Unit 13 Standard Template Library

Structure:

13.1 Introduction

In the previous unit you studied about the templates in C++. You got a clear idea of 'class' and 'function' templates. You also studied about the template instantiation, template specialization along with the template parameters. You also got some idea of using static members and variables in templates as well as using friend functions in templates. In this unit you will study in detail the standard template library.

Many people felt that C++ classes were inadequate in situations requiring containers for user-defined types, and methods for common operations on them. For example, you might need self-expanding arrays, which can easily be searched, sorted, added to or removed from without messing with memory reallocation and management. Other Object-oriented languages used *templates* to implement this sort of thing, and hence they were incorporated into C++.

To assist the users of C++ in generic programming, Alexender Stepnov and Meng Lee of Hewlett Packard developed a general purpose templatized classes and functions that could be used as a standard approach for the storing and processing of data. The standard template library is the collection of these generic classes and functions.

STL is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, i.e., its components are heavily parameterized: almost every component in the STL is a template. You should make sure that you understand how templates work in C++ before you use the STL.

Objectives:

After studying this unit you should be able to:

- explain the importance of STL
- list and explain the STL components
- list and explain types of containers
- explain the role of iterators in STL

13.2 STL Components

There are many components present in STL. But there are three major components in STL namely:

- Containers
- Algorithms
- Iterators.

These components work together to provide support to a variety of programming solutions. Figure 13.1 shows the relationship between the three components of STL. Algorithms employ iterators to perform operations stored in containers.

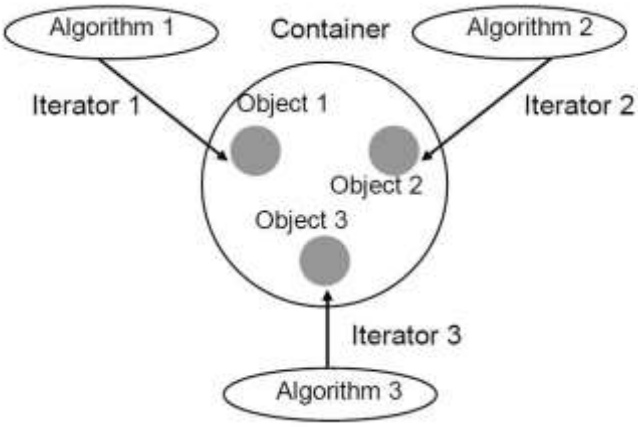

Figure 13.1: Relationship between the three STL components

13.2.1 Containers

The containers are objects that hold data of the same type. It is a way data is organized in memory. The STL containers are implemented by template classes, and hence, can be customized to hold many data types.

Very many container types are provided by STL which represents objects that contain other objects. The major ones are sequence containers, associative containers and derived containers. The classification of containers is shown in figure 13.2.

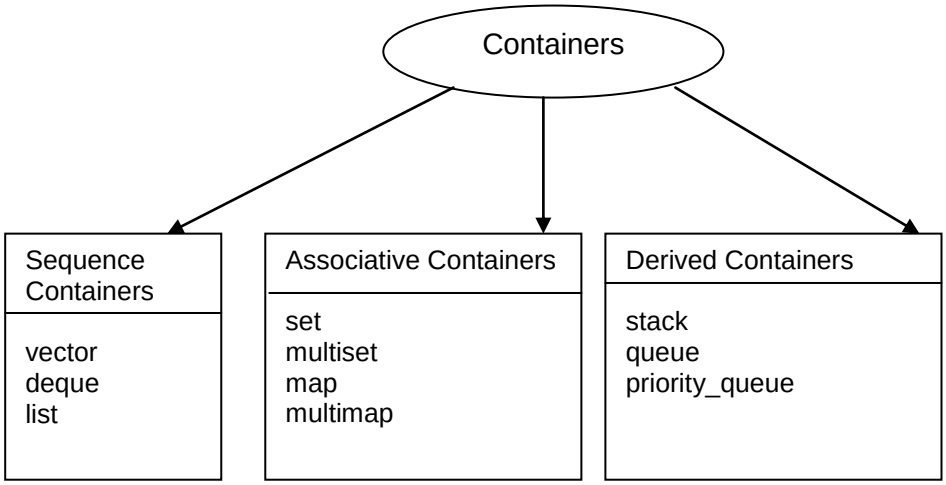

Figure 13.2: Three major categories of containers

As you can see in figure 13.2, the standard sequence containers include *vector*, *deque* and list. Data is stored in linear sequence in sequence containers. The standard associative containers can be categorized into set, multiset, map and multimap. Associative containers are a generalization of sequences. Sequences are indexed by integers; associative containers can be indexed by any type. The derived containers can be classified into three types i.e., stack, queue and priority queue.

13.2.2 Iterators

An iterator is an object (like a pointer) that points to an element in a container. Iterators can be used to move through the contents of the container. Iterators are handled just like pointers. Iterators can be incremented or decremented. They connect algorithms with containers, and play an important role in the manipulation of data stored in the container. Iterators are like location specifiers for containers or streams of data, in the same way that an int* can be used as a location specifier for an array of integers, or an ifstream can be used as a location specifier for a file.

The STL implements five different types of iterators. These are: *input iterators* (which can only be used to read a sequence of values), *output iterators* (which can only be used to write a sequence of values), *forward iterators* (which can be read, written to, and moved forward), *bidirectional iterators* (which are like forward iterators but can also move backwards) and *random access iterators* (which can move freely any number of steps in one operation). The table 13.1 shown below illustrates the iterators and their characteristics:

Table 13.1: Iterators and their characteristics

Iterator	Access Method	Direction of movement	I/O capability	Remark
Input	Linear	Forward Only	Read Only	Cannot be saved
Output	Linear	Forward Only	Write only	Cannot be saved
Forward	Linear	Forward Only	Read/Write	Cannot be saved
Bidirectional	Linear	Forward and backward	Read/Write	Cannot be saved
Random	Random	Forward and backward	Read/Write	Cannot be saved

It is possible to have bidirectional iterators act like random access iterators, as moving forward ten steps could be done by simply moving forward a step

at a time, a total of ten times. However, having distinct random access iterators offers efficiency advantages. For example, a vector would have a random access iterator, but a list only a bidirectional iterator.

Iterators are the major features which allow the generality of the STL. For example, an algorithm to reverse a sequence can be implemented using bidirectional iterators, and then the same implementation can be used on lists, vectors and deques. User-created containers have to provide only an iterator which implements one of the 5 standard iterator interfaces, and all the algorithms provided in the STL can be used on the container.

This generality also comes at a price at times. For example, performing a search on an associative container such as a map or set can be much slower while using iterators than by calling member functions offered by the container itself. This is because an associative container's methods can take advantage of knowledge of the internal structure, which is opaque to algorithms using iterators.

13.2.3 Algorithms

Algorithms are functions that can be used across a variety of containers for processing their contents. The STL algorithms are template C++ functions to perform operations on containers. Although each container provides functions for its basic operations, more than sixty standard algorithms are provided by STL to support more extended or complex operations. Standard algorithms also allow us to work with two different types of containers at the same time. In order to be able to work with many types of containers, the algorithms do not take containers as arguments. Instead, they take iterators that specify part or all of a container. In this way we can use algorithms to work on entities which are not the containers. For example, we can use the copy function to copy data from standard input into a vector. Numerous algorithms used to perform operations such as searching and sorting are provided in the STL; each of them is implemented to require a certain level of iterator (and therefore will work on any container that provides an interface by iterators).

The algorithms include sorting operations (sort, merge, min, max, etc.), searching operations (find, count, equal, etc.), mutating operations (transform, replace, fill, rotate, shuffle), and generalized numeric operations (accumulate, adjacent difference, etc.).

13.2.4 Function objects

The function object is a function that has been wrapped in a class so that it looks like an object. The class would have only one member function, an overloaded() operator and no data. This class is templatized to enable its use with different data types. It is used as arguments to certain containers and algorithms. For example the statement

sort(array, array+5, greater<int>());

uses the function object greater<int>() to sort the elements contained in array in the descending order.

It is useful in keeping and retrieving state information in functions passed into other functions. Regular function pointers can also be used as function objects. Function objects are STL's way of representing "executable data". For example, one of the STL algorithms is *for_each*. This applies a function to each object in a container. You need to be able to specify its function to each object in the container.

Self Assessment Questions:

1. The three major components in STL are_____,_____ and _____.
2. The containers are objects that hold data of the same type.
3. _____ is an object that points to an element in a container.
4. The _____ are template C++ functions to perform operations on containers.

13.3 Sequence Containers

Sequence containers store elements in a linear sequence, like a line, as shown in figure 13.3 below:

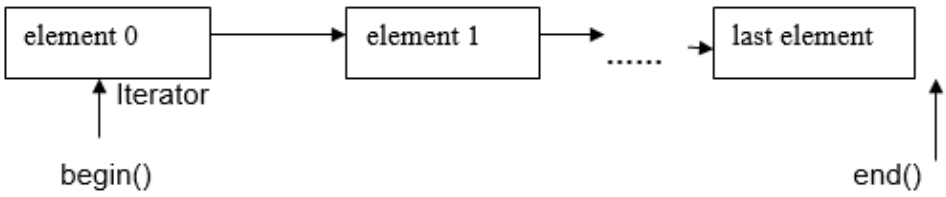

Figure 13.3: Elements in a sequence containers

Each element is related to other elements by its position along the line. They all expand themselves to allow insertion of elements, and all of them support a number of operations on them.

There are three types of *sequence containers* in the STL. These, as their name suggests, store data in linear sequence. They are the vector, deque and list:

- vector<Type>
- deque<Type>
- list<Type>

To choose a container, decide what sort of operations you will most frequently perform on your data, and then, use the following table to help you.

Table 13.2: Time overhead of operations on sequence containers

Operation	Vector	Deque	List
Access first element	*constant*	*constant*	*constant*
Access last element	*constant*	*constant*	*constant*
Access random element	*constant*	*constant*	*linear*
Add/delete at beginning	*Linear*	*constant*	*constant*
Add/delete at end	*constant*	*constant*	*constant*
Add/delete at random	*Linear*	*linear*	*constant*

Each container has attributes suited to particular applications. The subsections and code samples below should further clarify when and how to use each type of the sequence container.

13.3.1 Vector

#include <vector>

The vector class is similar to an array. Its syntax is also similar to an array e.g. vector[3]. Vector allows random access of the elements, with a constant time overhead, O(1). Performing insertion and deletion at the end of vectors is cheap. As with strings, no bounds-checking is performed with vectors when you use the operator [].

There is an overhead of O(N) to perform insertions and deletions at the end of the vector, where N is the number of vector elements. The reason for this

complexity is that the storage is contiguous, and to perform insertions and deletions in the middle, the elements have to be shuffled to make space for new entries. Memory overhead of a vector is very low and comparable to a normal array.

The table 13.3 shows some of the main vector functions:

Table 13.3: Vector functions

Vector access function	Purpose
begin()	Returns iterator pointing to the first element
end()	Returns iterator pointing _after_ the last element
push_back(…)	Add an element to the end of vector
pop_back(…)	Destroy an element at the end of vector
swap(,)	Swap two elements
insert(,)	Insert a new element
size()	The Number of elements in vector
capacity()	Element capacity before more memory is needed
empty()	True if the vector is empty
[ ]	Random access operator

The example below shows how vectors are used:

```
#include <stdlib.h>
// C++ STL Headers
#include <algorithm.h.>
#include <iostream.h>
#include <vector.h>
using namespace std;
int main(int argc, char *argv[])
 {
   int nitems = 0;
   int ival;
   vector<int> v;
   cout << "Enter integers, <Return> after each, <Ctrl>Z to finish:" << endl;
   while(cin >> ival, cin.good())
```

```
    {
        v.push_back(ival);
       cout.width(6);
       cout << nitems << ": " << v[nitems++] << endl;
    }
    if (nitems)
    {
       sort(v.begin(), v.end());
       for (vector<int>::const_iterator viter=v.begin(); viter!=v.end(); ++viter)
          cout << *viter << " ";
       cout << endl;
    }
    return(EXIT_SUCCESS);
}
```

Note how the element sort takes *v.begin()* and *v.end()* as range arguments. This is very common in the STL, and you will meet it again. The STL provides specialized variants of vectors: the *bitset* and *valarray*. The former allows a degree of array-like addressing for individual bits, and the latter is intended for numeric use with real or integer quantities. To use them, include the <*bitset*> or <*valarray*> header files (these are not always supported in current STL implementations). Be careful when you *erase()* or *insert()* elements in the middle of a vector. This can invalidate all existing iterators. To erase all elements in a vector, use the *clear()* member function.

13.3.2 Deque

#include <deque>

The double-ended queue, deque (pronounced "deck") has properties similar to those of a vector. But as you can observe from the name, it is possible to perform insertions and deletions at both the ends of a deque.

The table 13.4 shows some of the main deque functions:

Table 13.4: Deque functions

Deque access function	Purpose
begin()	Returns iterator pointing to the first element
end()	Returns iterator pointing _after_ the last element
push_front(…)	Add an element to the front of a deque
pop_front(…)	Destroy an element at the front of a deque
push_back(…)	Add an element to the end of a deque
pop_back(…)	Destroy an element at the end of a deque
swap(,)	Swap two elements
insert(,)	Insert a new element
size()	The number of elements in a deque
capacity()	Element capacity before more memory is needed
empty()	True if the deque is empty
[]	Random access operator

It is not easy to perform insertions and deletions at random positions in a deque. But it is easy - though not as efficiently as in a vector - to access the elements randomly using an array-like syntax. Like a vector, an *erase()* or *insert()* in the middle can invalidate all existing iterators.

The following program shows a deque representing a deck of cards. The queue is double-ended, so you could modify it to cheat and deal off the bottom.

```
#include <stdlib.h>
// C++ STL Headers
#include <algorithm>
#include <deque>
#include <iostream>
#ifdef _WIN32
using namespace std;
#endif
class Card
```

```
{
   public:
      Card() { Card(1,1); }
      Card( int s, int c ) { suit = s; card = c; }
      friend ostream & operator<<( ostream &os, const Card &card );
      int value() { return( card ); }
   private:
      int suit, card;
};
ostream & operator<<( ostream &os, const Card &card ) {
   static const char *suitname[] = { "Hearts", "Clubs", "Diamonds", "Spades"
};
   static const char *cardname[] = { "Ace", "2", "3", "4", "5", "6", "7",
                                     "8", "9", "10", "Jack", "Queen", "King" };
   return( os << cardname[card.card-1] << " of " << suitname[card.suit] );
}
class Deck {
   public:
      Deck() { newpack(); };
      void newpack() {
         for ( int i = 0; i < 4; ++i ) {
            for ( int j = 1; j <= 13; ++j ) cards.push_back( Card( i, j ) );
         }
      }
      // shuffle() uses the STL sequence modifying algorithm,
random_shuffle()
      void shuffle() { random_shuffle( cards.begin(), cards.end() ); }
      bool empty() const { return( cards.empty() ); }
      Card twist() { Card next = cards.front(); cards.pop_front(); return(next);
}
   private:
```

```
        deque< Card > cards;
};
int main( int argc, char *argv[] ) {
    Deck deck;
    Card card;
    int total, bank_total;
    char ch;
    //  End of declarations ...
        while ( 1 ) {
        cout << "\n\n ---- New deck ----" << endl;
        total = bank_total = 0;
        deck.shuffle();
        ch = 'T';
        while ( 1 ) {
            if ( total > 0 && total != 21 ) {
                cout << "Twist or Stick ? ";
                cin >> ch;
                if ( !cin.good() ) cin.clear(); // Catch Ctrl-Z
                    ch = toupper( ch );
            } else {
                if ( total == 21 ) ch = 'S'; // Stick at 21
            }
            if ( ch == 'Y' || ch == 'T' ) {
                card = deck.twist();
                total += card.value();
                cout << card << " makes a total of " << total << endl;
                if ( total > 21 ) {
                    cout << "Bust !" << endl;
                    break;
                }
            } else {
```

```
            cout << "You stuck at " << total << "\n"
                 << "Bank tries to beat you" << endl;
            while ( bank_total < total ) {
               if ( !deck.empty() ) {
                  card = deck.twist();
                  bank_total += card.value();
                  cout << card << " makes bank's total " << bank_total << endl;
                  if ( bank_total > 21 ) {
                     cout << "Bank has bust - You win !" << endl;
                     break;
                  } else if ( bank_total >= total ) {
                     cout << "Bank has won !" << endl;
                     break;
                  }
               }
            }
            break;
         }
      }
      cout << "New game [Y/N] ? ";
      cin >> ch;
      if ( !cin.good() ) cin.clear(); // Catch Ctrl-Z
      ch = toupper( ch );
      if ( ch != 'Y' && ch != 'T' ) break;
      deck.newpack();
   }

   return( EXIT_SUCCESS );
}
```

The card game is a version of pontoon, the idea being to get as close to 21 as possible. Aces are counted as one, and picture cards, as 10. Try to

modify the program to do smart addition and count aces as 10 or 1; use a vector to store your "hand" and give alternative totals.

Notice the check on the state of the input stream after reading in the character response. This is needed because if you hit, say, <Ctrl>Z, the input stream will be in an error state, and the next read will return immediately, causing a loop if you do not clear *cin* to a good state.

13.3.3 List

#include <list>

It is not possible with lists to access the elements randomly. It is best suited for applications in which it is required to add or delete elements to and from the middle. These elements are implemented as double linked list structures in order to support bidirectional iterators, and are the most memory-hungry standard containers, vector being the least so. In compensation, lists allow low-cost growth at either end, or in the middle.

Here are some main list functions:

Table 13.5: List functions

List access function	Purpose
begin()	Returns the iterator pointing to the first element
end()	Returns the iterator pointing _after_ the last element
push_front(…)	Add an element to the front of list
pop_front(…)	Destroy an element at the front of list
push_back(…)	Add an element to the end of list
pop_back(…)	Destroy an element at the end of list
swap(,)	Swap two elements
insert(,)	Insert a new element
size()	The number of elements in list
capacity()	Element capacity before more memory is needed
empty()	True if list is empty
sort()	Specific function because <algorithm>sort routines expect random access iterators

Following is an example with list in use:

```
#include <stdlib.h>
// C++ STL Headers
#include <algorithm>
#include <iostream>
#include <list>
#include <string>
using namespace std;
int main( int argc, char *argv[] ) {
   string things[] = { "JAF", "ROB", "PHIL", "ELLIOTT", "ANDRZEJ" };
   const int N = sizeof(things)/sizeof(things[0]);
   list< string > yrl;
   list< string >::iterator iter;
   for ( int i = 0; i < N; ++i) yrl.push_back( things[i] );
   for ( iter = yrl.begin(); iter != yrl.end(); ++iter ) cout << *iter << endl;
   //  Find "ELLIOTT"
   cout << "\nNow look for ELLIOTT" << endl;
   iter = find( yrl.begin(), yrl.end(), "ELLIOTT" );
   //  Mary should be ahead of Elliott
   if ( iter != yrl.end() ) {
      cout << "\nInsert MARY before ELLIOTT" << endl;
      yrl.insert( iter, "MARY" );
   } else {
      cout << "\nCouldn't find ELLIOTT" << endl;
   }
   for ( iter = yrl.begin(); iter != yrl.end(); ++iter ) cout << *iter << endl;
   return( EXIT_SUCCESS );
}
```

The loop over elements starts at yrl.begin() and ends just before *yrl.end()*. The STL *.end()* functions return iterators pointing just past the last element, and so, loops should do a != test, and not try to dereference this most likely invalid, position. Take care not to reuse (e.g. ++) iterators after they have

been used with *erase()* - they will be invalid. Other iterators, however, will still be valid after *erase()* or *insert()*.

Self Assessment Questions

5. The three types of sequence containers in the STL are_____, _____ and _____,
6. Vector allows _____ of the elements.
7. _____ are the best suited for applications in which it is required to add or delete elements to and from the middle.
8. _____list access function returns iterator pointing to the first element.

13.4 Associative Containers

Associative containers are designed to support direct access to elements using keys. They are not sequential. There are four types of associative containers. They are:

- Set
- Multiset
- Map
- Multimap

All these containers store data in a tree data structure which facilitates fast searching, deletion, and insertion. However, these are very slow for random access, and inefficient for sorting.

A number of items can be stored by set and multiset containers, and provides operations for manipulating them using the values as keys. For example, a set might store objects of the student class which are ordered alphabetically using names as keys. We can search for a desired student using his name as the key. The main difference between a set and a multiset is that a multiset allows duplicate items while a set does not.

Map and multipmap containers are used to store pairs of items, one called the key and the other called the value. The values can be manipulated using the keys associated with them. The values are sometimes called the mapped values. The main difference between a map and a multimap is that a map allows only one key for a given value to be stored while multiple keys are allowed by a multimap.

13.5 Derived Containers

Three derived containers are provided by STL. These are- stack, queue and priority queue. These are also known as container adapters. It is possible to create stack, queue and priority queues from different sequence containers. The derived containers are not supported by iterators, and hence, they cannot be used for data manipulation. However, two member functions i.e. push and pop, are supported by them for implementation of insertion and deletion operations.

Self Assessment Questions

9. The four types of associative containers are_____
10. The three derived containers provided by STL are _____.

13.6 Iterators

#include <iterator> // do not normally need you to include this yourself

An iterator you are already familiar with is a pointer into an array:

```
char name[] = "Word";
    char ch, *p;
    p = name;   // or &name[0] if you like
    ch = p[3];  // Use [] for random access
    ch = *(p+3);// Equivalent to the above
    *p = 'C';   // Write "through" p into name
    while ( *p && *p++ != 'r' ); // Read name through p
```

We can observe from the above program that the iterators are very flexible and powerful. As you can see, the above code uses variable ‘p’ in five different ways. We take it for granted that the compiler generates the appropriate offset for array elements, using the size of a single element.

The STL iterators you have already met are those returned by the *begin()* and *end()* container access functions, that let you loop over container elements. For example:

```
List<int> l;
List<int>::iterator liter; // Iterator for looping over list elements
for ( liter = l.begin(); liter != l.end(); ++liter ) {
   *liter = 0;
}
```

The end-of-loop condition is slightly different from normal. Usually the end condition would be a less than < comparison, but as you can see from the table of iterator categories below, not all iterators support <, so we increment the iterator from *begin()* and stop just before it becomes equal to *end()*. It is important to note that, for virtually all STL purposes, *end()* returns an iterator "pointing" to an element just after the last element, which it is not safe to dereference, but is safe to use in equality tests with another iterator of the same type. Sometimes, better performance is given by the pre increment operator (++). The reason for this is that there is no requirement of creating a temporary copy of the previous value, though the compiler usually optimizes this way.

Iterators can be termed as the generalized abstraction to the pointers which are designed to allow programmers to access different container types in a consistent way. To put it more simply, you can think of iterators as a "black box" between containers and algorithms. When you use a telephone to directly dial someone in another country, you do not need to know how the other phone system works. You can talk to the remote person provided it supports certain basic operations like dialing, ringing, reporting an engaged tone, hanging up after the call. . Similarly, if the minimum required iterator types for an algorithm are supported by a container class, then the algorithm will work with the container. This is important because it means that you can use algorithms such as the sort and random_shuffle as we've seen in the earlier examples. The authors of the algorithm are not required to know which containers they are acting on, provided we support the type of iterator required by that algorithm. The sort algorithm, for example, only needs to know how to move through the container elements, how to compare them, and how to swap them.

There are 5 categories of iterator:

- Random access iterators
- Bidirectional iterators
- Forward iterators
- Input iterators
- Output iterators

They are not all as powerful in terms of the operations they support - most do not allow random access- as we've seen in the difference between

vector and list. The figure 13.4 is the summary of the iterator hierarchy, the most capable at the top, and the operations supported on the right.

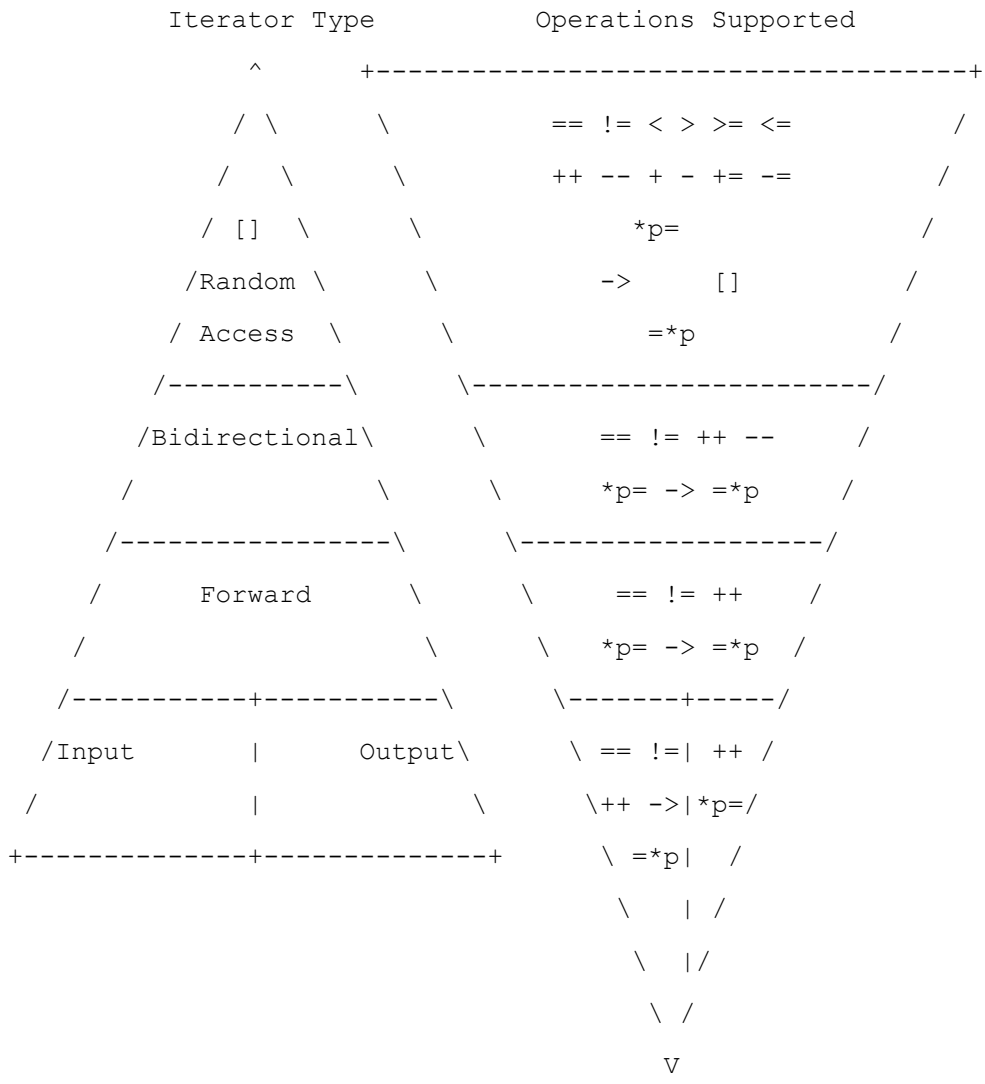

Figure 13.4: Iterator Hierarchy

The higher contains all the functionality of the layers below it. In addition, it contains some more functionalities. Only random iterators provide the ability to add or subtract an integer to or from the iterator, like *(p+3). If you write an iterator, it must provide all the operations needed for its category, e.g. if it is a forward iterator, it must provide ==, !=, ++, *p=, -> and =*p. It is to be taken care that ++p and p++ are different. ++p increments the iterator and

then it returns a reference to itself. p++ first returns a copy of itself, and then increments.

Operators must retain their conventional meaning, and elements must have the conventional copy semantics. In a nutshell, this means that the copy operation must produce an object that, when tested for equality with the original item, must match. Because only random iterators support integer add and subtract, all iterators except output iterators provide a *distance()* function to find the "distance" between any two iterators. The type of the value returned is:

```
template<class C> typename iterator_traits<C>::difference_type
```

This is useful if, for example, you *find()* a value in a container, and want to know the "position" of the element you have found.

```
map< key_type, data_type >::iterator im;
map< key_type, data_type >::difference_type dDiff;
im = my_map.find( key );
dDiff = distance( my_map.begin(), im );
```

This operation will be inefficient if the random access iterators are not supported by the containers because in that case, it will have to "walk through" the elements comparing the iterators.

Just as you can declare pointers to const objects, you can have iterators to const elements. The const_ prefix is used for this purpose.

```
e.g.        vector::iterator i;       // Similar to   my_type *i
            vector::const_iterator i; // Similar to   const my_type *i
```

The iterator_traits for a particular class is a collection of information, like the "iterator tag" category, which helps the STL "decide" on the best algorithm to use when calculating distances. The calculation is trivial for random iterators, but if you have only forward iterators, then it may be a case of slogging through a linked list to find the distance. If you write a new class of container, then this is one of the things you must be aware of. As it happens, the vector, list, deque, map and set - all provide at least Bidirectional iterators, but if you write a new algorithm, you should not assume any capability better than that which you really need. The lower the

category of iterator you use in your algorithm, the wider the range of containers your algorithm will work with.

Although the input and output iterators seem rather poor in capability, in fact they do add the useful ability to read and write containers to or from files. This is demonstrated in the program below:

```
#include <stdlib.h>
// C++ STL Headers
#include <algorithm>
#include <fstream>
#include <iostream>
#include <vector>
using namespace std;
int main( int argc, char *argv[] ) {
   int i, iarray[] = { 1,3,5,7,11,13,17,19 };
   fstream my_file("vector.dat",ios::out);// Add |ios::nocreate to avoid
                                          // creation if it doesn't exist
   vector<int> v1, v2;
   for (i = 0;i<sizeof(iarray)/sizeof(iarray[0]); ++i) v1.push_back(iarray[i]);
   //  Write v1 to file
   copy(v1.begin(),v1.end(), ostream_iterator<int,char>(my_file," "));
   cout << "Wrote vector v1 to file vector.dat" << endl;
   //  Close file
   my_file.close();
   //  Open file for reading or writing
   my_file.open( "vector.dat", ios::in|ios::out );
   //  Read v2 from file
   copy( istream_iterator<int,char>(my_file), // Start of my_file
        istream_iterator<int,char>(),        // Val. returned at eof
        inserter(v2,v2.begin()));
   cout << "Read vector v2 from file vector.dat" << endl;
   for ( vector<int>::const_iterator iv=v2.begin(); iv != v2.end(); ++iv )
      cout << *iv << " ";
```

```
    cout << endl;
    return(EXIT_SUCCESS);
}
```

The result of the possible restrictions on an iterator is that most algorithms have two iterators as their arguments, or (perhaps less safely) an iterator and a number of elements count. In particular, when you are using iterators, you need to be aware that it is not a good idea to test an iterator against NULL, or to test whether one iterator is greater than another. Testing for equality or inequality is safe except for output iterators, which is why the loops in the example code use iterator != x.end() as their termination test.

Self Assessment Questions

11. There are _____ categories of iterator.
12. Only random iterators provide the ability to add or subtract an integer to or from the iterator. (True/False)

13.7 Summary

- STL, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science.
- There are many components present in STL. But there are three major components in STL. They are: containers, algorithms and iterators.
- The containers are objects that hold data of the same type. It is the way data is organized in memory.
- An iterator is an object (like a pointer) that points to an element in a container. Iterators can be used to move through the contents of the container.
- Algorithms are functions that can be used across a variety of containers for processing their contents.
- The function object is a function that has been wrapped in a class so that it looks like an object.
- Sequence containers store elements in a linear sequence. There are three types of sequence containers in the STL- the vector, the deque and the list.
- The vector class is similar to an array. Vector allows random access of the elements, with a constant time overhead, O(1). It is possible to

perform insertions and deletions at both the ends of Lists. Lists are best suitable for the applications where it is required to add or delete the elements to and from the middle.

- Associative containers are designed to support direct access to elements using keys. They are not sequential. There are four types of associative containers: set, multiset, map and multimap.
- The derived containers provided by STL are: stack, queue and priority queue. These are also known as container adapters.

13.8 Terminal Questions

1. List and explain the STL components.
2. Explain sequence containers.
3. Explain associative and derived containers.
4. Discuss on iterators.

13.9 Answers

Self Assessment Questions

1. Containers, algorithms, iterators
2. Containers
3. Iterator
4. STL algorithms
5. Vector, deque and list
6. Random access
7. Lists
8. begin()
9. Set, multiset, amp and multimap
10. Stack, queue and priority queue
11. Five
12. True

Terminal Questions

1. STL provides a number of container types, representing objects that contain other objects. The STL contains sequence containers and associative containers. For more details refer section 13.2.
2. There are three types of sequence containers in the STL They are: the vector, the deque and the list. For more details refer section 13.3.

3. Associative containers are designed to support direct access to elements using keys. They are not sequential. There are four types of associative containers: set, multiset, map and multimap.

   Three derived containers are provided by STL. These are: stack, queue and priority queue. These are also known as container adapters. For more details refer section 13.4 and 13.5.

4. The STL iterators are returned by the *begin()* and *end()* container access functions. For more details refer section 13.6.

References:

- *Object-Oriented Programming with C++ - Sixth Edition*, by E Balagurusamy. Tata McGraw-Hill Education.
- *The C++ Programming Language, fourth edition*, by Bjarne Stroustrup. Addison-Wesley.
- *Object-Oriented Programming with ANSI and Turbo C++*, by Kamthane
- *http://clinuxpro.com/*