The Standard Template Library (STL)

The STL contains several kinds of entities. The three most important are **containers**, **algorithms**, and **iterators**.

• A **container** (collection) is a way that stored data is organized in memory.

Examples are stack, linked list, the array. The STL containers are implemented by **template classes** so they can be easily customized to hold different kinds of data.

• **Algorithms** are procedures that are applied to containers to process their data in various ways.

For example, there are algorithms to sort, copy, search, and merge data. In the STL, algorithms are represented by **template functions**. These functions are not member functions of the container classes.

• ***Iterators*** are a generalization of the concept of pointers: They point to elements in a container.

You can increment an iterator, as you can a pointer, so it points in turn to each element in a container.

Iterators are a key part of the STL because they connect algorithms with containers. The STL iterators are implemented by classes.

Example:

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

int main(){
  vector <int> v;                    // vector (array) of integers
  vector <float> f;                  // vector (array) of floats
          :                          // some operations on arrays (fill)
  sort ( v.begin(), v.end() );       // sort is an algorithm int the STL
  sort ( f.begin(), f.end() );
          :                          // other operations on arrays
}
```

Container Template class

Algorithm Template function

Methods of vector

In STL vector is a container (template class).
sort is an algorithm (template function).
begin() and end() are member functions of vector . They returns iterators (pointers) to the first and last element of the array.

Containers:

Containers in the STL fall into two categories: ***sequential*** and ***associative***.

The sequential containers are *vector*, *list*, and *deque*.

The associative containers are *set*, *multiset*, *map*, and *multimap*.

In addition, several containers are called *abstract data types*, which are specialized versions of other containers. These are *stack*, *queue*, and *priority_queue*.

Sequential containers: Elements of the sequential containers can be accessed by position, for example, by using an index. An ordinary *C/C++* array is an example of a sequence container.

One problem with an ordinary *C/C++* array is that you must specify its size at compile time, that is, in the source code. You must specify an array large enough to hold what you guess is the maximum amount of data.

When the program runs, you will either waste space in memory by not filling the array or run out of space.

The STL provides the **vector** container to avoid these difficulties.

The STL provides the **list** container, which is based on the idea of a linked list.

The third sequence container is the **deque**, which can be thought of as a combination of a stack and a queue. A deque combines these approaches so you can insert or delete data from either end. The word "deque" is derived from *Double-Ended QUEue*.

Basic Sequential Containers:

Container	Characteristic	Advantages and Disadvantages
Ordinary C++ array	Fixed size	Quick random access (by index number). Slow to insert or erase in the middle. Size cannot be changed at runtime.
vector	Relocating, expandable array	Quick random access (by index number). Slow to insert or erase in the middle. Quick to insert or erase at end.
list	Doubly linked list	Quick to insert or delete at any location. Quick access to both ends. Slow random access.
deque	Like vector, but can be accessed at either end	Quick random access (using index number). Slow to insert or erase in the middle. Quick to insert or erase (push and pop) at either the beginning or the end.

Associative containers:

An associative container is not sequential; instead it uses *keys* to access data. The keys, typically numbers or stings, are used automatically by the container to arrange the stored elements in a specific order.
For example if **m** is an STL **map** that stores students names and uses students IDs as a key, then the statement,

std::string name = m[1504125001];

initializes name to a string value in m associated with the key 1504125001.

There are two kinds of associative containers in the STL: **maps** and **sets**. A map associates a key with a value. For example student's number and the student's name.
A set is similar to a map, but it stores only the keys; there are no associated values. For example, only the number of students.

The **map** and **set** containers allow **only one key** of a given value to be stored. This makes sense in, say, a phone book where you can assume that multiple people don't have the same number.
On the other hand, the **multimap** and **multiset** containers allow multiple keys. In an English dictionary, there might be several entries for the word "set," for example.

Basic associative containers are: **map ,set, multimap, multiset.**

Container	Characteristics	Advantages and Disadvantages
Map	Associates key with element Only one key of each value allowed	Quick random access (by key). Inefficient if keys not evenly distributed.
Multimap	Associates key with element Multiple key values allowed	Quick random access (by key). Inefficient if keys not evenly distributed.
Set	Stores only the keys themselves Only one key of each value allowed	Quick random access (by key). Inefficient if keys not evenly distributed.
Multiset	Stores only the keys themselves Multiple key values allowed	Quick random access (by key). Inefficient if keys not evenly distributed.

Instantiating an STL container object is easy. First, you must include an appropriate header file. Then you use the template format with the kind of objects to be stored as the parameter. Examples might be

```
include <vector>
:
vector<ComplexT> cvect;            // create a vector of complex numbers

include <list>
:
list<Teacher> teacher_list;        // create a list of Teachers

include <map>
:
map<int,string> IntMap;            // create a map of ints and strings

include <multiset>
:
multiset<employee> machinists; // create a multiset of employees
```

Notice that there's no need to specify the size of STL containers. The containers themselves take care of all memory allocation.

Member Functions

Algorithms are template functions of the STL, carrying out complex operations such as sorting and searching. They are not members of any template class. However, containers also need member functions to perform simpler tasks that are specific to a particular type of container.
Some member functions common to all containers:

size()	Returns the number of items in the container.
empty()	Returns true if container is empty.
max_size()	Returns size of the largest possible container.
begin()	Returns an iterator to the start of the container for iterating forward through the container.
end()	Returns an iterator to the past-the-end location in the container, used to end forward iteration.
rbegin()	Returns a reverse iterator to the end of the container for iterating backward through the container.
rend()	Returns a reverse iterator to the beginning of the container, used to end backward iteration.

See Example eA2_1.cpp

Member Functions of Sequential Containers

Vectors:
- Smart arrays.
- They manage storage allocation for you, expanding and contracting the size of the vector as you insert or erase data.
- You can use vectors much like arrays, accessing elements with the [] operator.
- Such random access is very fast with vectors.
- It's also fast to add (or *push*) a new data item onto the end (the *back*) of the vector.
- When this happens, the vector's size is automatically increased to hold the new item.

Some member functions:
constructors,
push_back(),
size(),
operator [ ],
swap(),
empty(),
back(),
pop_back(),
insert(),
erase()

See Example eA2_2.cpp

See Example eA2_3.cpp

See Example eA2_4.cpp

Lists:
- Doubly linked list.
- Each element contains a pointer not only to the next element but also to the preceding one.
- The container stores the address of both the front (first) and the back (last) elements, which makes for fast access to both ends of the list.

Some member functions:
push_front(),
front(),
pop_front
reverse(),
merge(),
unique()

The **insert()** and **erase()** member functions are used for list insertion and deletion, but they require the use of iterators, so we will see these functions later.

See eA2_5.cpp

See eA2_6.cpp

Deques:
- A deque is a variation of a vector.
- Like a vector, it supports random access using the [ ] operator.
- Unlike a vector (but like a list), a deque can be accessed at the front as well as the back.
- It's a sort of double-ended vector, supporting **push_front(), pop_front(),** and **front().**
- Memory is allocated differently for vectors and queues. A vector always occupies a contiguous region of memory. A deque, on the other hand, can be stored in several noncontiguous areas; it is segmented.

See eA2_7.cpp

Iterators:

Iterators are "smart" pointers to items in containers.

In general, the following holds true of iterators:

- Given an iterator **iter**, ***iter** represents the object the iterator points to (alternatively, **iter->** can be used to reach the object the iterator points to).
- **++iter** or **iter++** advances the iterator to the next element. The notion of advancing an iterator to the next element is consequently applied: several containers have a *reversed* iterator type, in which the **iter++** operation actually reaches an previous element in a sequence.
- For the containers that have their elements stored consecutively in memory *pointer arithmetic* is available as well. This counts out the **list**, but includes the **vector, queue, deque, set** and **map**. For these containers **iter + 2** points to the second element beyond the one to which **iter** points.

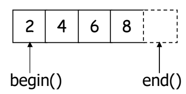
The STL containers include *typedefs* to define iterators. They also produce iterators (i.e., type **iterator**) using member functions **begin()** and **end()** and, in the case of reversed iterators (type **reverse_iterator**), **rbegin()** and **rend().**

```
vector<string> vs;                  // vs is an array of strings
vector<string>::iterator iter;      // iter is an iterator of vs
iter =  vs.begin();                 // iter points to the first element of vs
++iter;                             // iter points to the next element
```


Data access:

In containers that provide random-access iterators (vector and queue), it's easy to iterate through the container using the [ ] operator. Containers such as lists, which don't support random access, require a different approach.

```
int main() {
   int arr[ ] = { 2, 4, 6, 8 };         // array of ints
   list<int> iList(arr, arr+4);         // list initialized to array
   list<int>::iterator it;              // iterator to list-of-ints
   for(it = iList.begin(); it != iList.end();  it++)   cout << *it << ' ';
   return 0;
}
```



An equivalent approach, using a while loop instead of a for loop, might be

```
it = iList.begin();
while( it != iList.end() )
      cout << *it++ << ' ';
```

Data insertion:

```
int main() {
   list<int> iList(5);                                  // empty list holds 5 ints
   list<int>::iterator it;                              // iterator
   int data = 0;                                        // fill list with data
   for(it = iList.begin(); it != iList.end(); it++)
      *it = data += 2;
   for(it = iList.begin(); it != iList.end(); it++) cout << *it << ' ';     // display list
   return 0;
}
```

The first loop fills the container with the int values 2, 4, 6, 8, 10, showing that the overloaded * operator works on the left side of the equal sign as well as on the right. The second loop displays these values.

Example: Shapes with the STL: Inheritance and Polymorphism

See Example: eA2_8.cpp

Constant iterators:

The STL defines *const_iterator* types to be able to visit a range of the elements in a constant container. Whereas the elements of the list in the previous example could have been altered, the elements of the vector in the next example are immutable, and **const_iterator**s are required:

```
int main() {
   int arr[ ] = { 2, 4, 6, 8 };                    // array of ints
   const vector<int> v1(arr, arr+4); // vector initialized to array
   vector<int>::const_iterator it;                 // constant iterator to vector-of-ints
   for(it = v1.begin(); it != v1.end();  it++)   cout << *it << ' ';
}
```

Reverse iterators:

Suppose you want to iterate backward through a container from the end to the beginning. You might think you could say something like

```
list<int>::iterator it;              // normal iterator
it = iList.end();                    // start at end
while( it != iList.begin() )         // go to beginning
cout << *--i << ' ';                 // pre-decrement iterator
```

To iterate backward, the better way is to use a ***reverse iterator***.

Using the reverse operator

```
int main() {
   int arr[ ] = { 2, 4, 6, 8, 10 };             // array of ints
   list<int> iList(arr, arr+5);                 // list initialized to array
   list<int>::reverse_iterator revit;           // reverse iterator
   revit = iList.rbegin();                      // iterate backwards
   while( revit != iList.rend() )               // through list,
      cout << *revit++ << ' ';                  // displaying output
  return 0;
}
```

Reverse iterators act like pointers to elements of the container, *except that when you apply the increment operator to them, they move backward rather than forward.*
You must use the member functions rbegin() and rend() when you use a reverse iterator. (But don't try to use them with a normal forward iterator.) Confusingly, you're starting at the end of the container, but the member function is called rbegin(). Also, you must increment the iterator. Don't try to decrement a reverse iterator; revit-- doesn't do what you want. With a reverse_iterator, always go from rbegin() to rend() using the increment operator.

Algorithms:

- Template functions. They are not members of any class.
- These algorithms were designed to work with STL containers, but you can apply them also to ordinary C++ arrays.

Examples:

The find() Algorithm:
Function prototype: Iterator find(Iterator first, Iterator last, Type const &value);
Description: Element value is searched for in the range of the elements implied by the iterator range [first, last). An iterator pointing to the first element found is returned. If the element was not found, last is returned. The operator==() of the underlying data type is used to compare the elements.

```
#include <iostream>
#include <algorithm>                // must be included for algorithms
using namespace std;
int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };
int main() {
   int* ptr;
   ptr = find(arr, arr+8, 33);        // find first 33
   cout << "First object with value 33 found at offset " << (ptr-arr) << endl;
}
```

The output from this program is
First object with value 33 found at offset 2.

Using the **find()** algorithm with containers

```
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;
int main() {
  list<int> iList(5);                               // empty list holds 5 ints
  list<int>::iterator it;                           // iterator
  int data = 0;                                     // fill list with data
  for(it = iList.begin(); it != iList.end(); it++)
      *it = data += 2;
  it = find(iList.begin(), iList.end(), 8);         // look for number 8
  if( it != iList.end() ) cout << "\nFound 8";
    else                  cout << "\nDid not find 8."; }
```

As an algorithm, find() takes three arguments. The first two are iterator values specifying the range to be searched and the third is the value to be found.
Here I fill the container with the same 2, 4, 6, 8, 10 values as in the last example.
Then I use the find() algorithm to look for the number 8. If find() returns iList.end(), I know it's reached the end of the container without finding a match. Otherwise, it must have located an item with the value 8. Here the output is
Found 8

You can also use algorithms with user defined classes. But classes must include necessary operators, which are used by algorithms. For example the **find()** algorithm uses the **operator==()** of the underlying data type to compare the elements.

```
class ComplexT{
    float re,im;
  public:
    set(float r, float i){re=r; im=i;}
    bool operator==(const ComplexT &c) const{
          return re==c.re && im==c.im;
    }
};
```

```
int main(){
   ComplexT z[3];
   z[0].set(1.1, 1.2);
   z[1].set(2.1, 2.2);
   z[2].set(3.1, 3.2);
   ComplexT zSearch;
   zSearch.set(2.1, 2.2);
   ComplexT *result;
   result=find(z, z+3, zSearch);
   if (result == z+3) cout << "Not found";
      else            cout << "Found";
   return 0;
}
```

See Example: eA2_9.cpp

Example: **sort()**

Function prototypes:

```
void sort(Iterator first, Iterator last);
void sort( Iterator first, Iterator last, comp);
```

Description:

- The first prototype: the elements in the range [first, last) are sorted in ascending order, using the operator<() of the underlying data type.
- The second prototype: the elements in the range [first, last) are sorted in ascending order, using the comp function object to compare the elements.

Example:

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
int main()
{
   string words[]=  {"november", "kilo", "mike", "lima","oscar", "quebec", "papa"};
   sort(words, words +7);
   for(int i =0 ; i<7; i++)  cout << words[i] << endl;
   return 0;
}
```


Other prototype of **sort()** uses a given function to compare arguments.
void sort(Iterator first, Iterator last, **comp**);
In this case the elements in the range [first, last) are sorted in ascending(**?**) order, using the **comp** function to compare the elements.
The **comp** function can be a user written function:

```
#include <iostream>
#include <algorithm>
#include <string>
using namespace std;
bool after( const string &left, const string &right)
{
   return left > right;
}

int main()
{
   string words[]=  {"november", "kilo", "mike", "lima",  "oscar", "quebec", "papa"};
   sort(words, words +7, after);                    // after is a pointer to a function
   for(int i =0 ; i<7; i++) cout << words[i] << endl;
   return 0;
}
```

See Example: eA2_10.cpp

See Example: eA2_11.cpp

In this example elements are sorted in descending order because of **after** function.

Function Objects

Some algorithms can take something called a *function object* as an argument. A function object is actually an object of a template class that has a single member function: the overloaded **() operator**. The names of these classes can be used as function names.
In the header file functional there are many useful template classes which include a single member function the overloaded function call () operator. For example, a function object can be created from class **greater** to use with the sort algorithm:

```
template<class T>
struct greater {
   bool operator()(const T& x, const T& y) const
   {
      return x > y ;
   }
};
```

```
#include <iostream>
#include <algorithm>
#include <string>
#include <functional>
using namespace std;

int main()
{   string words[]=  {"november", "kilo", "mike", "lima",  "oscar", "quebec", "papa"};
    sort(words, words +7, greater<string>());       // greater () is a function object
    for(int i =0 ; i<7; i++) cout << words[i] << endl;
    return 0;
}
```

In this example elements are sorted in descending order because of **greater** function object.

Iterators and Algorithms

Besides acting as smart pointers to items in containers, iterators serve another important purpose in the STL. They determine which algorithms can be used with which containers.

In some theoretical sense, you should be able to apply every algorithm to every container. In fact, many algorithms will work with all the STL containers. However, some algorithms are inefficient (i.e., slow) when used with some containers. The sort() algorithm, for example, needs random access to the container it's trying to sort; otherwise, it would need to iterate through the container to find each element before moving it, a time-consuming approach.

Similarly, to be efficient, the reverse() algorithm needs to iterate backward as well as forward through a container.

Iterators provide an elegant way to match appropriate algorithms with containers. If you try to use an algorithm that's too powerful for a given container type, then you won't be able to find an iterator to connect them. If you try it, you will receive a compiler error alerting you to the problem.

The STL defines five types of iterators to make this scheme work.

- **InputIterators:** InputIterators can read elements from a container. The dereference operator is guaranteed to work as an rvalue in an expression, not as an lvalue.
- **OutputIterators:** OutputIterators can be used to write to a container. The de-reference operator is guaranteed to work as an lvalue in an expression, not as an rvalue.

- **ForwardIterators:** ForwardIterators combine InputIterators and OutputIterators. They can be used to traverse the container in one direction, for reading and/or writing.
- **BidirectionalIterators:** BidirectionalIterators allow the traversal of a container in both directions, for reading and writing.
- **RandomAccessIterators:** RandomAccessIterators provide access to any element of the container at any moment. An algorithm such as sort() requires a RandomAccessIterator, and can therefore not be used with lists or maps, which only provide BidirectionalIterators.

Iterator	**Step Forward**	**Read**	**Write**	**Step Back**	**Random Access**
Operation	++	value=*i	*i=value	--	[n]
Random-access iterator	x	x	x	x	x
Bidirectional iterator	x	x	x	x	
Forward iterator	x	x	x		
Output iterator	x		x		
Input iterator	x	x			

If you confine yourself to the basic STL containers, you will be using only two kinds of iterators. The vector and deque require a random-access iterator, whereas the list, set, multiset, map, and multimap require only bi-directional iterators.

When you define an iterator, you must specify what kind of container it will be used for. For example, if you've defined a list holding elements of type int,

```
list<int> iList;          // list of ints
```

then to define an iterator to this list you say

```
list<int>::iterator it; // iterator to list-of-ints
```

When you do this, the STL automatically makes this iterator a bi-directional iterator because that's what a list requires. An iterator to a vector or a deque, on the other hand, is automatically created as a random-access iterator.

Plugging iterators into ane algorithm:

Every algorithm, depending on what it will do to the elements in a container, requires a certain kind of iterator. If the algorithm must access elements at arbitrary locations in the container, it requires a random-access iterator. If it will merely step forward through the iterator, it can use the less powerful forward iterator.

Algorithm	Input	Output	Forward	Bidirectional	Random Access
for_each	x				
find	x				
count	x				
copy	x	x			
replace			x		
unique			x		
reverse				x	
sort					x
nth_element					x
merge	x	x			
accumulate	x				

Although each algorithm requires an iterator with a certain level of capability, a more powerful iterator will also work. The replace() algorithm requires a forward iterator, but it will work with a bi-directional or a random-access iterator as well.

•Instead of an InputIterator it is also possible to use a Forward-, Bidirectional- or RandomAccessIterator.

•Instead of an OutputIterator it is also possible to use a Forward-, Bidirectional- or RandomAccessIterator.

• Instead of a ForwardIterator it is also possible to use a Bidirectional- or RandomAccessIterator.

•Instead of a BidirectionalIterator it is also possible to use a RandomAccessIterator.

From the previous tables, you can figure out whether an algorithm will work with a given container. The table shows that the sort() algorithm, for example, requires a random-access iterator. The only containers that can handle random-access iterators are vectors and deques. There's no use trying to apply the sort() algorithm to lists, sets, maps, and so on.

Any algorithm that does *not* require a random-access iterator will work with any kind of STL container because all these containers use bi-directional iterators, which is only one grade below random access.

As you can see, comparatively few algorithms require random-access iterators. Therefore, most algorithms work with most containers.

for_each() algorithm:

The for_each() algorithm allows you to do something to every item in a container. You write your own function to determine what that "something" is. Your function can't change the elements in the container, but it can use or display their values.

Function prototype:

func for_each(InputIterator first, InputIterator last, Function func);

Description:

Each of the elements implied by the iterator range [first, last) is passed in turn to the *function* func. The function may not modify the elements it receives (as the used iterator is an input iterator). If the elements are to be transformed, transform() should be used.

Example: for_each() is used to convert all the values of an array from inches to centimeters and display them.

```
void in_to_cm(float in)                                 // convert and display as centimeters
{
   cout << (in * 2.54) << ' ';
}
int main()
{                                                       // array of inches values
   float array[] = { 3.5, 6.2, 1.0, 12.75, 4.33 };
   vector<float> inches (array, array+5);               // vector of inches values
   for_each(inches.begin(), inches.end(), in_to_cm);    // output as centimeters
   return 0;
}
```

See Example:eA2_12.cpp

Associative Containers

The two main categories of associative containers in the STL are maps and sets.

A map (sometimes called a *dictionary* or *symbol table*) stores *key* and *value* pairs. The keys are arranged in sorted order.

A set is similar to a dictionary, but it stores only keys; there are no values.

In both a set and a map, only one example of each key can be stored. It's like a dictionary that forbids more than one entry for each word.

A *multiset* and a *multimap* are similar to a set and a map, but can include multiple instances of the same key.

The advantages of associative containers are that, given a specific key, you can quickly access the information associated with this key; it is much faster than by searching item by item through a sequence container. On normal associative containers, you can also quickly iterate through the container in sorted order.

Associative containers share many member functions with other containers. However, some algorithms, such as lower_bound() and upper_bound(), exist only for associative containers. Also, some member functions that do exist for other containers, such as the push and pop family (push_back() and so on), have no versions for associative containers.

Set

The set class implements a set of (sorted) values. To use the set, the header file set must be included: #include <set>

A set is filled with values, which may be of any container-acceptable type. Each value can be stored only once in a set.

See Example: eA2_13.cpp

An important pair of member functions available only with associative containers is the lower_bound() and upper_bound().

```
set<string> city;
set<string>::iterator iter;   // iterator to set
city.insert("Trabzon");       // insert city names
      :
iter = city.begin();          // display set
while( iter != city.end() )
   cout << *iter++ << endl;

string lower, upper;          // display entries in range
cout << "\nEnter range (example A Azz): ";
cin >> lower >> upper;
iter = city.lower_bound(lower);
while( iter != city.upper_bound(upper) )
   cout << *iter++ << endl;
```

The program first displays an entire set of cities. The user is then prompted to type in a pair of key values, and the program will display those keys that lie within this range.

See Example: eA2_14.cpp

Map

The map class implements a (sorted) associative array. To use the map, the header file map must be included: #include <map>

A map is filled with *Key/Value* pairs, which may be of any container-acceptable type.

The key is used for looking up the information belonging to the key. The associated information is the *Value*. For example, a phonebook uses the names of people as the key, and uses the telephone number and maybe other information as the value.

Basically, the operations on a map are the storage of *Key/Value* combinations, and looking for a value, given a key. Each key can be stored only once in a map. If the same key is entered twice, the last entered *key/value* pair is stored, and the pair that was entered before is lost. **Example:** Cities and their plate numbers.

```
int main()
{                                         // set of string objects
   map<string,int> city_num;
   city_num["Trabzon"] = 61;    // insert city names and numbers
   city_num["Adana"] = 01;
   string city_name;
   cout << "\nEnter a city: ";
   cin >> city_name;
   if ( city_num.end() == city_num.find(city_name) )
      cout << city_name << " is not in the database" << endl;
   else
      cout << "Number of " << city_name << ": " << city_num[city_name];
}
```

See Example: eA2_15.cpp

Container Adaptors

It's possible to use basic containers to create another kind of container called a *container adaptor*. An *adaptor* is a sort of simplified or conceptual container that emphasizes certain aspects of a more basic container; it provides a different *interface* to the programmer.
The adaptors implemented in the STL are stacks, queues, and priority queues.
A **stack** restricts access to pushing and popping a data item on and off the top of the stack.
In a **queue**, you push items at one end and pop them off the other end.
In a **priority queue**, you push data in the front in random order, but when you pop the data off the other end, you always pop the largest item stored: The priority queue automatically sorts the data for you.
Adaptors are template classes that translate functions used in the new container (such as push and pop) to functions used by the underlying container.
Stacks, queues, and priority queues can be created from different sequence containers, although the deque is often the most obvious choice.
You use a template within a template to instantiate a new container. For example, here's a stack object that holds type int, instantiated from the deque class:

```
stack< int, deque<int> > int_stack;
```

By default, an STL stack adapts a deque. So you can define a stack as follows:

```
stack< int > int_stack;
```

We could force a stack to adapt a vector with the definition:

```
stack< int, vector<int> > int_stack;
```