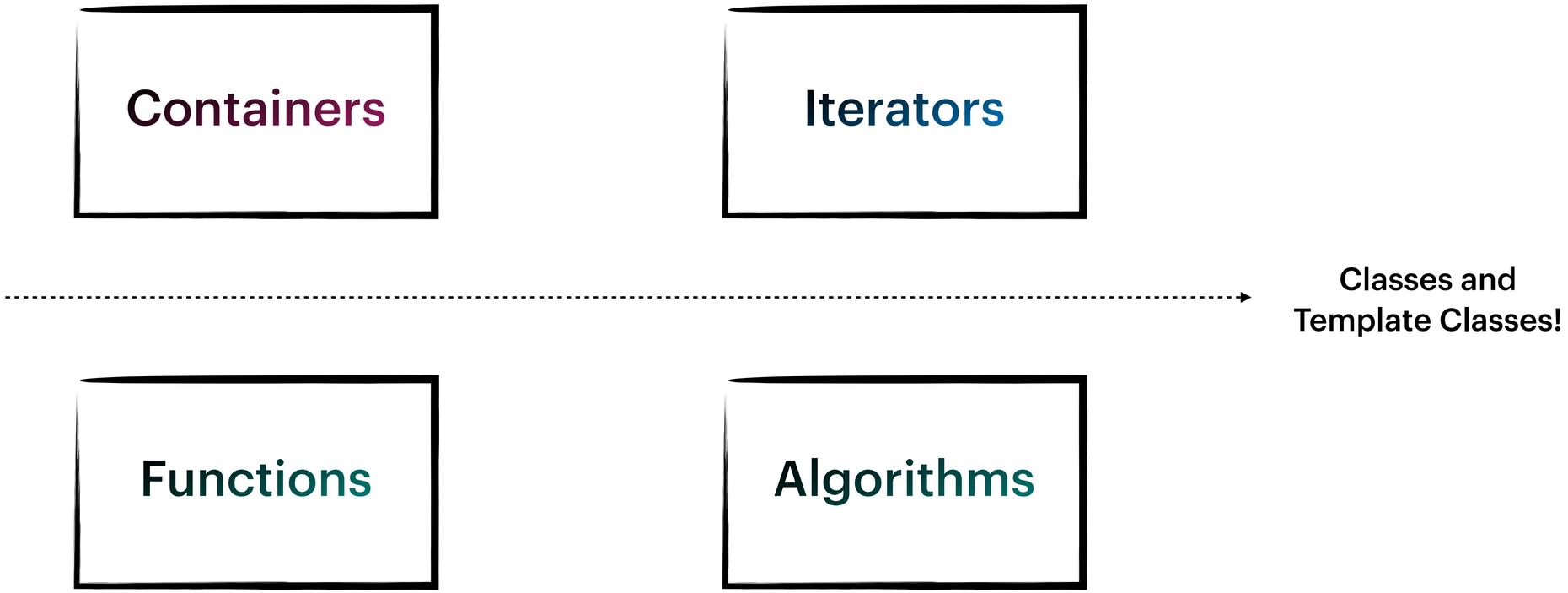
A class that is comprised of member variables of a general type/types.

```
template<class valueType>
RealVector<valueType>::RealVector()
{
    elems = new
valueType[kInitialSize];
    logical_size = 0;
    array_size = kInitialSize;
}
```

```
template<class valueType> class RealVector {
public:
    //type aliases: this is how the iterator works!
    using const_iterator = const valueType*;

    //initial size for our rev
    RealVector(size_t n, const valueType& val);
    //destructor
    ~RealVector();
…
```

So why did we take a break from the STL?



Generic Programming

Writing reusable, unique code with no duplication!

Generic C++

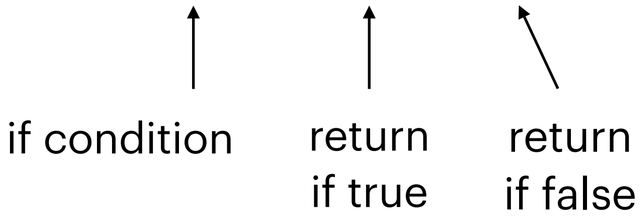
- Allow data types to be parameterized (C++ entities that work on any datatypes)
- Template classes achieve generic classes
- What about functions?
  - **How can we write methods that work on any data type?**

Let's write a function to get the min of two ints!

```
int myMin(int a, int b) {
  return a < b ? a : b;
}
```

Let's write a function to get the min of two ints!

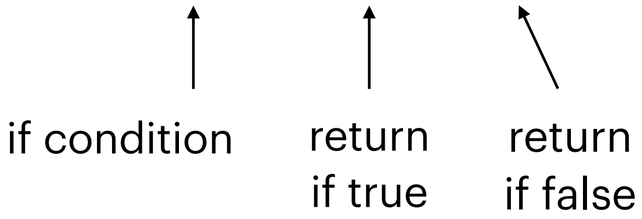
```
int myMin(int a, int b) {
  return a < b ? a : b;
}
```



sidenote: ternary operators

Let's write a function to get the min of two ints!

```
int myMin(int a, int b) {
  return a < b ? a : b;
}
```



sidenote: ternary operators

```
// equivalently,
int myMin(int a, int b) {
  if (a < b) {
    return a;
  }
  return b;
}
```

Can we handle different types?

```
int myMin(int a, int b) {
  return a < b ? a : b;
}

int main() {
  auto min_int = myMin(1, 2);
  auto min_name = myMin("Sathya", "Frankie");
}
```

Can we handle different types?

```
int myMin(int a, int b) {
  return a < b ? a : b;
}

int main() {
  auto min_int = myMin(1, 2);                    // 1
  auto min_name = myMin("Sathya", "Frankie");
}
```

Can we handle different types?

```
int myMin(int a, int b) {
  return a < b ? a : b;
}

int main() {
  auto min_int = myMin(1, 2);                    // 1
  auto min_name = myMin("Sathya", "Frankie");    // error!
}
```

One solution: overloaded functions

```
int myMin(int a, int b) {
  return a < b ? a : b;
}

// exactly the same except for types
std::string my_min(std::string a, std::string b) {
  return a < b ? a : b;
}

int main() {
  auto min_int = myMin(1, 2);                    // 1
  auto min_name = myMin("Sathya", "Frankie");    // Frankie
}
```

Overloading is having multiple functions of the same name, but with different return types and parameter types!

One solution: overloaded functions

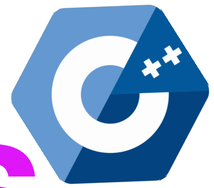
```
int myMin(int a, int b) {
  return a < b ? a : b;
}

// exactly the same except for types
std::string my_min(std::string a, std::string b) {
  return a < b ? a : b;
}

int main() {
  auto min_int = myMin(1, 2);                    // 1
  auto min_name = myMin("Sathya", "Frankie");    // Frankie
}
```

But what about comparing other data types, like doubles, characters, and complex objects?

Template Functions



Writing reusable, unique code with no duplication!

We can write generic, "template" functions!

```
template <typename Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}
```

We can write generic, "template" functions!

Declares that the next function declaration is a template

Specifies that "Type" is some ***generic type***

List of template arguments (types)

```
template <typename Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}
```

We can write generic, "template" functions!

Declares that the next function declaration is a template

Specifies that "Type" is some ***generic type***

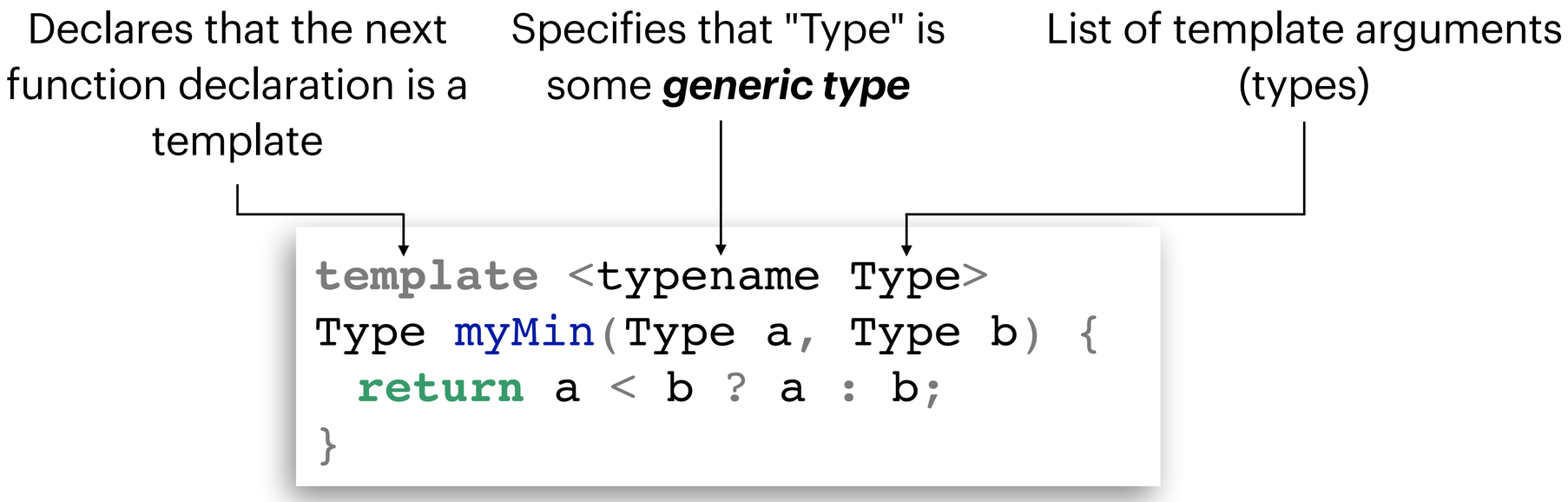
List of template arguments (types)

```
template <typename Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}
```

```
template <class Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}
```

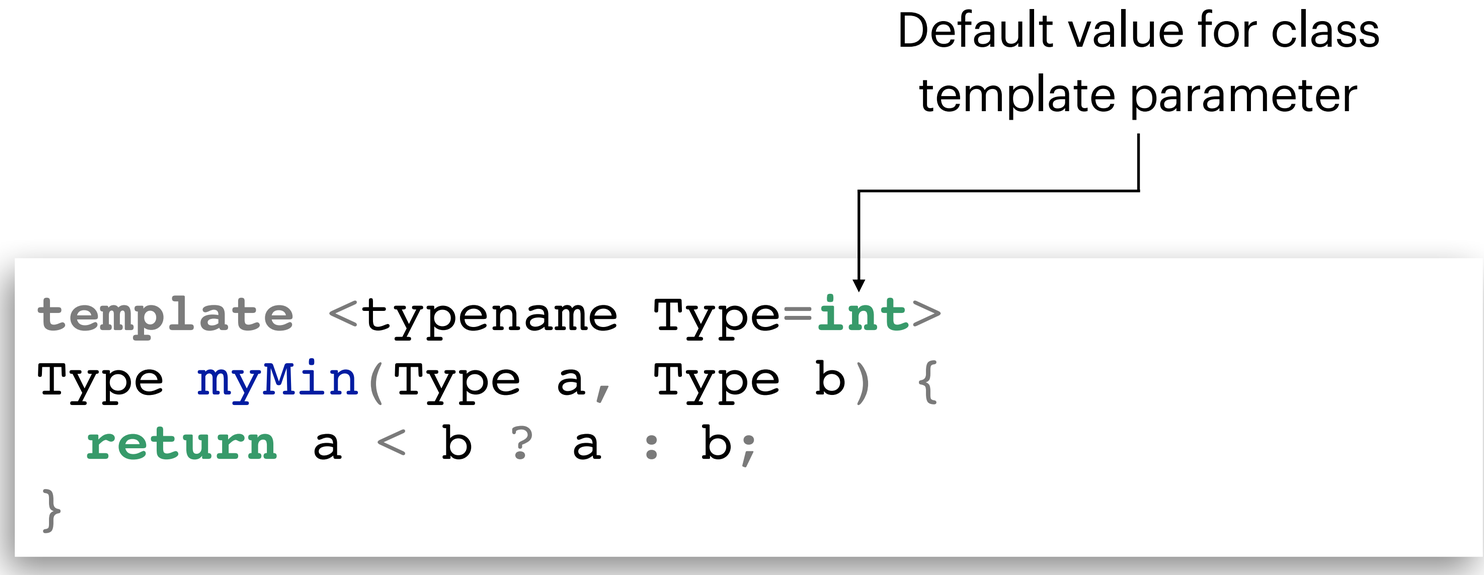
Here, "class" is an alternative keyword to typename. They're 100% equivalent in template function declarations!

We can write generic, "template" functions!



Scope of template arguments (e.g. Type) is limited to just this one function!

We can write generic, "template" functions!



So how do we use these functions?

Calling template functions

```
template <typename Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}

// int main() {} will be omitted from future examples
// we'll instead show the code that'd go inside it
cout << myMin<int>(3, 4) << endl; // 3
```


Explicit template parameter (Type)

- Template functions can be called (instantiated) implicitly or explicitly
- Template parameters can be explicitly provided or implicitly deduced

Calling template functions

let compiler deduce return type

```
template <typename T, typename U>
auto smarterMyMin(T a, U b) {
  return a < b ? a : b;
}

// int main() {} will be omitted from future examples
// we'll instead show the code that'd go inside it
cout << myMin(3.2, 4) << endl; // 3.2
```

Template parameter deduced from function parameters

- Template functions can be called (instantiated) implicitly or explicitly
- Template parameters can be explicitly provided or implicitly deduced

Let's review r-/lvalues and references

```
bool makeEven (int& x) {
  // make number even and return whether or not original number was odd.
  bool wasOdd = x % 2;
  x *= 2;
  return wasOdd;
}

int x = 5;
bool wasOdd = f(x);
```

Let's review r-/lvalues and references

```
bool makeEven (int& x) {
  // make number even and return whether or not original number was odd.
  bool wasOdd = x % 2;
  x *= 2;
  return wasOdd;
}

int x = 5;
bool wasOdd = f(x);
```

- lvalues are underlined
- if we change x, its creator (the lines below) will be affected

- lvalues
  - anything on the ***L***eft side of an equal sign
  - an object whose resource can't be reused without consequence
    - can't be changed without affecting creator

Let's review r-/lvalues and references

```
bool makeEven (int& x) {
  // make number even and return whether or not original number was odd.
  bool wasOdd = x % 2;
  x *= 2;
  return wasOdd;
}

int x = 5;
bool wasOdd = f(x);
```

- rvalues are underlined
- if we change 2, nothing will be affected.

- rvalues
  - anything on the ***R***ight side of an equal sign
  - an object whose resource can be reused at no cost to creator

Let's review r-/lvalues and references

- lvalue references - any reference to an lvalue (&)
- rvalue references - any reference to an rvalue (&&) (will be explained in our move semantics lecture)
- Pointers are objects that store a reference

```
int x = 5;
&x                    // lvalue reference
int* pointer = &x
```

Template type deduction - case 1

- If the template function parameters are regular, pass-by-value parameters:
  1. Ignore the "&"
  2. After ignoring "&", ignore const too.

```
template <typename Type>
Type addFive(Type a) {
  return a + 5;                          // only works for types that support "+"
}

int a = 5;
addFive(a);                              // Type is int
const int b = a;
addFive(b);                              // Type is still int
const int& c = a;
addFive(c);                              // even now, Type is still int
```

Template type deduction - case 2

- If the template function parameters are references or pointers, this is how types (e.g. Type) are deduced:
    1. Ignore the "&"
    2. Match the type of parameters to inputted arguments
    3. Add on const after

```
template <typename Type>
void makeMin(const Type& a, const Type& b, Type& minObj) {
  // set minObj to the min of a and b instead of returning.
  minObj = a < b ? a : b;
}

const int a = 20;
const int& b = 21;
int c;
myMin(a, b, c);              // Type is deduced to be int
cout << c << endl;          // 20
```

a and b are references to const values

Template Functions Demo

Template Functions: syntax and initialization

Template Functions (behind the scenes)

- Normal functions are created during compile time, and used in runtime.
- Template functions are not compiled until used by the code.

```
template <typename Type>
Type myMin(Type a, Type b) {
  return a < b ? a : b;
}
cout << myMin(3, 4) << endl; // 3
```

- The compiler deduces the parameter types and generates a unique function specifically for each time the template function is called.
- After compilation, the compiled code looks ***as if*** you had written ***each instantiated version*** of the function yourself.

Template Functions (behind the scenes)

- To recap instantiation, when you call a template function, either:
  - for explicit instantiation, compiler creates a function in the executable that matches the initial function call's template type parameters
  - for implicit instation, the compiler does the same,

What other types of code can we run during compilation?

Template Metaprogramming

Writing code that runs during compilation (instead of run time)

What is TMP?

- Normal code runs during run time.
- TMP -> run code during compile time
  - make compiled code packages smaller
  - speed up code when it's actually running

What is TMP?

- Normal code runs during run time.
- TMP -> run code during compile time
  - make compiled code packages smaller
  - speed up code when it's actually running

```
template<unsigned n>
struct Factorial {
   enum { value = n * Factorial<n - 1>::value };
};

template<> // template class "specialization"
struct Factorial<0> {
   enum { value = 1 };
};

std::cout << Factorial<10>::value << endl; // prints 3628800, but run during compile time!
```

How can TMP actually be used?

- TMP was actually discovered (not invented, discovered) recently!
- Where can TMP be applied?
  - Ensuring dimensional unit correctness
  - Optimizing matrix operations
  - Generating custom design pattern implementation
    - policy-based design (templates generating their own templates)

Why write generic functions?

Let's get some practice with making template functions by solving a problem you may see in the future!

Why write generic functions?

Count the # of times 3 appears in a `std::vector<int>`.
Count the # of times "Y" appears in a `std::istream`.
Count the # of times 5 appears in the second half of a `std::deque<int>`.
Count the # of times "X" appear in the second half of a `std::string`.

Why write generic functions?

Count the # of times 3 appears in a `std::vector<int>`.
Count the # of times "Y" appears in a `std::istream`.
Count the # of times 5 appears in the second half of a `std::deque<int>`.
Count the # of times "X" appear in the second half of a `std::string`.

By using generic functions, **we can solve each of these problems with a single function**!

Counting Occurrences: Attempt 1

```
int count_occurrences(std::vector<std::string> vec, std::string target){
    int count = 0;
    for (size_t i = 0; i < vec.size(); ++i){
        if (vec[i] == target) count++;
    }
    return count;
}
```

Usage: `count_occurrences({"Xadia", "Drakewood", "Innean"}, "Xadia");`

Counting Occurrences: Attempt 1

```
int count_occurrences(std::vector<std::string> vec, std::string target){
    int count = 0;
    for (size_t i = 0; i < vec.size(); ++i){
        if (vec[i] == target) count++;
    }
    return count;
}


Usage: count_occurrences({"Xadia", "Drakewood", "Innean"}, "Xadia");
```

🤔 What if we wanted to generalize this beyond just strings?

Counting Occurrences: Attempt 2

```
template <typename DataType>
int count_occurrences(const std::vector<DataType> vec, DataType target){
    int count = 0;
    for (size_t i = 0; i < vec.size(); ++i){
        if (vec[i] == target) count++;
    }
    return count;
}
```

Usage: `count_occurrences({"Xadia", "Drakewood", "Innean"}, "Xadia");`

Counting Occurrences: Attempt 2

```
template <typename DataType>
int count_occurrences(const std::vector<DataType> vec, DataType target){
    int count = 0;
    for (size_t i = 0; i < vec.size(); ++i){
        if (vec[i] == target) count++;
    }
    return count;
}

Usage: count_occurrences({"Xadia", "Drakewood", "Innean"}, "Xadia");
```

🤔 What if we wanted to generalize this beyond just vectors?

Counting Occurrences: Attempt 3

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

Usage: count_occurrences({"Xadia", "Drakewood", "Innean"}, "Xadia");
```

🤔 What's wrong with this?

Exactly! The collection may not be indexable. How do we solve this?

Counting Occurrences: Attempt 4

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType target){
    int count = 0;
    for (initialization; end-condition; increment){
        if (element access == target) count++;
    }
    return count;
}

vector<std::string> lands = {"Xadia", "Drakewood", "Innean"};
Usage: count_occurrences(arg1, arg2, "Xadia");
```

Counting Occurrences: Attempt 4

```
template <typename InputIt, typename DataType>
int count_occurrences(InputIt begin, InputIt end, DataType target){
    int count = 0;
    for (auto iter = begin; iter != end; ++iter){
        if (*iter == target) count++;
    }
    return count;
}

vector<std::string> lands = {"Xadia", "Drakewood", "Innean"};
Usage: count_occurrences(lands.begin(), lands.end(), "Xadia");
```

Nice work!

We manually pass in ***begin*** and ***end*** so that we can customize our search bounds.

count_occurrences demo

Template Functions: syntax and initialization

Are we done?

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

Usage: std::string str = "Xadia";
       count_occurrences(str.begin(), str.end(), 'a');
```

Could we reuse this to find how many vowels are in “Xadia”, or how many odd numbers were in a std::vector<int>?

Lambdas and STL Algorithms

Next lecture!