STL and more..

Prof. Seokin Hong

Iterators

STL and Iterator

- **STL has containers, algorithms and Iterators**
  - Containers hold objects, all of a specified type
  - Generic algorithms act on objects in containers
  - Iterators provide access to objects in the containers

- **Iterators**
  - Not a pointer but usually implemented using pointers
  - Treating iterators as pointers typically is OK
  - Each container defines an appropriate iterator type

Basic Iterator Operations

- **Basic operations shared by all iterator types**
  - ++ (pre- and postfix) : advance to the next data item
  - == and != : test whether two iterators point to the same data item
  - * (dereferencing operator): provides data item access
    - ***p** access may be read-only or read-write

  - c.begin( ) : returns an iterator pointing to the first element of container c
  - c.end( ) : returns an iterator pointing past the last element of container c

Basic Iterator Operations (Cont.)

```
//Program to demonstrate STL iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;
int main()
{
    vector<int> container;

    for (int i = 1; i <= 4; i++)
        container.push_back(i);

    cout << "Here is what is in the container:\n";
    vector<int>::iterator p;
    for (p = container.begin(); p != container.end(); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Setting entries to 0:\n";
    for (p = container.begin(); p != container.end(); p++)
         *p = 0;

    cout << "Container now contains:\n";
    for (p = container.begin(); p != container.end(); p++)
        cout << *p << " ";
    cout << endl;

    return 0;
}
```

Kinds of Iterators

- **Forward iterators**
  - provide basic operations

- **Bidirectional iterators**
  - provide basic operations and the -- operator to move to the previous data item
  -

- **Random access iterators**
  - provide basic operations and -- operator
  - Indexing p[2] returns the third element in the container
  - p + 2 : an iterator to the third element in the container

Kinds of Iterators (Cont.)

```
//Program to demonstrate bidirectional and random access iterators.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main()
{
    vector<char> container;

    container.push_back('A');
    container.push_back('B');
    container.push_back('C');
    container.push_back('D');

    for (int i = 0; i < 4; i++)
        cout << "container[" << i << "] == "
             << container[i] << endl;

    vector<char>::iterator p = container.begin();
    cout << "The third entry is " << container[2] << endl;
    cout << "The third entry is " << p[2] << endl;
    cout << "The third entry is " << *(p + 2) << endl;

    cout << "Back to container[0].\n";
    p = container.begin();
    cout << "which has value " << *p << endl;
    cout << "Two steps forward and one step back:\n";
    p++;
    cout << *p << endl;
    p++;
    cout << *p << endl;
    p--;
    cout << *p << endl;

    return 0;
}
```

Three different notations for the same thing.

This notation is specialized to vectors and arrays.

These two work for any random access iterator.

This is the decrement operator. It works for any bidirectional iterator.

Constant and Mutable Iterators

- **Categories of iterator divide into constant and mutable iterator.**
  - Constant Iterator does not allow assigning element at p

```
using std::vector<int>::const_iterator;
const_iterator cp = v.begin( );
*cp = something; // illegal
```

  - Mutable iterator p allows changing the element at p

```
using std::vector<int>::iterator;
iterator p = v.begin( );
*p = something; // OK
```

Using auto

- **The C++11 auto keyword can simplify variable declarations for iterators**
  - Example

```
vector<int>::iterator p = v.begin();
```

- **We simply use:**

```
auto p = v.begin();
```

Reverse Iterators

- Reverse iterators reverse the usual behavior of ++ and –
- rp-- moves the reverse iterator rp towards the beginning of the container
- rp++ moves the reverse iterator rp towards the end of the container

```
reverse_iterator rp;
for(rp = c.rbegin( ); rp != c.rend( ); rp++)
    process_item_at (rp);
```

```
//Program to demonstrate a reverse iterator.
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main()
{
    vector<char> container;

    container.push_back('A');
    container.push_back('B');
    container.push_back('C');

    cout << "Forward:\n";
    vector<char>::iterator p;
    for (p = container.begin(); p != container.end(); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Reverse:\n";
    vector<char>::reverse_iterator rp;
    for (rp = container.rbegin(); rp != container.rend(); rp++)
        cout << *rp << " ";
    cout << endl;

    return 0;
}
```

Containers

Containers

- **The STL provides three kinds containers:**
  - Sequential Containers
    - the ultimate position of the element depends on where it was inserted, not on its value.
  - Container Adapters
    - use the sequential containers for storage, but modify the user interface to stack, queue or other structure.
  - Associative Containers
    - maintain the data in sorted order to implement the container's purpose. The position depends on the value of the element.

Sequential Containers

- **Three sequential containers:**
  - stl::list, stl::vector and stl::deque

- **Sequential means,**
  - container has a first element, a second element and so on

- **stl::list is a doubly linked list**
- **stl::vector is essentially a dynamic array**
- **stl::deque is “double ended queue”**
  - Data can be added or removed at either end and the size can change while the program runs.

Sequential Containers (Cont.)

```
//Program to demonstrate the STL template class list.
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;

int main()
{
    list<int> list_object;

    for (int i = 1; i <= 3; i++)
        list_object.push_back(i);

    cout << "List contains:\n";
    list<int>::iterator iter;
    for (iter = list_object.begin(); iter != list_object.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    cout << "Setting all entries to 0:\n";
    for (iter = list_object.begin(); iter != list_object.end(); iter++)
        *iter = 0;

    cout << "List now contains:\n";
    for (iter = list_object.begin(); iter != list_object.end(); iter++)
        cout << *iter << " ";
    cout << endl;

    return 0;
}
```

Common Container Members

- **The STL sequential containers support these members:**
  - container( ); // creates empty container
  - ~container( ); // destroys container, erases all members
  - c.empty( ) // true if there are no entries in c
  - c.size( ) const; // number of entries in container c
  - c = v; //replace contents of c with contents of v

More Common Container Members

- c.swap(other_container); // swaps contents of c and other_container.
- c.push_back(item); // appends item to container c
- c.begin( ); // returns an iterator to the first element in container c
- c.end( ); // returns an iterator to a position beyond the end of the container c.
- c.rbegin( ); // returns an iterator to the last element in the container.
- c.rend( ); // returns an iterator to a position beyond the of the container.
- c.front( ); // returns the first element in the container (same as *c.begin( );)
- c.back( ); // returns the last element in the container same as *(--c.end( ));
- c.insert(iter, elem); //insert copy of element elem before iteIr
- c.erase(iter); //removes element iter points to, returns an iterator to element
  // following erasure. returns c.end( ) if last element is removed.

More Common Container Members

- c.clear( ); // makes container c empty
- c1 == c2 // returns true if the sizes equal and corresponding elements in c1 and c2 are equal
- c1 != c2 // returns !(c1==c2)
- c.push_front(elem) // insert element elem at the front of container c.
  // NOT implemented for vector due to large run-time that results

Operation Support

Operation	Function	vector	List	deque
Insert at front	push_front(e)	-	O	O
Insert at back	push_back(e)	O	O	O
Delete at front	pop_front( )	-	O	O
Delete at back	pop_back( )	O	O	O
Insert in middle	insert(e)	(O)	O	(O)
Delete in middle	erase(iter)	(O)	O	(O)
Sort	sort( )	O	-	O

(O) Indicates this operation is significantly slower.

Operation Support (Cont.)

Template Class Name	Iterator Type Names	Kind of Iterators	Library Header File
slist Warning: slist is not part of the STL.	slist<T>::iterator slist<T>::const_iterator	mutable forward constant forward	<slist> Depends on implementation and may not be available.
list	list<T>::iterator list<T>::const_iterator list<T>::reverse_iterator list<T>::const_reverse_iterator	mutable bidirectional constant bidirectional mutable bidirectional constant bidirectional	<list>
vector	vector<T>::iterator vector<T>::const_iterator vector<T>::reverse_iterator vector<T>::const_reverse_iterator	mutable random access constant random access mutable random access constant random access	<vector>
deque	deque<T>::iterator deque<T>::const_iterator deque<T>::reverse_iterator deque<T>::const_reverse_iterator	mutable random access constant random access mutable random access constant random access	<deque>

The Container Adapters : stack and queue

- Container Adapters use sequence containers for storage but supply a different user interface.
- stl::stack uses a Last-In-First-Out discipline.
- stl::queue uses a First-In-First-Out discipline.
- stl::deque is the default container for both stl::stack and stl::queue.

std::stack

- **Declarations:**
  - #include <stack>
  - std::stack<T> s; // uses deque as underlying store
  - std::stack<T, underlying_container> t ; //uses the specified container as underlying
    //container for stack
  - stack::stack<T> s (sequence_container); // initializes stack to elements in
    // sequence_container.

std::stack (cont.)

Sample Member Functions	
Member function	**Returns**
s.size( )	number of elements in stack
s.empty( )	true if no elements in stack else false
s.top( )	reference to top stack member
s.push(elem)	void Inserts copy of *elem* on stack top
s.pop( )	void function. Removes top of stack.
s1 = = s2	true if sizes same and corresponding pairs of elements are equal, else false

std::stack (cont.)

```
//Program to demonstrate use of the stack template class from the STL.
#include <iostream>
#include <stack>
using std::cin;
using std::cout;
using std::endl;
using std::stack;

int main()
{
    stack<char> s;

    cout << "Enter a line of text:\n";
    char next;
    cin.get(next);
    while (next != '\n')
    {
        s.push(next);
        cin.get(next);
    }

    cout << "Written backward that is:\n";
    while ( ! s.empty() )
    {
        cout << s.top();
        s.pop();
    }
    cout << endl;

    return 0;
}
```

The member function pop removes one element, but does not return that element. pop is a void function. So, we needed to use top to read the element we remove.

std::queue

- **Declarations:**
  - include <queue>
  - std::queue<T> q; // uses deque as underlying store
  - std::queue<T, underlying_container> q ; //uses the specified container as underlying
    // container for queue
  - std::queue<T> s (sequence_container); // initializes queue to elements in
    // sequence_container.

std::queue (cont.)

Sample Member Functions	
Member function	Returns
q.size( )	number of elements in queue
q.empty( )	true if no elements in queue else false
q.front( )	reference to front queue member
q.push(elem)	void adds a copy of *elem* at queue rear
q.pop( )	void function. Removes front of queue.
q1 == q2	true if sizes same and corresponding pairs of elements are equal, else false

Associative Containers: std::set and std::map

- Associative containers keep elements sorted on a some property of the element called the key.

- The order relation to be used may be specified:

```
std::set<T,  OrderRelation> s;
```

- The default order is the < relational operator for both std::set and std::map.

std::set

- **Declarations:**
  - #include <set>
  - set<T> s; // uses deque as underlying store
  - set<T, Ordering> s ; //uses the specified order relation to sort elements in the set
    // uses < if no order is specified.
- **Iterators:**
  - iterator, const_iterator, reverse_iterator, const_reverse_iterator

std::set (cont.)

function	Returns
s.size( )	number of elements in set
s.empty( )	true if no elements in set else false
s.insert(el)	Insert *elem* in set. No effect if *el* is a member
s.erase(itr)	Erase element to which *itr* refers
s.erase(el)	Erase element *el* from set. No effect if *el* is not a member
s.find(el)	Mutable iterator to location of *el* in set if present, else returns s.end( )
s1 == s2	true if sizes same and corresponding pairs of elements are equal, else false

std::set (cont.)

```
//Program to demonstrate use of the set template class.
#include <iostream>
#include <set>
using std::cout;
using std::endl;
using std::set;

int main()
{
    set<char> s;

    s.insert('A');
    s.insert('D');
    s.insert('D');
    s.insert('C');
    s.insert('C');
    s.insert('B');

    cout << "The set contains:\n";
    set<char>::const_iterator p;
    for (p = s.begin(); p != s.end(); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Removing C.\n";
    s.erase('C');
    for (p = s.begin(); p != s.end(); p++)
        cout << *p << " ";
    cout << endl;

    return 0;
}
```

No matter how many times you add an element to a set, the set contains only one copy of that element.

std::map

- **A map is a function given as a set of ordered pairs <first, second>**
- **First and second can be different data types**
  - Example: <string, int>
- **std::map is an associative array.**
  - Example,
    - numbermap["c++"] = 5

      // associates the integer 5 with the string "c++"

```
//Program to demonstrate use of the map template class.
#include <iostream>
#include <map>
#include <string>
using std::cout;
using std::endl;
using std::map;
using std::string;

int main()
{
    map<string, string> planets;

    planets["Mercury"] = "Hot planet";
    planets["Venus"] = "Atmosphere of sulfuric acid";
    planets["Earth"] = "Home";
    planets["Mars"] = "The Red Planet";
    planets["Jupiter"] = "Largest planet in our solar system";
    planets["Saturn"] = "Has rings";
    planets["Uranus"] = "Tilts on its side";
    planets["Neptune"] = "1500 mile-per-hour winds";
    planets["Pluto"] = "Dwarf planet";

    cout << "Entry for Mercury - " << planets["Mercury"]
         << endl << endl;

    if (planets.find("Mercury") != planets.end( ))
       cout << "Mercury is in the map." << endl;
    if (planets.find("Ceres") == planets.end( ))
       cout << "Ceres is not in the map." << endl << endl;

    cout << "Iterating through all planets: " << endl;
    map<string, string>::const_iterator iter;

     for (iter = planets.begin(); iter != planets.end(); iter++)
     {
        cout << iter->first  << " - " << iter->second << endl;
     }
     return 0;
}
```

std::map (cont.)

Function	Returns
m.size( )	number of pairs in the map
m.empty( )	true if no pairs are in the map else false
m.insert(el) *el* is a pair<key, T>	Inserts *el* into map. Returns <iterator, bool>. If successful, bool is true, iterator points to inserted pair. Otherwise bool is false
m.erase(key)	Erase element with key value *key* from map.
m.find(el)	Mutable iterator to location of *el* in map if present, else returns m.end( )
m1 = = m2	true if maps contain the same pairs, else false
m[target]	Returns a reference to the map value associated to a key of target.

Performance Comparison

Container	Insert Head	Insert Tail	**Insert**	Remove Head	Remove Tail	**Remove**	Index Search	Find
vector	n/a	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(log n)
list	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	n/a	O(n)
deque	O(1)	O(1)	n/a	O(1)	O(1)	O(n)	n/a	n/a
queue	n/a	O(1)	n/a	O(1)	n/a	n/a	O(1)	O(log n)
stack	O(1)	n/a	n/a	O(1)	n/a	n/a	n/a	n/a
map	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)	O(log n)
multimap	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)*	O(log n)
set	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)	O(log n)
multiset	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)*	O(log n)

https://stackoverflow.com/questions/730498/iterator-access-performance-for-stl-map-vs-vector/730524

Tip: Use ranged-for, auto with containers

- **C++11 ranged-for loop and auto keyword make it easier to iterate through containers.**

```
std::map<int, string> personIDs = { {1,"Walt"}, {2,"Kenrick"}};
std::set<string> colors = {"red","green","blue"};

for (auto p : personIDs)
      cout << p.first << " " << p.second << endl;
for (auto p : colors)
      cout << p << " ";
```

New C++ Features

C++ Is Evolving

- **The International Standards Organization ratifies proposed changes to the language**
  - C++11, C++14, C++17

- **Examples of some additions**
  - std::array
  - Threads
  - Regular Expressions
  - Smart Pointers

std::array

- **std::array allows you to use a vector-like notation for random access into a fixed-size sequence of elements**
- **Provides safe array access with the performance and minimal storage**
- **The following creates an array of 4 ints:**

```
std::array<int, 4> a = {1, 2, 3, 4};
```

- **Use a.size() to get the number of elements and use [] to access elements:**

```
// Output each element in the array
for (int i = 0; i < a.size(); i++)
    cout << a[i] << endl;
```

- **No harmful effects accessing outside the boundaries of the array**
  - a[100] = 10; // Ignored, no memory write

Regular Expressions

- **For our purposes, a regular expression provides a way to match patterns of text**
  - Formally, a regular expression describes a language from the class of regular languages
  - Some compilers still missing regex support

```
#include <regex>
using std::regex;
```

Regular Expressions

Regular Expression	Meaning
Letter or digit	The same letter or digit. For example, the regular expression `a` matches the text `a`, and the regular expression `abc123` matches the text `abc123`.
.	Matches any single character
|	Union or logical OR
R?	The regular expression `R` appears 0 or 1 time
R+	The regular expression `R` repeats consecutively 1 or more times
R*	The regular expression `R` repeats consecutively 0 or more times
R{n}	The regular expression `R` repeats consecutively n times
R{n,m}	The regular expression `R` repeats consecutively n to m times
^	Beginning of the text
$	End of the text
[list of elements]	Match any of the elements. For example, `[abcd]` would match `a`, `b`, `c`, or `d`.
[element1-elementN]	Match any of the elements in the range. For example, `[a-zA-Z]` would match any uppercase or lowercase letter.
()	Precedence and expression grouping

Regular Expression Examples

Description	Regular Expression
Three a's followed by three b's	`aaabbb` or `a{3}b{3}`
Any sequence of zero or more a's	`a*`
One or more a's followed by any sequence of b's	`a+b*`
The rules for an identifier, i.e., a letter or underscore followed by any sequence of letters, digits, or underscores	`[a-zA-Z_]+[a-zA-Z0-9_]*`

Example – Matching a Phone Number

```
string phonePattern = R"(\d{3}-\d{3}-\d{4})";
string twoWordPattern = R"(\w+\s\w+)";
regex regPhone(phonePattern);
regex regTwoWord(twoWordPattern);

string s;
cout << "Enter a string to test the phone pattern." << endl;
getline(cin, s);
if (regex_match(s, regPhone))
            cout << s << " matches " << phonePattern << endl;
else
            cout << s << " doesn't match " << phonePattern << endl;

cout << endl;
cout << "Enter a string to test the two word pattern." << endl;
getline(cin, s);
if (regex_match(s, regTwoWord))
            cout << s << " matches " << twoWordPattern << endl;
else
            cout << s << " doesn't match " << twoWordPattern << endl;
```

Threads

- **A thread is a separate computational process that runs concurrently**
  - Useful for performance reasons and to prevent your program from blocking while waiting for input

```
#include <thread>
using std::thread;
void func(int a)
{
    cout << "Hello World: " << a << endl;
}

int main()
{
      thread t1(func, 10);  // Runs func(10) in a thread
      thread t2(func, 20);  // Runs func(20) in a thread
      t1.join();  // Waits for thread 1 to finish
      t2.join();  // Waits for thread 2 to finish
}
```

Threads (Cont.)

- **Mutex**
  - You can use a mutex to give a thread exclusive access to run a block of code.
  - Other threads will wait for the mutex to be unlocked before entering

```
#include <mutex>
using std::mutex;
mutex globalLock;

void func(int a)
{
    globalLock.lock();
    cout << "Hello World: " << a << endl;
    globalLock.unlock();
}

int main()
{
    thread t1(func, 10);
    thread t2(func, 20);
    t1.join();
    t2.join();
}
```

Smart Pointers

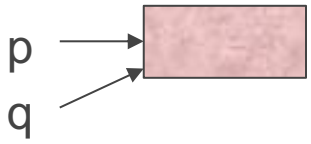
- **A template class that automatically frees up memory allocated to dynamic variables when they go out of scope**

- **Uses a technique called reference counting that counts how many pointers reference an allocated node**

```
p = new Node();
```

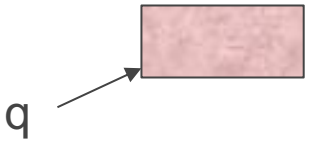

```
q = p;
```

Reference Counting



Ref count= 2

p = nullptr;



Ref count= 1

q = nullptr;

Ref count= 0
Memory deallocated

Smart Pointers

- **Old code without smart pointers**

```
Node *p = new Node();
p->callFunction();
delete p;    // delete when done with the pointer
```

- **Converted to smart pointers**

```
#include <memory>
  using std::shared_ptr;
  shared_ptr<Node> p(new Node());  // Template class
  p->callFunction();               // Use like a regular pointer
  //delete p;          → No longer needed, will be deleted automatically
                       // when reference count reaches 0
```

NEXT ?

Term Project
Final Presentation