COMP 3522

Object Oriented Programming in C++

Week 6

Agenda

1. auto keyword
2. Ranged for
3. Intro to the STL
4. STL containers

COMP 3522

auto KEYWORD

The auto keyword

- When used as a variable type, auto specifies that **the type of the variable will be deduced automatically from its initializer**.
- When used as a function return type, auto specifies that **the return type will be deduced from the return statements**

auto

```
double sum = 5.0;
auto a; //ERROR, auto requires initializer
auto d = 5.0;
auto i = 1 + 2;

int add(int x, int y) { return x + y; }
int main()
{
    auto sum = add(5, 6);
}
```

auto can't be used with function parameters

```
void add_and_print(auto x, auto y)
{
    std::cout << x + y;
}
```

This won't work because the compiler can't infer types for function parameters x and y at compile time

auto can be used with function return types

```
auto add(int x, int y)
{
    return x + y;
}
```

I would like to discourage this:

- Using auto for variables is fine because the object is right there
- Using auto for functions means we have to dig into the function to find out what it's supposed to be returning.

Some programmers like to do this:

Instead of this:

```
int add(int x, int y);
```

They like to do this:

```
auto add(int x, int y) -> int;
```

In this case, auto does not perform type inference, it is just part of the syntax to use a **trailing return type**. But why, though?

So we can do this (so easy to read!)

```
auto add(int x, int y) -> int;
auto divide(double x, double y) -> double;
auto print_something() -> void;
auto calculate_that(int x, double d) -> string;
```

Additional reading: http://en.cppreference.com/w/cpp/language/auto

RANGED FOR

“The ranged for” aka for-each loop

- Identical to Java
- Some of you have already been using it
- Formally it executes a for loop over a specified range

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

for (int i=0; i<v.size(); i++)
     std::cout << v[i] << ‘ ‘;

for (const int i : v) //i is a copy of element in v
     std::cout << i << ‘ ‘;
```

“The ranged for” aka for-each loop

- Identical to Java
- Some of you have already been using it
- Formally it executes a for loop over a specified range

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};

for (int i=0; i<v.size(); i++)
     std::cout << v[i] << ‘ ‘;

for (const int& i : v) // const reference
     std::cout << i << ‘ ‘;
```

“The ranged for” aka for-each loop

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};
for (auto i : v) // access by value, i is int
     std::cout << i << ' ';

// the initializer may be a braced-init-list
for (int n : {0, 1, 2, 3, 4, 5})
     std::cout << n << ' ';
std::cout << '\n';
```

“The ranged for” aka for-each loop

```
// the initializer may be an array
int a[] = {0, 1, 2, 3, 4, 5};
for (int n : a)
     std::cout << n << ' ';

// the loop variable doesn’t have to be used
for (int n : a)
     std::cout << something_unrelated << ' ';
std::cout << '\n';
```

INTRO TO THE STL

Standard Template Library

- C++ **STL**
- Like the Java Collections Framework **SUPERPOWERED**
- One of the most fun and interesting reasons to work with C++
- Composed of:
  1. Containers – classes that store objects and data
  2. Iterators - used for working on a sequence of values
  3. Algorithms - functions specially designed to be used on a range of elements

https://www.geeksforgeeks.org/the-c-standard-template-library-stl/

About the STL

- Uses **value semantics** – containers get a **copy** of the object we are putting in it
- This means our element class must have:
  - Copy constructor
  - Assignment operator
  - Destructor
- STL **performs almost no checking** – the programmer is responsible for meeting preconditions
- STL uses **half-open ranges** **[**included, not included**)**
  - Imagine array size 5. array**[**0,5**)**
  - When iterating through include 0^{th} index, exclude 5^{th} index

Containers

1. **Sequence** containers
   1. We specify the order
   - `array, vector, deque, list, forward_list`
2. **Associative** containers
   1. Objects are automatically sorted
   2. Can be searched with O(log n) complexity
   - `set, multiset, map, multimap`
3. **Unordered associative** containers
   1. Stored using hash
   2. Can be searched O(1) to O(n) worst case
   - `unordered_set, unordered_multiset, unordered_map, unordered_multimap`

Suppose we have a container object c…

c.insert(x); // Inserts a copy of x into an *associative container*
c.insert(position, x); // Inserts a copy of x into a *sequence container*

c.begin(); // Returns an iterator pointing to the first element in the container if it exists
c.end(); // Returns an iterator one past the end of the container

for (**auto** it = c.begin(); it != c.end(); ++it) { // process! }

About vectors (again…)

A vector is a **dynamic array** that offers random access and insertion/deletion

```
#include <vector>
vector<int> v;
v.push_back(2);

for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
      cout << *it << endl;
}
```

More about vectors…

```
vector<int> v2 {2, 4, 6, 8, 10};
v2.insert(v2.begin(), 6); // 6, 2, 4, 6, 8, 10
v2.insert(v2.begin() + 3, -4); // 6, 2, 4, -4, 6, 8, 10
v2.erase(v2.begin() + 1); //6, 4, -4, 6, 8, 10


int a [] = { 3, 6, 9, 12, 15 };
vector<int> v3 {a, a + 5 }; // copies in the range [first, last)
```

(Re-)familiarize yourself with the vector

- empty()
- size()
- capacity()
- clear()
- insert()
- push_back() and pop_back()
- front() and back()

http://www.cplusplus.com/reference/vector/vector/

STL CONTAINERS

I'm so tired of vectors

- We've talked about vectors enough
- Let's visit some different containers in the STL
- Java has the Java Collections Framework
- C++ has the **Standard Template Library**
- Java has collections
- C++ has **containers**

Container classes

- **First-class containers**
  - Vector
  - List
  - Deque
  - **Map** and Multimap
  - **Set** and Multiset
- **Container adaptors (modify and restrict first class container)**
  - Stack (default implementation is the *deque*)
  - Queue (default implementation is also the *deque*)
  - Priority_queue (default implementation is the *vector*)

STL CONTAINERS:

Pair, map

Side-note: pair

- <utility>
- Object that can hold two values of different types
- Has member functions (accessors) called first and second
- See **pair.cpp** for a sample

- There is a built-in C++ function to simplify the creation of a pair: **make_pair**.

http://www.cplusplus.com/reference/utility/pair/

map

- <map>
- std::map
- Sorted associative container
- Provides a collection of 1-to-1 mappings, i.e. a collection of key/value pair objects
- **value_type** is a **pair** type that combines **key** and **value**
- Keys must be unique
- Keys are sorted using a comparison function (like the Java Comparator)
- Logarithmic speed for search, insertion, and removal (**fast!**)

http://www.cplusplus.com/reference/map/map/

map

- The **value_type** of a map is a **Pair**
- How do we add something to a map?
- Suppose we have a phonebook that maps strings to strings:

```
phonebook.insert(map<string, string>::value_type(“Sam”, “6045551212”));
```

- This fails if an element with the same key is already in the map:

```
phonebook.insert(make_pair(“Sam”, “2505551212”));
```

Map.cpp

STL CONTAINERS:

Set

Set

- <set>
- std::set
- Associative container. Contains a sorted set of **unique** objects of type Key
- The value of an element also identifies it
- Elements **cannot be edited** after being added (are **const**)
- Sorting performed using key comparison function
- Logarithmic speed for search, insertion, and remove (fast!)
- Typically implemented as a binary search tree

Comparing (and sorting) set elements

- The C++ compare concept:
  - Type T satisfies Compare if it:
    1. Satisfies **BinaryPredicate** (evaluates to true/false)
    2. Induces a **strict weak ordering**
- Suppose we have a struct called myPair that stores 2 ints, x and y
- We need to write a **< operator** to use it in a set

```
bool operator<(const myPair& lhs, const myPair& rhs) {
    return lhs.x + lhs.y < rhs.x + rhs.y;
} // Sorts myPairs by sum of components
```

* https://en.wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings

STRICT WEAK ORDERING

Note: Strict weak ordering

- Almost all C++ STL functions/containers require the ordering to satisfy the standard mathematical definition of a **strict weak ordering.**
- It satisfies strict weak ordering if your logic follows all three rules
- Let **lessThan**(left, right) be a comparison function. The compare function is in strict weak ordering iff:

1. IRREFLEXIVITY: **lessThan(x, x) == false**
   - If **x** is passed into both parameters, the expected output is false
   - **x** can not be less than itself

Note: Strict weak ordering

2. ANTISYMMETRY: **if lessThan(x, y) then !lessThan(y, x)**
   - If **x** is less than **y**, then **y** can not be less than **x**

3. TRANSITIVITY: **if lessThan(x, y) and lessThan(y, z) then lessThan(x, z)**
   - If **x** is less than **y**, and **y** is less than **z**, then **x** is less than **z**

Defining strict weak ordering

- We can do this in three ways:

1. Define **operator<**(const Obj& lhs, const Obj& rhs) inside the class
2. Define a **custom comparison function** that is a binary predicate (takes two elements and returns a boolean)
3. Implement **operator( )** as a comparison function in a separate struct
   - Functor – object that behaves like a function

STL CONTAINERS & TYPEDEFS

Review: typedef

The **typedef** keyword creates an **alias** that can be used anywhere instead of a (possibly) complex type name

```
typedef int int_t; // declares int_t to be an alias for the type int
int_t myT; //initializes an int

typedef int arr_t[3]; // arr_t is array of 3 int
arr_t myArr; //initializes an int array of size 3
myArr[0] = 999;
myArr[1] = 1;
myArr[2] = 5;
```

Review: typedef

The **typedef** keyword creates an **alias** that can be used anywhere instead of a (possibly) complex type name

```
pair <map<string, string>::iterator it, map<string, string>::iterator it> pairMapIters;

                                    //compared to below

//mapIterator is a new type. Alias for map<string, string>::iterator it
typedef map<string, string>::iterator it mapIterator;

pair <mapIterator, mapIterator> pairMapIters;
```

STL typedef Restaurant Analogy

Fast Food
- LCDMenu

Fancy Restaurant
- PaperMenu

Food Truck
- BoardMenu

What's a **menu**?

What's a **menu**?

What's a **menu**?

Me:
Can I get a **menu**?

STL typedef Restaurant Analogy

Fast Food - **LCDMenu**	Fancy Restaurant - PaperMenu	Food Truck - BoardMenu
OK!	What's an **LCDmenu**?	What's an **LCDmenu**?

Me:
Can I get an **LCDMenu**?

STL typedef Restaurant Analogy

Fast Food
- LCDMenu

- typedef LCDMenu menu

Fancy Restaurant
- PaperMenu

- typedef PaperMenu menu

Food Truck
- BoardMenu

- typedef BoardMenu menu

//All restaurants implement a typedef so that "menu" is a common name/alias for their internal implementation of menu

STL typedef Restaurant Analogy

Fast Food
- LCDMenu

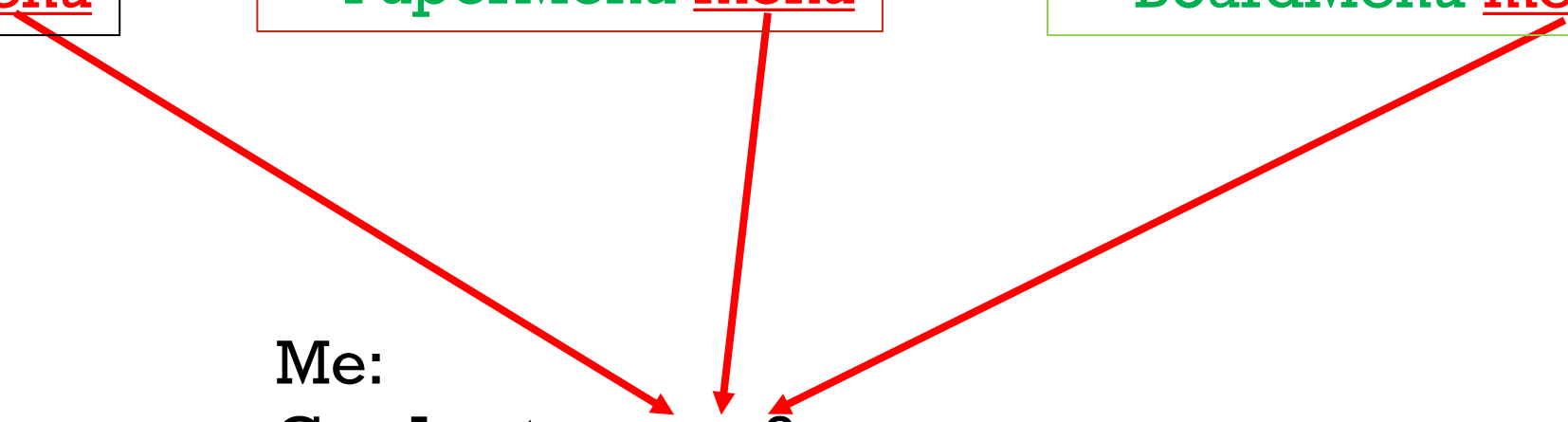
- typedef LCDMenu menu

Fancy Restaurant
- PaperMenu

- typedef PaperMenu menu

Food Truck
- BoardMenu

- typedef BoardMenu menu



Me:
Can I get a menu?

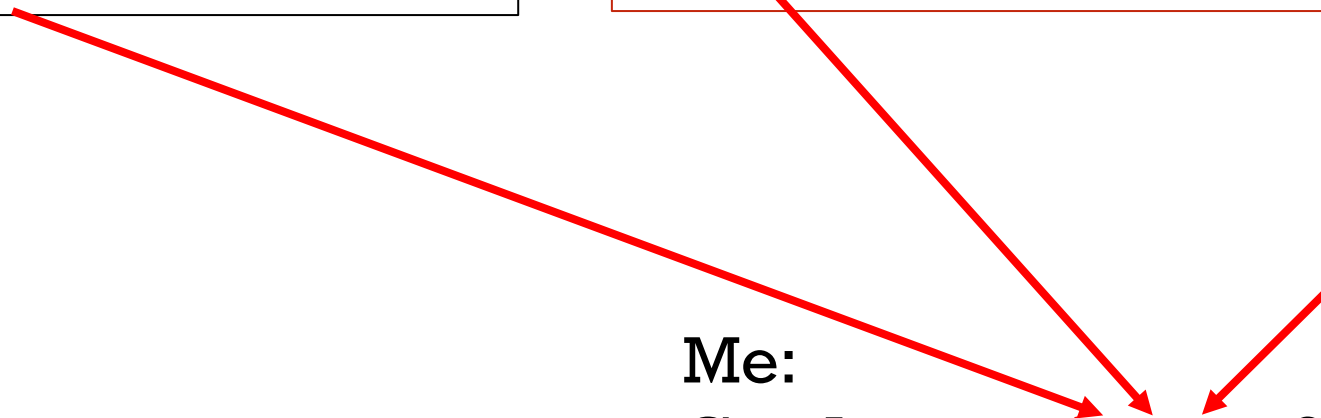
STL typedef Restaurant Analogy

*psuedocode

Vector*
- RandomAccessIterator
- typedef RandomAccessIterator iterator

Map*
- BiDirectionalIterator
- typedef BiDirectionalIterator iterator

Set*
- BiDirectionalIterator
- typedef BiDirectionalIterator iterator



Me:
Can I get an iterator?

STL typedef Restaurant Analogy

*psuedocode

Vector*
- RandomAccessIterator

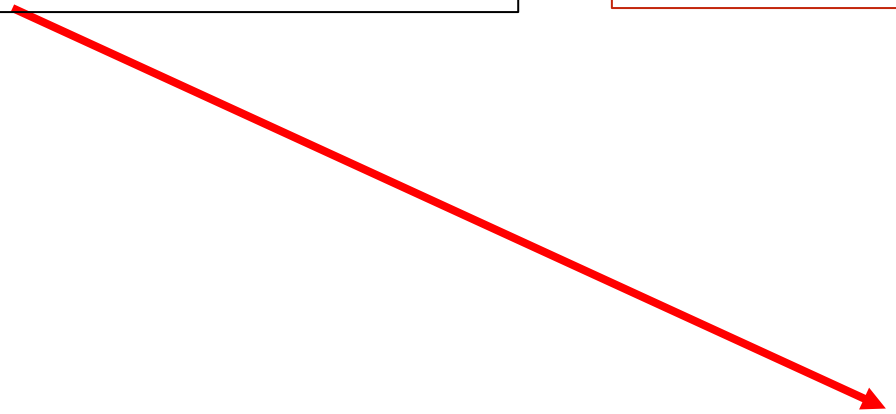
- typedef RandomAccessIterator iterator

Map*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator

Set*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator



vector<int>::iterator myIter;

Gets RandomAccessIterator using iterator typedef

STL typedef Restaurant Analogy

*psuedocode

Vector*
- RandomAccessIterator

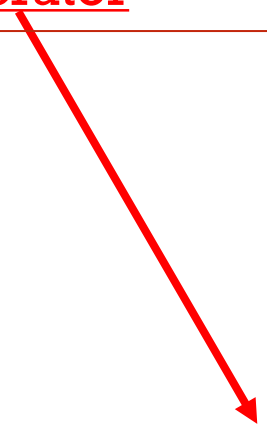
- typedef RandomAccessIterator iterator

Map*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator

Set*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator



map<string, int>::iterator myIter;

Gets BiDirectionalIterator using iterator typedef

typedefs.cpp

Containers and typedefs

- STL containers have standard typedefs
  - size_type
  - value_type
  - **iterator**
  - pointer
  - reference and const_reference
  - difference_type, etc.
- This makes it possible to write generic functions that work on containers
- If we create a container, we should implement these standard typedefs

https://cs.smu.ca/~porter/csc/ref/stl/containers_typedefs.html

Why? **Consistency**!

- Using the typedef mechanism means we can give the **same name to the same conceptual entity** across different container classes
- For example, consider the iterator:
  - vector<int>::**iterator** gives us a **random access iterator** for a vector of integers
  - list<string>::**iterator** gives us a **bidirectional iterator** for a list of strings

size_type

- Unsigned integer type
- Sufficiently large to hold the size of any object of that class
- Appears in all first-class containers and in the container adaptors.

Loop iteration with int using vector

```
void print(const vector<int>& vec)
{
    for (int i = 0; i < vec.size(); ++i) {
        cout << vec[i] << ‘ ‘;
    }
    cout << endl;
}
```

Loop iteration with size_type using vector

```
void print(const vector<int>& vec)
{
    for (vector<int>::size_type i = 0; i < vec.size(); ++i) {
        cout << vec[i] << ' ';
    }
    cout << endl;
}
```

iterator

- An iterator of the default type for a (first-class) container type

```
vector<int> myVec = {1,2,3};
vector<int>::iterator vecIter = myVec.begin();
```

const_iterator

- A const iterator of the default type for a (first-class) container type
  - Prevents modification to elements during iteration

reverse_iterator

- A reverse iterator of the default type for a (first-class) container type
  - Iterates backwards starting from last element with `rbegin()`
  - Incrementing moves the iterator towards beginning of the container

```
vector<int>::reverse_iterator = myVec.rbegin();
```

const_reverse_iterator

- A const iterator of the default type for a (first-class) container type
  - Prevents modification to elements during iteration

Agenda

1. STL Iterators
2. Algorithms

COMP 3522

STL ITERATORS

What about iterators?

- We've looked at first-class containers and container adaptors
- We've looked at the STL standard typedefs
- We still have to look at iterators

Reminder: what's an iterator?

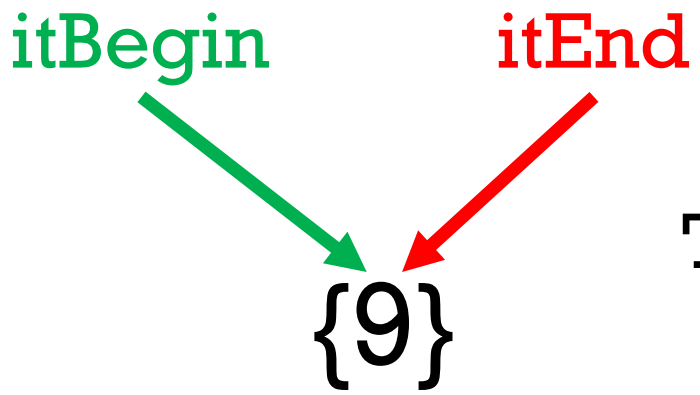
- An object
- Points to some element in a range of elements
- Can iterate (loop) through the range of elements
- Has operators like increment (++) and dereference (*)
- Why important?
  - Iterate through containers generically
  - **Extensively used when calling STL algorithms**

```
some_algorithm(iterBegin, iterEnd, func)
```

Iterator begin and end?

```
vector <int> intVec = {9};
vector <int>::iterator itBegin = intVec.begin();
vector <int>::iterator itEnd = intVec.end();

for(itBegin; itBegin != itEnd; itBegin++)
{
    cout << *itBegin << endl;
}
```

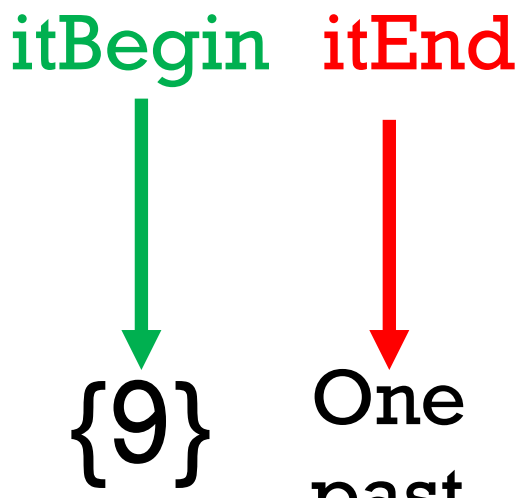


That doesn't look right…

Iterator begin and end?

```
vector <int> intVec = {9};
vector <int>::iterator itBegin = intVec.begin();
vector <int>::iterator itEnd = intVec.end();

for(itBegin; itBegin != itEnd; itBegin++)
{
    cout << *itBegin << endl;
}
```



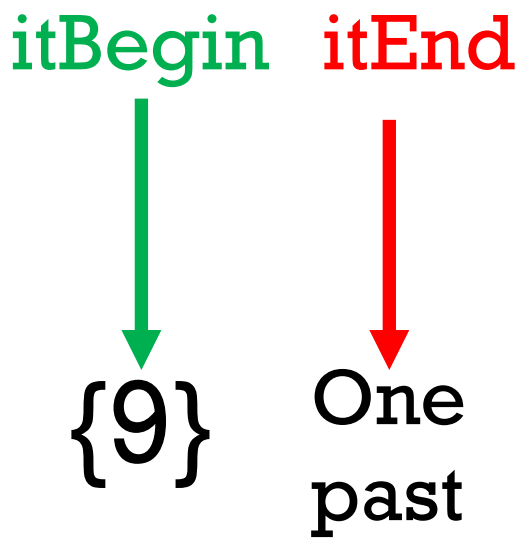
end() returns iterator pointing to theoretical one element past last element in range of values

itEnd does not point to element. **Don't dereference end**

Iterator begin and end?

```
vector <int> intVec = {9};
vector <int>::iterator itBegin = intVec.begin();
vector <int>::iterator itEnd = intVec.end();

for(itBegin; itBegin != itEnd; itBegin++)
{
    cout << *itBegin << endl;
}
```



1st pass:

itBegin != itEnd

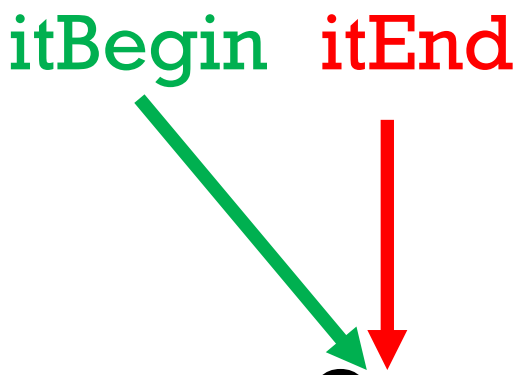
Do loop code

Output: 9

Iterator begin and end?

```
vector <int> intVec = {9};
vector <int>::iterator itBegin = intVec.begin();
vector <int>::iterator itEnd = intVec.end();

for(itBegin; itBegin != itEnd; itBegin++)
{
    cout << *itBegin << endl;
}
```



2nd pass:

itBegin == itEnd

Leave loop

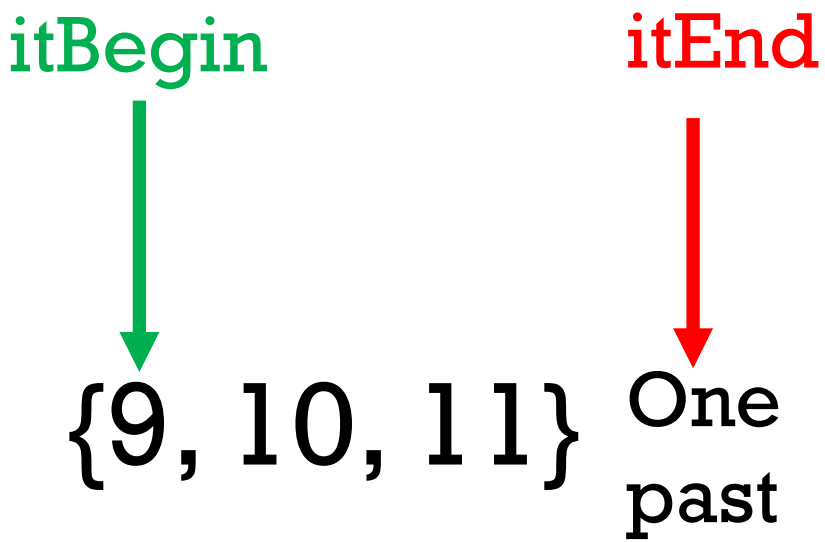
Output: 9

Iterator begin and end?

```
vector <int> intVec = {9, 10, 11};
vector <int>::iterator itBegin = intVec.begin();
vector <int>::iterator itEnd = intVec.end();

for(itBegin; itBegin != itEnd; itBegin++)
{
    cout << *itBegin << endl;
}
```

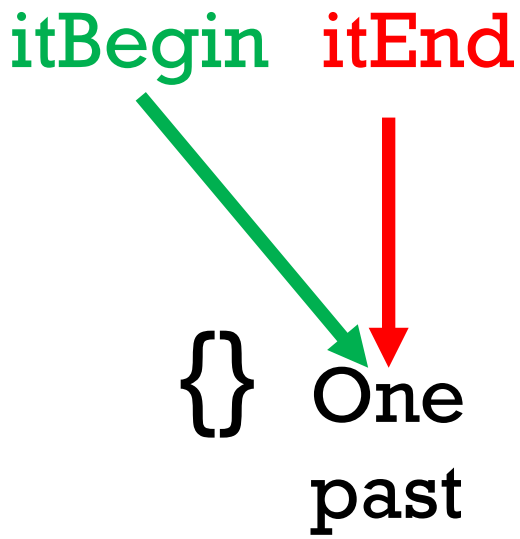
Example of more than 1 element in vector



Iterator begin and end?

```
vector <int> intVec = {};
vector <int>::iterator itBegin = intVec.begin();
vector <int>::iterator itEnd = intVec.end();

for(itBegin; itBegin != itEnd; itBegin++)
{
    cout << *itBegin << endl;
}
```



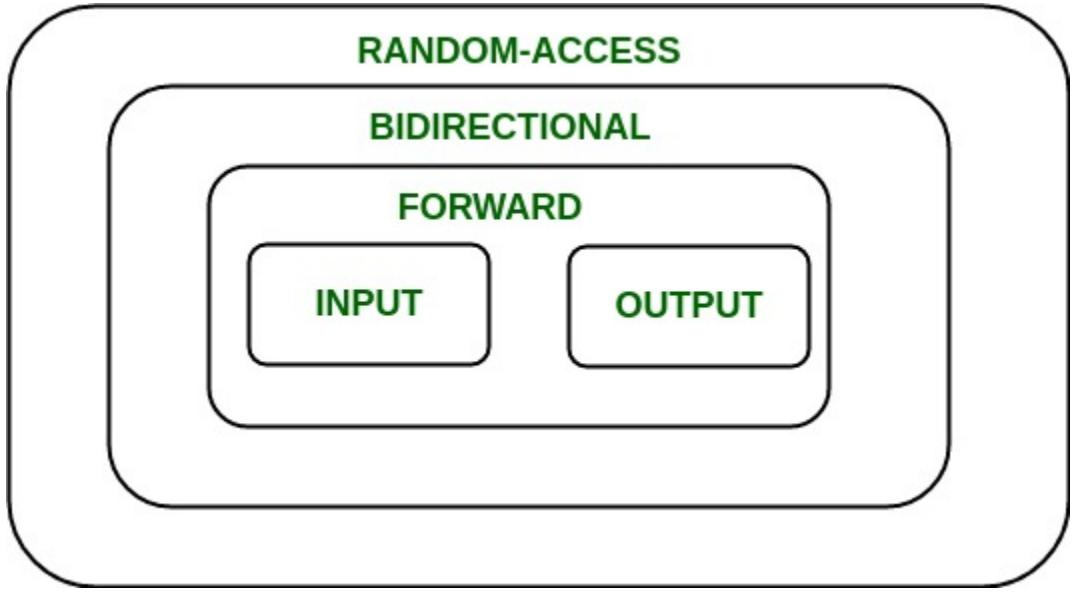
If empty:

itBegin == itEnd

Reminder: what's an iterator?

There are 5 kinds of iterators in C++

- Input Iterators
- Output Iterators
- Forward Iterators
- Bidirectional Iterators
- Random Access Iterators



https://www.geeksforgeeks.org/introduction-iterators-c/

Characteristics of iterators

- Each kind of iterator is defined by the operations that can be performed on it
    1. Read
    2. Write
    3. Increment (with or without multiple passes)
    4. Decrement
    5. Random access.

INPUT & OUTPUT ITERATORS

1. Input iterator I

- **Single pass** forward direction sequential input operations: ==, !=, ++, *it
- **Only reading**
- Incrementing

- **Note**: there is not a single type of input iterator. Each container defines its own specific iterator type that can loop through and access all the elements.

1. Input iterator II – i and j are iterators

Supported Expression	Returns	Equivalent Expression
i != j	bool	!(i == j)
*i	value_type	
i->m	m	(*i).m
++i,	It	++i; return It temp = i;
i++	It	It temp = i; ++i; return temp;
*i++	value_type	value_type x = *i; ++i; return x;

2. Output iterator I

- **Single pass** forward direction sequential output operations: *it, ++, ***it = value**
- **Only writing**
- Incrementing

- **Note**: there is not a single type of output iterator, either. Each container defines its own specific iterator type that can loop through and access all the elements.

2. Output iterator II – i is an iterator

Supported Expression	Returns	Equivalent Expression
*i = some value		
++i	It	++i; return It temp = i;
i++	It	It temp = i; ++i; return temp;
*i++ = some value		*i = some value; ++i;

FORWARD, BIDIRECTIONAL, & RANDOM ITERATORS

3. Forward iterator

- **Multi-pass** forward direction sequential output operations: ==, !=, ++, *it, *it = value
- **Reading AND Writing**, Incrementing
- **We can make a copy of the iterator and dereference the same iterant:**

```
iterSaved = iter; //copies iter to iterSaved
iter++;
cout << "Previous element is " << (*iterSaved) << endl;
cout << "Current element is " << (*iter) << endl;
```

4. Bidirectional iterator I

- **Multi-pass** bidirectional direction sequential output operations: ==, !=, ++, **--**, *it, *it = value
- Reading AND Writing
- Incrementing, **Decrementing**

- **Note**: there is not a single type of bidirectional iterator, either. Each container defines its own specific iterator type that can loop through and access all the elements.

iterator.cpp

4. Bidirectional iterator II – i is an iterator

- Can be used like an input iterator, output iterator, forward iterator, and supports the following additional expressions:

Supported Expression	Returns	Equivalent Expression
--i	It	
i--	It	It temp = i; --i; return temp;
*i--	value_type	

5. Random access iterator I

- **Multi-pass** random access output operations: ==, !=, ++, --, *it, *it = value, it + n or it - n where n is an int, <, <=, …
- Reading AND Writing
- Incrementing and Decrementing
- **Random access**(HOORAY!)

- **Note**: there is not a single type of random access iterator, either. Each container defines its own specific iterator type that can loop through and access all the elements.

5. Random access iterator II

- Can be used like bidirectional iterator, and supports the following additional expressions:

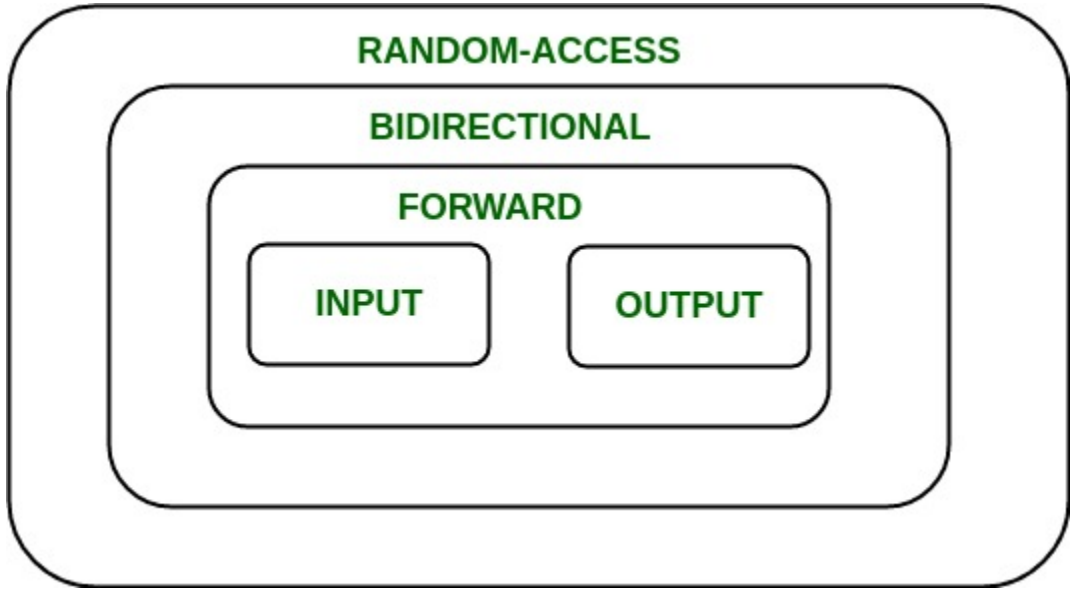
Supported Expression	Returns	Equivalent Expression
i += n, i -= n where i = It&	It	
i + n, i – n	It	It temp = i; return temp += n;
i - j	difference_type	return difference
i[n]	value_type	*(i + n)
i < j, i <= j, i > j, i >= j	bool	

i is an iterator, j is another iterator, n is a number

RandomAccessIterator.cpp

Some notes

- Only random-access iterators permit an integer value to be added to or subtracted from an iterator (iter1 - 5)
- Only random access iterators permit one iterator to be subtracted from another (results in a difference_type) (iter1 – iter2)



https://www.geeksforgeeks.org/introduction-iterators-c/

ALGORITHMS

The algorithm

- I cannot possibly do justice to what you will learn in your algorithms course
- So…
- We will explore some of the common algorithms in C++
- We will identify some patterns (parameters, what they do, etc.)

The STL algorithm

- Used on an array or an **STL container**
- A **function** that can be used on a **range of elements**
- Always in the range **[first, last)**
- Does not change the size of the container
- Used with **iterators or pointers**

- **Use www.cplusplus.com/reference/algorithm**

Types of algorithms I of II

1. Sequential, non-modifying (find( ), for_each( ))
2. Sequential, modifying (copy( ), remove_if( ))
3. Partitioning (**partition( )**, is_partitioned( ))
4. Sorting (sort( ), is_sorted( ))

Types of algorithms II of II

5. Binary Search (binary_search( ))
6. Merge (merge( ), inplace_merge( ))
7. Heap (make_heap( ), push_heap( ))
8. Min/Max (min( ), max( ))
9. Generalized numeric operations

Where are they?

1. <algorithm> (most of them)
2. <numeric> (a few generalized numeric operations)

PARTITION ALGORITHM

Let's explore an algorithm

- Let's check out **std::partition** in <algorithm>
- My top two sources of information are always:
  - http://www.cplusplus.com/reference/algorithm/partition/
  - http://en.cppreference.com/w/cpp/algorithm/partition
- Reorders the elements in [first, last). The elements for which a predicate p is true all precede elements for which the predicate p returns false
- Translation:
  - Sorts the elements using a function
  - The function evaluates each element and returns true or false
  - All the elements that are "true" come before elements that are "false"

Partitions

- Relative ordering of the elements is not maintained
- There are three parameters
  1. An iterator or pointer to the **first element** in the range
  2. An iterator or pointer to **one past the final element** in the range
  3. A pointer to a **function** that accepts a single parameter of the type being iterated over and returns a boolean
- Returns an **iterator** pointing to the first element in the second group (the "false" group), or the end of the container if there are no members of this group

Odd number partition example

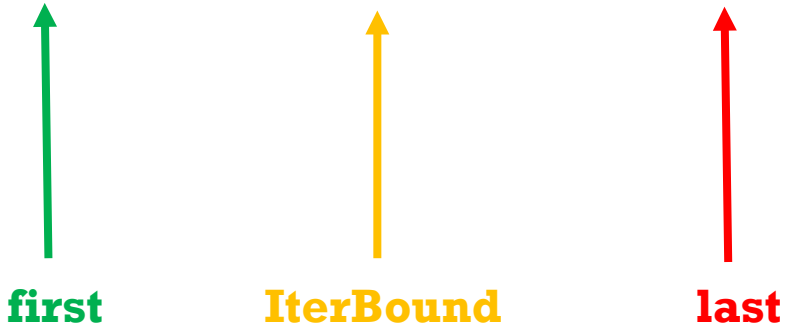
- Vector with 10 numbers
- Create a function IsOdd to return true/false if number is odd
- Want to partition the vector so odd numbers appear first
- What should happen after calling std::partition(vector.begin(), vector.end(), IsOdd)
  - Vector re-ordered and iterator returned (iterBound), pointing to first even number.

Partition **odd numbers** example

Before: {1,2,3,4,5,6,7,8,9,10}

Odd numbers appear before even numbers. **IterBound** points to first **EVEN** number

After {1,9,3,7,5,**6**,4,8,2,10}



“Possible” implementation *

```
template<class ForwardIt, class UnaryPredicate>
ForwardIt partition(ForwardIt first, ForwardIt last, UnaryPredicate p)
{
      first = std::find_if_not(first, last, p); //find first false
      if (first == last) return first;
      for (ForwardIt i = std::next(first); i != last; ++i)
      {
            if (p(*i)) {
                  std::iter_swap(i, first);
                  ++first;
            }
      }
      return first;
}
```

* http://en.cppreference.com/w/cpp/algorithm/partition

GOAL

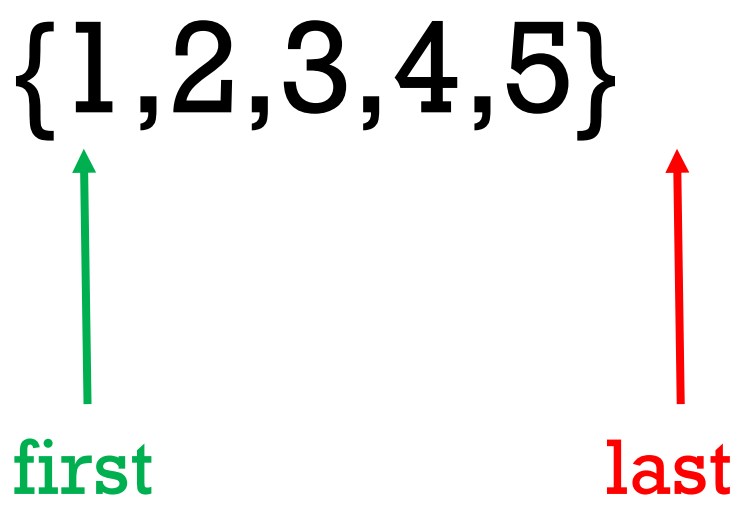
{1,2,3,4,5}

- Sort all odd numbers to the front of the list
- Even numbers in the back
- Provide
  - Iterator to beginning of list (first)
  - Iterator to end (last)
  - Function to determine if number is odd (IsOdd())

Odd number partition example

- Move iterator to first even number
- **p** is our function **IsOdd()**

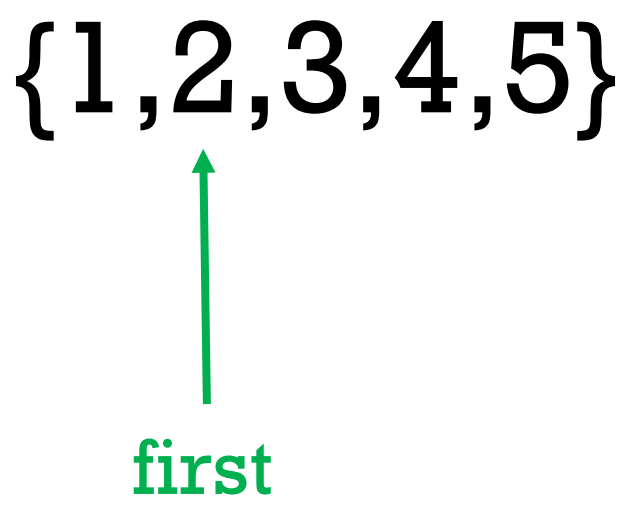
```
first = std::find_if_not(first, last, p); //find first even number
for (ForwardIt i = std::next(first); i != last; ++i)
	{
		if (p(*i)) //isOdd(3)?
		{
			std::iter_swap(i, first);
			++first;
		}
	}
```



Odd number partition example

- Move iterator to first even number
- p is our function IsOdd()

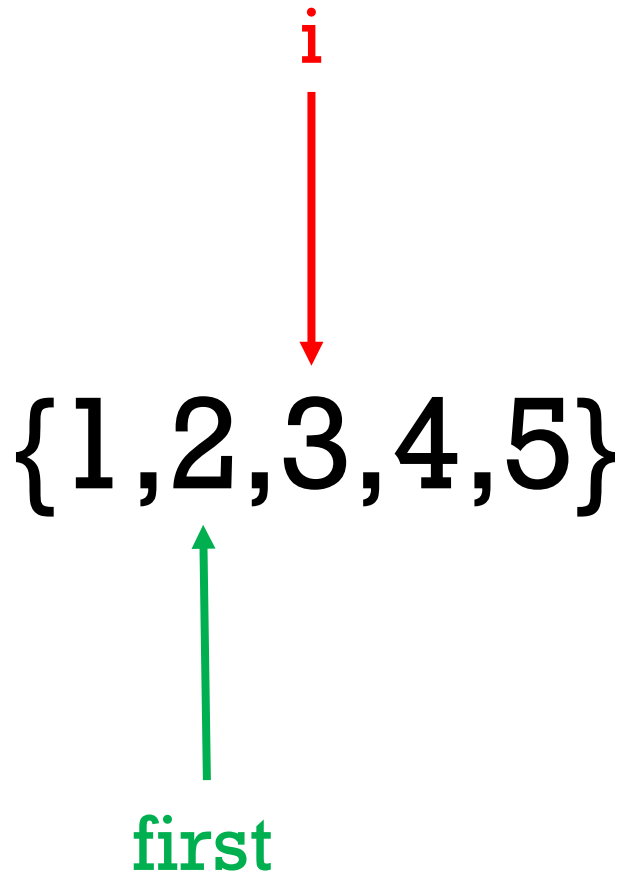
```
first = std::find_if_not(first, last, p); //find first even number
for (ForwardIt i = std::next(first); i != last; ++i)
	{
		if (p(*i)) //isOdd(3)?
		{
			std::iter_swap(i, first);
			++first;
		}
	}
```



Odd number partition example

- Move iterator **i** to number next to first
- Iterator **i** will seek ahead for odd numbers

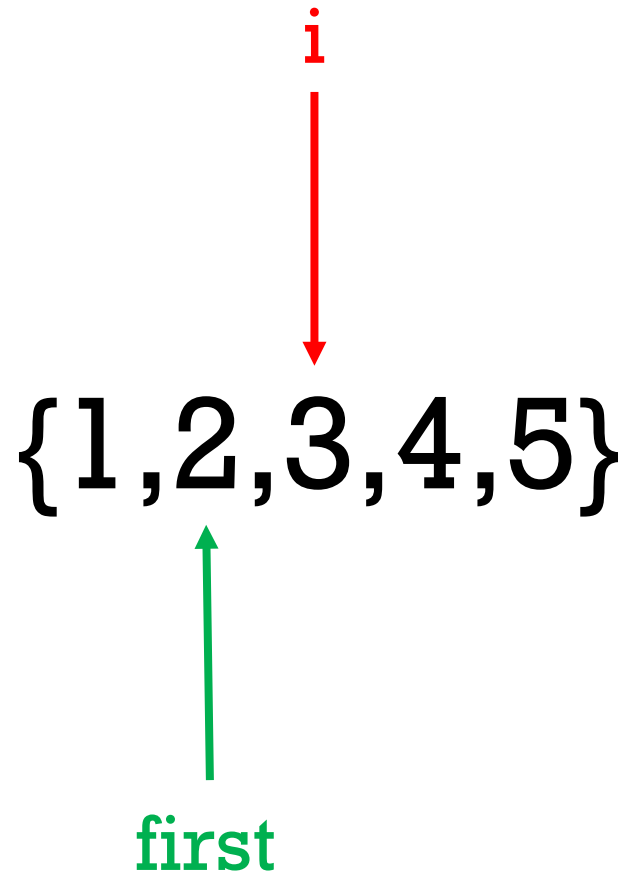
```
for (ForwardIt i = std::next(first); i != last; ++i)
        {
                if (p(*i)) //isOdd(3)?
                {
                        std::iter_swap(i, first);
                        ++first;
                }
        }
```



Odd number partition example

- Check if number i points at is an odd number
- p is our function IsOdd()

```
for (ForwardIt i = std::next(first); i != last; ++i)
    {
        if (p(*i)) //isOdd(3)?
        {
            std::iter_swap(i, first);
            ++first;
        }
    }
```

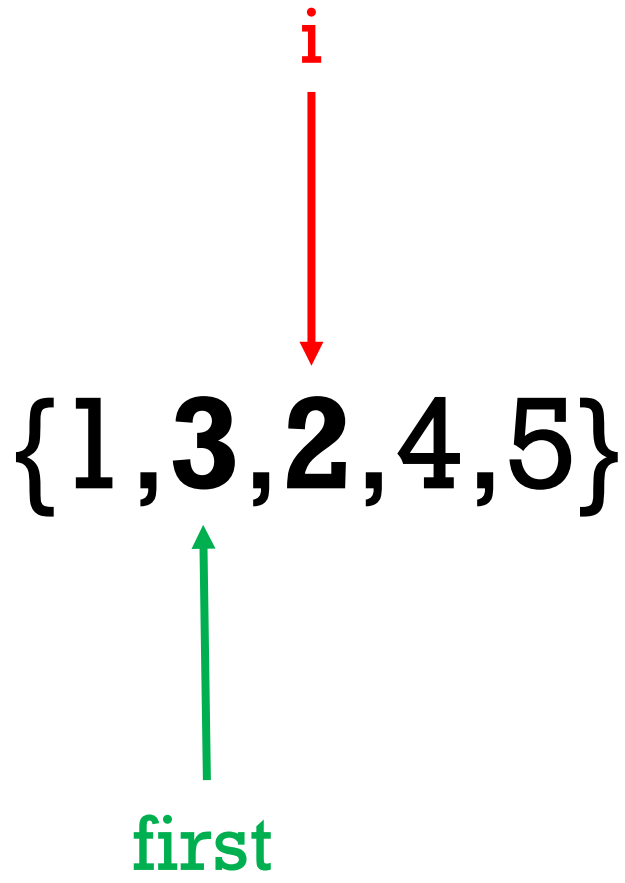



```
p(*i) //isOdd(3)? TRUE
```

Odd number partition example

- Swap the number iterators **i** and **first** are pointing at since we've found an odd number

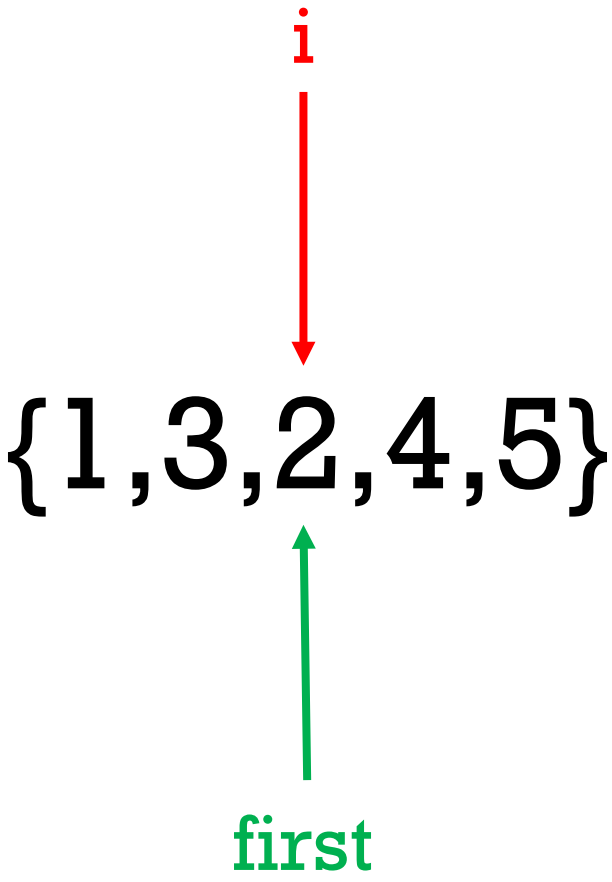
```
for (ForwardIt i = std::next(first); i != last; ++i)
        {
                if (p(*i))
                {
                        std::iter_swap(i, first);
                        ++first;
                }
        }
```



Odd number partition example

- After the swap, increment **first** to point to the next number

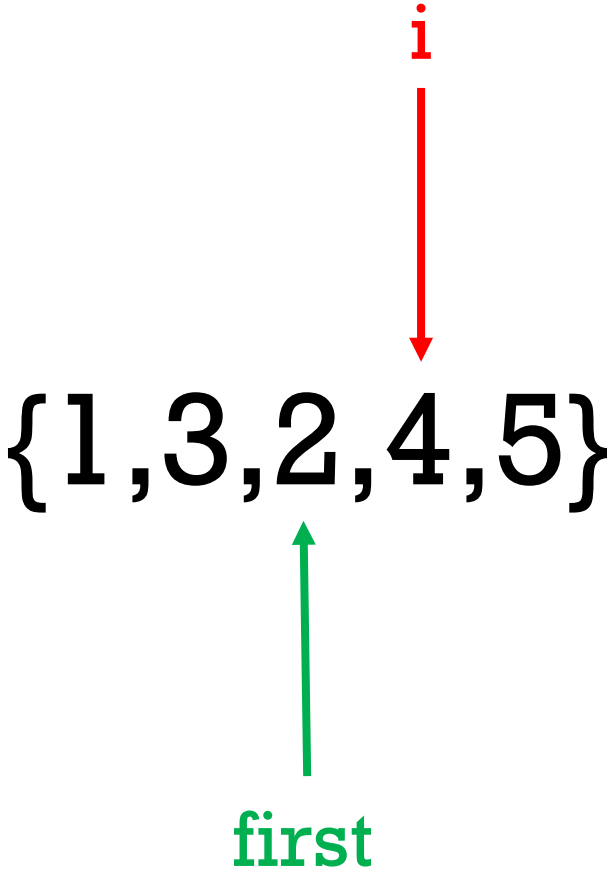
```
for (ForwardIt i = std::next(first); i != last; ++i)
        {
                if (p(*i))
                {
                        std::iter_swap(i, first);
                        ++first;
                }
        }
```



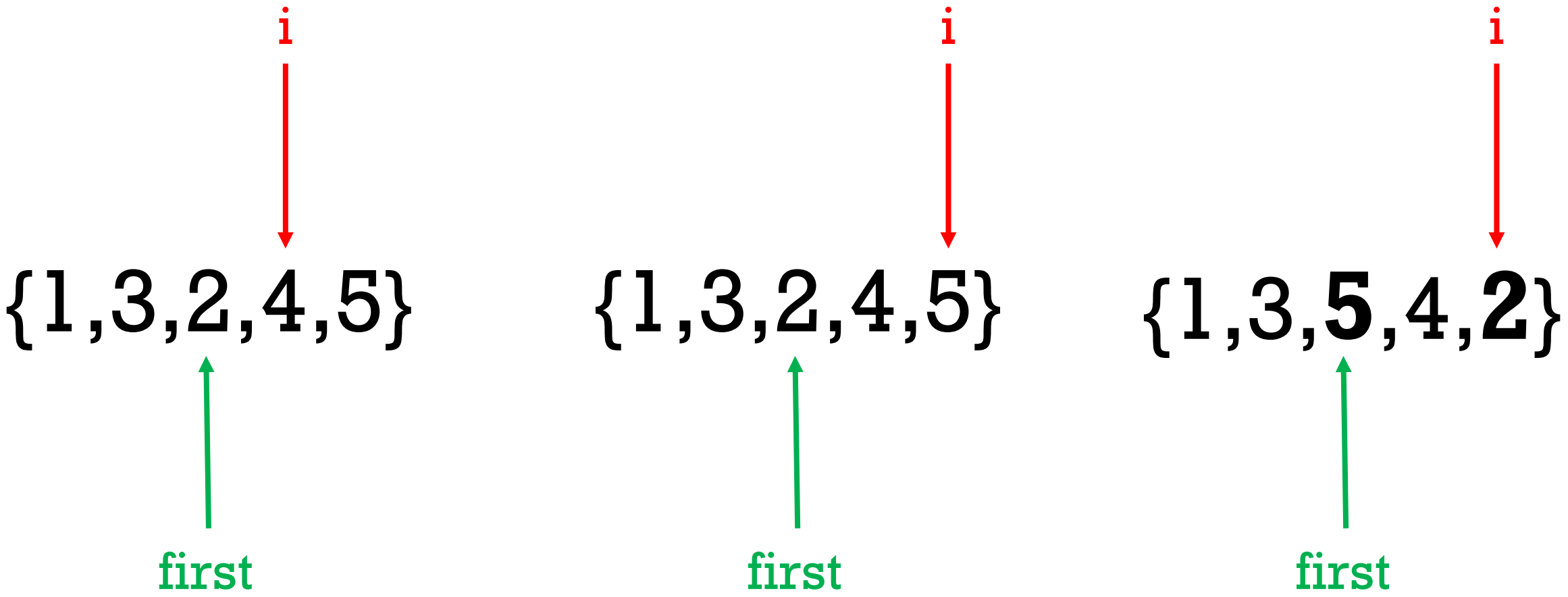
Odd number partition example

- Increment i to continue searching for the next odd number

```
for (ForwardIt i = std::next(first); i != last; ++i)
        {
                if (p(*i))
                {
                        std::iter_swap(i, first);
                        ++first;
                }
        }
```



Odd number partition example

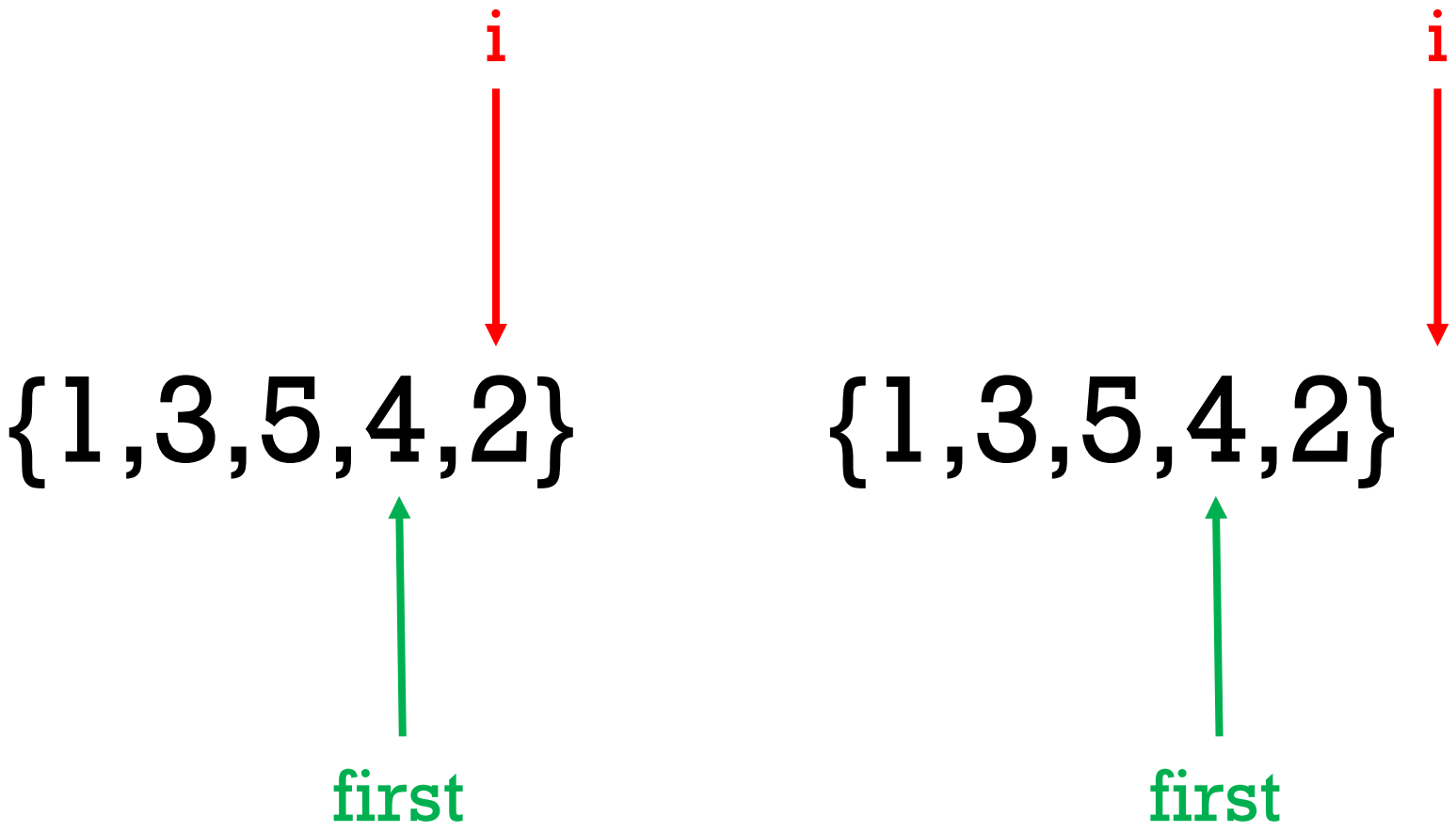



```
p(*i)? //isOdd(4)? FALSE
```

```
++i
p(*i)? //isOdd(5)? TRUE
```

```
std::iter_swap(i, first);
```

Odd number partition example



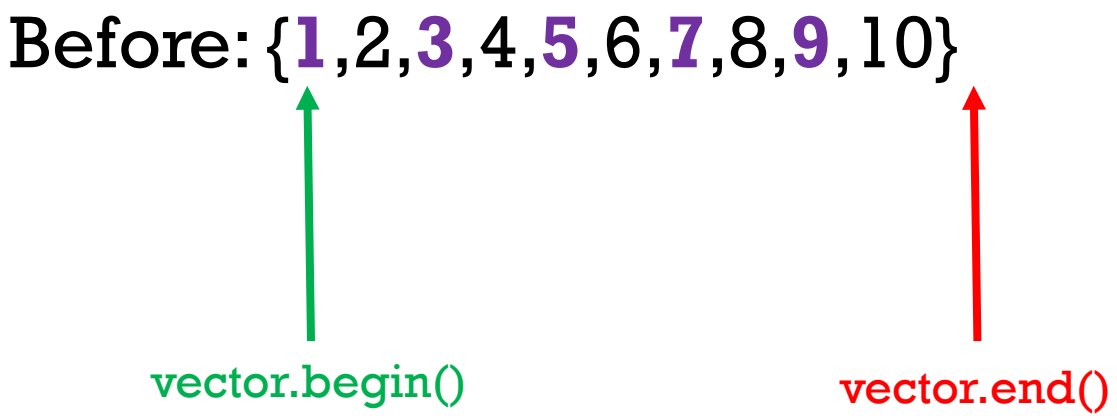

```
++first;
```

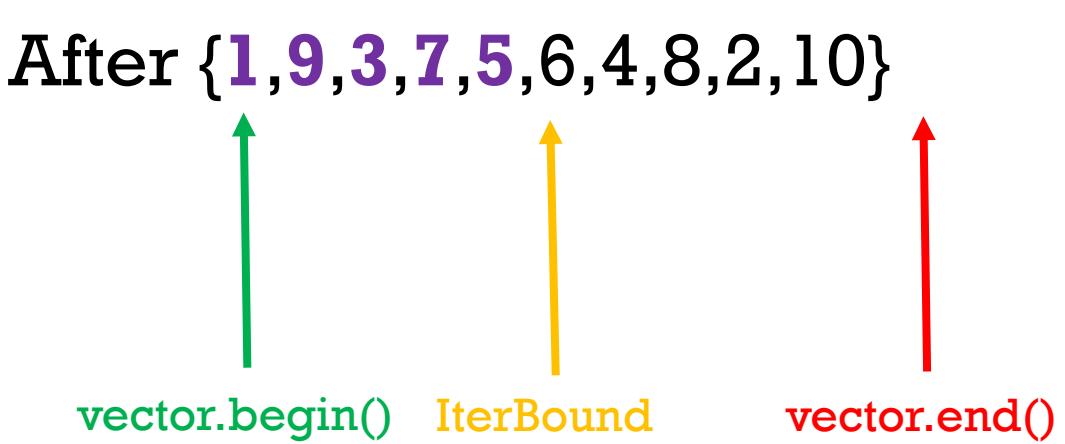
```
++i
i != last ? FALSE //leave loop
```

Odd number partition example

- Vector with 10 numbers
- Create a function IsOdd to return true/false if number is odd
- Want to partition the vector so odd numbers appear first
- What should happen after calling std::partition(vector.begin(), vector.end(), IsOdd)
  - Vector re-ordered and iterator returned (iterBound), pointing to first even number.

Odd number partition example





partitionOdd.cpp

ACTIVITY

1. Examine partition.cpp
2. This program generates cities in 2D space and partitions them based on distance
3. Examine how it uses different algorithms to achieve its output
   - partition algorithm
   - for_each algorithm

ACTIVITY

1. I made a simple Date class
2. Examine Date.hpp, Date.cpp, DateTester.cpp
3. Implement operator< in Date so that it has a strict weak ordering.
4. What is happening in the main method? Comment the code.

ACTIVITY

1. Select TWO data structures from the STL
2. Open the webpage http://www.cplusplus.com/reference/algorithm/
3. Select TWO algorithms
4. Write TWO little programs. Each program should use ONE data structure to demonstrate how ONE of the algorithms works.