THE STANDARD TEMPLATE LIBRARY

Standard Template Library

- The STL or standard template library is a collection of useful templates that are part of the C++ standard namespace
- In order to use each template in the STL, you need to include the appropriate header file
- For example, in order to use the vector template, you need to do:

#include <vector>
using namespace std;

• The STL supports a variety of *data structures* and numerical algorithms.

Today

- Today we delve deeper into the use of templates with a look at the *Standard Template Library*.
- In the same way that the standard library adds functionality to the basic C/C++ language , the template library provides templates.
- Both save you having to write lots of code from scratch.
- Good references are:
 - -http://www.learncpp.com/cpp-tutorial
 - -http://www.cplusplus.com/doc/tutorial
 - -http://www.cppreference.com/index.html
- You can also look at chapters 6 and 7 in the Pohl textbook.

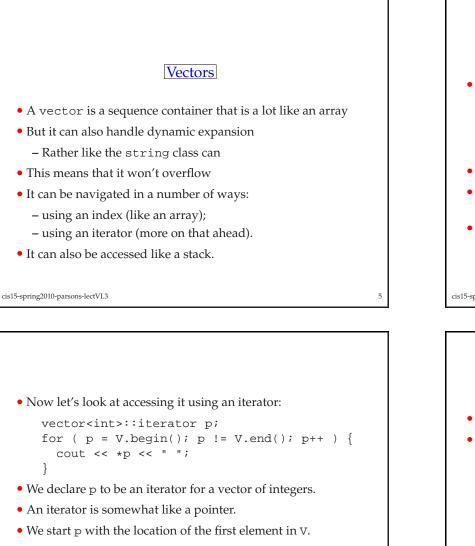
cis15-spring2010-parsons-lectVI.3

Containers

- Containers are classes that store groups of like elements.
- Kind of like fancy, more capable arrays
- There are two types of containers:
 - *sequence* containers which are: vector, list, deque
 - *associative* containers which are: set, multiset, map multimap and bitset
- We will look at both of these kinds of container.

cis15-spring2010-parsons-lectVI.3

cis15-spring2010-parsons-lectVI.3



- Then we print the element that p indicates, increase the value of p, and continue until we get to the last element of V.
- Iterators are particularly useful because vectors are dynamic.

```
• Here's a first example:
```

```
vector<int> V(10);
for ( int i=0; i<10; i++ ) {
    V[i] = i * 10;
}
```

- We declare V to be a vector with 10 elements.
- We then use an index, just like for an array, to set every element of the vector.
- (This and other examples can be found in vector.cpp.

```
cis15-spring2010-parsons-lectVI.3
```

- What do we mean by dynamic?
- Well, this:

```
V.push_back(10);
```

adds the element 10 to the end of the vector, and this:

```
V.pop_back();
```

removes the last element from the vector.

• Accessing using iterators and V.begin() and V.end() saves us from having to keep track of the length of the vector.

```
cis15-spring2010-parsons-lectVI.3
```

7

```
Also worth noting are:
V.size() which gives the size of the vector;
V.empty() which returns true if V doesn't have any elements;
V.front() which returns the first element in the vector.
V.end() which returns the last element in the vector.
So we can write:
while(!V.empty()){
    cout << "Size is " << V.size() << " ";
    cout << "First element is " << V.front() << " ";
    cout << "Last element is " << V.back() << endl;
    V.pop_back();
    }
</li>
I always think this lends itself to a recursive approach.
```

```
• There are many other functions for vector, but we will just look at some variations on the constructor which can be useful.
```

• Remember we started with:

vector<int> V(10);

which created a vector V with 10 unspecified elements.

• We also have:

```
vector<int> W(10, 20);
```

which creates a vector W and instantiates it with 10 copies of the integer 20.

and

```
vector<int> X(V.begin(), V.end());
```

which creates a vector X and instantiates it with the contents of V between V.begin() and V.end().

10

12

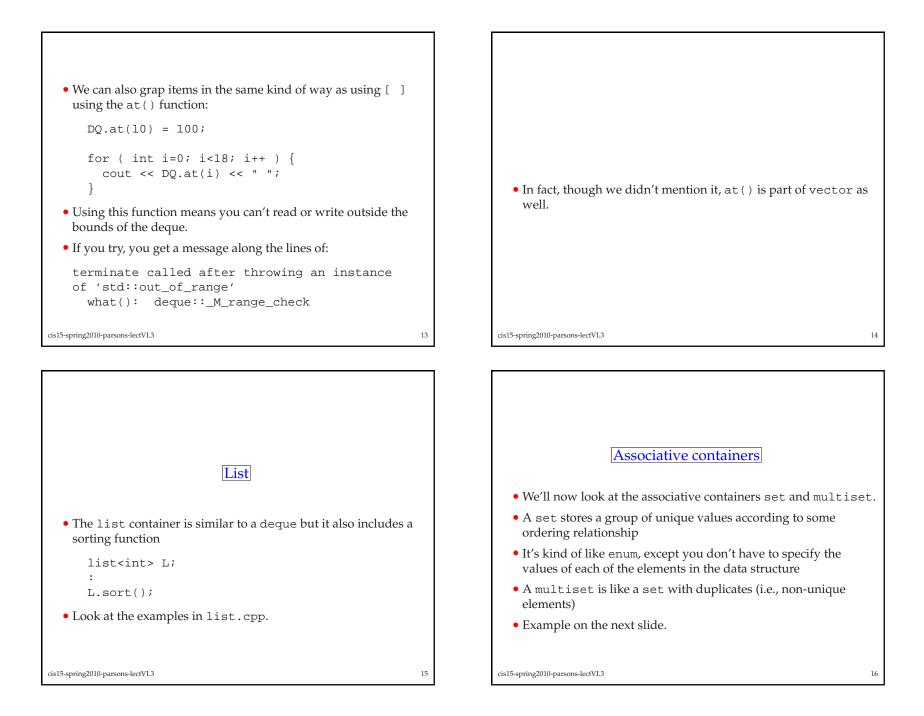
cis15-spring2010-parsons-lectVI.3

```
Here's code for adding elements to a deque:
for ( int i=0; i<10; i++ ) {
    DQ.push_front( i * 10 );
    }
for ( int i=0; i<10; i++ ) {
    DQ.push_back( i + 10 );
    }
And code for taking elements from a deque:
    DQ.pop_front();
    DQ.pop_back();
As with vector we have empty()
```

Deque

- A deque is another sequence container.
- You can think of it as an extension of a vector.
- With a vector you can only add items at the end.
- With a deque you can add items at either end,
- (There is a price to pay for that you can't use the index operator [] with a deque.)
- The following examples are all in deque.cpp.

```
cis15-spring2010-parsons-lectVI.3
```



```
#include <iostream>
#include <set>
using namespace std;
int main() {
   set<int> S;
   for ( int i=0; i<10; i++ ) {
     S.insert( i * 10 );
   }
   set<int>::iterator p;
   for ( p = S.begin(); p != S.end(); p++ ) {
     cout << *p << " ";
   }
   cout << endl;
}
cisl5-spring2010-parsons-lectVL3</pre>
```

```
      Map and multimap

      • Two more associative containers.

      • A map stores elements in "key-value" pairs

      • Instead of using numeric indexes, like arrays or vectors, to access elements, the "key" is used as a symbolic index

      • With a map, each key and value pair is unique

      • With a multimap, a single key may correspond to multiple values

      • Example on the next slide.

      test5-spring2010-parsons-lectVL3

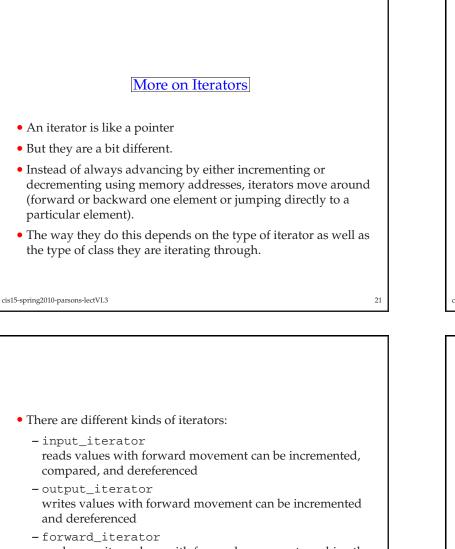
      • And the output is:

      (alex, 12)
      (jen, 15)

      (suz, 19)
```

```
#include <iostream>
    #include <map>
    using namespace std;
    struct strCmp {
                                                                                       • Note that elements are listed in alphabetical order based on the
     bool operator()( const char* s1, const char* s2 ) const {
        return( strcmp( s1, s2 ) < 0 );
                                                                                         key value.
      }
                                                                                       • This is because of the strCmp comparison operator that is part
    };
                                                                                         of the map definition.
    int main() {
     map<const char *, int, strCmp> M;
                                                                                       • If we reversed the operator, e.g., changed
     M["suz"] = 19;
                                                                                         return( strcmp( s1, s2 ) < 0 );
     M["alex"] = 12;
     M["jen"] = 15;
                                                                                         to
      map<const char *,int, strCmp>::iterator p;
      for ( p = M.begin(); p != M.end(); p++ ) {
                                                                                         return( strcmp( s2, s1 ) < 0 );
       cout << "(" << p->first << "," << p->second << ")\t";
                                                                                         then the output would be reversed:
      }
      cout << endl;
                                                                                                               (jen,15)
                                                                                                                                      (alex,12)
    }
                                                                                         (suz,19)
                                                                                                                                                           20
cis15-spring2010-parsons-lectVI.3
                                                                      19
                                                                                    cis15-spring2010-parsons-lectVI.3
```

17



reads or writes values with forward movement combine the functionality of input and output iterators with the ability to store the iterators value

• Compare:

```
int i;
for ( i=0; i<N; ++i ) {
    ...
}
with:
```

```
vector<int>::iterator p;
for ( p=v.begin(); p != v.end(); ++p ) {
```

cis15-spring2010-parsons-lectVI.3

. . .

-bidirectional_iterator

reads and writes values with forward and backward movement like forward iterators, but can also be incremented and decremented

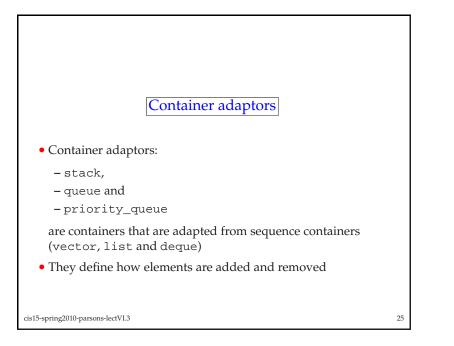
- random_iterator reads and writes values with random access
- reverse_iterator
 either a random iterator or a bidirectional iterator that moves
 in reverse direction
- All containers have a shared *interface* (i.e., the public functions); these are:
 - * Constructor and destructor
 - * Functions to access, insert and delete elements
 - * Iterators (which we will get to later)

cis15-spring2010-parsons-lectVI.3

23

24

22



Stack

- A stack is a "LIFO" data structure: "last in, first out"
- Which means that items are added to the front of the stack and also removed from the front of the stack.
- We have talked about stacks in the past this semester and used the analogy of a stack of plates in a cafeteria: new plates are added to the top; plates are also removed from the top
- The STL stack has the following members:

constructor
empty()
pop()
push()
size()
top()

cis15-spring2010-parsons-lectVI.3

Priority Queue

26

28

- Like a queue, except that the items are ordered according to a comparison operator that is specified when a priority queue object is instantiated
- So elements are inserted in order.
- Has the following members:

constructor
empty()
pop()
push()
size()
top()

cis15-spring2010-parsons-lectVI.3

27

cis15-spring2010-parsons-lectVI.3

Queue

- A queue is a "FIFO" data structure: "first in, first out"
- Which means that items are added to the back of the queue and are removed from the front of the queue
- A queue is just like a conventional line (of humans) (also called a "queue" if you live in the UK)
- Has the following members:

constructor back() empty() front() pop() push() size()

