HO#14B
Supplementary notes
Gary Chan
Fall 2022

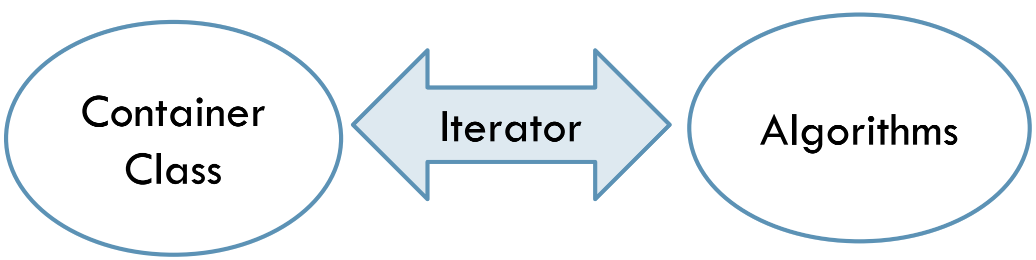
Standard Template Library (STL) Examples: Vector, List and Deque

N:5,9; D:18,22

STL (Standard Template Library)

Components:

- Containers:
  - Generic "off-the-shelf" class templates for storing collections of data
- Algorithms:
  - Generic "off-the-shelf" function templates for operating on containers
- Iterators:
  - Generalized "smart" pointers that allow algorithms to operate on almost any container



Containers in Standard Template Library

- Sequence containers
  - Represent linear data structures
  - Start from index/location 0
- Associative containers
  - Nonlinear containers
  - Store key/value pairs
- Container adapters
  - Implemented as constrained sequence containers
- “Near-containers” C-like pointer-based arrays
  - Exhibit capabilities similar to those of the sequence containers, but do not support all their capabilities
  - strings, bitsets and valarrays

Kind of Container	STL Containers
Sequential	vector, list, deque,
Associative	map, multimap, multiset, set
Adapters	priority_queue, queue, stack
Near-containers	bitset, valarray, string

The `vector` Container

- A type-independent pattern for an array class
  - Capacity can expand
  - Self contained
- Can be conceptualized as a powerful array
- C-style pointer-based arrays have great potential for errors and several shortcomings
  - C++ does not support continuous insertion of an elements into the array
  - Two arrays cannot be meaningfully compared with equality or relational operators (e.g., `a1 > a2`)
  - One array cannot be assigned to another using the assignment operators (e.g., `a1=a2`)

The `vector` Container

- Requires header file <vector>
- A data structure with contiguous memory locations
  - Efficient, direct access to any element via subscript operator
- Commonly used when data must be sorted and easily accessible via indices (subscripts)
- When additional memory is needed
  - Transparently allocates larger contiguous memory, copies elements and de-allocates old memory (behind user's back)
- Supports random-access *iterators*
- All STL algorithms can operate on vectors

The vector Container

- Declaration

```
template <typename T>
class vector
{   . . .   }
```

- Constructors

```
vector<int> v,             // empty vector
            v1(100),       // with capacity of 100 int
            v2(100, val),  // 100 copies of val
            v3(fptr,lptr); // copy to v3
                           // elements in memory
                           // locations from fptr to
                           // lptr (excluding lptr)
```

```
int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
vector<int> v1( &array[2], &array[4]);//gets 3 and 4
```

Vector Operations

- Information about a vector's contents
  - `v.size()        // current # of items`
  - `v.empty()`
  - `v.capacity()   // max. storage space(no less than v.size())`
  - `Etc.`
- Adding, removing, accessing elements
  - `v.push_back(X)   // push as back`
  - `v.pop_back()     // take away the back`
  - `v.front()        // peep the front`
  - `v.back()         // peep the back`
- Declaring different types of vectors:
  - `vector<int> iv;  // empty integer vector`

Vector Operations

- Assignment
  - v1 = v2
- Swapping
  - v1.swap(v2)
- Relational operators
  - == or != implies element by element equality or inequality
  - less than <, <=, >, >= behave like string comparison
- Accessing an element
  - With `[]`: E.g., v[0], v[1], etc.
  - With the member function at(i): E.g., v.at(0), v.at(1), etc.
- The member function `at()` has boundary checking:

```
vector<int> iv;  // empty vector of size 0

for( i = 0; i < 10; i++ ){
   cout << iv[i]; // segmentation fault (due to out-of-range access)
   cout << iv.at(i); // graceful termination with an exception msg:
   //terminate called after throwing an instance of 'std::out_of_range'
  }
```

Increasing Capacity of a Vector

- When vector `v` becomes full
  - Capacity is increased automatically when item is added
- Algorithm to increase capacity of `vector<T>`
  - Allocate new array to store vector's elements
  - Copy existing elements to new array
  - Destroy old array in `vector<T>`
  - Make new array the `vector<T>`'s storage array
- Allocate new array
  - Capacity doubles when more space is needed
    - 0 → 1 → 2 → 4 → 8 → 16, etc.
  - Can be wasteful for a large vector to double – use `resize()` to resize the vector, e.g.,

```
v.resize( 10 ); // the elements beyond the size will be
                // truncated/erased
```

2-D vector

- Accessing element as `v[i][j]`
- Creating a 100x1000 matrix. Method 1:

```
  int i, j;
  vector<vector<int> > v2D; // Note the space between > >

// creating a 1000x100 matrice
  for(i = 0; i < 1000; i++){
    v2D.push_back( vector<int> () ); // a row element
    for(j=0; j<100; j++)
      v2D[i].push_back(i+j);  // pushing column elements
  }
```

- Method 2:

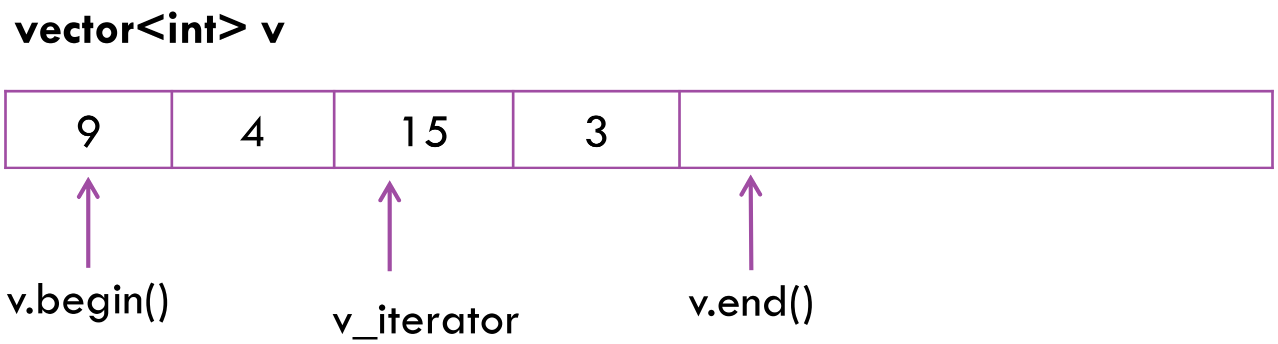
```
vector<vector<int> > v2d;
v2d.resize(1000);
for( i = 0; i < 1000; i++ )
  v2d[i].resize(100);
```

Iterators

- Note that a subscript operator is provided for vector, e.g., v[2]
  - BUT … this is not a *generic* way to access container elements
  - This is because some containers do NOT have `[]` connotations, and hence their `[]` operator is not overloaded (list, etc.)
- STL provides objects called iterators
  - ```
    vector<int>::iterator foo;
    vector<int>::const_iterator foo;
    vector<int>::reverse_iterator foo;
    vector<int>::const_reverse_iteractor foo;
    ```
  - can point at an element
  - can access the value within that element
  - can move from one element to another
- They are independent of any particular container … thus a generic mechanism as a uniform way to access elements
- A constant iterator is an iterator which you will not or cannot change the content it points to
    ```

Iterators

- Given a vector which has had values placed in the first 4 locations:



- `v.begin()` will return the iterator value for the first slot
- `v.end()` for the next empty slot
- `for( v_iterator = v.begin(); v_iteractor < v.end(); v_iterator++)…`

Iterators

- Each STL container declares an **`iterator`** type
  - can be used to define **`iterator`** objects
- To declare an **`iterator`** object, the identifier **`iterator`** must be preceded by
  - name of container, e.g., `vector<int>`
  - scope operator **`::`**
- Example:

```
vector<int>::iterator vecIter = v.begin()
vector<int>::const_iterator cvecIter = v.begin()
```

Iterators

- A pointer
- Basic operators that can be applied to iterators:
  - Increment operator ++
  - Decrement operator --
  - Dereferencing operator *
  - Assignment =
  - Addition, subtraction +, -, +=, -=
    `vecIter + n` returns iterator positioned **`n`** elements away
  - Subscript operator [ ]
    `vecIter[n]` returns reference to nth element from current position

Iterators vs. Subscript for Vector

Subscript:

```
ostream & operator<<(ostream & out, const
  vector<double> & v)
{
  for (int i = 0; i < v.size(); i++)
     out << v[i] << "  ";
  return out;
}
```

Iterators:

```
for (vector<double>::iterator it = v.begin();
  it != v.end(); it++)  // can also it < v.end()
       out << *it << "  ";
```

Iterator Functions

- Insert and erase elements anywhere in the vector with iterators is as inefficient as for arrays because shifting is required

Function Member	**Description**
v.begin()	Return an iterator positioned at v's first element
v.end()	Return an iterator positioned past v's last element
v.rbegin()	Return a reverse iterator positioned at v's last element
v.rend()	Return a reverse iterator positioned before v's first element
v.insert(iter, value)	Insert *value* into *v* at the location specified by *iter*
v.insert(iter, n, value)	Insert *n* copies of *value* into *v* at the location specified by *iter*
v.erase(iter)	Erase the value in *v* at the location specified by *iter*
v.erase(iter1, iter2)	Erase values in *v* from the location specified by *iter1* to that specified by *iter2* (*not including* *iter2*)
+ other insert /erase overload functions	

Iterator does not move with vector; Need to be re-located

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

int main(){

  vector<int> v(2,1);  // two 1s
  vector<int>::iterator vit, it;
  int a[] = {1, 2, 3, 4, 5};

  vit = v.begin();

  cout << "v address = " << v.begin() << "; content = " << v[0] << endl;
  cout << "vit address = " << vit << "; content = " << *vit << endl;

  v.insert( vit, a, a+4 );
  for( it = v.begin(); it < v.end(); it++ )
    cout << *it << " ";
  cout << endl;

  cout << "v address = " << v.begin() << "; content = " << v[0] << endl;
  cout << "vit address = " << vit << "; content = " << *vit << endl;
  return 1;
}
```

```
v address = 0x3fcc8; content = 1
vit address = 0x3fcc8; content = 1
1 2 3 4 1 1
v address = 0x3fd68; content = 1
vit address = 0x3fcc8; content = 261328
```

Template Function and Its Call

```
#include <iostream>
#include <vector>

using namespace std;

template< class A>
void printv( vector< A > a ){
  typename vector< A >::const_iterator it;
  // need typename here as A is a template

  for( it = a.begin(); it < a.end(); it++ )
    cout << *it << " ";
  cout << "\n";
}

template< class A, class B>
void print2( void ){
  A a = 3;
  B b = "hi there";

  cout << a << "\n";
  cout << b << "\n";

}
```

```
int main(){
  vector<int> vint;
  int i;

  for( i = 0; i < 10 ; i++ )
    vint.push_back( i );
  printv( vint );

  print2<int, char *>();

  return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
3
hi there
```

Some Common Vector Member Functions

- `vector::assign`
  - Assign values to vector
- `vector::at(i)`
  - ith element of the vector (start at 0)
- `vector::back()`
  - The reference of the last element
- `vector::begin()`
  - The first element for iterator
- `vector::capacity()`
  - Storage capacity of the vector
- `vector::clear()`
  - Clear the content
- `vector::empty()`
  - Whether the vector is empty
- `vector::end()`
  - The last element for iterator
- `vector::erase`
  - Remove elements
- `vector::front()`
  - Return the reference to the first element
- `vector::insert`
  - Insert elements into the vector
- `vector::operator[]`
  - foo[i] is the ith element of the vector
- `vector::pop_back()`
  - pop out the last element
- `vector::push_back( X )`
  - Push X as the last element
- `vector::rbegin()`
  - Reverse begin for iterator
- `vector::rend()`
  - Reverse end for iterator
- `vector::size()`
  - The number of elements
- `vector::swap( v2 )`
  - v1.swap(v2) swaps v1 and v2

Vectors vs. Arrays

Vectors

- Capacity can increase
- A self-contained object having function members to do tasks
- Is a class template

(Primitive) Arrays

- Fixed size, cannot be changed during execution
- Cannot "operate" on itself: must write functions to work on it
- Must "re-invent the wheel" for most actions for each array element type

STL's `list` Container

- Requires header file `<list>`
- Implemented internally as a doubly-linked list
  - Provides efficient insertion and deletion operations at any location
- Supports bidirectional iterators
  - Can be traversed forward and backward

Creating a vector of list

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

int main(){

  vector<list<int>> vl(10);
  int i,j;

  for( i = 0; i < 10; i++)
    for(j = 0; j<5; j++)
      vl[i].push_back(j); // create a vector of identical lists

  list<int>::reverse_iterator lit = vl[3].rbegin();
  for( ; lit != vl[3].rend(); lit++ )
    cout << *lit;    // print out 43210

  return 1;
}
```

Converting a reverse iterator to a normal iterator

- Note that the forward iterator is one position ahead of the reverse iterator
  - `it.begin()` points to the first element, while `rit.rend()` points to the position just before the first one
  - `it.end()` points to the next one after the last element, while `rit.rbegin()` points to the last one.
- It is often useful to convert iterator ←→ reverse_iterator
- Iterator → reverse_iterator: use the constructor of the reverse_iterator, e.g.,

```
list<int>::reverse_iterator rit( it );
```

- Reverse_iterator → iterator: use the `base()` member function in the reverse_iterator, e.g.,

```
it = rit.base();
```

- In all the cases, the iterator after the conversion is always one position higher than the reverse_iterator.

```
int main(){

  list<int> coll;

  // insert elements from 1 to 9
  for (int i=1; i<=9; ++i) {
    coll.push_back(i);
  }

  // find position of element with value 5
  list<int>::iterator pos;
  pos = find (coll.begin(), coll.end(),    // range
              3);                          // value

  // print value of the element
  cout << "pos:   " << *pos << endl;  // get 3

  // convert forward iterator to reverse iterator using its constructor
  list<int>::reverse_iterator rpos(pos);

  // print value of the element to which the reverse iterator refers
  cout << "rpos:  " << *rpos << endl; // get 2!

  // convert reverse iterator back to normal iterator
  list<int>::iterator rrpos;
  rrpos = rpos.base();

  // print value of the element to which the normal iterator refers
  cout << "rrpos: " << *rrpos << endl; // get 3


}
```

list Member Function `sort()`

- By default, arranges the elements in the list in ascending order
- Can take a binary predicate (i.e., boolean function) as argument to determine the sorting order
  - Called like a function pointer
  - E.g., `mylist.sort(compare_alg)`, where `compare_alg(x,y)` returns true if x is ordered *before* y.

```
// list::sort
#include <iostream>
#include <list>
using namespace std;

// reverse sort (sort in decreasing order)
bool reverse_sort (int left, int right)
{
  if (left > right)
    return true;   // first comes before second

  return false;
}

int main ()
{
  list<int> lst;
  list<int>::const_iterator it;

  lst.push_back( 2 );
  lst.push_back( 1 );
  lst.push_back( 7 );
  lst.push_back( 9 );
  lst.push_back( 6 );
  lst.push_back( 2 );
```

```
  lst.sort();

  cout << "lst sorted in increasing order:";
  for (it=lst.begin(); it!=lst.end(); ++it)
    cout << " " << *it;
  cout << endl;

  lst.sort(reverse_sort);

  cout << "lst sorted in decreasing order:";
  for (it=lst.begin(); it!=lst.end(); ++it)
    cout << " " << *it;
  cout << endl;

  return 0;
}
```

```
lst sorted in increasing order: 1 2 2 6 7 9
lst sorted in decreasing order: 9 7 6 2 2 1
```

list Member Function `unique()`

- Removes duplicate elements from the list
- List must first be *sorted*
- Can take an argument which specifies a binary predicate (i.e., boolean function) to determine whether two elements are *equal*
  - Called like a function pointer
  - Scanning the list from the head and compare the most recently retained element with a new one. Delete the new one if it is "the same" as the retained one.
  - Define an equal function, say `bool equal(x,y)` , which returns true if `x` is defined to be equal to `y`. In the context of list, `x` is the retained element right before `y`. Then a call of

  `unique( equal )` removes `y` if `equal` returns true, and not otherwise.

```
// list::unqiue
#include <iostream>
#include <list>
using namespace std;
// definition of equal
// if left is less than or equal to a factor 2 of right, they are the same
// left is always before right in the list and they are +ve integers
bool factor2 (int left, int right)
{
  cout << left << " " << right
       << endl;

  if (left *2 > right){
    cout << "true!\n"; // equal
    return true; // delete remove
  }
  return false;
}

int main (){
  list<int> lst;
  list<int>::const_iterator it;

  lst.push_back( 2 );
  lst.push_back( 1 );
  lst.push_back( 7 );
  lst.push_back( 9 );
  lst.push_back( 3 );
  lst.push_back( 2 );
```

```
  lst.sort();
  lst.unique();
  cout << "lst after unique call:";
  for (it=lst.begin(); it!=lst.end(); ++it)
    cout << " " << *it;
  cout << endl;

  lst.unique(factor2);
  cout << "lst after unique(factor2) call:";
  for (it=lst.begin(); it!=lst.end(); ++it)
    cout << " " << *it;
  cout << endl;

  return 0;
}
```

```
lst after unique call: 1 2 3 7 9
1 2
2 3
true!
2 7
7 9
true!
lst after unique(factor2) call: 1 2 7
```

Some list Member Functions

- `list::assign`
- `list::back()`
- `list::begin()`
- `list::clear()`
- `list::empty()`
- `list::end()`
- `list::erase()`
- `list::front()`
- `list::insert`
- `list::merge`
  - v1.merge(v2) merges the two *sorted* lists to form a new sorted list v1
- `list::operator=`
- `v1=v2`
- `list::pop_back()`
- `list::pop_front()`
- `list::push_back( X )`
- `list::push_front( X )`
- `list::rbegin()`
- `list::remove()`
- `list::remove_if( foo )`
  - Remove all elements for the function foo returning true
- `list::rend()`
- `list::reverse()`
  - Reverse the order of the list
- `list::size()`
- `list::sort( foo )`
  - Sort the element of the list
- `list::swap( list2 )`
  - Swap the two lists
- `list::unique( foo )`
  - Remove all the duplicates in a *sorted* list

STL's deque Container

- Requires header file <deque>
- As an ADT, a deque is a double-ended queue
  - Pronounced as "deck"
- It is a sequential container
  - Additional storage may be allocated at either end
  - Noncontiguous memory layout (dynamic allocation on the heap)
- Acts like a queue (or stack) on both ends
- It is an ordered collection of data items
- Items usually are added or removed at the ends
- Provides many of the benefits of vector and list in one container
  - Reasonably efficient indexed access using subscripting
  - Reasonably efficient insertion and deletion operations at front and back

Deque Operations

- Construct a deque (usually empty)

```
 deque<double> dq;            // empty deque
deque<int> first (3,100); // three ints with a value of 100
deque<int> second (5,200); // five ints with a value of 200
```

- *Empty*: return true if the deque is empty
- Add
  - *push_front*: add an element at the front of the deque
  - *push_back*: add an element at the back of the deque
- Retreive
  - *front*: peep the element at the front of the deque. Can be lvalue.
  - *back*: peep the element at the back of the deque. Can be lvalue.
- Remove
  - *pop_front*: remove the element at the front of the deque
  - *pop_back*: remove the element at the back of the deque

Deque Class Template

- Has the same operations as `vector<T>` except some member functions (there is no `capacity()` and no `reserve()`)
- Has two new operations:
  - `d.push_front(value);` - Push copy of value at `front` of `d`
  - `d.pop_front();` - Remove the element at the front of `d`
- Like STL's `vector`, it has
  - [ ] subscript operator
  - insert and delete at arbitrary points in the list (*insert* and *erase*)
- Insertion and deletion in the middle of the deque are not guaranteed to be efficient

Some deque Member Functions

- deque::assign
- deque::at(i)
  - The ith element (starting from 0)
- deque::back()
  - Return the last element
- deque::begin()
- deque::clear()
  - Delete the whole deque
- deque::empty()
- deque::end()
- deque::erase
  - Remove either a single element (erase(i)) or a range of element (erase(i,j))
- deque::front()
  - Return the first element
- deque::insert
- deque::operator=
  - for d1 = d2;
- deque::operator[]
  - for d[i]
- deque::pop_back()
  - delete the last element
- deque::pop_front()
  - delete the first element
- deque::push_back(X)
- deque::push_front(X)
- deque::rbegin()
- deque::rend()
- deque::size()
  - Return the number of elements
- deque::swap(dq2)

Efficiency Consideration and Performance Comparison

- Which STL to use depends on the access pattern of your applications
- Their insertion and deletion are all through iterators
- Vector (Implemented as a contiguous array)
  - `insert` and `erase` in the middle of vector are not efficient (involves moving of elements and may lead to memory re-allocation and copying)
  - Insertion and deletion at the *end* are fast (e.g., `push_back` operation)
  - Random *access* is fast (array indexing, e.g., `front`, `back` and `[]`)
- List (Implemented as doubly linked list)
  - `insert` and `erase` in the middle of the list given an iterator are efficient (involving only a few pointer movements)
  - Insertion and deletion at *both ends* are fast (`push_front` and `push_back` operations)
  - Random access is slow (has to use iterator to traverse the list to the get the element)
- Deque
  - Implementation involves a combination of pointers and array (blocks of contiguous memory chunks), probably in the form of a linked list with array in each node
  - `insert` and `erase` in the middle are reasonably fast
  - Insertion and deletion at both ends are reasonably fast (`push_front` and `push_back` operations)
  - Random access is reasonably fast (using `[]`)
  - Intermediate performance between vector and list

Q&A