Standard Template Library

Logistics

• Project
  – Part 1 (clock and design) due Sunday, Sept 25th
  – Part 2 (water) due Sunday, Oct 16th
  – Partners for Parts 2-3
• Questions?

Logistics

• Important date:
  – THURSDAY is Exam 1
  – Will cover:
    • C++ environment / architecture
    • C++ variables, pointers, references
    • Aggregates (Arrays, struct) static, const
    • Makefiles
  – Will not cover
    • Classes
    • Operator overloading
    • Constructors, Destructors, operator=
    • Templates

Exam 1

• Review session
  – Wednesday, Sept 28th
  – 9-10am / 8-9pm
  – 70-3435

Before we begin

• Any questions?

The Plan

• Today: STL 1
• Tomorrow: STL 2
• Thursday: Exam 1
A quick intro to Templates

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

Datatype to be filled in later

---

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
- 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

```cpp
#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

```cpp
#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

```cpp
#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

```cpp
#include <deque>
#include <algorithm>
#include <iostream>
using namespace std;

deque<int> d;
dequeue_for(25, d.push_back);
dequeue_for(25, d.pop_front);
for (int i = 0; i < 25; i++) d[i] = i;
dequeue_for(25, d[pop_front] = d.front());
```
Deque

```c++
#include <deque>
#include <iostream>

using namespace std;

deque<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();
}
```

Adaptor

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

Queue

```c++
// 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

```c++
// 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
  - Random access 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
  – Function Objects