Standard Template Library (STL)

The Standard Template Library (STL) is the heart of the C++ standard library.

It includes the **containers**, part of the **iostream libraries**, **function objects**, and **algorithms**.

The STL is an example of **generic** programming. While the **object-oriented** programming concentrates on the **data aspect** of programming, **generic** programming concentrates on **algorithms**.

A goal of generic programming is to write code that is **independent of data types**. Templates are the C++ tools for creating generic program. Templates let us define a function or class in terms of a generic type.

The STL goes further by providing a generic representation of **algorithms**.

STL Component

STL has three key components - containers (popular templatized data structures), iterators and algorithms.

1. **Containers**   
   Containers manage collection of element. To meet different needs, the STL provides different kinds of containers.   
   * 1. **Sequential containers**   
        These are **ordered collections** in which every element has a certain position. This position depends on the time and place of the insertion, but it is independent of the value of the element. These include **vector**, **deque**, and **list**.
     2. **Associative containers**   
        These are **sorted collections** in which the actual position of an element depends on its value due to a sorting criterion. These include **set**, **multiset**, **map**, and **multimap**.
2. **Iterators**   
   Iterators are objects that can navigate over elements. An iterator represents a certain position in a container. Iterators are divided into five groups, based on the operations they support.  
   * 1. **Input iterators**   
        These are read-only iterators where each iterated location may be read only once. The most common manifestation of input iterator is **istream\_iterators**.
     2. **Output iterators**   
        These are write-only iterators where each iterated location may be written only once. The most common manifestation of output iterator is **ostream\_iterator**.
     3. **Forward iterators**   
        These have the capabilities of both input and output iterators, but they can read or write a single location repeatedly. They don't support **operator--**, so they can move only forward.
     4. **Bidirectional iterators**   
        These are just like forward iterators, except they can go backward as well as forward. The standard associative containers all offer bidirectional iterators.
     5. **Random access iterators**   
        These do everything bidirectional iterators do, but they also offer **iterator arithmetic**, i.e., the ability to jump forward or backward in a single step. **vector**, **string**, and **deque** each provide random access iterators.
3. **Algorithms**   
   The STL provides several standard algorithms for the processing of elements of collections. They can search, sort, modify, or simply use the element for different purpose. Algorithms use iterators. So, an algorithm has to be written only once to work with arbitrary containers because the iterator interface for iterators is common for all container types.

Vectors

A **vector** manages its elements in a dynamic array. It enables random access, which means we can access each element directly with index.

Appending and removing elements at the end of the array is very fast. But inserting an element in the middle or at the beginning of the array takes time because all the following elements should be moved to make room for it while maintaining the order.

Deques

**Double-ended-queue (deque)** is a dynamic array that is implemented so that it can grow in both directions. So, inserting element at the end and at the beginning is fast. Inserting elements in the middle, however, takes time because element must be moved.

Lists

A **list** is implemented as a doubly linked list of element. In other words, each element in a list has its own segment of memory and refers to its predecessor and its successor. Lists do **not** provide **random** access. General access to an arbitrary element takes linear time and this is a lot worse than vectors and deques.

The advantage of a list is that the insertion or removal of an element is fast at any position. Only the links must be changed. This implies that moving an element in the middle of a list is very fast compared with moving an element in a vector or a deque.

Vector

Initializing Vectors

#include <vector>

int main ()

{

std::vector<int> vec1; // empty vector of ints

std::vector<int> vec2 (3); // 3 ints

std::vector<int> vec2 (3,10); // 3 ints with value 10

std::vector<int> vec3 (vec2.begin(),vec2.end()); // iterating via vec2

std::vector<int> vec4 (vec3); // a copy of vec3

int myInt[] = {1,2,3}; // construct from arrays:

std::vector<int> vec5 (myInt, myInt + sizeof(myInt) / sizeof(int) );

return 0;

}

Using the push\_back()/pop\_back()/size()/clear()/empty() member function

#include <vector>

#include <iostream>

#include <string>

int main()

{

std::vector<std::string> Scientist;

Scientist.push\_back("James Maxwell");

Scientist.push\_back("Edwin Hubble");

Scientist.push\_back("Charles Augustin de Coulomb");

Scientist.push\_back("Louis Pasteur");

std::cout << "Now, we have " << Scientist.size() << " scientists.\n";

Scientist.pop\_back();

std::cout << "Now, we have " << Scientist.size() << " scientists.\n";

std::vector<std::string>::iterator iter;

for (iter = Scientist.begin(); iter != Scientist.end(); ++iter)

std::cout << \*iter << std::endl;

Scientist.clear();

if(Scientist.empty())

std::cout << "Nothing in the list\n";

else

std::cout << "You have something in the list\n";

return 0;

}

Output is:

Now, we have 4 scientists.

Now, we have 3 scientists.

James Maxwell

Edwin Hubble

Charles Augustin de Coulomb

Nothing in the list

The **push\_back()** adds a new element to the end of a vector. As a result, Scientist[0] is now equal to "James Maxwell", and Scientist[3] is now equal to "Louis Pasteur".  
The **pop\_back()** removes the last element of a vector and reduces the vector size by one. In the example, the size of "Scientist" is reduced from 4 to 3.  
The **clear()** member function removes all of the elements of a vector and sets its size to **0**.  
The **empty()** returns "true" if the vector is empty, otherwise, it returns "false."

Iterators

We can declare **iterator** as following: **container type::iterator new\_iterator**

vector<string>::iterator iter;

**Iterators** are values that identify an element in a container. We can access/change the value of the element using iterator.

We can also declare another iterator:

vector<string>::const\_iterator iter;

This **constant iterator** is just like a regular iterator except that we can think of a constant iterator as providing read-only access. The iterator itself, however, can change. In other words, we can move **iter** all around the vector. But we can't change the value of any of the elements through **iter**.

begin() and end()

In the above example, we assign the return value of **Scientist.begin()** to **iter**. The **begin()**returns an iterator that refers to a container's first element. So, the statement assigns an iterator that refers to the first element of **Scientist** to **iter**. Then, while looping through the elements, we test the return value of **Scientist.end()** against **iter** to make sure the two are equal.

The **end()** returns an iterator one past the last element in a container. This means the loop will continue until **iter** has looped through all of the elements in **Scientist**. In the action statement in the loop, **++iter**, increments **iter**, which traverse it to the next element in the vector. Depending upon the iterator, we can perform other mathematical operations on iterators to traverse them around a container.

In the body of the loop, we send **\*iter** to **cout**. By placing the dereference operator(\*) in front of **iter**, we display the value of the element to which the iterator refers.

insert() and erase()

We can insert an item, for example, an the beginning:

Scientist.insert(Scientist.begin(),"Leonardo da Vinci");

not or we can remove an item from the list, an element not at the end but from the middle:

Scientist.erase(Scientist.begin() + 2);

Vector - Performance

Vectors grow dynamically, and every vector has a specific size. When we add a new element to a vector, the computer reallocates memory and may even copy all of the vector elements into this new memory, and this can cause a performance hit.

capacity()

The **capacity()** returns the capacity of a vector (the number of elements that a vector can hold before a program must allocate more memory for it). So, a vector's capacity is not the same thing as its size which is the number of elements a vector currently holds. In short, **capacity()**is the size of the container and the **size()** is the currently filled level. The **capacity()** is always equal to or larger than the size. The difference between them is the number of elements that we can add to the vector before the array under the hood needs to be reallocated.

reserve()

Before we look into the **reserve()** we need to know what's happening whenever a vector needs more space. It's doing similar to **realloc** operation. New memory allocation, copy from the old to the new, destruct old objects, deallocate old memory, invalidation of iterators. It's expensive!

The **reserve()** increases the capacity of a vector to the number supplied as an argument. The **reserve()** gives us control over when a reallocation of additional memory occurs.

Scientist.reserve(20); // reserve memory for 20 **additional** elements

By using **reserve()** to keep a vector's capacity large enough for our purposes, we can delay memory reallocation.

According to Scott Meyers, the following code requires 2-18 reallocations:

vector<int> v;

for(int i = 0; i < 1000; ++i) v.push\_back(i);

So, he suggested we should use **reserve()** to reduce the costs:

vector<int> v;

v.reserve(1000);

for(int i = 0; i < 1000; ++i) v.push\_back(i);

insert() and erase()

Adding or removing an element from the end of a vector using **push\_back()** or **pop\_back()** is extremely efficient. Adding or removing an element at any other element in a vector, however, can require more work because we may have to move multiple elements to accommodate the insertion or deletion. With large vectors this can cause a performance hit.

Vector – Matrix

Vector - Matrix Initialization

Here is another example of vector of vector, 3x2, matrix initialization:

#include <iostream>

#include <vector>

using namespace std;

#define ROW 3

#define COL 2

int main()

{

// vector with ROW rows, each row has COL columns with initial value of 99

vector<vector<int> > v2D(ROW, vector<int>(COL,99));

for(int i = 0; i < ROW; ++i) {

for(int j = 0; j < COL; ++j) {

cout << v2D[i][j] << " ";

}

cout << endl;

}

return 0;

}

Output is:

99 99

99 99

99 99

Vector vs List

1. **vector**  
   1. Contiguous memory.
   2. Pre-allocates space for future elements, so extra space may be required.
   3. Unlike a list where additional space for a pointer is needed, each element only requires the space for the element type itself.
   4. Can re-allocate memory for the entire vector at any time that we add an element.
   5. Insertions at the end are constant, but insertions elsewhere are a costly O(n).
   6. Erasing an element at the end of the vector is constant time, but for the other locations it's O(n).
   7. We can randomly access its elements.
   8. Iterators, pointers, and references are invalidated if we add or remove elements to or from the vector.

vector::iterator it = v.begin();

for(it = v.begin(); it != v.end(); ++it) {

if(\*it == 5) v.erase(it);

}

So, after the \*it == 5, this may crash.

* 1. We can easily get at the underlying array if we need an array of the elements:

vector<int> v;

for(int i = 0; i < 10; ++i) v.push\_back(i);

int \*a = &v[0];

vector is very similar to the array, so the following line is true:

&v[i] == &v[0] + i;

1. **list**  
   1. Non-contiguous memory.
   2. No pre-allocated memory. The memory overhead for the list itself is constant.
   3. Each element requires extra space for the node which holds the element, including pointers to the next and previous elements in the list.
   4. Never has to re-allocate memory for the whole list just because we add an element.
   5. Insertions and erasures are cheap no matter where in the list they occur.
   6. It's cheap to combine lists with splicing.
   7. We cannot randomly access elements, so getting at a particular element in the list can be expensive.
   8. Iterators remain valid even when we add or remove elements from the list.
   9. If we need an array of the elements, we'll have to create a new one and add them all to it, since there is no underlying array.