ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

The Standard Template Library
Savitch chapter 19

other useful books

The C++ standard library : a tutorial and handbook by Nicolai Josuttis (This is a very useful reference. There is one copy to consult in the library. I have the other library copy and will bring it to labs)

C++ templates : the complete guide by David Vandevoorde and Nicolai M. Josuttis (also one copy in the library)

The Standard Template Library

- STL is part of the ANSI C++ standard

- The STL includes a library of standard abstract data types
  - Many of them container classes like stacks and queues
    - Some containers available in the Standard Template Library are:

      `vector`, `list`, `deque` (*pronounced deck*), `set`, `map`, `dictionary`, `stack`, `queue`
    - All of these containers have different features, and all can be implemented as containers to hold any type (as an array can)
  - STL containers are part of the toolbox used by every programmer.
  - Containers include the idea of an `Iterator`, an object that cycles through the elements of a container.

- The STL also includes implementations of many important generic algorithms
  - such as searching and sorting

STL Containers - Common features

- Containers grow as needed
- Containers hold elements of any type (may be a primitive type or a user defined type such as a class)
- The iterator provides a common way to access all elements of a container

- Every container has methods `begin()` `end()` `size()` `empty()`, `swap()`
  - All the methods must be implemented in constant time (not dependent on the number of items in the container)
  - `end()` returns a ‘NULL pointer’ that acts as an end-marker
  - `swap()` takes as an argument another container of the same type

- Every container supports operators as follows (*as long as the element-type it contains supports the operators*)
  - `==` , `!=` (equal if containers of same size and all corresponding elements equal)
  - `<` `>` `<=` `>=` return true if true when applied to each element in the 2 containers

Iterators

- A generalisation of a pointer
  - You will not go far wrong if you think of and use the iterator as a pointer
- The actual behaviour of the iterator depends on the type of container it iterates over.
  - Some container iterators have more functionality than others
- Overloaded operators that can be used with iterators over most container types
  - Prefix and postfix operators (++ and --) which move the iterator along
  - Equal and not equal comparators (== and !=) to see if 2 iterators are pointing to the same data item
  - De-referencing operator (*) to access the data pointed to by the iterator
- Container classes themselves have member functions to get the iterator process started.
  - `begin()` and `end()` methods
- processing elements in a container using an iterator

```
// where p is an iterator variable of the type for the
// container object c
for (p = c.begin(); p!= c.end(); p++)
   process *p //*p is the current data item
```

Constant and Mutable Iterators

- With a constant iterator, the de-referencing operator produces a read-only (constant) version of the element
  - This means it is NOT AN L-VALUE
  - the element in the container cannot be changed through a constant iterator pointer
  - The element pointed to by the constant iterator can still be printed, or assigned to another variable
- Most types of containers support both constant and mutable (i.e. non-constant) iterators
- A constant container object will only support constant iterators

Examples:

An iterator over a set of integers is of type

```
set<int>::iterator
```

A constant iterator over a vector of Employee objects is of type

```
vector<Employee>::const_iterator
```

STL sequential container classes

- Elements of a sequential container are organised in a linear arrangement
  - There is a first and a last element, and each element has a predecessor and a successor
- There are 3 sequential containers in the STL, **`vector`, `list`, `deque`**
  - **`vector`** provides random access by using the subscript or index to get to any position
  - **`deque`** is a double-ended queue, used by default to implement both a queue and a stack, but it also allows random-access by subscript at any position

    The interface for a deque is thus similar to a vector, but the deque differs internally from the vector (memory is not contiguous), so that it is more efficient for some operations, and less efficient for others.
  - **`list`** does not provide random (sub-script) access
- the methods which <u>*must*</u> be implemented for a sequential container
  - Every sequential container must support **`insert()`** and **`erase()`**

`iterator insert(iterator pos, const item_Type& item)`

- Inserts before the position referenced by `pos`, returns an iterator referencing the newly inserted item

`iterator erase(iterator pos)`

- removes the item referenced by `pos`, and returns iterator to item following this one.

functions available for *some* sequential containers

`void push_back(const Item_type& item)`

`void pop_back()`

- inserts/removes an item as the last in the container
- is available for all 3 defined seq. containers

`void (push_front(const Item_type& item)`

`void pop_front()`

- available for the `list` and `deque`, but not the `vector`

`Item_Type& front()`

`Item_Type& back()`

- returns reference to first or last item in the seq. container
- is available for all 3 defined seq. containers

`Item_Type& at(int index)`

- Randomly accesses an item via an index. the value of the index is validated
- Can also use subscript access with index brackets [] , as for an array, to access the item, *faster*, *but the index is not validated*
- Only available for `vector` and `deque`, but not `list`

Code example

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

int main()
{
   vector<int> vcont;
   list<double> lcont;
   vector<int>::iterator vit;
   list<double>::iterator lit

   double nextinlist = 1.1;

   //initialise the containers
   for (int i = 0; i <4; i++)
   {
    vcont.push_back(i);
    lcont.push_back(nextinlist);
    nextinlist+=1.1;
   }

   cout << "here are the
      contents of the vector\n";
   vit = vcont.begin();
   while ( vit != vcont.end() )
      cout << *vit++ << endl;

   cout << "\nhere are the
        contents of the list\n";
   lit = lcont.begin();
   while ( lit != lcont.end() )
     cout << *lit++ << endl;

   cout << endl << endl;
   return 0;
}
```

STL Associative Container classes

- The **set** and the **map** are two associative containers in the STL
- Associative Containers store data in which each data item has an associated value known as its key
  - In simplest case, each data item is its own key (this is the case for a set)
  - As another example, the data item might be a `struct`, the key one of the fields of that `struct`
  - A map is a special case in which each element is an ordered pair consisting of a value and a key, where each key defines a unique value (or no value)
- Items are retrieved from the container on the basis of the key
  - Basically, associative containers work like a very simple database
- The C++ standard library uses a special kind of binary search tree, called a balanced binary search tree, to implement sets and maps
  - Could also be implemented using a hash table
  - It is proposed in the current revision of the C++ standard that a hash container class will be included in the STL

The set container class

value is also the key

- Stores elements without repetition
  - No duplicate elements
  - Every item is its own key
  - Items can be retrieved based on their value
- To work efficiently, the set elements are stored internally in sorted order
  - the method of ordering can be specified
  - if it is not, then the ordering is assumed to use the < relational operator

From Savitch

SAMPLE MEMBER FUNCTIONS

MEMBER FUNCTION (s IS A SET OBJECT)	MEANING
`s.insert(`*`Element`*`)`	Inserts a copy of *`Element`* in the set. If *`Element`* is already in the set, this has no effect.
`s.erase(`*`Element`*`)`	Removes *`Element`* from the set. If *`Element`* is not in the set, this has no effect.
`s.find(`*`Element`*`)`	Returns an iterator located at the copy of *`Element`* in the set. If *`Element`* is not in the set, `s.end( )` is returned. Whether the iterator is mutable or not is implementation dependent.
`s.erase(`*`Iterator`*`)`	Erases the element at the location of the *`Iterator`*.
`s.size( )`	Returns the number of elements in the set.
`s.empty( )`	Returns `true` if the set is empty; otherwise, returns `false`.
`s1 == s2`	Returns `true` if the sets contain the same elements; otherwise, returns `false`.

The `set` template class also has a default constructor, a copy constructor, and other specialized constructors not mentioned here. It also has a destructor that returns all storage for recycling, and a well-behaved assignment operator.

Sets: some sample code

```
set<char> vowels;
set<char>::iterator p;

vowels.insert('e');
vowels.insert('a');
vowels.insert('i');
vowels.insert('e'); // will be ignored since 'e' is
                   // already in the set
for (p = vowels.begin(); p != vowels.end(); p++)
  cout << *p << ' ' ; // iterator is like a pointer, de-
                      // reference it get the set element
cout << endl;

if ( (p = vowels.find('o'))  != vowels.end())
   cout << *p << " is in the set\n";
else
   cout << "there are " << vowels.size() << " items in the
  set and \'o\' is not one of them\n";
```

The map container class

- The STL class `pair<T1, T2>` has objects that are pairs of values, the first of type T1 and the second of type T2
  - The 1st element of the pair `p is p.first`, the 2nd element is `p.second`
  - `first` and `second` are public, so can be accessed directly as above (pair is actually implemented as a struct with some constructors included)

- The STL class `map<T1, T2>` is essentially a function that is given as a **set of ordered pairs** (The first value of the pair (the key) uniquely determines the second value)
- Example 1: a map where the key is a char, and each char is associated with at most one int value

```
map<char, int> m;
```

- Example 2: A map where the key is a string, and each key maps to at most one Employee object (for example the string might be the Employee id)

```
map<string, Employee> emps;
```

One way to insert into the map **emps**

```
Employee mary;
…
emps["0045"] = mary;
```

Another way to insert into the map **emps**

```
Employee mary;
…
pair<string, Employee> emppair(
                    "0045", mary);
emps.insert(emppair);
```

From Savitch

Sample Member Functions

MEMBER FUNCTION (m IS A MAP OBJECT)	MEANING
`m.insert(Element)`	Inserts *Element* in the map. *Element* is of type `pair<KeyType, T>`. Returns a value of type `pair<iterator, bool>`. If the insertion is successful, the second part of the returned pair is `true` and the iterator is located at the inserted element.
`m.erase(Target_Key)`	Removes the element with the key *Target_Key*.
`m.find(Target_Key)`	Returns an iterator located at the element with key value *Target_Key*. Returns `m.end( )` if there is no such element.
`m[Target_Key]`	Returns a reference to the object associated with the *Target_Key*. If the map does not already contain such an object then a default object of type T is inserted.
`m.size( )`	Returns the number of pairs in the map.
`m.empty( )`	Returns `true` if the map is empty; otherwise, returns `false`.
`m1 == m2`	Returns `true` if the maps contain the same pairs; otherwise, returns `false`.

The `map` template class also has a default constructor, a copy constructor, and other specialized constructors not mentioned here. It also has a destructor that returns all storage for recycling, and a well-behaved assignment operator.

Another example of a map

(the Types used in the declarations are shown in blue)

```
map<string, Electrical> prods;

Electrical gadget(“foo", 34.56);

pair<string, Electrical> prodpair("0045", gadget);
prods.insert(prodpair);

// 2 lines above can be replaced with
// prods["0045"] = gadget;

//or can use an ‘anonymous’ pair
//prods.insert(pair<string, Electrical> ("0045", gadget) );

map<string, Electrical>::iterator pm;
pm = prods.find("0045");
cout << pm->second.getPrice() << endl;
```

The 'same' map using polymorphism?

```
map<string, Product*> prods;  //the elements in the map are now pointers to
                                products
Electrical gadget("foo", 34.56);
// or could say Electrical *gadget = new Electrical(("foo", 34.56);
// then we wouldn't use & before gadget in the code following
// because it already is a pointer!
pair<string, Product*> prodpair("0045", &gadget);
prods.insert(prodpair);

// 2 lines above can be replaced with
// prods["0045"] = &gadget;

// or could use an anonymous pair, replace the 2 lines above with
// Prods.insert(pair<string, Product*>("0045", &gadget) )

map<string, Product*>::iterator pm;
pm = prods.find("0045");
cout << pm->second->getPrice() << endl;  // the get price method of Electrical
                                           // product used polymorphically
```

When to use which Container:
Some rules of thumb

- By default use a vector
  - Simplest internal structure
  - Provides random access
  - Data access convenient and flexible, and processing often fast enough
- If you insert/remove often at beginning and end of a sequence, use a deque
  - Also if it is important that the amount of memory shrinks when an element is removed
- If you insert/remove and move elements in the middle of a container often, consider a list
  - Has functions to move items from 1 container to another in constant time
  - But no random access means there may be performance overheads accessing items in the middle of the list
- If you often need to search for elements according to a certain criteria, use a set or a multiset that sorts elements according to that criteria.
  - uses a binary search internally
  - Multiset accepts duplicates
- To process key-value pairs use a map or multimap

(from Josuttis chap. 6, abridged)

Container Adapters `stack` and `queue`

- These are template classes that are implemented on top of other classes
  - Recall, in SDEV5 we saw how we might implement a stack using the vector class, or using a linked list, and last week we used an array for the template stack class we developed.
- When we use the stack template class, we can specify what type of container it should be built on
  - If we don't specify, a deque is used as the default for both stack and queue
  - We should not suggest a vector as the underlying class for a queue (why not??), but we could specify a list

Examples:

a queue of ints, using the default deque as the underlying container

```
queue<int>
```

a queue of ints, using a list as the underlying container

```
queue<int, list>
```

- The underlying container used does not change in any way the interface we can use for the stack and the queue

The stack adapter class

From Savitch

Sample Member Functions

Member function (s is a stack object)	Meaning
`s.size( )`	Returns the number of elements in the stack.
`s.empty( )`	Returns `true` if the stack is empty; otherwise, returns `false`.
`s.top( )`	Returns a mutable reference to the top member of the stack.
`s.push(`*Element*`)`	Inserts a copy of ***Element*** at the top of the stack.
`s.pop( )`	Removes the top element of the stack. Note that `pop` is a `void` function. It does not return the element removed.
`s1 == s2`	True if `s1.size( ) == s2.size( )` and each element of `s1` is equal to the corresponding element of `s2`; otherwise, returns `false`.

The queue adapter class

from Savitch

SAMPLE MEMBER FUNCTIONS

MEMBER FUNCTION (q IS A QUEUE OBJECT)	MEANING
`q.size( )`	Returns the number of elements in the queue.
`q.empty( )`	Returns `true` if the queue is empty; otherwise, returns `false`.
`q.front( )`	Returns a mutable reference to the front member of the queue.
`q.back( )`	Returns a mutable reference to the last member of the queue.
`q.push(`*`Element`*`)`	Adds *`Element`* to the back of the queue.
`q.pop( )`	Removes the front element of the queue. Note that `pop` is a `void` function. It does not return the element removed.
`q1 == q2`	True if `q1.size( ) == q2.size( )` and each element of `q1` is equal to the corresponding element of `q2`; otherwise, returns `false`.

STL algorithms

- Standard algorithms for processing the elements of a collection
  - stand-alone functions, not part of any container class, but have arguments which are *iterators* over a container
  - Will work with many container types, and with any type of data the container might hold
  - Process one or more *ranges* of elements

- Example 1: `sort()` being used with a *`vector`* container type holding data of type *`int`*.

```
vector<int> v;
sort(v.begin(), v.end() ) ;
```

- Example 2: `sort()` being used with a *`deque`* of *`double`* data

```
deque<double> d;
sort(d.begin(), d.end() ) ;
```

- `sort` is an algorithm that can only be used for random-access container types, (i.e vector and deque) so not with the `list` container type.
- Random access iterators are also provided for ordinary arrays (as a pointer to an element of the array) and for strings – (which are implemented internally as character arrays)

The sort algorithm - more

- By default, It sorts a container in descending order by using the operator <
  - The basic form of `sort()` is passed 2 iterators, which reference the first item in the container, and the end of the container

```
vector<int> v
sort(v.begin(), v.end() ) ;
```

  - `sort()` could be used to sort a portion of the vector only. The call below sorts the portion of the vector from 9th item to 19th item

```
sort(v.begin + 8, v.begin() + 19);
```

- `sort()` may optionally be passed a ‘binary predicate’ to replace the default < operator used for sorting
  - *binary* means it takes 2 arguments, *predicate* means it returns true or false
  - For example, to sort integers depending on their first digit, (so that 100 will come before 9) we must provide a function to use as the sorting criteria

```
bool digitWiseLessThan(int &i, int &j)
{
    // convert i and j to c-strings, and compare first chars
    // code goes here
}
…
```

and pass it as a third argument to the sort algorithm

```
sort(v.begin(), v.end(), digitWiseLessThan ) ;
```

Other algorithms in the STL

- All the algorithms are passed a range consisting of an iterator to the first element in the range, and to *one after* the final element in the range.
- One of the most useful is the `for_each()` algorithm, which is passed a range, and an operation which is applied to each item in the range.
- The operation is in the form of a user-defined function

```
void printElement(int element)
{
    cout << element;
}

void square (int &element)
{
    element = element * element;
}
```

```
vector<int> v;
…
for_each(v.begin(), v.end(), printElement);
for_each(v.begin(), v.end(), square);
```

Other algorithms in the STL (2)

- Counting algorithms
  - Count the no of elements of the given value within the range

```
count(iterator begin, iterator end, const T& value)
```

  - Count the number of elements for which a given operation returns true

```
count(iterator begin, iterator end, unaryPredicate op)
```

  - `op` is the name of a function which takes one argument of the collection item type, and returns true or false

```
bool isEven(int elem)
{
    return elem % 2 == 0;
}
```

```
vector<int> coll;
…
num = count(coll.begin(), coll.end(), 4)
cout << "no. of elements equal to 4 is " << num << endl;


num = count(coll.begin(), coll.end(), isEven)
cout << "no. of even elements is " << num << endl;
```

Other algorithms in the STL (3)

Minimum and Maximum

```
iterator min_element(iterator begin, iterator end)
iterator min_element(iterator begin, iterator end,
  CompFunc op)

iterator max_element(iterator begin, iterator end)
iterator max_element(iterator begin, iterator end,
  CompFunc op)
```

  - the compare function will be a binary predicate as in the sort() algorithm

- Other algorithms include those
  - to search a range for a given value
  - to copy one range to another
  - to swap elements in one range with another
  - We could go on …..