CS 105C: Lecture 7

Last Time...

Templates implement parametric polymorphism in C++

The *same* code for all types.

```
1  template <class T>
2  void swap(T& x, T& y){
3     T temp;
4     temp = y;
5     y = x;
6     x = temp;
7  }
```

Templates need to be instantiated--for this, the definition needs to be known

We need to put template definitions in the headers--this is exactly the opposite of how headers are usually used!

```
1  // File swap.h
2  template <typename T>
3  void swap(T& v1, T& v2);
```

```
1  // File main.cpp
2  #include "swap.h"
3
4  int main(){
5    int a = 3, b = 8;
6    swap(a,b);
7  }
```

```
1  // File swap.cpp
2  #include "swap.h"
3
4  template <typename T>
5  void swap(T& a, T& b){
6    T temp;
7    temp = a;
8    a = b;
9    b = temp;
10 }
```

Templates can compile fine with one set of types, but cause a type error with another.

```
1 template <typename T, typename U>
2 void addElem(T<U>& collection, const U& toAdd){
3   collection.insert(toAdd);
4 }
```

```
1 std::vector<int> vec;
2 addElem(vec, 5);
```

Okay!

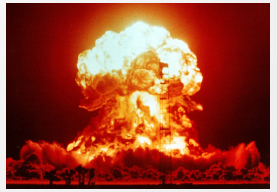
```
1 std::set<int> set;
2 addElem(set, 5);
```

Okay!

```
1 std::list<int> lst;
2 addElem(lst, 5);
```

Okay!

```
1 std::array<int> arr;
2 addElem(arr, 5);
```



Questions!

Q: What are some common use cases for templates?

Q: What are some common use cases for templates?

std::abs
std::all_of
std::any_of
std::atoi
std::begin
std::boolalpha
std::cerr
std::cin
std::cout
std::distance
std::end
std::endl
std::ends
std::exception
std::fabs
std::find
std::fixed
std::forward
std::fstream
std::getline
std::hex
std::ifstream
std::ios
std::iostream
std::is
std::isalpha
std::isdigit
std::istream
std::iterator
std::map
std::max
std::min
std::move
nullptr
std::ostream
std::ostringstream
std::pair
std::pow
std::printf
std::rand
std::range
std::regex
std::remove
std::set
std::setfill
std::setprecision
std::setw
std::size
std::sort
std::min
std::sprintf
std::sqrt
std::srand
std::stoi
std::strcmp
std::strerror
std::string
std::stringbuf
std::stringstream
std::strrchr
std::strstr
std::strtok
std::terminate
std::tolower
std::tuple
std::uppercase
std::vector
std::ws

Q: What are some common use cases for templates?

std::abs
std::all_of
std::any_of
std::atoi
std::begin
std::boolalpha
std::cerr
std::cin
std::cout
std::distance
std::end
std::endl
std::ends
std::exception
std::fabs
std::find
std::fixed
std::forward
std::fstream
std::getline
std::hex
std::ifstream
std::ios
std::iostream
std::is
std::isalpha
std::isdigit
std::istream
std::iterator
std::map
std::max
std::min
std::move
nullptr
std::ostream
std::ostringstream
std::pair
std::pow
std::printf
std::rand
std::range
std::regex
std::remove
std::set
std::setfill
std::setprecision
std::setw
std::size
std::sort
std::min
std::sprintf
std::sqrt
std::srand
std::stoi
std::strcmp
std::strerror
std::string
std::stringbuf
std::stringstream
std::strrchr
std::strstr
std::strtok
std::terminate
std::tolower
std::tuple
std::uppercase
std::vector
std::ws

Q: What are some common use cases for templates?

std::abs
std::all_of
std::any_of
std::atoi
std::begin
std::boolalpha
std::cerr
std::cin
std::cout
std::distance
std::end
std::endl
std::ends
std::exception
std::fabs
std::find
std::fixed
std::forward
std::fstream
std::getline
std::hex
std::ifstream
std::ios
std::iostream
std::is
std::isalpha
std::isdigit
std::istream
std::iterator
std::map
std::max
std::min
std::move
nullptr
std::ostream
std::ostringstream
std::pair
std::pow
std::printf
std::rand
std::range
std::regex
std::remove
std::set
std::setfill
std::setprecision
std::setw
std::size
std::sort
std::min
std::sprintf
std::sqrt
std::srand
std::stoi
std::strcmp
std::strerror
std::string
std::stringbuf
std::stringstream
std::strrchr
std::strstr
std::strtok
std::terminate
std::tolower
std::tuple
std::uppercase
std::vector
std::ws

Q: What are some common use cases for templates?

std::abs
std::all_of
std::any_of
std::atoi
std::begin
std::boolalpha
std::cerr
std::cin
std::cout
std::distance
std::end
std::endl
std::ends
std::exception
std::fabs
std::find

std::fixed
std::forward
std::fstream
std::getline
std::hex
std::ifstream
std::ios
std::iostream
std::is
std::isalpha
std::isdigit
std::istream
std::iterator
std::map
std::max
std::min

std::move
nullptr
std::ostream
std::ostringstream
std::pair
std::pow
std::printf
std::rand
std::range
std::regex
std::remove
std::set
std::setfill
std::setprecision
std::setw

std::size
std::sort
std::min
std::sprintf
std::sqrt
std::srand
std::stoi
std::strcmp
std::strerror
std::string
std::stringbuf
std::stringstream
std::strrchr
std::strstr
std::strtok

std::terminate
std::tolower
std::tuple
std::uppercase
std::vector
std::ws

Q: What are some common use cases for templates?

std::abs
std::all_of
std::any_of
std::atoi
std::begin
std::boolalpha
std::cerr
std::cin
std::cout
std::distance
std::end
std::endl
std::ends
std::exception
std::fabs
std::find
std::fixed
std::forward
std::fstream
std::getline
std::hex
std::ifstream
std::ios
std::iostream
std::is
std::isalpha
std::isdigit
std::istream
std::iterator
std::map
std::max
std::min
std::move
nullptr
std::ostream
std::ostringstream
std::pair
std::pow
std::printf
std::rand
std::range
std::regex
std::remove
std::set
std::setfill
std::setprecision
std::setw
std::size
std::sort
std::min
std::sprintf
std::sqrt
std::srand
std::stoi
std::strcmp
std::strerror
std::string
std::stringbuf
std::stringstream
std::strrchr
std::strstr
std::strtok
std::terminate
std::tolower
std::tuple
std::uppercase
std::vector
std::ws

Q: Does the following code work?

```
1  struct Animal { ... };
2
3  struct Dog : public Animal { ... };
4
5  template <typename T>
6  swap(T& a, T& b){
7    T temp;
8    temp = a;
9    a = b;
10   b = temp;
11 }
12
13 swap(Dog(), Animal());
```

A: No. Even if the template could infer the correct type, we cannot swap an Animal and a Dog, since there's no guarantee the Animal is actually a Dog.

Q: Why can't we just use auto instead of this whole template business?

```
1 auto swap (auto& x, auto& y){
2   auto temp = x;
3   x = y;
4   y = temp;
5 }
```

A: They have different meanings. 'auto' means "there is exactly one type that can go here. Please figure it out for me."

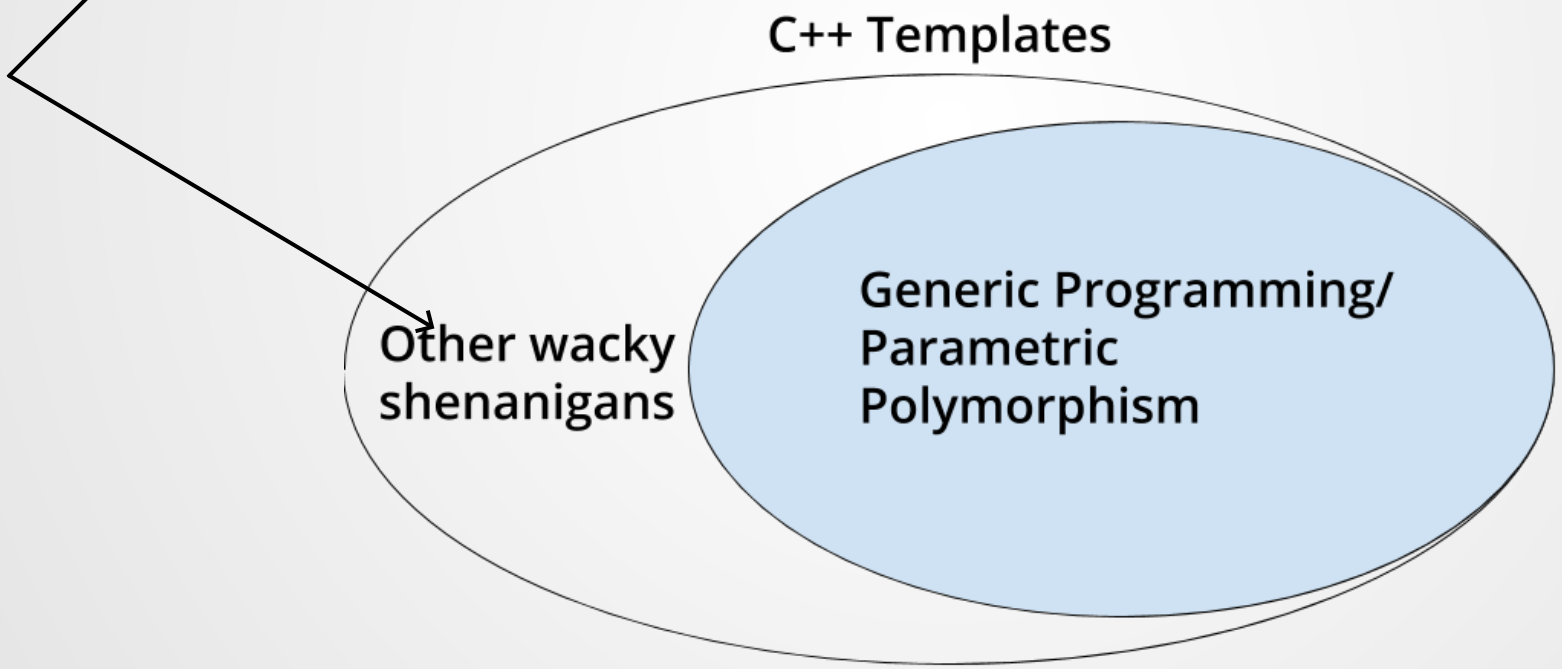
Templates mean "any type can go here."

Caveat: C++20 introduces new syntax with concepts that allows this to happen. However, you cannot constrain the types for x and y (e.g. force them to be the same type).

Q: Why do you pass ints to template parameters if they're used for parametric polymorphism?

```
1  template <unsigned int n>
2  ...
```

A: I should have clarified this more



Templates in C++ serve similar purposes to generics in Java, but have additional functionality that made them much more powerful.

CS 105C: Lecture 7

<iterator>, <algorithm>, lambdas

or

How to use <algorithm>

Hypothetical Situation

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 );                    (1)  (until C++20)

template< class InputIt, class T >
constexpr typename iterator_traits<InputIt>::difference_type
              count( InputIt first, InputIt last, const T &value );          (1)  (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 );   (2)  (since C++17)

template< class InputIt, class UnaryPredicate >
typename iterator_traits<InputIt>::difference_type
    count_if( InputIt first, InputIt last, UnaryPredicate p );               (3)  (until C++20)

template< class InputIt, class UnaryPredicate >
constexpr typename iterator_traits<InputIt>::difference_type
              count_if( InputIt first, InputIt last, UnaryPredicate p );     (3)  (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 );   (4)  (since C++17)
```

...and then you write a for-loop.

Standard library functions are often not very pretty

...which is a shame, because they're usually pretty useful, and not that hard to use once you understand what's going on!

Remove all bricks that collide with the ball

```
1 auto bbb = ball.getBoundingBox();
2 auto collides_with_ball = [bbb](const Brick &b) {return bbb.intersects(b.getBoundingBox());}
3 auto last = std::remove_if(bricks.begin(), bricks.end(), collides_with_ball);
4 bricks.erase(last, bricks.end());  # Cleanup due to how std::remove_if works
```

How did <algorithm> and its surrounding structures even come about?

Iterator

How do I add one to each element of a vector?

```
1 int begin = 0;
2 int i = begin;
3 while(i != vec.size()){
4   vec[i]++;
5   i++;
6 }
```

How do I add one to each element of a linked list?

```
1 Node* begin = linkedlist.head();
2 Node* target = begin;
3 while(target != nullptr){
4   target->data++;
5   target = target->next;
6 }
```

How do I add one to each element of a binary tree?

```
1 Node* begin = tree.root();
2 Node* target = begin;
3 while(target != nullptr){
4   target->data++;
5   target = tree.in_order_traversal().next(target);
6 }
```

...or...

```
1 Node* begin = tree.root();
2 Node* target = begin;
3 while(target != nullptr){
4   target->data++;
5   target = tree.pre_order_traversal().next(target);
6 }
```

What do these solutions all have in common?

```
1 int begin = 0;
2 int i = begin;
3 while(i != vec.size()){
4   vec[i]++;
5   i++;
6 }
```

```
1 Node* begin = linkedlist.head();
2 Node* target = begin;
3 while(target != nullptr){
4   target->data++;
5   target = target->next;
6 }
```

```
1 Node* begin = tree.root();
2 Node* target = begin;
3 while(target != nullptr){
4   target->data++;
5   target = tree.in_order_traversal().next(target);
6 }
```

A first valid element

A current element

A first invalid element

A way to access the data within the element

A way to get the next element

An associated data structure

What do these solutions all have in common?

A first valid element - begin()

A current element - (no special method)

A first invalid element - end()

A way to access the data within the element - * [dereference]

A way to get the next element - next() or ++

An associated data structure

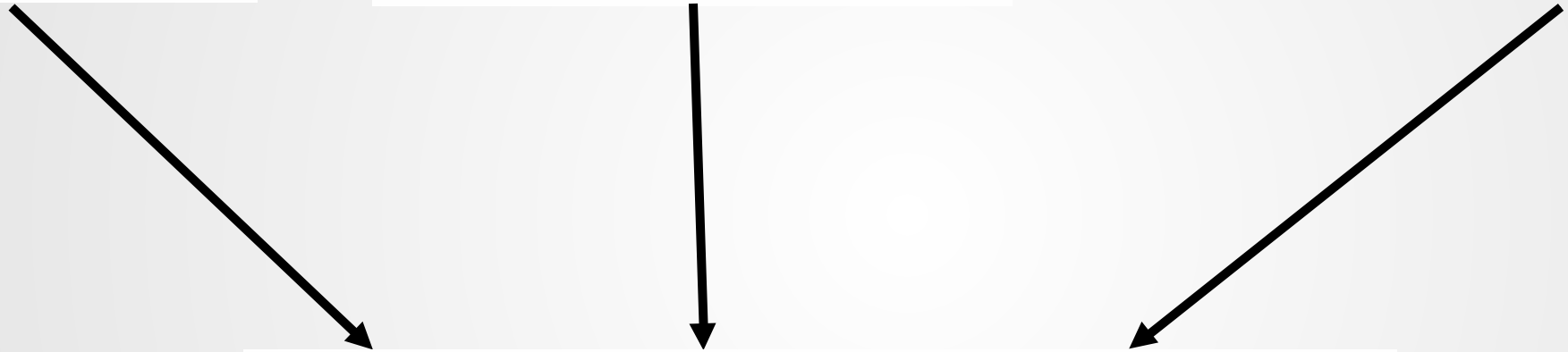
These traits form the core of the iterator

How to add one to every element of a ~~vector~~ ~~linked list~~ ~~tree~~ collection?

```cpp
int begin = 0;
int i = begin;
while(i != vec.size()){
  vec[i]++;
  i++;
}
```

```cpp
Node* begin = linkedlist.head();
Node* target = begin;
while(target != nullptr){
  target->data++;
  target = target->next;
}
```

```cpp
Node* begin = tree.root();
Node* target = begin;
while(target != nullptr){
  target->data++;
  target = tree.in_order_traversal().next(target);
}
```



```cpp
template<typename T>
void add_one(T& collection){
  T::iterator it = collection.begin();
  while(it != collection.end()){
    *it++;
    it = it.next();
  }
}
```

In fact, we can do even better!

```
1 T::iterator it = collection.begin();
2 while(it != collection.end()){
3   auto& elem = *it;
4   /* Do computation */
5   it = it.next();
6 }
```

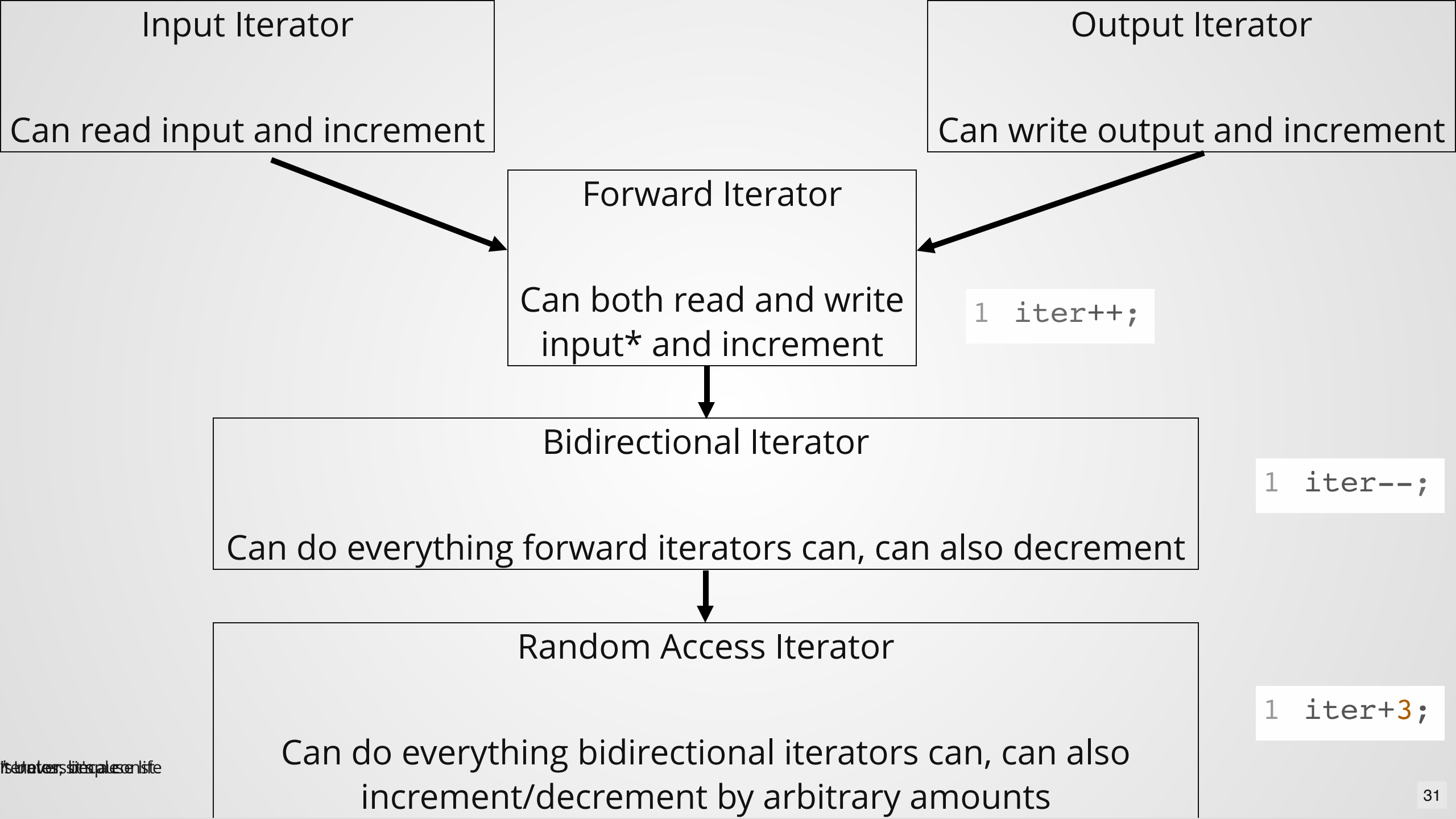
This pattern is common enough that
we gave it a special coat of paint:

```
1 for (auto& elem : collection){
2   /* Do computation */
3 }
```

What other capabilities do iterators have?

The Three Rs

Reading and Riting and Rithmetic


Input Iterator
Can read input and increment
Output Iterator
Can write output and increment
Forward Iterator
Can both read and write input* and increment
1 iter++;
Bidirectional Iterator
Can do everything forward iterators can, can also decrement
1 iter--;
Random Access Iterator
Can do everything bidirectional iterators can, can also increment/decrement by arbitrary amounts
1 iter+3;

<algorithm>s based on iterators

count, find, fill

std::count, std::count_if

Defined in header <algorithm>

```
[redacted]
count( InputIt first, InputIt last, const T &value );                    (1)   (until C++20)
[redacted]
count( InputIt first, InputIt last, const T &value );                          (since C++20)
```

How do we call this function?

std::count, std::count_if

Defined in header <algorithm>

```
[redacted]
count( InputIt first, InputIt last, const T &value );                    (1)   (until C++20)
[redacted]
count( InputIt first, InputIt last, const T &value );                          (since C++20)
```

```
1 std::vector<int> vec = {1,3,5,7,9,12,14,16,27,3,2,4,19,3};
2
3 int num_equals_3 = std::count(vec.begin(), vec.end(), 3);
```

std::find, std::find_if, std::find_if_not

Defined in header <algorithm>

```
[redacted]
InputIt find( InputIt first, InputIt last, const T& value );                    (1)   (until C++20)
[redacted]
constexpr InputIt find( InputIt first, InputIt last, const T& value );          (1)   (since C++20)
```

Returns iterator to *first* instance of value, if it exists

```
std::vector<int> vec = {1,3,5,7,1,9,12,14,16,1,27,3,2,4,19,3};

// Let's be honest: who actually wants to type out this type signature??
std::vector<int>::iterator iter = std::find(vec.begin(), vec.end(), 3);

auto iter2 = std::find(vec.begin(), vec.end(), 3);
```

What if it doesn't?

```
if (iter2 == vec.end()){
  /* Did not find value in the vector */
}
```

std::fill

Defined in header <algorithm>

```
void fill( ForwardIt first, ForwardIt last, const T& value );                (1)   (until C++20)
constexpr void fill( ForwardIt first, ForwardIt last, const T& value );            (since C++20)
```

Why does this need a ForwardIterator?

```
1 std::vector<int> vec = {1,3,5,7,1,9,12,14,16,1,27,3,2,4,19,3};
2
3 auto iter1 = std::find(vec.begin(), vec.end(), 3);
4 auto iter2 = std::find(next(iter1), vec.end(), 3);
5 std::fill(iter1, iter2+1, 5);
```

What does this code do?

Suppose that vec has

- no copies of 3
- only one copy of 3
- two or more copies of 3

Bonus Iterators!

Useful little tools in the STL

std::copy, std::copy_if

Defined in header <algorithm>

```
template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );                    (1)   (until C++20)
template< class InputIt, class OutputIt >
constexpr OutputIt copy( InputIt first, InputIt last, OutputIt d_first );                (since C++20)
```

Copies the elements between *first* and *last* into the range indicated by *d_first*

```
1 auto iter = first;
2 while(iter != last){
3   *d_iter = *iter;
4   ++iter;
5   ++d_iter;
6 }
```

What if we don't want to overwrite elements in the destination?

Solution: Use an insert_iterator

Note: using std::inserter() requires that the underlying class implements an insert() method.

If the class implements a push_back() method, you can also use std::back_inserter()

```cpp
// Copy all elements from src into dest
std::vector<int> src = /* initialization */;
std::vector<int> dest;

// If we don't reserve, we run off the end
dest.reserve(src.size());
std::copy(src.begin(), src.end(), dest.begin());
```

```cpp
// Copy all elements from src into dest
std::vector<int> src = /* initialization */;
std::set<int> dest;

// How do I insert the element into a set?
// I can't preallocate space anymore...
```

```cpp
// Copy all elements from src into dest
std::vector<int> src = /* initialization */;
std::set<int> dest;

auto dest_inserter = std::inserter(dest.end());
std::copy(src.begin(), src.end(), dest_inserter);
```

Reverse Iterators

Just an iterator that runs in the opposite direction of the "normal" iterator

Class must support at least bidirectional iterators to have a reverse iterator

```
1 bool is_palindrome(const std::string& a){
2   return std::equals(a.begin(), a.end(),
3                      a.rbegin(), a.rend());
4 }
```

Computing Functions for Algorithm

Computing Functions for Algorithm

So far, we can count all elements in a collection equal to something (5, or "mystring", or false)

But what if we want to count all even elements? Or all elements that are Fibonacci numbers?

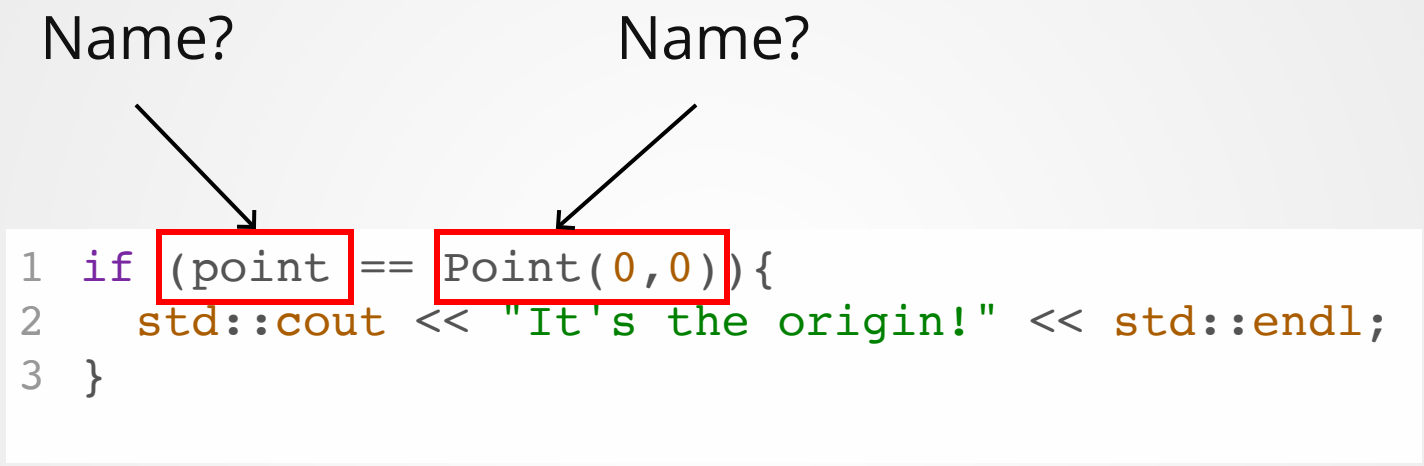
What if we want to sort things?

```
1 int neven = std::count_if(vec.begin(), vec.end(), /* What type of thing goes here? */);
2
3 int nodd = std::count_if(vec.begin(), vec.end(), /* What type of thing goes here? */);
```

Introducing Lambdas

(starting in C++14)

Lambdas are anonymous functions



The second point has no name: it is an
anonymous object

Lambdas are anonymous functions

```
1 std::count_if(vec.begin(), vec.end(), [](int x){return x%2==0;});
```

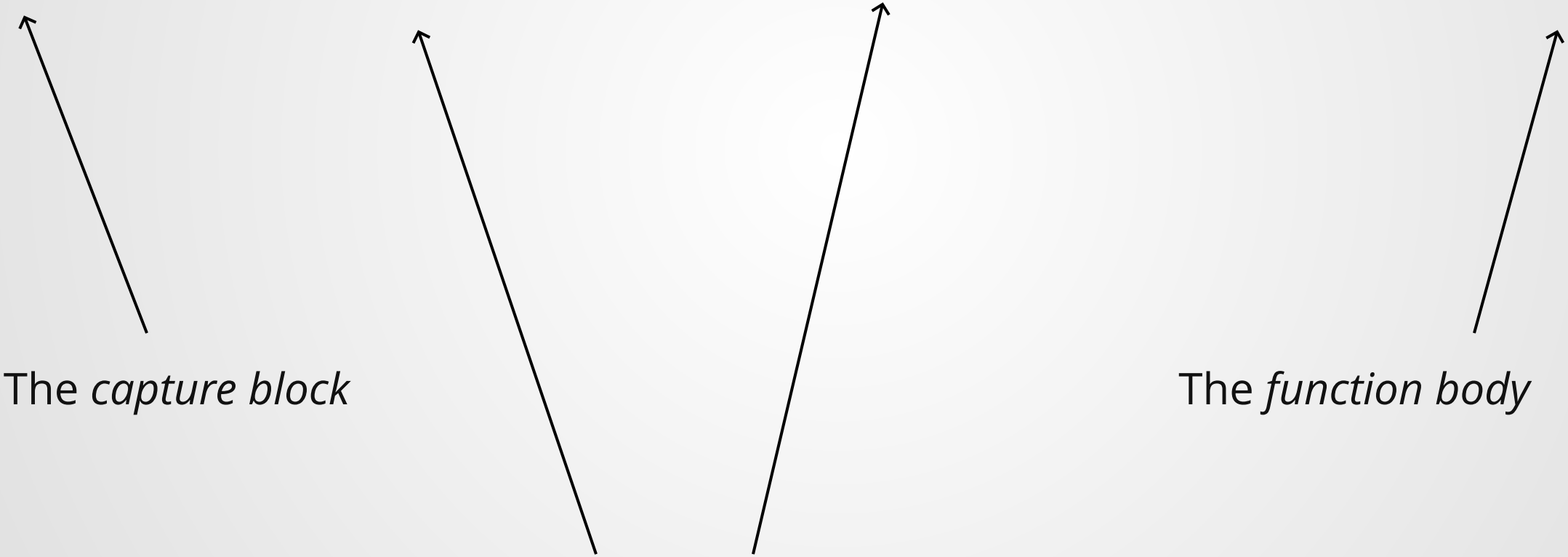
no name!

We can still bind them to names though!

```
1 auto is_even = [](int x){return x%2==0;}
2 bool three_is_even = is_even(3);
```

A C++ Lambda has three parts

```
[=](int x, int y) -> bool { return x <= y; }
```



The function body

```
[=](int x, int y) -> bool { return x <= y; }
```

Works pretty much like a normal function body: sequence of statements separated by semicolons.

What variables can we access?

- Anything declared in the lambda arguments
- Any variables that would normally be accessible in a standalone function (e.g. global variables, static class variables)
- Any variables declared in the *capture block*

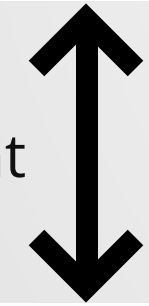
The parameters and return value

```
[=](int x, int y) -> bool { return x <= y; }
```

Similar to how they work in regular functions (in fact, the arrow syntax is legal for regular functions as well!)

```
[=](int x, int y) { return x <= y; }
```

equivalent



If we don't specify a return type, the compiler will automatically deduce one.

```
[=](int x, int y) -> auto { return x <= y; }
```

The captures clause

```
[=](int x, int y) -> bool { return x <= y; }
```

Lambdas can *capture* values from their current scope.

Think of them as being saved when the lambda is being *created* and used when the lambda is *called*.

A capture clause is a comma-separated list of variable names which should be captured by the lambda.

Example Captures

```
1  #include<vector>
2  #include<string>
3  #include<iostream>
4
5  std::vector<std::string> adjectives = {"mean", "green", "lean"};
6
7  int main(){
8    std::string suffix;
9    std::cin >> suffix;
10
11   auto append_suffix = [suffix](std::string& f){return f + suffix;};
12   assert(append_suffix("hey") == std::string("hey") + suffix);
13
14   std::transform(adjectives.begin(), adjectives.end(), append_suffix);
15 }
```

What are the captures? Parameter types?

Example Captures

```
1 auto bbb = ball.getBoundingBox();
2 auto collides_with_ball =
3   [bbb](const Brick &b) {return bbb.intersects(b.getBoundingBox());}
4
5 auto last = std::remove_if(bricks.begin(), bricks.end(), collides_with_ball);
```

What are the captures? Parameter types?

Auto Capture

Sometimes, you don't want to write out a list of 30 million variables (though if you're capturing that many, you might already be doing something wrong)

C++ gives us two special capture characters to automatically generate the capture list:

- = means automatically capture all unresolved variables by value
- & means automatically capture all unresolved variables by reference

These are generally safe to use all over the place, though be careful if the variable names overlap (e.g. you have both a capture and an input named the same thing)

Reading The Docs

Rule 1: You can pretend the ExecutionPolicy overloads don't exist (unless you want to run these in parallel!)

std::unique_copy

Defined in header <algorithm>

```
template< class InputIt, class OutputIt >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first );                                  (1)  (until C++20)
template< class InputIt, class OutputIt >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first );                              (since C++20)
```

[redacted]

```
template< class InputIt, class OutputIt, class BinaryPredicate >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first, BinaryPredicate p );               (3)  (until C++20)
template< class InputIt, class OutputIt, class BinaryPredicate >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first, BinaryPredicate p );           (since C++20)
```

[redacted]

Rule 2: Look at the types and names and think about what they mean! *Always* look at the overloads without **Predicate**s or **Op**s first

std::unique_copy

Defined in header <algorithm>

```
template< class InputIt, class OutputIt >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first );                                  (1)  (until C++20)
template< class InputIt, class OutputIt >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first );                              (since C++20)
```

```
template< class InputIt, class OutputIt, class BinaryPredicate >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first, BinaryPredicate p );               (3)  (until C++20)
template< class InputIt, class OutputIt, class BinaryPredicate >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first, BinaryPredicate p );           (since C++20)
```

Rule 3: The output structure is assumed to be large enough to hold all the data.

(If this is false, make the output large enough or use an insert iterator)

std::unique_copy

Defined in header <algorithm>

```
template< class InputIt, class OutputIt >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first );                                  (1)  (until C++20)
template< class InputIt, class OutputIt >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first );                              (since C++20)
```

```
template< class InputIt, class OutputIt, class BinaryPredicate >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first, BinaryPredicate p );               (3)  (until C++20)
template< class InputIt, class OutputIt, class BinaryPredicate >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first, BinaryPredicate p );           (since C++20)
```

Rule 3.5: If two input ranges are needed, and there is not a last iterator for the second range, it is assumed the two ranges are the same size.

std::equal

Defined in header <algorithm>

```
template< class InputIt1, class InputIt2 >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2 );                                   (1)  (until C++20)

template< class InputIt1, class InputIt2 >
constexpr bool equal( InputIt1 first1, InputIt1 last1,
                      InputIt2 first2, InputIt2 last2 );         (since C++20)
```

Rule 4: **Unary** lambdas take one operand, **Binary** lambdas take two

Predicates return booleans, and **Op**s return anything.

std::unique_copy

Defined in header <algorithm>

```
template< class InputIt, class OutputIt >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first );                                  (1)  (until C++20)
template< class InputIt, class OutputIt >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first );                              (since C++20)
```

```
template< class InputIt, class OutputIt, class BinaryPredicate >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first, BinaryPredicate p );               (3)  (until C++20)
template< class InputIt, class OutputIt, class BinaryPredicate >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first, BinaryPredicate p );           (since C++20)
```

Copies the elements from the range `[first, last)`, to another range beginning at `d_first` in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied.

1) Elements are compared using `operator==`. The behavior is undefined if it is not an equivalence relation.

3) Elements are compared using the given binary predicate `p`. The behavior is undefined if it is not an equivalence relation.

2,4) Same as (1,3), but executed according to `policy`. This overload only participates in overload resolution if `std::is_execution_policy_v<std::decay_t<ExecutionPolicy>>` is true

std::unique_copy

Defined in header `<algorithm>`

```
template< class InputIt, class OutputIt >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first );                                   (1)
```
(until C++20)

```
template< class InputIt, class OutputIt >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first );
```
(since C++20)

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2 >
ForwardIt2 unique_copy( ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last,   (2)
                        ForwardIt2 d_first );
```
(since C++17)

```
template< class InputIt, class OutputIt, class BinaryPredicate >
OutputIt unique_copy( InputIt first, InputIt last,
                      OutputIt d_first, BinaryPredicate p );                (3)
```
(until C++20)

```
template< class InputIt, class OutputIt, class BinaryPredicate >
constexpr OutputIt unique_copy( InputIt first, InputIt last,
                                OutputIt d_first, BinaryPredicate p );
```
(since C++20)

```
template< class ExecutionPolicy, class ForwardIt1, class ForwardIt2, class BinaryPredicate >
ForwardIt2 unique_copy( ExecutionPolicy&& policy, ForwardIt1 first, ForwardIt1 last,   (4)
                        ForwardIt2 d_first, BinaryPredicate p );
```
(since C++17)

Putting it all together

Scrabble code!

Note: the following code is *completely agnostic* to the underlying data structures: the only rule is that your data structure has to implement the appropriate iterator types.

The user enters several words they want to try playing. The program makes sure that all the scrabble words are legal, then picks the highest-scoring one.

Note: code does not 100% compile--ask me if you want to see a compiling version (it needs templates)

```
constexpr auto legalWords = gen_scrabble_legal_words();
unsigned int scoreWord(std::string word){
  /* Compute scrabble point value */
}

int main(){
  /* Some collection of strings: again, we don't care
  what the format is as long as it has iterators */
  auto userWords = get_user_input();

  // Use built-in string comparison
  std::sort(userWords.begin(), userWords.end());

  auto illegalWords;
  std::set_difference(userWords.begin(), userWords.end()
                      ,legalWords.begin(), legalWords.end()
                      ,std::inserter(illegalWords.begin()));

  if(!illegalWords.empty()){
     std::cout << "Illegal word in input!" << std::endl;
  }
  ...
  // Continued on next slide
```

```
  // Continue from previous slide

  auto legalUserWords;
  auto is_legal_word = [&legalWords](std::string word) -> bool {
    return std::find(legalWords.begin(), legalWords.end(), word) != legalWords.end();
  };
  std::copy_if(userWords.begin(), userWords.end(), std::inserter(legalUserWords.begin())
              ,is_legal_word);

  auto compare_words = [](const std::string& a, const std::string& b){
    return scoreWord(a) < scoreWord(b);
  };

  // Iterator to largest element
  auto maxIter = std::max_element(legalUserWords.begin(), legalUserWords.end(), compare_words);

  std::cout << "The highest-scoring legal word that you entered was " << *maxIter << std::endl;
} // End of main()
```

Summary

Iterators

A way to abstract out "go through every element in the collection."

Have different capabilities: read/write and motion.

category				properties	valid expressions
all categories				*copy-constructible*, *copy-assignable* and *destructible*	`X b(a);` `b = a;`
				Can be incremented	`++a` `a++`
Random Access	Bidirectional	Forward	Input	Supports equality/inequality comparisons	`a == b` `a != b`
				Can be dereferenced as an *rvalue*	`*a` `a->m`
			Output	Can be dereferenced as an *lvalue* (only for *mutable iterator types*)	`*a = t` `*a++ = t`
				default-constructible	`X a;` `X()`
				Multi-pass: neither dereferencing nor incrementing affects dereferenceability	`{ b=a; *a++; *b; }`

				Can be decremented	`--a`
					`a--`
					`*a--`

				Supports arithmetic operators + and -	`a + n`
					`n + a`
					`a - n`
					`a - b`

				Supports inequality comparisons (<, >, <= and >=) between iterators	`a < b`
					`a > b`
					`a <= b`
					`a >= b`

				Supports compound assignment operations += and -=	`a += n` `a -= n`
				Supports offset dereference operator ([])	`a[n]`

Iterators

std::begin() or collection.begin() gets us an iterator to the *first valid element*.

std::end() or collection.end() gets us an iterator to the *first invalid element*.

not the last valid element!

Use std::inserter() or std::back_inserter() to get an insertion iterator.

C++ Lambdas

An anonymous function that can be constructed and used in a local scope.

```
[=](int x, int y) -> bool { return x <= y; }
```

Capture Clause

Parameters

Function Body

Upcoming things

Project 2 is still due next week.

If you haven't started thinking about the garbage collector yet, **start worrying.**

==========

No quiz next week! There will be two back-to-back quizzes at some point in the future.

Additional Readings

- Simple Intro to Iterators
- The CPPReference for functions in <algorithm>
- A human-readable guide to useful functions in <algorithm>
- The C++ Primer, Appendix A.2 has a great categorization of all the functions in <algorithm>, along with explanations for how to read the API doc and what each function does.

  It lists the algorithms not by name or arguments, but by functionality (e.g. "Algorithms to find an item", "Partitioning and Sorting Algorithms", etc.)

Notecards

- Name and EID
- One thing you learned today (can be "nothing")
- One question you have about the material. **If you leave this blank, you will be docked points.**

  If you do not want your question to be put on Piazza, please write the letters **NPZ** and circle them.