CMSC 341
Lecture 6 – STL, Stacks, & Queues

Based on slides by Lupoli, Dixon & Gibson at UMBC

Templates

Common Uses for Templates

- Some common algorithms that easily lend themselves to templates:
  - Swap
  - … what else?
  - Sort
  - Search
  - FindMax
  - FindMin

`maxx()` Overloaded Example

```
float    maxx ( const float a, const float b );
int      maxx ( const int a, const int b );
Rational maxx ( const Rational& a, const Rational& b);
myType   maxx ( const myType& a, const myType& b);
```

- Code for each looks the same…

```
if ( a < b )
    return b;
else
    return a;
```

we want to reuse this code for **all** types

What are Templates?

- Templates let us create functions and classes that can work on “generic” input and types

- This means that functions like **`maxx()`** only need to be written once
  - And can then be used for almost anything

Indicating Templates

- To let the compiler know you are going to apply a template, use the following:

```
template <class T>
```

- What this line means overall is that we plan to use "**`T`**" in place of a data type
  - *e.g.*, **`int`**, **`char`**, **`myClass`**, etc.
- This template prefix needs to be used before function declarations and function definitions

Template Example

Function Template

```
template <class T>
T maxx ( const T& a, const T& b)
{
  if ( a < b )
    return b;
  else
    return a;
}
```

Notice how '**T**' is mapped to '**int**' everywhere in the function…

Compiler generates code based on the argument type

```
cout << maxx(4, 7) << endl;
```

Generates the following:

```
int maxx ( const int& a, const int& b)
{
  if ( a < b )
    return b;
  else
    return a;
}
```

Using Templates

- When we call these templated functions, nothing looks different:

```
SwapVals(intOne,     intTwo);
SwapVals(charOne,    charTwo);
SwapVals(strOne,     strTwo);
SwapVals(myClassA,  myClassB);
```

Templating Classes

- Want to be able to define classes that work with various types of objects
- Shouldn't matter what kind of object it stores

- Generic "collections" of objects
  - Linked List
  - Stack
  - Vector
  - Binary Tree (341)
  - Hash Table (341)

Making a Templated Class

- Three key steps:
  - Add template line
    - Before class declaration
  - Add template line
    - Before each method in implementation
  - Change class name to include template
    - Add **`<T>`** after the class name wherever it appears

Example: Templated Node

```
template <class T>
class Node
{
   public:
      Node( const T& data );
      const T& GetData();
      void SetData( const T& data );
      Node<T>* GetNext();
      void SetNext( Node<T>* next );

   private:
      T m_data;
      Node<T>* m_next;
};

template <class T>
Node<T>::Node( const T& data )
{
   m_data = data;
   m_next = NULL;
}
```

```
template <class T>
const T& Node<T>::GetData()
{
   return m_data;
}

template <class T>
void Node<T>::SetData( const T& data )
{
   m_data = data;
}

template <class T>
Node<T>* Node<T>::GetNext()
{
   return m_next;
}

template <class T>
void Node<T>::SetNext( Node<T>* next )
{
   m_next = next;
}
```

Example: Templated Stack

```
template <class T>
class Stack
{
   public:
      Stack();
      void Push(const T& item);
      T Pop();

   private:
      Node<T>* m_head;
};
```

```
template <class T>
Stack<T>::Stack()
{
   m_head = NULL;
}
```

```
template <class T>
void Stack<T>::Push(const T& item)
{
   Node<T>* newNode = new Node<T>(item);
   newNode->SetNext(m_head);
   m_head = newNode;
}
```

```
template <class T>
T Stack<T>::Pop()
{
   T data = m_head->GetData();
   Node<T>* temp = m_head;
   m_head = temp->GetNext();
   delete temp;
   return data;
}
```

Using the Templated Stack

```
int main()
{
   Stack<int>    nums;
   Stack<string> names;

   nums.Push(7);
   nums.Push(8);
   cout << nums.Pop() << endl;
   cout << nums.Pop() << endl;

   names.Push("Freeman");
   names.Push("Hrabowski");
   cout << names.Pop() << endl;
   cout << names.Pop() << endl;

   return 0;
}
```

Multiple Templated Types

Example: Pair

```
template < class Key, class Data >
class Pair
{
  public:
    Pair( );
    ~Pair( );
    Pair( const Pair<Key, Data>& pair);
    bool operator== (const Pair<Key, Data>& rhs) const;

  private:
    Key   m_key;
    Data m_data;
};

// Pair's equality operator
template <class K, class D>
bool Pair<K, D>::operator== (const Pair<K,D>& rhs) const
{
    return m_key == rhs.m_key && m_data == rhs.m_data;
}
```

Using the Pair Template

```
int main ( )
{
   string name1 = "Thunder";
   string name2 = "Jasper";

   // use pair to associate a string and its length
   Pair< int, string > dog (name1.length(), name1);
   Pair< int, string > cat (name2.length(), name2);

    // check for equality
   if (dog == cat)
      cout << "All animals are equal!" << endl;
    return 0;
}
```

Using the Pair Template (Example 2)

```
int main ( )
{
   // use Pair for names and Employee object
   Employee john, mark;

   Pair< string, Employee > boss  ("John", john);
   Pair< string, Employee > worker("Mark", mark);

   if (boss == worker)
      cout << "A real small company" << endl;

    return 0;
}
```

Miscellaneous Extra Template Info

Templates as Parameters

- Not much different from a “regular” variable

```
template <class T>
void Sort ( SmartArray<T>& theArray )
{
   // code here
}
```

- Make sure that the behaviors used in the function are defined for the type you’re using

Standard Template Library (STL)

Standard Template Library (STL)

- The Standard Template Library (*STL*) is a C++ library of container classes, algorithms, and iterators
- Provides many of the basic algorithms and data structures of computer science

From: https://www.sgi.com/tech/stl/stl_introduction.html

Considerations of the STL

- Containers replicate structures very commonly used in programming.
- Many containers have several member functions in common, and share functionalities.

From: http://www.cplusplus.com/reference/stl/

Considerations of the STL

- The decision of which type of container to use for a specific need depends on:
  - the functionality offered by the container
  - the efficiency of some of its members (complexity)

From: http://www.cplusplus.com/reference/stl/

Types of Containers

Focus of Today

- Sequence containers
  - Array, vector, deque, forward_list, list
- Container adapters
  - Stacks, queues, priority_queues
- Associative containers (and the unordered)
  - Set, multiset, map, multimap

Standard Containers

- Sequences:
  - **vector**: Dynamic array of variables, struct or objects. Insert data at the end.
  - **list**: Linked list of variables, struct or objects. Insert/remove anywhere.
  - Sequence **<u>means order does matter</u>**

Container Adapters

- Container adapters:
  - **stack** LIFO
  - **queue** FIFO
  - adapter means ***VERY LIMITED*** functionality

Will we use STL?

- Today we are going to talk about the ways that we can implement stacks and queues

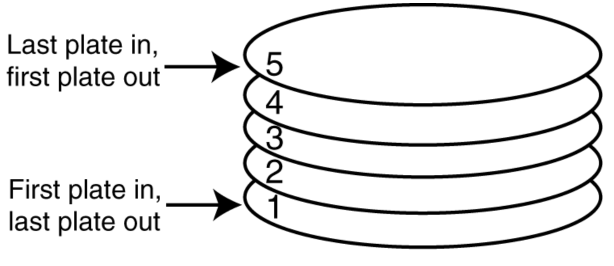
- 3 Ways to Create a Stack or Queue
  - Create a static stack or queue using an array
  - Create a dynamic stack or queue using a linked list
  - Create a stack or queue using the STL

Stacks

Stacks

Introduction to Stacks

- A *stack* is a data structure that stores and retrieves items in a last-in-first-out (LIFO) manner.



Applications of Stacks

- Computer systems use stacks during a program’s execution to store function return addresses, local variables, etc.
- Some calculators use stacks for performing mathematical operations.

Implementations of Stacks

- Static Stacks
  - Fixed size
  - Can be implemented with an array
- Dynamic Stacks
  - Grow in size as needed
  - Can be implemented with a linked list
- Using STL (dynamic)

Stack Operations

- Push
  - causes a value to be stored in (pushed onto) the stack
- Pop
  - retrieves and removes a value from the stack

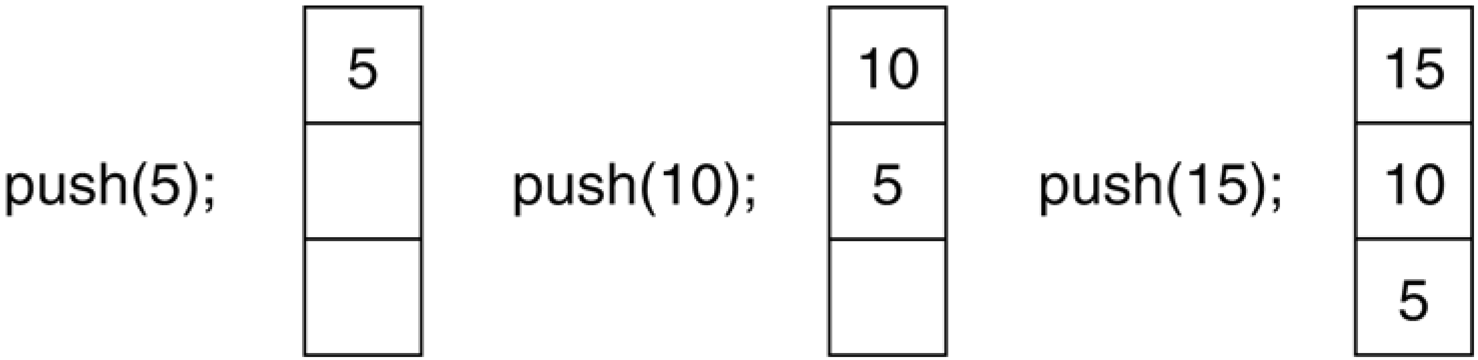
The Push Operation

- Suppose we have an empty integer stack that is capable of holding a maximum of three values. With that stack we execute the following push operations.

```
push(5);
push(10);
push(15);
```

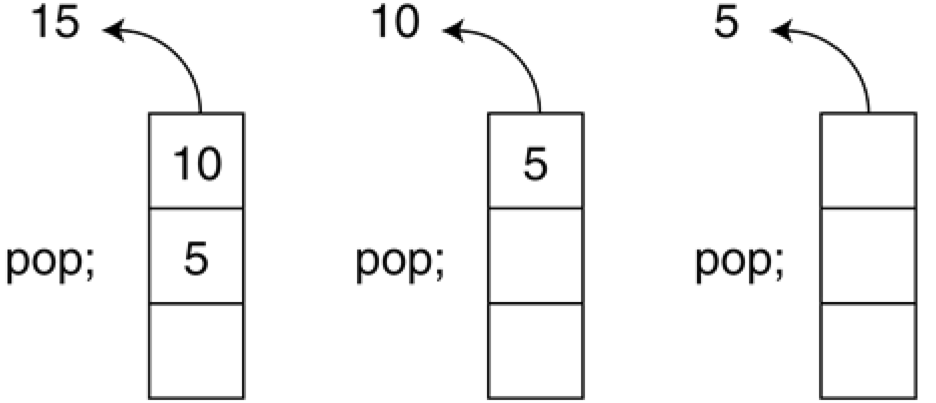
The Push Operation

The state of the stack after each of the `push` operations:



The Pop Operation

- Now, suppose we execute three consecutive pop operations on the same stack:



Other Stack Operations

- **`isFull()`**: A Boolean operation needed for static stacks. Returns true if the stack is full. Otherwise, returns false.
- **`isEmpty()`**: A Boolean operation needed for all stacks. Returns true if the stack is empty. Otherwise, returns false.

Static Stacks

Static Stacks

- A *static stack* is built on an array
  - As we are using an array, we must specify the starting size of the stack
  - The stack may become full if the array becomes full

Member Variables for Stacks

- Three major variables:
  - **`Pointer`** Creates a pointer to stack
  - **`size`** Tracks elements in stack
  - **`top`** Tracks top element in stack

Member Functions for Stacks

- **`CONSTRUCTOR`** Creates a stack
- **`DESTRUCTOR`** Deletes a stack
- **`push()`** Pushes element to stack
- **`pop()`** Pops element from stack
- **`isEmpty()`** Is the stack empty?
- **`isFull()`** Is the stack full?

Static Stack Definition

```
#ifndef INTSTACK_H
#define INTSTACK_H

class IntStack
{
private:
        int *stackArray;
        int stackSize;
        int top;

public:
        IntStack(int);
        ~IntStack()
           {delete[] stackArray;}
        void push(int);
        void pop(int &);
        bool isFull();
        bool isEmpty();
};

#endif
```

pointer
size()
top()

Member Variables

Constructor
Destructor

push()
pop()
isFull()
isEmpty()

Member Functions

Dynamic Stacks

Dynamic Stacks

- A *dynamic stack* is built on a linked list instead of an array.
- A linked list-based stack offers two advantages over an array-based stack.
  - No need to specify the starting size of the stack. A dynamic stack simply starts as an empty linked list, and then expands by one node each time a value is pushed.
  - A dynamic stack will never be full, as long as the system has enough free memory.

Member Variables for Dynamic Stacks

- Parts:
  - `Linked list` Linked list for stack (nodes)
  - `size` Tracks elements in stack

Member Functions for Dynamic Stacks

`CONSTRUCTOR`	Creates a stack
`DESTRUCTOR`	Deletes a stack
`push()`	Pushes element to stack
`pop()`	Pops element from stack
`isEmpty()`	Is the stack empty?
`top()`	What is the top element?

What happened to `isFull()`?

Dynamic Stack

```
class DynIntStack
{
private:
     struct StackNode
     {
          int value;
          StackNode *next;
     };

     StackNode *top;

public:
     DynIntStack(void)
          {    top = NULL; }
     void push(int);
     void pop(int &);
     const Elem& top() const throw(StackEmpty);
     bool isEmpty(void);
};
```

Linked list of elements

value

pointer

top

Member Variables

Constructor

push()

pop()

top()

isEmpty()

Member Functions

Common Problems with Stacks

- Stack underflow
  - no elements in the stack, and you tried to pop
- Stack overflow
  - maximum elements in stack, and tried to add another
  - not an issue using STL or a dynamic implementation

Queues

Introduction to the Queue

- Like a stack, a queue is a data structure that holds a sequence of elements.
- A queue, however, provides access to its elements in *first-in, first-out (FIFO)* order.
- The elements in a queue are processed like customers standing in a line: the first customer to get in line is the first one to be served (and leave the line).

Example Applications of Queues

- In a multi-user system, a queue is used to hold print jobs submitted by users, while the printer services those jobs one at a time.
- Communications software also uses queues to hold information received over networks. Sometimes information is transmitted to a system faster than it can be processed, so it is placed in a queue when it is received.

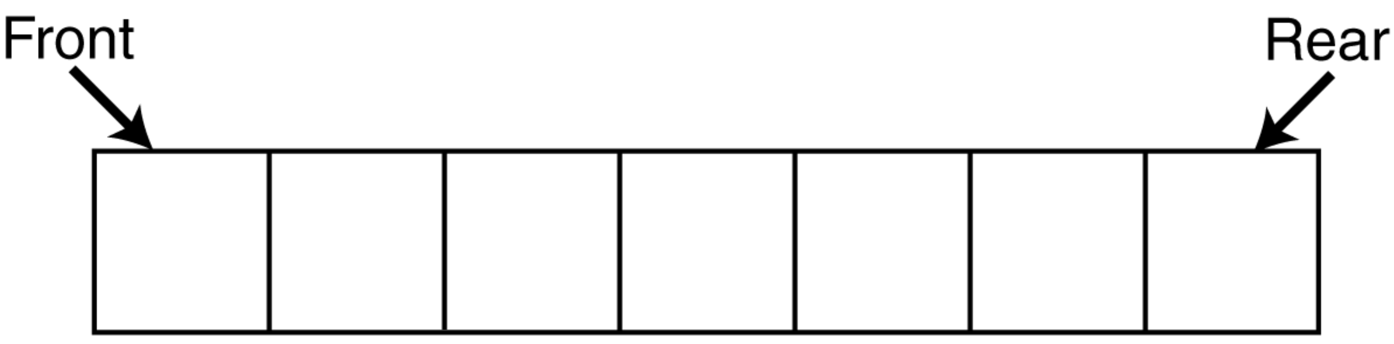
Implementations of Queues

- Static Queues
  - Fixed size
  - Can be implemented with an array
- Dynamic Queues
  - Grow in size as needed
  - Can be implemented with a linked list
- Using STL (dynamic)

Just like stacks!

Queue Operations

- Think of queues as having a front and a rear.
  - rear: position where elements are added
  - front: position from which elements are removed



Queue Operations

- The two primary queue operations are *enqueuing* and *dequeuing*.
- To *enqueue* means to insert an element at the rear of a queue.
- To *dequeue* means to remove an element from the front of a queue.

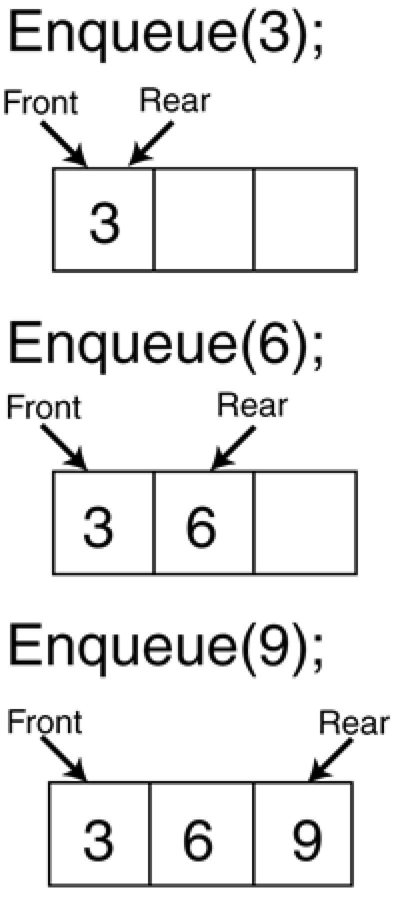
Queue Operations

- Suppose we have an empty static integer queue that is capable of holding a maximum of three values. With that queue we execute the following enqueue operations.

```
Enqueue(3);
Enqueue(6);
Enqueue(9);
```

Queue Operations - Enqueue

- The state of the queue after each of the enqueue operations.



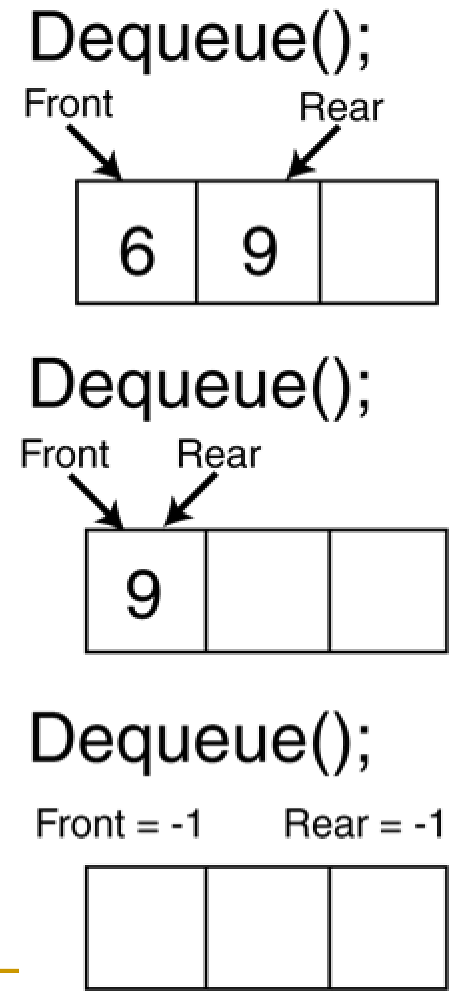
Queue Operations - Dequeue

- Now let's see how dequeue operations are performed. The figure on the right shows the state of the queue after each of three consecutive dequeue operations
- An important remark
  - After each dequeue, remaining items shift toward the front of the queue.

3 removed

6 removed

9 removed



Efficiency Problem of Dequeue & Solution

- Shifting after each dequeue operation causes inefficiency.
- Solution
  - Let front index move as elements are removed
  - let rear index "wrap around" to the beginning of array, treating array as circular
    - Similarly, the front index as well
  - Yields more complex enqueue, dequeue code, but more efficient
  - Let's see the trace of this method on the board for the enqueue and dequeue operations given on the right (queue size is 3)

```
Enqueue(3);
Enqueue(6);
Enqueue(9);
Dequeue();
Dequeue();
Enqueue(12);
Dequeue();
```

Implementation of a Static Queue

- The previous discussion was about static arrays
  - Container is an array
- Class Implementation for a static integer queue
  - Member functions
    - **`enqueue()`**
    - **`dequeue()`**
    - **`isEmpty()`**
    - **`isFull()`**
    - **`clear()`**

Member Variables for Static Queues

- Five major variables:
  - **`queueArray`** Creates a pointer to queue
  - **`queueSize`** Tracks capacity of queue
  - **`numItems`** Tracks elements in queue
  - **`front`**
  - **`rear`**
    - The variables front and rear are used when our queue “rotates,” as discussed earlier

Member Functions for Queues

- **`CONSTRUCTOR`** Creates a queue
- **`DESTRUCTOR`** Deletes a queue
- **`enqueue()`** Adds element to queue
- **`dequeue()`** Removes element from queue
- **`isEmpty()`** Is the queue empty?
- **`isFull()`** Is the queue full?
- **`clear()`** Empties queue

Static Queue Example

```
#ifndef INTQUEUE_H
#define INTQUEUE_H

class IntQueue
{
private:
      int *queueArray;
      int queueSize;
      int front;
      int rear;
      int numItems;
public:
      IntQueue(int);
      void enqueue(int);
      void dequeue(int &);
      bool isEmpty() const;
      bool isFull() const;
      void clear();
};
#endif
```

Member Variables:

- `pointer`
- `queueSize()`
- `front`
- `rear`
- `numItems`

Member Functions:

- `Constructor`
- `enqueue()`
- `dequeue()`
- `isEmpty()`
- `isFull()`
- `clear()`

STL Queues

STL Queues

- Another way to implement a queue is by using the standard library
- An STL queue leverages the pre-existing library to access the data structure
- Much easier to use

STL Queue Example

```
#include <iostream>       // std::cin, std::cout
#include <queue>          // std::queue
using namespace std;

int main ()
{
  std::queue<int> myqueue;
  int myint;

  std::cout << "Please enter some integers (enter 0 to 
end):\n";

  do {
    std::cin >> myint;
    myqueue.push (myint);
  } while (myint);

  std::cout << "myqueue contains: ";
  while (!myqueue.empty())
  {
    std::cout << ' ' << myqueue.front();
    myqueue.pop();
  }
  std::cout << '\n';

  return 0;
}
```

Iterators

Iterators

- An *iterator* in C++ is a concept that refines the iterator design pattern into a specific set of behaviors that work well with the C++ standard library.
- The standard library uses iterators to expose elements in a range, in a consistent, familiar way.

From: https://cise.ufl.edu/class/cop3530fa10/ITERATORS.ppt

Iterators

- Anything that implements this set of behaviors is called an iterator.
  - Allows Generic Algorithms
  - Easy to implement your own iterators and have them integrate smoothly with the standard library.

From: https://cise.ufl.edu/class/cop3530fa10/ITERATORS.ppt

Encapsulation

- *Encapsulation* is a form of information hiding and abstraction
- Data and functions that act on that data are located in the same place (inside a class)

- *Ideal*: separate the interface/implementation so that you can use the former without any knowledge of the latter

Iterator Pattern

- The iterator pattern describes a set of requirements that allows a consumer of some data structure to access elements in it with a familiar interface, regardless of the internal details of the data structure.
- The C++ standard library containers (data structures) supply iterator interfaces, which makes them convenient to use and interoperable with the standard algorithms.

From: https://cise.ufl.edu/class/cop3530fa10/ITERATORS.ppt

Iterators

- The iterator pattern defines a handful of simple requirements. An iterator should allow its consumers to:
  - Move to the beginning of the range of elements
  - Advance to the next element
  - Return the value referred to, often called the referent
  - Interrogate it to see if it is at the end of the range

From: https://cise.ufl.edu/class/cop3530fa10/ITERATORS.ppt

Using Iterators

- **`begin()`** returns a bidirectional iterator that represents the first element of the container.
- **`end()`** returns an iterator that represents the end of the elements (not the "last" element)
  - The end is a position <u>behind</u> the last element
  - Defining it this way gives us a simple ending criteria for our loops (as we'll see) and it avoids special handling for empty ranges of elements

Iterators in C++

- The C++ standard library provides iterators for the standard containers (for example, list, vector, deque, and so on) and a few other noncontainer classes. You can use an iterator to print the contents of, for example, a vector like this:

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

From: https://cise.ufl.edu/class/cop3530fa10/ITERATORS.ppt

C++ Iterators

- C++ iterators permit the same operations as the iterator pattern requires, but not literally.
- It's all there: move to the beginning, advance to the next element, get the referent, and test to see if you're at the end.
- In addition, different categories of iterators support additional operations, such as moving backward with the decrement operators (--it or it--), or advancing forward or backward by a specified number of elements.

From: https://cise.ufl.edu/class/cop3530fa10/ITERATORS.ppt

Iterator Types

- 5 main types of Iterators in C++
  - Read only
  - Write only
  - Forward Iterator
  - Reverse or Backwards Iterator
  - Random Access Iterator
- With exception of Read and Write, as we go down every iterator is a superset of the previous one in terms of functionality.
- Common e.g. -> Pointers are a type of random access iterators.

From: https://cise.ufl.edu/class/cop3530fa10/ITERATORS.ppt

Forward Iterators

- Essentially only need to traverse over elements
- However to make STL – compliant, or to be able to interface with STL Algorithms, an iterator over a data structure needs to implement the following functionality

From: https://cise.ufl.edu/class/cop3530fa10/ITERATORS.ppt

Forward Iterators

- Required Functionality (Forward Iterator)
  - Assignment
  - Tests for Equality
  - Forward advancement using the prefix and postfix forms of the ++ operator
  - dereferencing that returns an rvalue (value) or an lvalue (address)