Standard Template Library

Reminder

- Final exam
  - The date for the Final has been decided:
  - Saturday, November 16th
  - 8am – 10am
  - 01-2000

Announcement

- Exam 2
  - Has been moved to Monday October 28th

Project

- Congratulations!
  - You got past Submission 2!
  - Farmer problem: due October 30th
  - Any questions on the project

New plan

- Today: STL 1
- Monday: STL 2
- Tuesday: IOStreams 1

- Thursday: IOStreams 2
- Monday: Exam 2

A quick intro to Templates

```
template <class T>
class Queue
{
 private:
    T *q;
    int n;
    …
public:
    void enqueue (T i);
    T dequeue();
   …
}
```

Datatype to be filled in later

A quick intro to Templates

- To use this template:

```
// a queue of ints
Queue<int> iqueue;

// a queue of doubles
Queue<double> dqueue;

// a queue of aClass objects
Queue<aClass> aqueue

// a queue of pointers to aClass
Queue<aClass *> aptrqueue
```

The Standard Template Library

- A general-purpose C++ library of algorithms and data structures
- Based on a concept known as **generic programming**
- Implemented by means of the C++ **template** mechanism
- Part of the standard ANSI C++ library
- util package for C++

STL Components

- containers
  - classes that hold stuff
- iterators
  - Used to iterate through containers
  - Generalization of C++ pointers
- generic algorithms
  - Templated functions

STL Components

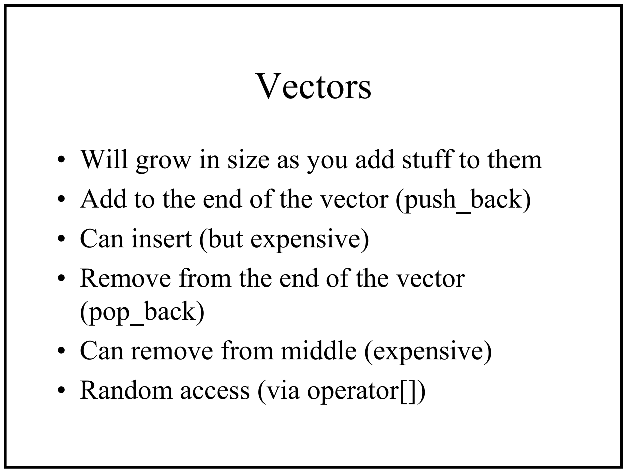
- function objects (Functors)
  - Objects that overload operator();
  - Substitute for pointers to functions
  - Beyond the scope of this course
- adaptors
  - adapt other components to special purposes.
  - Queues and stacks are adaptors
- Allocators
  - encapsulate a memory model.
  - decouple the algorithms from assumptions about a particular model.

Sample code using STL

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

  vector<int> v;
  for (int i = 0; i < 25; i++) v.push_back(i);

  random_shuffle(v.begin(), v.end());

  for (int j = 0; j < 25; j++) cout << v[j] << " ";

 ...
```

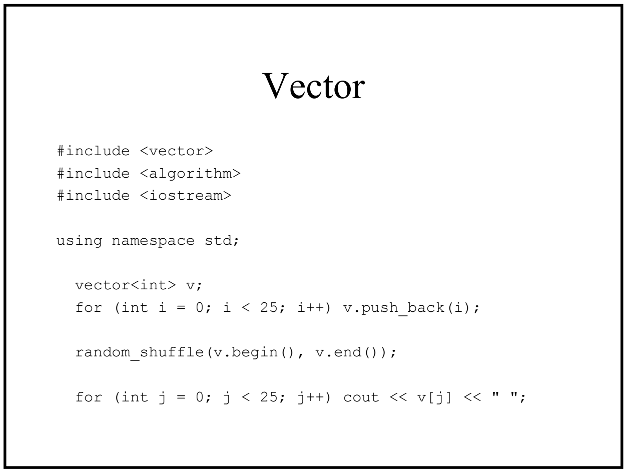
Simple Containers

- vector
  - Smart array
  - Grows dynamically
  - Random access (overrides [])
- list
  - Doubly-linked list
  - Sequential access
- deque
  - Double ended queue.
  - Best of both vector and list



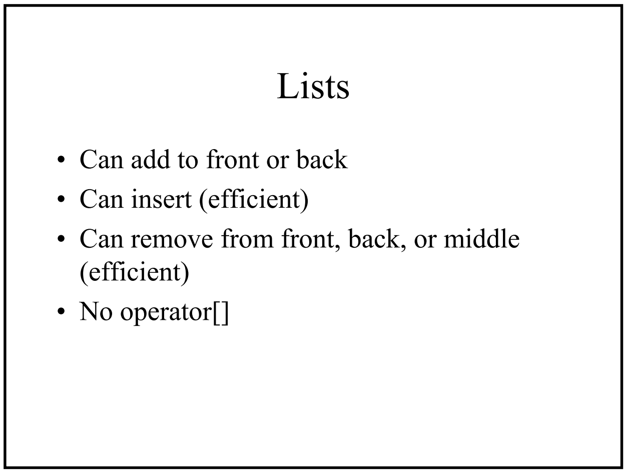
Vectors

- Will grow in size as you add stuff to them
- Add to the end of the vector (push_back)
- Can insert (but expensive)
- Remove from the end of the vector (pop_back)
- Can remove from middle (expensive)
- Random access (via operator[])



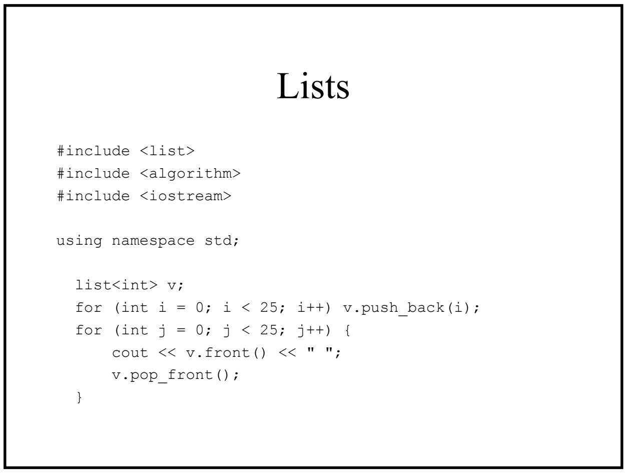
Vector

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

  vector<int> v;
  for (int i = 0; i < 25; i++) v.push_back(i);

  random_shuffle(v.begin(), v.end());

  for (int j = 0; j < 25; j++) cout << v[j] << " ";
```



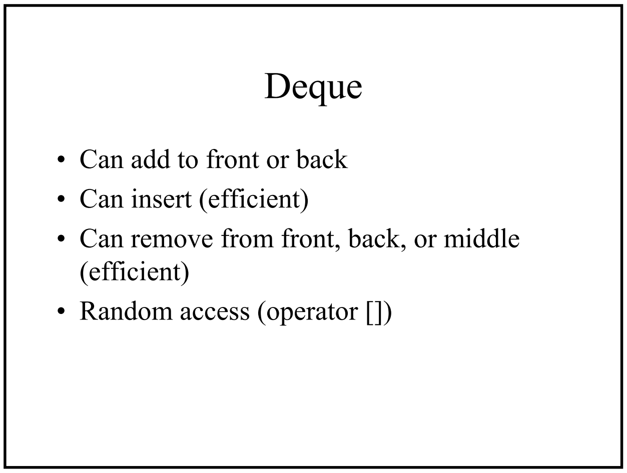
Lists

- Can add to front or back
- Can insert (efficient)
- Can remove from front, back, or middle (efficient)
- No operator[]



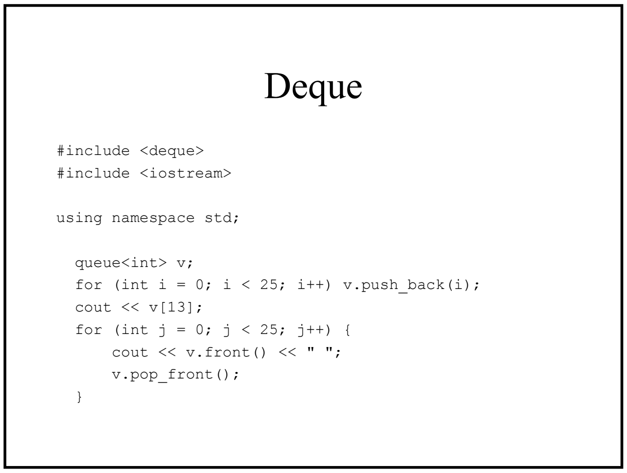
Lists

```
#include <list>
#include <algorithm>
#include <iostream>

using namespace std;

  list<int> v;
  for (int i = 0; i < 25; i++) v.push_back(i);
  for (int j = 0; j < 25; j++) {
      cout << v.front() << " ";
      v.pop_front();
  }
```



Deque

- Can add to front or back
- Can insert (efficient)
- Can remove from front, back, or middle (efficient)
- Random access (operator [])



Deque

```
#include <deque>
#include <iostream>

using namespace std;

  queue<int> v;
  for (int i = 0; i < 25; i++) v.push_back(i);
  cout << v[13];
  for (int j = 0; j < 25; j++) {
      cout << v.front() << " ";
      v.pop_front();
  }
```

Adaptor

- Wrapper class
- Converts the interface of one object to another
- Hides the interface of the original object

Adaptor

Queues and Stacks are Adaptors
-- Take in Container Templates
-- replaces it's own methods

```
template <class T, class Container =
  deque<T> >
class queue {...}
```

Queue

```
// Accessors

   bool empty () const;
   size_type size () const;
   value_type& front ();
   const value_type& front () const;
   value_type& back ();
   const value_type& back () const;
   void push (const value_type&);
   void pop ();
```

Stack

```
// Accessors
bool empty () const;
size_type size () const;
value_type& top ();
const value_type& top () const;
void push (const value_type&);
void pop ();
```

Questions?

Iterators

- Iterators are used to step through elements in STL containers

- Written to emulate C/C++ pointers
  - `operator++` to iterate forward
  - `operator--` to iterate backwards
  - `operator*` to dereference.

Iterator Types

- Some pointers are smarter than others
  - forward_iterators
  - reverse_iterators
  - bidirectional_iterators
  - const iterators

Iterator Types

- All container methods that return a position in the container will return it as iterators

- Each container has predefined types for the iterators it returns.

```
list<int> I;
list<int>::iterator it =
  I.begin();
```

Getting Iterators – List

```
// Iterators

        iterator begin ();
        const_iterator begin () const;
        iterator end ();
        const_iterator end () const;
        reverse_iterator rbegin ();
        const_reverse_iterator rbegin () const;
        reverse_iterator rend ();
        const_reverse_iterator rend () const;
```

Using Iterators

```
list<int> I;
list<int>::iterator it = begin();
while (it != I.end()) {
     cout << (*it);
     it++;
}
```

Random Access Iterators

- Allow for C-style pointer arithmetic’

```
list<int> I;
list<int>::iterator it = begin();
it+= 4;

// Prints out 5th element of I.
cout << (*it);
```

Summary

- Standard Template Library
- Simple Containers
- Iterators

- Next Time:
  - More complex containers
  - Algorithms
  - Reading the docs
- Have a good weekend!