Unit 9 (ch 18)

Standard Template Library & Intro. to C++11

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electronics & Electrical Engr.
Nat'l Yang Ming Chiao Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nycu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

- *9.1 Intro. to STL*
- 9.2 Iterators
- 9.3 Containers
- 9.4 Generic Algorithms
- 9.5 Intro. to C++11

Introduction to STL

- STL: Standard Template Library
  - Not part of the C++ core language
  - But is included in the C++ standard
    - Available in most C++ compilers
- It mainly consists of
  - Containers: hold objects, all of a specified type
    - Sequence
    - Associative
  - Iterators: a generic pointer to access objects in containers
  - Generic algorithms: act on objects in containers
- Save you a lot of efforts to use those common data structures / algorithms

A Common Example: Vectors

- Vectors look like arrays, but they can change size as your program runs
- To declare an empty vector with base type int:

  vector<int> v;
  - <int> identifies vector as a template class
  - You can use any base type in a template class:

    Ex: vector<string> v;
- vector class is included in the <vector> library
  - Vector names are placed in the standard namespace

    using namespace std;

Vector as a "Better" Array

- C-style pointer-based arrays have great potential for errors and are not flexible
  - A program can easily "walk off" either end of an array, because C++ does not check the subscripts range
  - Two arrays cannot be meaningfully compared with equality operators or relational operators
  - The size of the array must be passed as an additional argument when an array is passed to another function
  - One array cannot be assigned to another with the assignment operator(s)
- C++ Standard class template **vector** represents a more robust type of array with additional capabilities

Accessing vector Elements

- Vectors elements are indexed starting with 0
  - [ ]'s are used to read or change the value of an item:

```
v[i] = 42;
cout << v[i];
```

  - [ ]'s cannot be used to initialize a vector element
- Elements are added to a vector using the member function *push_back( )*
  - Adds an element in the next available position
  - Example:

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```

- When a vector runs out of space, its capacity is automatically increased

Constructor for vectors

- To initialize a specified number of elements in a vector, use its constructor function
  - Example: `vector<int> v(10);`
    → initializes 10 elements and set them to 0
    - [ ]'s can now be used to assign elements 0 through 9
    - *Push_back* will assign elements to the location after 9
- More constructor usage are listed as follows:
  - `vector<double> d;` // empty vector of doubles
  - `vector<int> v2(6,100);` // six ints with value 100
  - `vector<int> v3(v2);` // a copy of v2
  - `v2 = v;` // copy assignment operator

Example: Demonstrate vector Usage

```
//DISPLAY 8.9 Using a Vector
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> next;
    }

    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

Sample Dialogue

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size( ) = 1
4 added. v.size( ) = 2
6 added. v.size( ) = 3
8 added. v.size( ) = 4
You entered:
2 4 6 8
```

Size vs. Capacity in vector

- Size is the number of elements that actually used
  - Example: for (int i= 0; i < sample.size( ); i++)
    cout << sample[i] << endl;
- A vector's capacity is the number of elements allocated in memory
  - Accessible using the *capacity( )* member function
- When a vector runs out of space, the capacity is automatically increased
  - Member function *reserve( )* increases the capacity manually
  - Example: v.reserve(32); // at least 32 elements
    v.reserve(v.size( ) + 10); // at least 10 more
- Function *resize()* can be used to shrink a vector
  - Example: v.resize(24); // elements beyond 24 are lost

Ex: Compare Size & Capacity

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> myvector;
  for (int i=0; i < 100; i++)
    myvector.push_back(i);
  cout << "size: " << myvector.size() << '\n';
  cout << "capacity: " << myvector.capacity() << '\n';
  return 0;
}
```

The two numbers are not the same. (capacity ≥ size)

size → 100
capacity → 128

More Member Functions for vector

- front/back: access the first/last element
- pop_back: erase the last element
- insert/erase: insert/erase elements
- clear: clear content (erase all elements)
- empty: test whether vector is empty
- swap: swap content between two vectors
- several iterator-related ones … (discussed later)

Operations with vector Objects

- *vector* objects can be compared with another one using the equality (==) operators
- You can use the assignment (=) operator on vectors
  - Element by element copy of the right hand vector
- You can create a new vector object that is initialized with a copy of an existing vector
- As with C-style arrays, C++ does not perform bound checking while using [ ] to access vector elements
- Similar to string, vector provides member function *at* for bound checking. Ex: v1[2] → v1.at(2)
  - Throws an exception for invalid subscript

Relational Operators

- Vector also has non-member operator overloading

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> foo(3,100);
  vector<int> bar(2,200);
  if (foo == bar)  cout << "foo and bar are equal\n";
  if (foo != bar)  cout << "foo and bar are not equal\n";
  if (foo < bar)  cout << "foo is less than bar\n";
  if (foo > bar)  cout << "foo is greater than bar\n";
  if (foo <= bar)  cout << "foo is less than or equal to bar\n";
  if (foo >= bar)  cout << "foo is greater than or equal to bar\n";
  return 0;
}
```

lexicographical comparisons

Overview

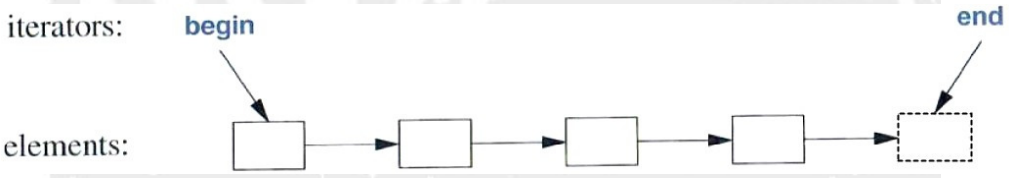
- 9.1 Intro. to STL
- *9.2 Iterators*
- 9.3 Containers
- 9.4 Generic Algorithms
- 9.5 Intro. to C++11

Iterators

- STL has containers, algorithms and iterators
  - Iterators provide access to objects in the containers yet hide the internal structure of the container
- An iterator is a generalization of pointer
  - Not a pointer but usually implemented using pointers
    - Treating iterators as pointers typically is OK
  - Each container defines an appropriate iterator type
    - The pointer operations may be overloaded for behavior appropriate for the container internals
- Why? This is one of the STL philosophy
  - The semantics, meaning, and syntax for iterator usage are uniform across all container types

Basic Iterator Operations

- Basic operations shared by all iterator types
  - ++ (pre- and postfix): advance to the next data item
  - == and != operators: test whether two iterators point to the same data item
  - * dereferencing operator: provides data item access
  - 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.
    - You can apply -- to the iterator returned by c.end( ) to get an iterator pointing to last element in the container



Demo Basic Iterator Operations

```
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main( )
{
    vector<int> v1;

    for (int i = 1; i <= 4; i++)
        v1.push_back(i);

    cout << "Here is what is in the container:\n";
    vector<int>::iterator p;
    for (p = v1.begin( ); p != v1.end( ); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Setting entries to 0:\n";
    for (p = v1.begin( ); p != v1.end( ); p++)
        *p = 0;

    cout << "Container now contains:\n";
    for (p = v1.begin( ); p != v1.end( ); p++)
        cout << *p << " ";
    cout << endl;

    return 0;
}
```

Sample Dialogue

```
Here is what is in the container:
1 2 3 4
Setting entries to 0:
Container now contains:
0 0 0 0
```

More about Iterator Operations

- **--** (pre- and postfix): moves to previous data item
- ***p** access may be read-only or read-write depending on the definition of the iterator p
- Each STL container may define different iterator types appropriate to the container internals
  - Forward iterators: provide the basic operations
  - Bidirectional iterators: provide the basic operations and the -- operators to move to the previous data item
  - Random access iterators provide
    - The basic operations and iterator arithmetic
      Ex: p+2 returns an iterator to the 3_{rd} element in the container
    - Indexing p[2] returns the third element in the container
  - And many more different iterators (discussed later)

Constant and Mutable Iterators

- Iterators can be classified into constant iterator and mutable iterator
- Constant Iterator *cp* does not allow assigning element at cp, ex:

```
using std::vector<int>::const_iterator;
const_iterator cp = v.begin( );  // OK. Just change address
*cp = something;   // Illegal. Cannot change data
```

- Mutable iterator *p* does allow changing the element at p, ex:

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

Bidirectional & Random Access Iterators

```
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main( )
{
   vector<char> v2;
   v2.push_back('A');
   v2.push_back('B');
   v2.push_back('C');
   v2.push_back('D');

   for (int i = 0; i < 4; i++)
      cout << "container[" << i << "] == "
           << v2[i] << endl;

   vector<char>::iterator p = v2.begin( );

   cout << "The third entry is " << v2[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 = v2.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;
}
```

Sample Dialogue

```
container[0] == A
container[1] == B
container[2] == C
container[3] == D
The third entry is C
The third entry is C
The third entry is C
Back to container[0].
which has value A
Two steps forward and one step back:
B
C
B
```

Using auto in C++11

- The C++11 auto keyword can simplify variable declarations for iterators, ex:

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

- Examples to use auto in C++11:

```
for (auto p = v.begin(); p != v.end(); ++p)  // C++11
    cout << *p << ' ';
cout << endl;

for (auto x : v)   // C++11, range-based for loop
    cout << x << ' ';
cout << endl;
```

Reverse Iterators

- A reverse iterator allows you to traverse a container from the end to the beginning.
- However, 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
- Ex: assume object c is a container with bidirectional iterators

```
reverse_iterator rp;
for (rp = c.rbegin( ); rp != c.rend( ); rp++)
    process_item_at (rp);   // from end to beginning
```

Demo Reverse Iterator

```
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;

int main( )
{
    vector<char> v3;

    v3.push_back('A');
    v3.push_back('B');
    v3.push_back('C');

    cout << "Forward:\n";
    vector<char>::iterator p;
    for (p = v3.begin( ); p != v3.end( ); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Reverse:\n";
    vector<char>::reverse_iterator rp;
    for (rp = v3.rbegin( ); rp != v3.rend( ); rp++)
        cout << *rp << " ";
    cout << endl;

    return 0;
}
```

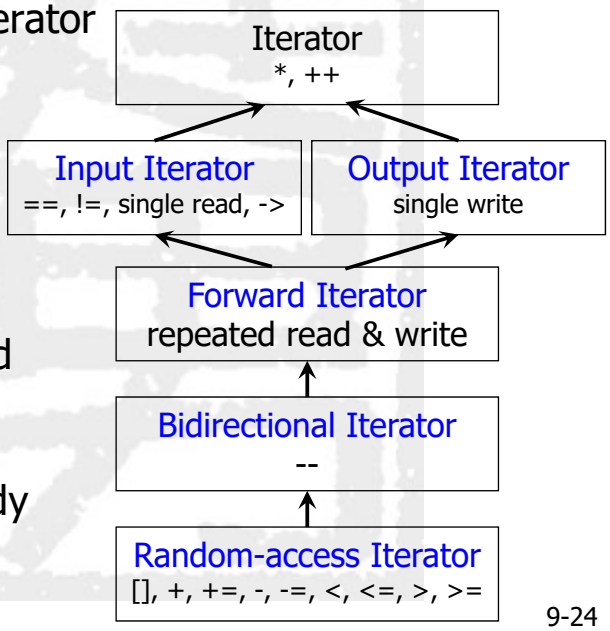
same code with different meaning

Sample Dialogue

```
Forward:
A B C
Reverse:
C B A
```


Other Kinds of Iterators

- There 5 main types of iterators in C++ STL
  - Different containers may use different iterators
  - vector uses random-access iterator
- Input/output iterators are the weakest and simplest
  - An input iterator is a forward iterator that can be used with input streams
  - An output iterator is a forward iterator that can be used with output streams
  - They are left for advance study



Overview

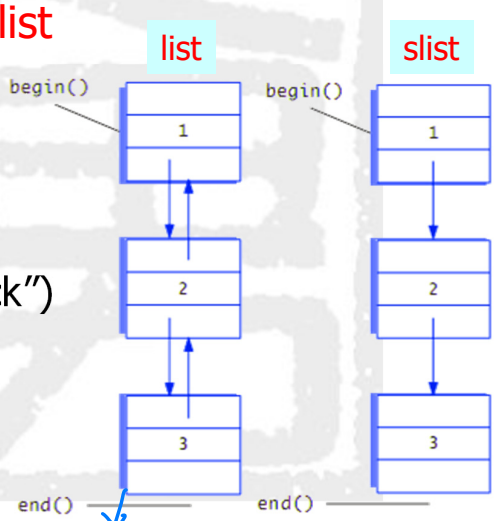
- 9.1 Intro. to STL
- 9.2 Iterators
- *9.3 Containers*
- 9.4 Generic Algorithms
- 9.5 Intro. to C++11

STL Containers

- The STL provides three kinds containers:
  - Sequential Containers: the position of the element depends on where it was inserted, not on its value
    - Ex: vector, list, deque
  - Container Adapters: use the sequential containers for storage, but modify the user interface
    - Ex: stack (LIFO), queue (FIFO), priority queue
  - Associative Containers: store data in sorted order, i.e. the position depends on the value of the element
    - Ex: set, multiset, map, multimap
    - C++11 also supports unordered associative containers

Sequential Containers

- Sequential means the container has a first element, a second element and so on
- An STL *vector* is essentially an array whose allocated space can grow while the program runs
- An STL *list* is a doubly linked list
  - Both ++ and -- are defined
- The single linked list, *slist*, is not in the STL
  - ++ defined, -- not defined
- An STL *deque* ("d-que" or "deck") is a "double ended queue"
  - Data can be added or removed at either end and the size can change while the program runs



沒辦法指到特定位置，但適合插入儲存資料

Demo the Basic **list** Operations

```
#include <iostream>
#include <list>
using std::cout;
using std::endl;
using std::list;

int main( )
{
   list<int> listObject;

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

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

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

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

   return 0;
}
```

Sample Dialogue

```
List contains:
1 2 3
Setting all entries to 0:
List now contains:
0 0 0
```

Common Container Members (1/3)

- The following members are supported by almost all STL sequential containers
  - 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
  - c1 == c2; // returns true if the sizes equal and // corresponding elements in them are equal
  - c1 != c2; // returns !(c1==c2)

Common Container Members (2/3)

- c.begin( ); // returns an iterator to the first element in 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 c // serves to as start of reverse traversal
- c.rend( ); // returns an iterator to a position which // indicates the end of reverse traversal in c
- c.front( ); // returns the first element in the container c // same as *c.begin( );
- c.back( ); // returns the last element in the container c // same as *(--c.end( ));

Common Container Members (3/3)

- c.swap(other_container); // swaps contents of c and
  // other_container.
- c.push_back(item); // appends item to container c
- c.insert(iter, elem); //insert element *elem* before iteIr
- c.erase(iter); //removes element that *iter* points to,
  // and returns an iterator to next element
  // returns c.end( ) if last element is removed
- c.clear( ); // makes container c empty
- c.push_front(elem); // insert element *elem* at the front of
  // container c. NOT implemented for
  // *vector* due to large run-time overhead

Iterators in Sequential Containers

DISPLAY 18.6 STL Basic Sequential Containers

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>

Supported Operations

Operation	Function	vector	list	deque
Insert at front	push_front(e)	–	√	√
Insert at back	push_back(e)	√	√	√
Delete at front	pop_front( )	–	√	√
Delete at back	pop_back( )	√	√	√
Insert in middle	insert(e)	(√)	√	(√)
Delete in middle	erase(iter)	(√)	√	(√)
Sort	Sort( )	√	–	√

(√) Indicates this operation is significantly slower.

Specific Functions for **list** -- remove

- list::remove(val) removes all items that equal to *val*
  - list::erase(iter) erases the item by location
- Ex:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    int a[] = {26,87,9,14};
    list<int> L(a,a+4);      // insert all items in array a into the list
    list<int>::iterator iter;
    L.remove(87);            // remove the items that equal to 87
    for (iter = L.begin(); iter != L.end(); ++iter)
       cout << " " << *iter;
    cout << endl;
    return 0;
}
```

Output:

```
26 9 14
```

Specific Functions for **list** -- reverse

- list::reverse() reverses the order of elements in a list
- Ex:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> L;
    list<int>::iterator iter;
    for (int i = 1; i < 10; i++)  L.push_back(i);
    for (iter = L.begin(); iter != L.end(); ++iter)
        cout << " " << *iter;
    cout << endl;
    L.reverse();     // reverse the order in the list
    for (iter = L.begin(); iter != L.end(); ++iter)
        cout << " " << *iter;
    cout << endl;
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1
```


Specific Functions for **list** -- merge

- list::merge(L1) transfers all elements of L1 into the list
  - If both lists are sorted, merge() will keep them ordered
- Ex:

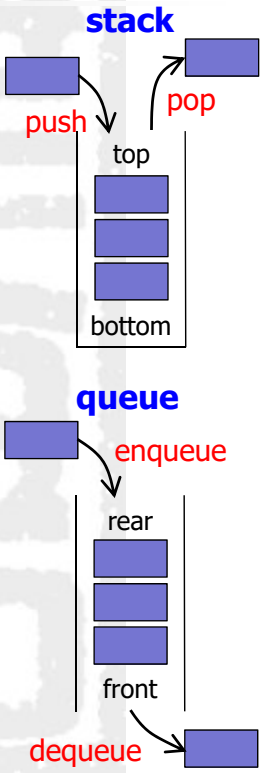
```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<double> L1, L2;
    list<double>::iterator iter;
    L1.push_back(3.2); L1.push_back(1.8); L1.push_back(2.3);
    L2.push_back(3.5); L2.push_back(6.3); L2.push_back(1.4);
    L1.merge(L2);     // reverse the order in the list
    // L2 becomes empty after merge
    for (iter = L1.begin(); iter != L1.end(); ++iter)
        cout << " " << *iter;
    cout << endl;
    return 0;
}
```

Output:

```
3.2 1.8 2.3 3.5 6.3 1.4
```

The Container Adapters

- Sequence container + different interface
- A **stack** uses a Last-In-First-Out (LIFO) discipline
  - Default container is **deque**
- A **queue** uses a First-In-First-Out (FIFO) discipline
  - Default container is **deque**
- A **priority queue** keeps its items sorted on the priority of the items
  - The highest priority item is removed first
  - Default container is **vector** to support removing items at the desired location



Stack vs Queue

- Declarations:
  - stack<T> s; // uses deque
  - stack<T, container> t;
    // use the specified container
  - stack<T> s (container);
    // initialize stack to
    // elements in container
- Header:

  #include <stack>
- Defined types:

  value_type, size_type
- No iterators are defined

- Declarations:
  - queue<T> q; // uses deque
  - queue<T, container> q;
    // use the specified container
  - queue<T> q (container);
    // initialize queue to
    // elements in container
- Header:

  #include <queue>
- Defined types:

  value_type, size_type
- No iterators are defined

stack Member Functions

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

queue Member Functions

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 corresonding pairs of elements are equal, else false

Demo the Use of **stack**

```
#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;
}
```

Sample Dialogue

```
Enter a line of text:
straw
Written backward that is:
warts
```

Demo the Use of **queue**

```
#include <iostream>
#include <queue>
using std::cin;
using std::cout;
using std::endl;
using std::queue;

int main( )
{
   queue<char> q;

   cout << "Enter a line of text:\n";
   char next;
   cin.get(next);
   while (next != '\n')
   {
      q.push(next);
      cin.get(next);
   }
   cout << "Written output is:\n";
   while ( ! q.empty( ) )
   {
      cout << q.front( );
      q.pop( );
   }
   cout << endl;

   return 0;
}
```

Sample Dialogue

```
Enter a line of text:
straw
Written output is:
straw
```

Associative Containers

- Associative containers keep elements sorted on a property of the element called the ***key***
- **Set**: store elements following a specific order of key without repetition
  - Addition insertions after the first element have no effect
- **Map**: store elements formed by a combination of a key value and a mapped value
  - Keys are unique in a map, too
  - One-to-one mapping between key and its mapped value (lookup table?)
- The order relation to be used may be specified:
  - Ex: set<T, OrderRelation> s;
  - The default order is the < relational operator

Set vs Map

- Declarations:
  - set<key> s; // uses deque
  - stack<key, ordering> s;
    // use specified order relation
- Header:
  #include <set>
- Defined types:
  value_type, size_type
- Iterators:
  - iterator, const_iterator, reverse_iterator, const_reverse_iterator
- Similar storage capability, but data are sorted in set

- Declarations:
  - map<key, T> m; // uses deque
  - map<key, T, ordering> m;
    // use specified order relation
- Header:
  #include <map>
- Defined types:
  value_type, size_type
- Iterators:
  - iterator, const_iterator, reverse_iterator, const_reverse_iterator
- Each element has an extra associative value

set Member Functions

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

Demo the Use of **set**

```
#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');   // duplicate items skipped
   s.insert('C');
   s.insert('C');   // duplicate items skipped
   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;
}
```

sorted already

Sample Dialogue

```
The set contains:
A B C D
Removing C.
A B D
```

map Member Functions

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 object associated to a key of target.

Maps as Associative Arrays

- An alternative interpretation is that a map is an associative array
  - For example, numbermap["c++"] = 5 associates the integer 5 with the string "c++"
    - "c++" → 5 (string to int)
- The easiest way to add and retrieve data from a map is to use the [] operator
  - If key is not already in the map, map[key] will create a new entry
  - Unlike array that allows number index only, the key in a map can be any legal type !!

key	map[key]
'a'	"Andy Liu"
'b'	"Bruce Lee"
'c'	"Caroline Yu"
'd'	"Diana Chen"
...	.....

Demo the Use of **map** (1/2)

```
#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;
```

Demo the Use of **map** (2/2)

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

```
Entry for Mercury - Hot planet

Mercury is in the map.
Ceres is not in the map.

Iterating through all planets:
Earth - Home
Jupiter - Largest planet in our solar system
Mars - The Red Planet
Mercury - Hot planet
Neptune - 1500 mile-per-hour winds
Pluto - Dwarf planet
Saturn - Has rings
Uranus - Tilts on its side
Venus - Atmosphere of sulfuric acid
```

Look-up Table !!

Specific Functions for **map** -- count

- map::count(key) checks if the *key* exists in the map
  - Return 1 if this key exists; otherwise, return 0
- Ex:

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<char,int> m;  // a map from char to int
    char c;
    m['a']=11;            // associate 'a' to 11
    m['c']=22;            // associate 'c' to 22
    m['f']=33;            // associate 'f' to 33
    for (c = 'a'; c <= 'h'; c++) {
        cout << c;
        if (m.count(c) > 0)  cout << " yes.\n";
        else cout << " no.\n";
    }
    return 0;
}
```

Output:

```
a yes.
b no.
c yes.
d no.
e no.
f yes.
g no.
h no.
```


Use Ranged-for, auto with Containers

- Ranged-for loop and auto keyword make it easier to iterate through containers (C++11 only)
- Consider the following map and set:

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

- We can iterate through each container conveniently with a ranged for loop and auto:

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

Overview

- 9.1 Intro. to STL
- 9.2 Iterators
- 9.3 Containers
- *9.4 Generic Algorithms*
- 9.5 Intro. to C++11

Generic Algorithms

- "Generic Algorithm" are a template functions that use iterators as template parameters
- There are about 100+ algorithms available in STL
  - Defined in <algorithm>
- Algorithms are roughly classified into
  - Nonmodifying Sequence Algorithms (count, find, …)
  - Modifying Sequence Algorithms (swap, reverse, …)
  - Sorting and Related Algorithms (sort, binary_search, …)
  - Numeric Algorithms (min_element, max_element, …)
- Only few of them are discussed here
  - See Josuttis for more information on algorithms …

Running Times and Big-O Notation

- How to evaluate the efficiency of an algorithm?
  - Time a program with a stop watch?
  - Calls to the system clock to determine running time?
  - The results vary with the problem size
- Be useful, you must specify time as a function of the problem size → time complexity
- We often use "worst case running time" to evaluate the time complexity → Big-O notation
  - Count only "steps" or "operations"
  - Mostly we agree to count =, <, &&, !, [ ], ==, ++, …
- This simplified assumption works well in practice

An Example to Evaluate Run Time

- Assume target is not in array
  - Worst case !!
- In loop, 6 operations per iteration
  <, &&, ! , [i] , ==, ++
- After N iterations, exit condition requires 3 more operations
  <, &&, and **!**
- Yes, there are some objections
  - Not all operations take the same time
  - Ignore some operations that might be significant
- What we really want in is the growth rate of the running time as a function of the problem size
  - Not precise computations for a particular architecture

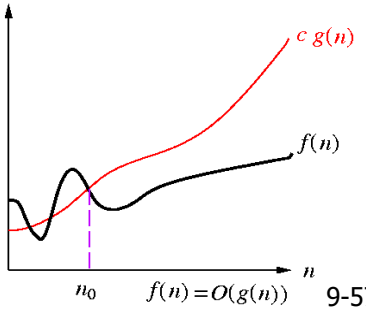
Total: 6N + 3 operations

```
int i = 0;
bool found = false;
while (( i < N ) && !(found))
    if (a[i] == target)
        found = true;
    else
        i++;
```

Big-O Notation

- Run time estimation is usually expressed with the Big-O notation
  - The O is the letter Oh, not the digit zero, 0
- Our loop in a previous slide runs in O(6N + 3)
- However, Big-O estimates do not distinguish 6N + 3 and 100N by its definition
- Def: *f(n)= O(g(n))* if ∃ c >0 and n_0 > 0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n_0

  → $2n^2 + 3n$ = O(n²), $3n\sqrt{n}$ = O(n²)
- Intuition: ignore constant multiples and small values of n



What's the Meaning of Big-O?

- Assume T(N) is the complexity function, you will see a big difference when N goes large
- Linear running time
  - T(N) = aN + b
- Quadratic running time
  - T(N) has highest term N^2
- O(log N): running time is fast even in large case
- Assume 1000 MIPS (Yr: 200x), 1 instruction per op

Time	Big-Oh	$n = 10$	$n = 100$	$n = 10^3$	$n = 10^6$
500	$O(1)$	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec
$3n$	$O(n)$	3×10^{-8} sec	3×10^{-7} sec	3×10^{-6} sec	0.003 sec
$n \log n$	$O(n \log n)$	3×10^{-8} sec	2×10^{-7} sec	3×10^{-6} sec	0.006 sec
n^2	$O(n^2)$	1×10^{-7} sec	1×10^{-5} sec	0.001 sec	16.7 min
n^3	$O(n^3)$	1×10^{-6} sec	0.001 sec	1 sec	3×10^5 cent.
2^n	$O(2^n)$	1×10^{-6} sec	3×10^{17} cent.	∞	∞
$n!$	$O(n!)$	0.003 sec	∞	∞	∞

Some Examples of Access Time

- **vector**: push_back(el), pop_back( ) have O(1) (constant upper bound) running time
- **vector**: insert at the front is O(N)
- **deque**: push_back(el), push_front(el), pop_back( ), push_back(el) are all O(1)
- **vector, deque**: insert in the middle is O(N)
- **list**: insert anywhere is O(1)
- **list**: Finding the location of an element has O(N) running time
- Most **set** and **map** operations are O(log N)

Nonmodifying Sequence Algorithms

- Nonmodifying algorithms that do not modify the container they operate upon
- **find**: Locates an element within a sequence
- **count**: Counts occurrences of a value in a sequence
- **equal**: Asks are elements in two ranges equal?
- **search**: Looks for the first occurrence of a match sequence within another sequence
- **binary_search**: Searches for a value in a container sorted using less. If the container was sorted using another predicate, this predicate must be supplied to binary_search.

Example for find()

- Definition: find(Iter first, Iter last, const T& value);
  - The first two specify a range: **[first, last)**, the third specifies a target **value** for the search.
  - If requested value is found, **find( )** returns an iterator that points to the first element with **value**.
  - If not found, it returns an iterator pointing one element past the final element. (same as **end()**)

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main() {
    list<int> L;
    list<int>::iterator it;
    L.push_back(10);
    L.push_back(20);
    L.push_back(30);
    it = find(L.begin(), L.end(), 30);
    if (it == L.end())
        cout << "data not found\n";
    else
        cout << *it << endl;
    return 0;
}
```

Output:

```
30
```


Example for count()

- Definition: count(Iter first, Iter last, const T& value);
  - The first two specify a range: **[first, last)**, the third specifies a target **value** for the count.
  - **count( )** returns the number of elements that equal to **value** within the specified range

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    int a[7] = {1,3,2,4,1,2,1};
    vector<int> v(a,a+7);
    int count1;
    count1 = count(v.begin(), v.end(), 1);
    cout << "count of 1 = " << count1
         << endl;

    return 0;
}
```

Output:

```
count of 1 = 3
```

Example for search()

- Definition: search(Iter s1, Iter s2, Iter t1, Iter t2);
  - The first two specify a range: **[s1, s2)** of the first container.
  - The following two specify a range: **[t1, t2)** of the second container.
  - **search()** returns a series of iterators that point to the elements in the first container that appear in the second container, within the specified range

```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    int a[7] = {1,3,2,5,1,2,1};
    vector<int> v(a, a+7);
    vector<int>::iterator it;
    list<int> L;
    L.push_back(5); L.push_back(1); L.push_back(2);
    it = search(v.begin(), v.end(), L.begin(), L.end());
    if (it != v.end())   // found
        cout << *it << ' ' << *(it+1) << ' '
             << *(it+2) << endl;
    return 0;
}
```

Output:

```
5 1 2
```


Container Modifying Algorithms

- Container modifying algorithms change the contents of the elements or their order
- **copy**: Copies from a source range to another location. **copy_n** can specify the number of copied elements
- **remove**: Removes all elements in a range that equal to the given value
- **swap**: Swaps the elements of two containers (same type)
- **reverse**: Reverses the order of the elements in the specified range
- **random_shuffle**: Randomly shuffles the elements of a sequence in the specified range

Example for copy()

- Definition: copy(Iter first, Iter last, Iter dest);
  - The first two specify a range: **[first, last)**, the third specifies the starting location **dest** for the copied sequence.
  - If **copy_n( )** is used, the second argument is replaced by the number of elements to be copied.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    int values[] = {1,2,3,4,5,6,7,8,9};
    vector<int> v1(values, values+9), v2(9), v3(5);
    copy(v1.begin(), v1.end(), v2.begin());
    // v2 is now 1,2,3,...,9
    copy_n(v1.begin(), 5, v3.begin());
    // v3 is now 1,2,3,4,5
    for (auto i : v2)
        cout << i <<' ';
    cout << endl;
    for (auto i : v3)
        cout << i <<' ';
    return 0;
}
```

Output:

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5
```


Example for reverse()

- Definition: reverse(Iter first, Iter last);
  - The first two specify a range: **[first, last]**, in which the order of elements are reversed.
  - If **reverse_copy( )** is used, you can specify a third argument as the starting point for the reversed sequence. The original sequence will be modified.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    int a[] = {1,5,4,9,8,6,1,3,5,4};
    reverse(a, a+10);
    // a is now 4,5,3,1,6,8,9,4,5,1
    reverse(a, a+5);
    // a is now 6,1,3,5,4,8,9,4,5,1
    vector<int> v1(a, a+10), v2(10);
    reverse(v1.begin(), v1.end());
    // v1 is now 1,5,4,9,8,4,5,3,1,6
    reverse_copy(v1.begin(), v1.end(), v2.begin());
    // v2 is now 6,1,3,5,4,8,9,4,5,1
    return 0;
}
```

Example for random_shuffle()

- Definition: random_shuffle(Iter first, Iter last);
  - The first two specify a range: **[first, last]**, in which the value of each element is swapped with that of another randomly picked element.
  - You can provide your own random function as the third argument.
  - The outputs shown in this example may be different to the results at your computer.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> myvec;
    for (int i=1; i<10; i++) myvec.push_back(i);
    // myvec now has 1,2,3,4,5,6,7,8,9
    random_shuffle(myvec.begin(), myvec.end()-3);
    // myvec now has 3,1,5,4,6,2,7,8,9
    random_shuffle(myvec.begin(), myvec.end());
    vector<int>::iterator it;
    cout << "My vector contains:\n";
    for (it = myvec.begin(); it != myvec.end(); ++it)
        cout << *it << ' ';
    cout << endl;
    return 0;
}
```

Output:

```
My vector contains:
4 3 1 6 8 9 2 7 5
```


Sorting Algorithms

- Theoretically, the runtime of the most efficient sort algorithms is O(N log(N))
  - N is the number of elements being sorted
- All the STL sorting algorithms are required to be O(N log(N)) → as fast as possible
- **sort**: Sorts elements within the given range in nondescending order
  - You can provide your own binary predicate to determine the order
- **merge**: Merges two sorted source ranges into a single destination range and still keep it sorted

Example for sort()

- sort(Iter first, Iter last, bp(e1,e2)): sorts the elements by user-defined order
  - If bp returns 1, e1 is kept at left and e2 is put at right
- Ex:

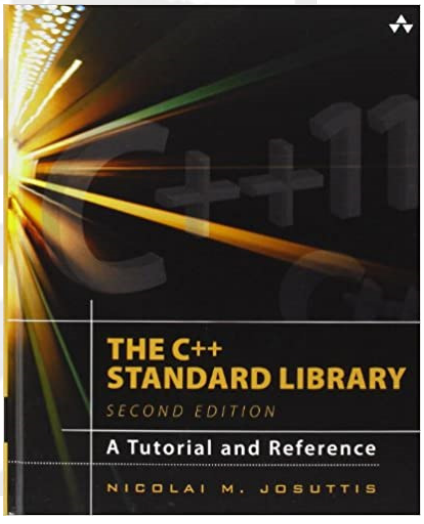
```
#include <iostream>
#include <algorithm>
using namespace std;
bool mygt(int i, int j)  { return i > j; }

int main() {
    int a[8] = {32,71,12,45,26,80,53,33};
    sort(a, a+8);          // a is sorted in increasing order
    cout << "default sorting results: ";
    for (int i=0; i<8; i++)  cout << a[i] << ' ';
    sort(a, a+8, mygt);  // a is sorted in decreasing order
    cout << "\nuser-defined sorting results: ";
    for (int i=0; i<8; i++)  cout << a[i] << ' ';
    cout << endl;
    return 0; }
```

Output:

```
default sorting results:
12 26 32 33 45 53 71 80
user-defined sorting results:
80 71 53 45 33 32 26 12
```


Recommended STL Reference Book

- Nicolai M. Josuttis, "The C++ Standard Library - A Tutorial and Reference", 2nd Edition
- Addison Wesley Longman, 2012, ISBN: 978-0321-62321-8
  - 1100+ pages
  - Chinese edition available
- Many online references are also available, ex:
  - www.cplusplus.com/reference/stl



Overview

- 9.1 Intro. to STL
- 9.2 Iterators
- 9.3 Containers
- 9.4 Generic Algorithms
- *9.5 Intro. to C++11*

C++ Is Evolving

- The International Standards Organization (ISO) ratifies proposed changes to the language
  - C++11, C++14, C++17 are some of the new versions
  - The number indicating the year of the draft standard
- Here we introduce some additions in C++11
  - std::array
  - Threads
  - Regular Expressions
  - Smart Pointers
- More changes are left for further study

The Standard Array Container

- std::array provides a vector-like notation for access into a fixed-size sequence of elements
  - It is a template-based class
- Provides safe array access with the performance and minimal storage of a regular array
- Ex: create an array of 4 integers:

```
#include <array>
using std::array;
array<int, 4> a = {1, 2, 3, 4};
// typical array: int a[4] = {1, 2, 3, 4};
// vector: vector<int> a(4);
```

Examples of Using std::array

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

```
for (int i = 0; i < a.size(); i++)
    cout << a[i] << endl;
```

- Given an array, index 3 initialized to zero:

```
array<int, 4> a = {1, 2, 3};  // missing 4th item
cout << a[2] << endl;         // 3
cout << a[3] << endl;         // 0
```

- No harmful effects accessing outside the boundaries of the array:

```
a[100] = 10;   // Ignored, no memory write
```

Regular Expressions

- A regular expression provides a way to describe the "patterns" for matching a sequence of text
  - Formally, a regular expression describes a language from the class of regular languages
  - Some compilers still do not support <regex> library
- Without regular expressions, it could be difficult to process the text for complicated patterns
  - Useful for reading files in some specific format
- This is a large topic, only a brief introduction is given here!

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

→ regex_match()
→ regex_search()
→ regex_replace()

Basic 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

Regular Expression	Meaning
\d	A single digit
\D	A non-digit
\s	A whitespace character (e.g., tab, newline, space)
\w	A word character
\\	A single \

- For example:
  - aaabbb or a{3}b{3} → three a's followed by three b's
  - a* → any sequence of zero or more a's
  - a+b* → One or more a's followed by any sequence of b's
  - [a-zA-Z_]+[a-zA-Z0-9_]* → The rules for an identifier
    - A letter or underscore followed by any sequence of letters, digits, or underscores

Ex: Regular Expression Matching

```
#include <iostream>
#include <regex>
#include <string>
using namespace std;
using std::string;
using std::regex;

int main( )
{
   // number format xxx-xxx-xxxx
   string phonePattern = R"(\d{3}-\d{3}-\d{4})";
   // two words separated by whitespace
   string twoWordPattern = R"(\w+\s\w+)";
   // create regex objects
   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 << "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;

    return 0;
}
```

Sample Outputs

Sample Dialogue 1

```
Enter a string to test the phone pattern.
907-867-5309
907-867-5309 matches \d{3}-\d{3}-\d{4}
Enter a string to test the two word pattern.
word up
word up matches \w+\s\w+
```

Sample Dialogue 2

```
Enter a string to test the phone pattern.
867-5309
867-5309 doesn't match \d{3}-\d{3}-\d{4}
Enter a string to test the two word pattern.
oneword
oneword doesn't match \w+\s\w+
```

Threads

- A thread is a separate computational process that runs concurrently
  - You can think of a thread as a program that can run at the same time (in parallel) as other threads
  - Useful for performance reasons and to prevent your program from blocking while waiting for input
- It is also possible to run a class in a thread
  - See textbook for example

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

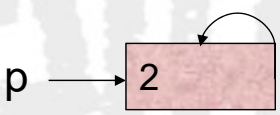
Sample Program of Threads

- This program runs func() in two threads and main runs in a third 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
}
```

Smart Pointers

- A template class that automatically frees up dynamic memory when they go out of scope
- Uses a technique called reference counting
  - Count how many pointers reference an allocated node
- Fails if there is a circular reference



Ref count = 2

- If p is reassigned, the memory won't be deleted
  - The reference count is stuck at 1 instead of zero

Reference Counting

- p = new Node();

- p = nullptr;

- q = p;

- q = nullptr;

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 is no longer needed.
// Will be deleted automatically when reference count reaches 0
```