CS2141 – Software Development using C/C++

The Standard Template Library
STL Content

- Data Structures – Template classes for common structures such as lists and stacks
- Iterators – A generalization of a pointer used to access containers without knowing anything about the internal structure of the container
- Function Objects – Objects that overload `operator()` so they can be used like a function
- Generic Algorithms – Functions that can work with many different data structures
STL Data Structures

• There are several groups of data structures:
  • Sequence structures
    • List or array type things
    • Includes vectors, lists, and deques
  • Sequence adapters
    • Built on top of sequence structures
    • Includes stacks and queues
  • Associative structures
    • Store key-value pairs
    • Includes maps and sets
Vectors

- A **vector** is a resizable array
  - Provides efficient random access
  - Insertions and deletions in the middle are slow

- Vector constructors:
  - `vector<T> v;`
  - `vector<T> v( int size );`
  - `vector<T> v( int size, T initial_value );`
    - Initializes elements to `initial_value`.
  - `vector<T> v( vector<T> oldvector );`
Vector Functions

- **Accessing elements:**
  - `v[index]` Return element at index.
  - `v.at(index)` Like [], but does range checking.
  - `v.front()` Return the first element.
  - `v.back()` Return the last element.

- **Size:**
  - `v.size()` Return number of elements.
  - `v.empty()` Return true if empty.
  - `v.resize(newsize)` Set number of elements.
Vector Functions cont.

- Insert and remove:
  - `v.push_back(value)` Append value
  - `v.pop_back()` Remove the last element
  - `v[pos] = value` Set element at pos to value
  - `v.insert(iterator, value)` Insert element at the position indicated by the iterator
  - `v.erase(iterator)` Remove the element at the position indicated by the iterator
  - `v.clear()` Remove all elements from the vector
Example

```cpp
#include <vector>    // Header for vectors
#include <iostream>
using namespace std;

int main( ) {
    vector<int> v;    // A vector that stores ints
    int i;

    while( !cin.eof( ) ) {  // Read any number of ints
        cin >> i;
        v.push_back( i );  // Store the ints in the vector
    }

    for( i = 0; i < v.size( ); ++i ) {    // Print out the
        cout << v[i] << " ";  // vector one element
        // at a time.
    }
    cout << endl;
}
```
Lists and Deques

- A **list** is a doubly linked list structure
  - Optimized for insertions and deletions
  - Does not allow random access (no [] operator)
- A **deque** is a double-ended vector
  - Operations at either end are efficient like **list**
  - Subscripting is efficient like a **vector**
Sequence Adapters

- The **stack** and **queue** both can take a sequence structure as a template parameter
  - Both use deques by default
  - A **vector** cannot be used with a **queue**
- They are adapters as they provide a specialized interface to a more general structure
  - Adapters do not provide iterators
  - Intended to only be used through their interfaces
- There is also a **priority_queue** type
Associative Structures

- A `map` stores key-value pairs
  - Map operations use a `pair` data type
    - `pair.first` returns the key
    - `pair.second` returns the value
- A `set` is like a `map`, but only stores keys
- Both are implemented as binary trees, so operations are very efficient
- A `multi_map` or `multi_set` allow keys to appear more than once in the structure
Iterators

• An iterator is a generic way of accessing a data structure without knowing anything about how the structure works

• Iterators are used a lot like pointers:
  • They can be dereferenced with the * operator
  • They can be incremented and decremented
  • They can be subscripted (sometimes)

• Pointers are iterators
Iterators cont.

- A pair of iterators refers to a range of locations:

  ```c
  int moo[30];
  
  int * begin = moo;
  int * end = moo + 30;
  ```

- The begin iterator refers to the first element in the data structure.

- The end iterator refers to after the last element.
Iterators cont.

- Iterators might access locations that are not necessarily contiguous in memory
- Iterators can be used without knowing anything about the underlying data structure
Iterators cont.

- `begin()` and `end()` are used to get iterators from an STL data structure.
  - `begin()` returns an iterator for the first element.
  - `end()` returns an iterator for after the last element.

```cpp
vector<int> v;
...
int sum = 0;
vector<int>::iterator start = v.begin();
vector<int>::iterator stop = v.end();
for ( ; start != stop; ++start )
    sum += *start;
```
Iterators cont.

- There are two major types of iterators:
  - Bi-directional
    - Can increment and decrement, but no random access
    - Returned by lists, sets, and maps
  - Random access:
    - Can do whatever - increment, decrement, subscript...
    - Returned by vectors, strings, and deques
- Some generic algorithms require random access iterators and can't be used with list, sets, or maps
Function Objects

- Any object overloading the parenthesis operator can be used as a function

```cpp
class Bigger {
    public:
    Bigger( int v = 0 ) : val( v ) { }
    bool operator()( int test )
    { return test > val; }

    private:
    int val;
};

Bigger byte( 8 );
if( byte( 3 ) )...
```
Generic Algorithms

- STL generic algorithms are based on templates, iterators, and function objects, so they can be used with a wide variety of data structures.

- One algorithm is find, which takes a start iterator, an end iterator, and a value to look for:

```cpp
int nums[100];
...
int * pos = find( nums, nums + 100, 45 );
if( pos != (nums + 100) )
    cout << "Found 45 in the array" << endl;
else
    cout << "Couldn't find 45" << endl;
```
Generic Algorithms cont.

- Generic algorithms are just template functions:

  ```cpp
template <class iterator, class T>
  iterator find( iterator first, iterator last, T & val )
  {
    while( first != last && *first != val )
      ++first;
    return first;
  }
```

- Some other algorithms:

  - count, copy, sort, count_if, replace, generate, equal, fill, random_shuffle, search, reverse, inner_product, for_each, includes, max