

# COMP 3522

Object Oriented Programming in C++  
Week 7 Day 2

# Agenda

1. Review
2. STL Iterators
3. Algorithms

# COMP

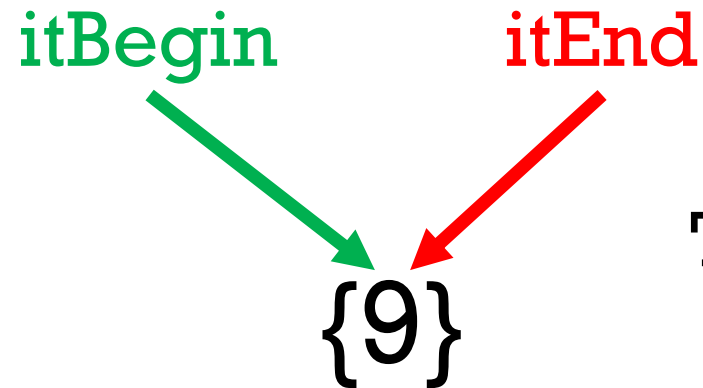
# 3522

REVIEW

# 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;  
}
```

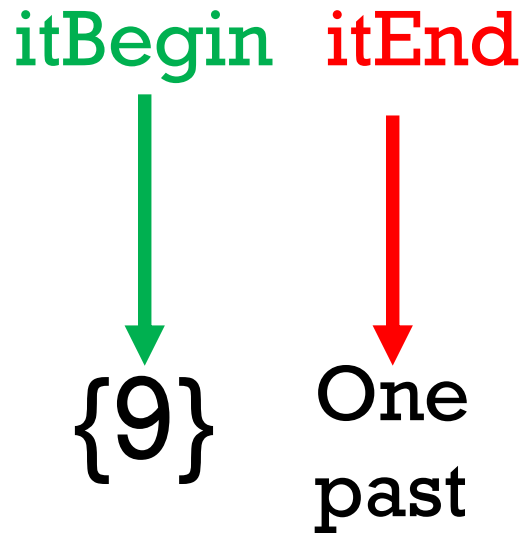


That doesn't look right...

# 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;  
}
```



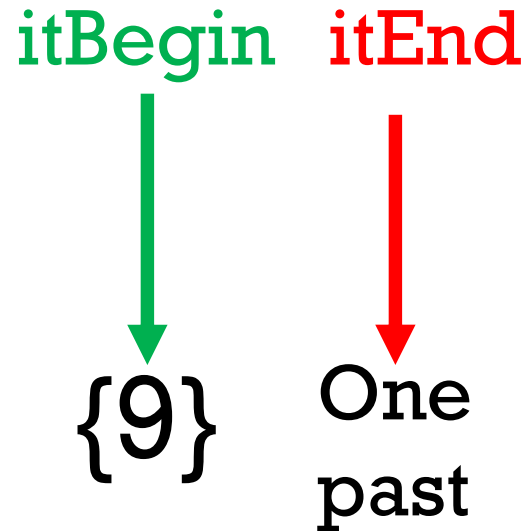
**end()** returns iterator pointing to theoretical one element past last element in range of values

`itEnd` does not point to element. Don't dereference

# 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;  
}
```



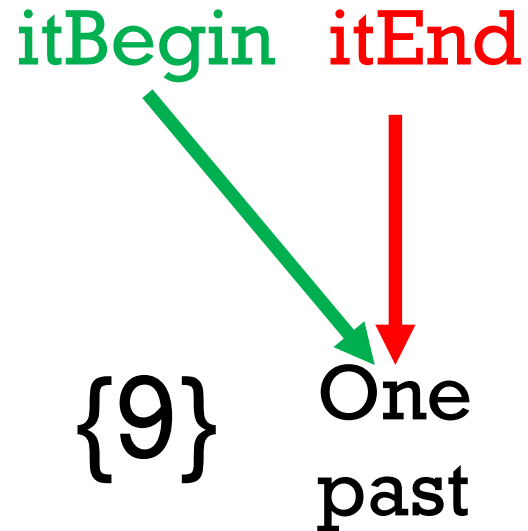
1st pass:  
`itBegin != itEnd`  
Do loop code

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;  
}
```



2<sup>nd</sup> 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;  
}
```

itBegin



{9, 10, 11}

itEnd



One  
past

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`

`}`

# 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

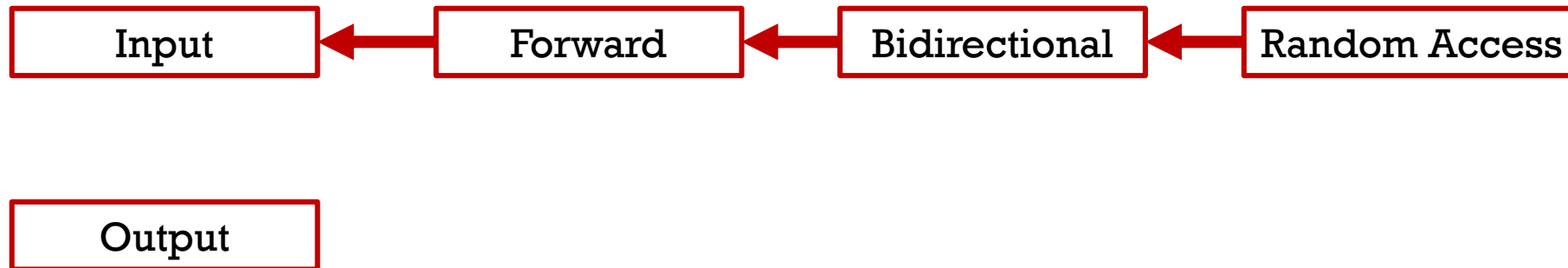
# Reminder: what's an iterator?

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

# Reminder: what's an iterator?

There are 5 kinds of iterators in C++

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



# 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.

# 1. Input iterator I

- Single pass forward direction sequential input operations: `==`, `!=`, `++`, `*it`
- 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

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



## 2. Output iterator I

- Single pass forward direction sequential output operations: `*it`, `++`, `*it = value`
- 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

<b>Supported Expression</b>	<b>Returns</b>	<b>Equivalent Expression</b>
<code>*i = some value</code>		
<code>++i</code>	It	<code>++i, return It temp = i;</code>
<code>i++</code>	It	<code>It temp = i; ++i; return temp;</code>
<code>*i++ = some value</code>		<code>*i = some value; ++i;</code>

### 3. Forward iterator

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

```
iterSaved = iter;
```

```
iter++;
```

```
cout << "Previous element is " << (*iterSaved) << endl;
```

```
cout << "Current element is " << (*iter) << endl;
```

## 4. Bidirectional iterator I

- 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.

## 4. Bidirectional iterator $\text{It} - i$ 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
<code>--i</code>	<code>It</code>	
<code>i--</code>	<code>It</code>	<code>It temp = i;</code> <code>--i;</code> <code>return temp;</code>
<code>*i--</code>	<code>value_type</code>	

## 5. Random access iterator I

- 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 bidirectional iterator, either. Each container defines its own specific iterator type that can loop through and access all the elements.

## 5. Random access iterator II

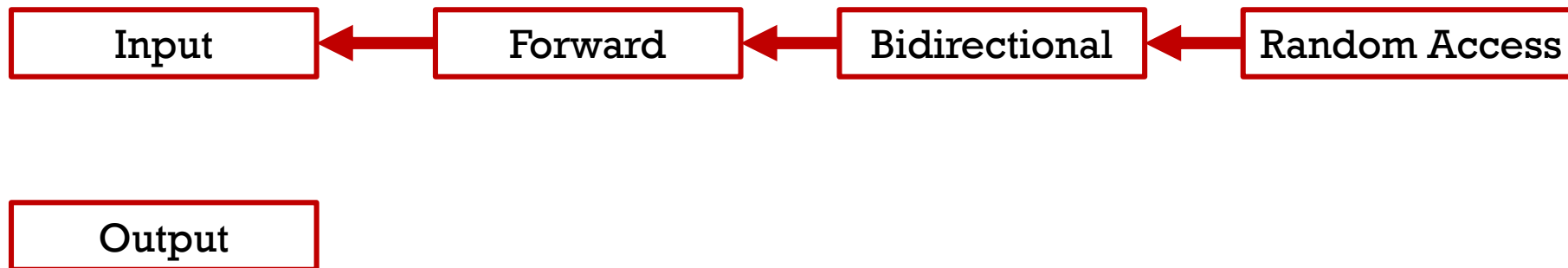
- Can be used like bidirectional iterator, and supports the following additional expressions:

Supported Expression	Returns	Equivalent Expression
$i += n, i -= n$ where $i = \text{It\&}$	It	
$i + n, i - n$	It	It temp = i; return temp += n;
$i2 - i$	difference_type	return difference
$i[n]$	value_type	$*(i + n)$
$i < i2, i \leq i2, i > i2, i \geq i2$	bool	

*i* is an iterator, *i2* is another iterator, *n* is a number

# 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`)



**RandomAccessIterator.cpp**



# Iterator adaptors for streams

- Sometimes a class will have the functionality you seek but not the right interface
- For example, the STL `copy()` algorithm requires a pair of **input iterators** as its first two parameters to give the range of values to be copied

```
template<class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result)
{
    while (first!=last) {
        *result = *first;
        ++result; ++first;
    }
    return result;
}
```

# Iterator adaptors for streams

- An istream object can act as a source of such data values but it does not have any iterators that the copy algorithm can use
- Similarly the third parameter of the copy() algorithm is an **output iterator** that directs the copied values to their proper destination. That destination could be an output stream but output streams do not directly provide any output iterators.

```
...  
OutputIterator copy (InputIterator first, InputIterator last, OutputIterator result)  
{  
...  
}
```

# Iterator adaptors

- An *adaptor class* is one that acts like a "translator" by "adapting" the messages you want to send to produce messages that the other class object wants to receive.
- An iterator adaptor class called `istream_iterator` provides the interface that the `copy()` algorithm expects for input and translates requests from this algorithm into appropriate `istream` operations
- Another called `ostream_iterator` provides the interface that the `copy()` algorithm expects for output and translates requests from the algorithm into appropriate `ostream` operations

# istream\_iterator

- Single pass input iterator
- Reads successive objects from an `std::basic_istream` (like `cin`) by calling the appropriate operator`>>`
- The first object is read when the iterator is constructed
- Future reads are performed when the iterator is incremented, not when it is dereferenced
- Constructing an `istream_iterator` using the default constructor creates a well know object called the end-of-stream iterator
- This is just like `container.end( )` for a stream iterator
- See **IteratorAdaptor.cpp**

# In C++, can we delete while iterating?

- **Yes!**
- Recall in Java we need to use the iterator's remove function
- **In C++ we use the container's erase function, which returns the iterator to the next element**
- Check out **delete.cpp**

# How about sorting words and removing dups?

- What if we have a source
- We want to **sort**
- We want to **remove duplicates**
- **Sounds like a job for a set!**
- Check out **setsort.cpp**

# ACTIVITY

1. Select STL containers that use forward, bidirectional, and random access iterators.
2. Write a small program that instantiates each container and uses C++-style random number generation to fill it with 20 random doubles.
3. Use the iterator to iterate over the list. Showcase the abilities, i.e., read, write, increment, decrement, random access, delete.

# STL Containers

- Sequence Containers:
  - vector
  - list
  - deque
  - arrays
  - forward\_list
- Associative Containers
  - set
  - multiset
  - map
  - multimap
- Container Adaptors
  - queue
  - priority\_queue
  - stack



# STL Iterators

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

# 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](http://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. Generalize numeric operations**

# Where are they?

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

# 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) so that 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”



# Okay that sounds exciting but how's it work?

```
template <class ForwardIterator, class UnaryPredicate>  
ForwardIterator partition ( ForwardIterator first,  
                           ForwardIterator last,  
                           UnaryPredicate pred);
```

Looks complicated...let's break it down

# 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

# “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

- Find iterator to first number that's even
- p is our function IsOdd()

```
first = std::find_if_not(first, last, p);
```

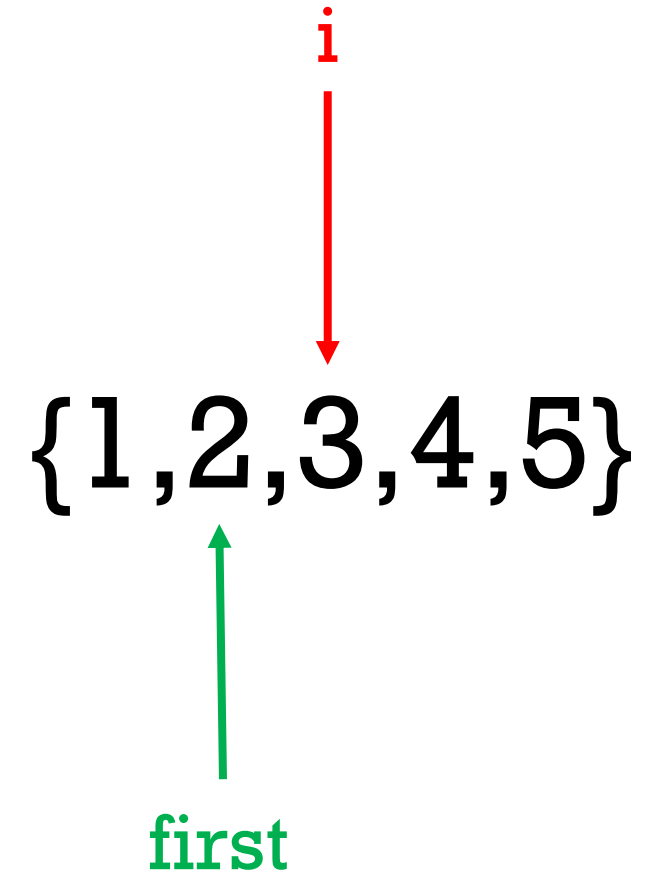
{1,2,3,4,5}



first

# Odd number partition example

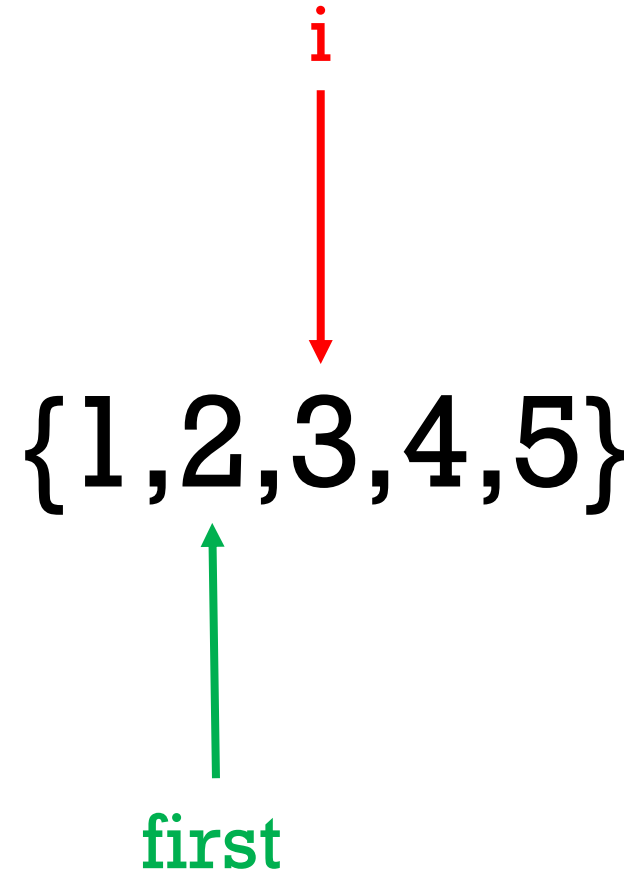
```
ForwardIt i = std::next(first)
```



# Odd number partition example

- `p` is our function `IsOdd()`

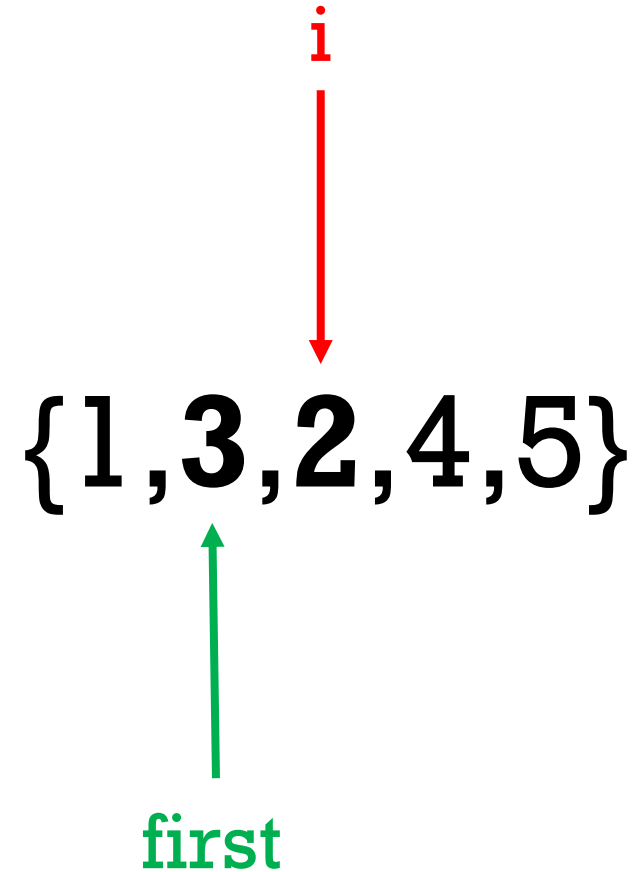
```
for (ForwardIt i = std::next(first); i != last; ++i)
{
    if (p(*i)) {
        std::iter_swap(i, first);
        ++first;
    }
}
```



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

# Odd number partition example

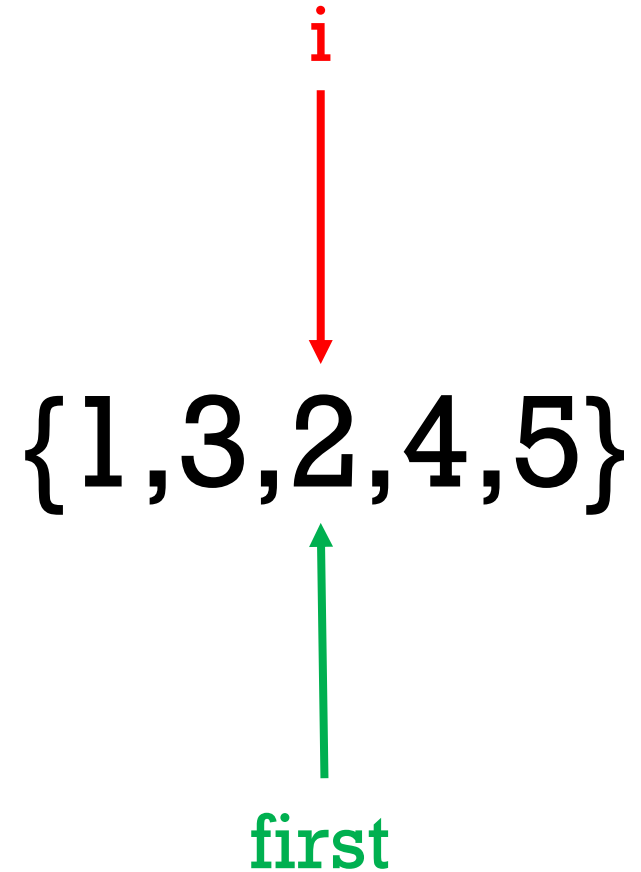
```
for (ForwardIt i = std::next(first); i != last; ++i)
{
    if (p(*i)) {
        std::iter_swap(i, first);
        ++first;
    }
}
```





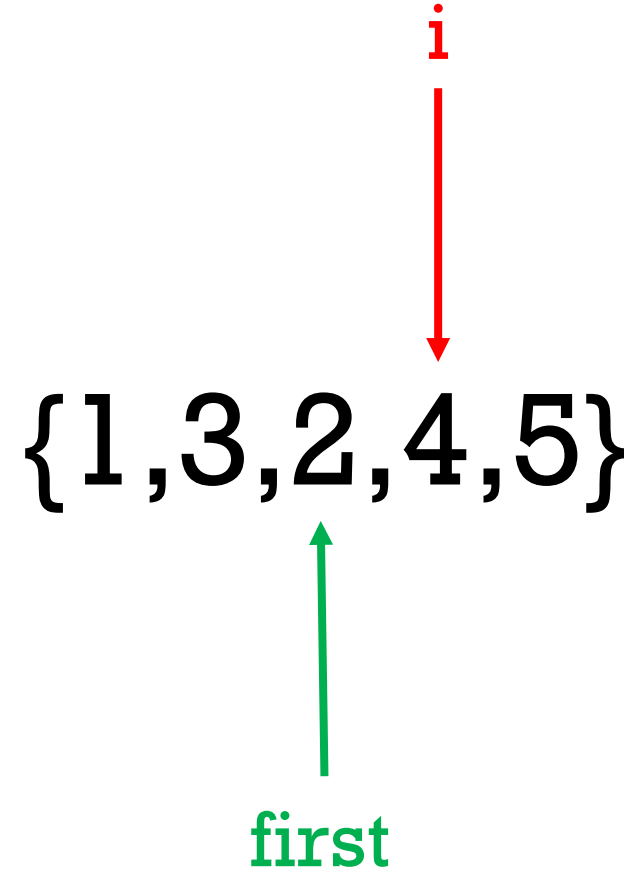
# Odd number partition example

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

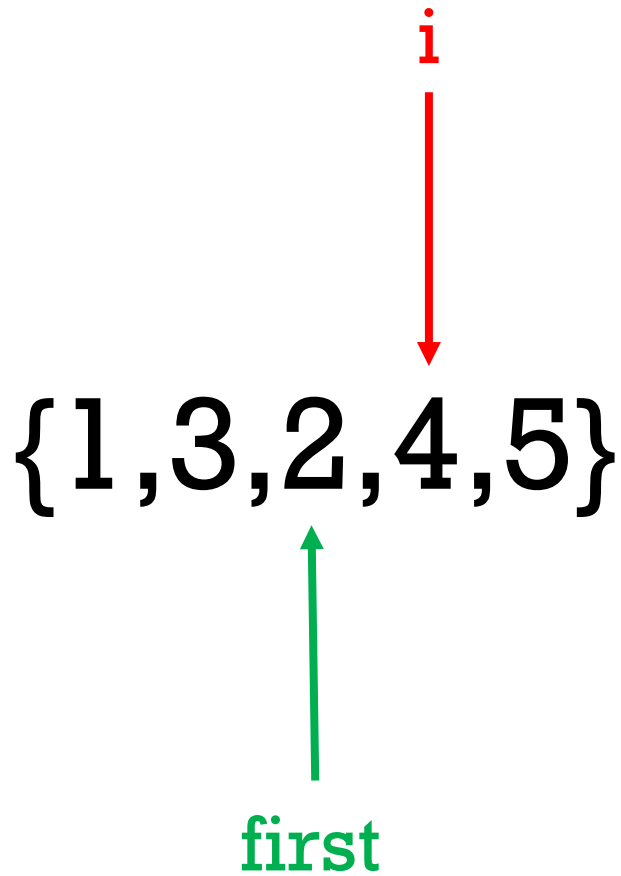


# Odd number partition example

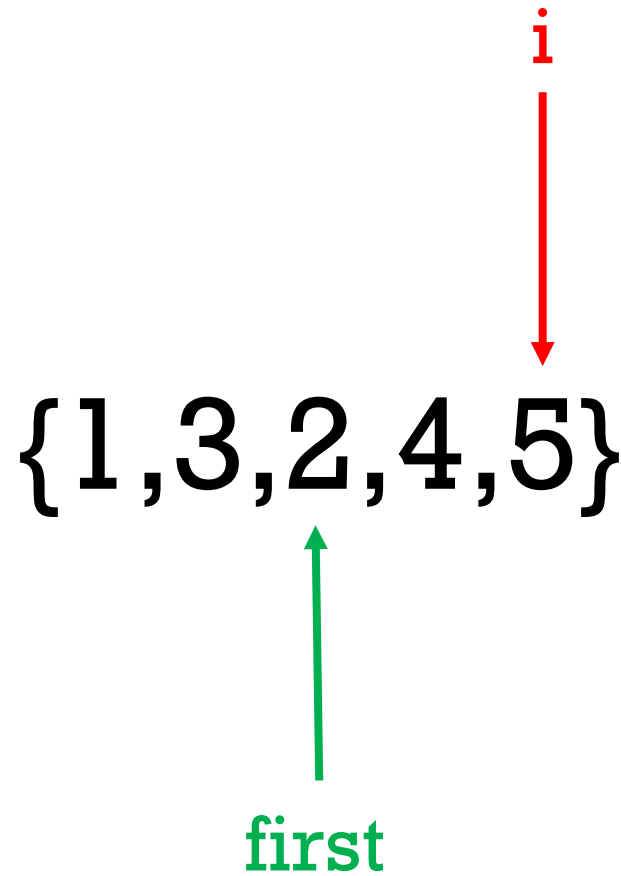
```
for (ForwardIt i = std::next(first); i != last; ++i)
{
    if (p(*i)) {
        std::iter_swap(i, first);
        ++first;
    }
}
```



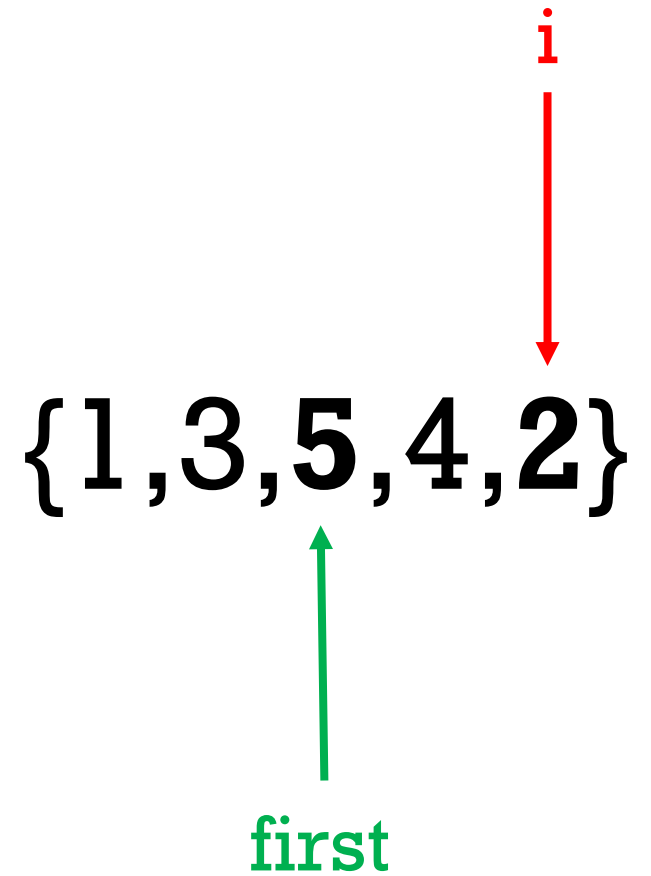
# Odd number partition example



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

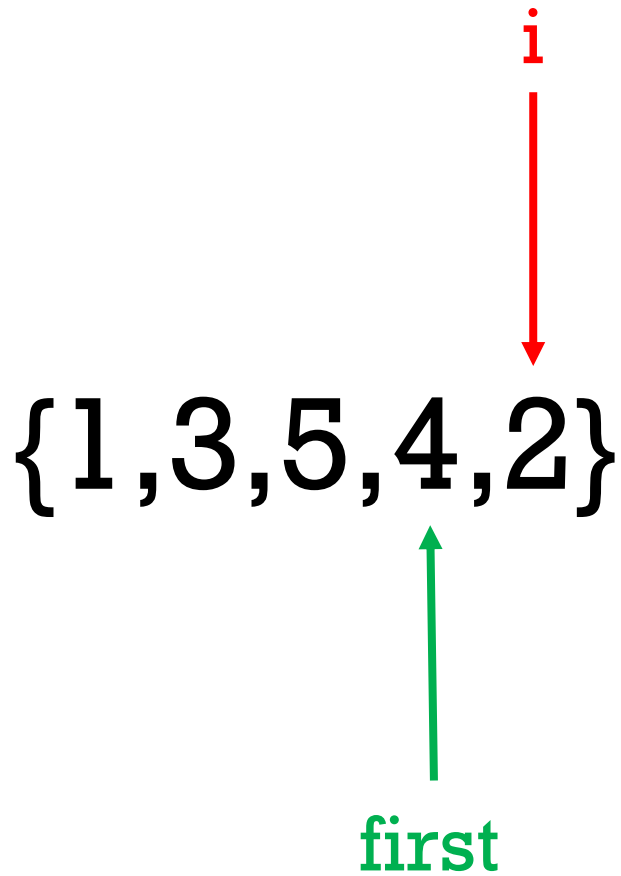


```
++i  
p(*i)? //isOdd(5)? TRUE
```

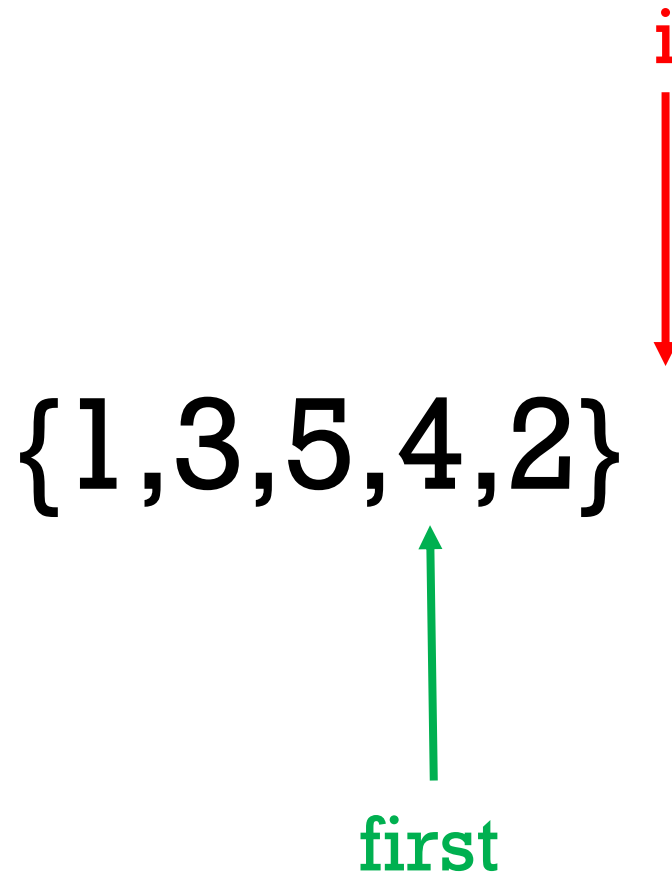


```
std::iter_swap(i, first);
```

# 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


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


  
`vector.begin()`

  
`vector.end()`

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

  
`vector.begin()`

  
`IterBound`

  
`vector.end()`

**partitionOdd.cpp**

# So I wrote a little program

- I chose a container (a list)
  - I chose a type of object to store in the list (a point in 2D space)
  - I chose a predicate (the point is within a specific distance from the origin)
  - I started coding...
- 
- I wrote **partition.cpp**

# Note: Strict weak ordering

- Almost all C++ STL functions/containers require the ordering to satisfy the standard mathematical definition of a strict weak ordering.
- Let `compare(left, right)` be a comparison function. The compare function is in strict weak ordering iff:
  1. IRREFLEXIVITY: **`compare(x, x) == false`**
  2. ANTISYMMETRY: **if `compare(x, y)` then `!compare(y, x)`**
  3. TRANSITIVITY: **if `compare(x, y)` and `compare(y, z)` then `compare(x, 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
    - Functors! We'll talk about this later in the term

# 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.