University of KwaZulu-Natal
Pietermaritzburg Campus
Examinations

Advanced Programming

COMP315P1

Time and Date:
14h00, Tuesday, 31^{st} May, 2011.

Examiners:
Hugh Murrell and Conrad Mueller

time limit: 3 hours **max marks: 100**

This paper consists of 15 pages, excluding this one.
Make sure that no pages are missing.
Candidates may attempt all questions.

Write your answers in the GREEN book
Use the TURQUOISE book for rough work

Question 1.1

Give sample C++ declarations for the following: [4]

a) a pointer to an int

b) an array of 10 ints

c) a pointer to an array of 10 ints

d) an array of 10 pointers to ints

```
int* p1;
int p2[10];
int* p3 = new int[10];
int** p4 = new int*[10];
```

Question 1.2

The problem with *arrays* is that they do not know their own size. Write a `C++` function, `reverseArray(int* a, int n)` that will reverse the contents of an array containing `n` ints *in place*.

For example, after executing:

```
int arr[] = {1,2,3,4,5,6}
reverseArray(arr, 6);
```

`arr` should contain {`6,5,4,3,2,1`}. [5]

```
void reverseArray(int* a, int n){
    int i=0;
    int j=n-1;
    while(i<j){
        int temp=a[j];
        a[j]=a[i];
        a[i]=temp;
        ++i;
        --j;
    };
    return;
}
```

Question 2.1

Assume that you are coding in `C++` before the advent of the Standard Template Library. You decide to design a class called `vector` for storing arrays of elements of type `double`.

Your class must provide a constructor that creates space for a user specified number of doubles on the heap and initializes each of them to zero. You must be able to index a vector as you would an array and a vector must be able to grow at run time.

Study the header for `vector` given below. Some of the members of `vector` have completed code. Some members are incomplete.

```
class vector {

    int sz;     // the number of elements (the size)
    double* elem;   // pointer to the first element
    int space;      // size + free_space

    public:
        vector();   // constructor
        vector(int s);  // constructor
        vector(const vector&) ; // copy constructor:
        ~vector(){ delete[ ] elem; };  // destructor
        int size() const { return sz; };    // get the current size
        double get(int n) { return elem[n]; };  // access: read
        void set(int n, double v) { elem[n]=v; };   // access: write
        void reserve(int newalloc); // get more space
        int capacity() const { return space; }  // get available space
        void resize(int newsize);   // grow (or shrink)
        vector& operator=(const vector& a); // copy assignment
        double& operator[ ](int n) { return elem[n]; };  // access:
        void push_back(double d);   // add element

};
```

a) give code for the `reserve` member. [5]

```
void vector::reserve(int newalloc)
// make space for newalloc elements

{

    if (newalloc<=space) return;
    // never decrease allocation

    double* p = new double[newalloc];
    // allocate new space

    for (int i=0; i<sz; ++i) p[i]=elem[i];
    // copy old elements

    delete[ ] elem;
    // deallocate old space

    elem = p;

    space = newalloc;

}
```

b) give code for the `resize` member. [5]

```
void vector::resize(int newsize)

    // make the vector have newsize elements
    // initialize each new element 0.0

{
    reserve(newsize);
    // make sure we have sufficient space

    for(int i = sz; i<newsize; ++i) elem[i] = 0;
    // initialize new elements

    sz = newsize;

}
```

c) give code for the `pushback` member. [5]

```
void vector::push_back(double d)

    // increase vector size by one
    // initialize the new element with d

{

    // no space: grab some
    if (sz==0)
        reserve(8);

    // no more free space: get more space
    else if (sz==space)
        reserve(2*space);

    // add d at end
    elem[sz] = d;

    // and increase the size (sz is the number of elements)
    ++sz;

}
```

d) Show how to generalize your `vector` of `double` class by writing down the `C++` header file for a `template` version of `vector`. . [5]

```
template<class T> class vector {

    int sz;     // the size
    T* elem;        // pointer to the elements
    int space;      // size+free_space

public:
    // default constructor
    vector() : sz(0), elem(0), space(0);
    // explicit constructor
    explicit vector(int s)
        :sz(s), elem(new T[s]), space(s) {}

    // copy constructor
    vector(const vector&);
    // copy assignment
    vector& operator=(const vector&);

    // destructor
    ~vector() { delete[ ] elem; }

    // get and set
    T get(int n){ return elem[n]; }
    void set(int n, T v) { elem[n]=v; }
    // access: return reference
    T& operator[ ] (int n) { return elem[n]; }
    // the current size
    int size() const { return sz; }
    // add new element
    void push_back(T d);

};
```

Question 3.1

Write a paragraph explaining the significance of the STL *iterator*. In your expanation make reference to the C++ code snippet below:

```
string str ("Test string");
string::iterator itr;
for ( itr=str.begin() ; itr < str.end(); itr++ )
    cout << *itr;
```

[6]

At a high level, an iterator is like a cursor in a text editor. An iterator has a well-defined position inside a container, and can move from one element to the next. Also like a cursor, an iterator can be used to read or write a range of data one element at a time.

Every STL container class exports a member function begin() which yields an iterator pointing to the first element of that container.

By initializing the iterator to str.begin(), we indicate to the C++ compiler that the itr iterator will be traversing elements of the string container str.

Each STL container exports a special member function called end() that returns an iterator to the element one past the end of the container. In this case one char PAST the end of the string str.

The entity *itr is known as an iterator dereference and means the element being iterated over by itr. As itr traverses the elements of the container, it will proceed from one element to the next in sequence until all of the elements have been visited. At each step, the element being iterated over can be yielded by prepending a star to the name of the iterator. In the above context, we dereference the iterator to yield the current char of the string str.

Question 3.2

A function designed to sell half of any stock you feel you own too much of, might be implemented in this way:

```
void sellStocks(map<string, int>& stocks, int threshold) {
    map<string, int>::iterator curr = stocks.begin();
    while (curr != stocks.end()) {
        if (curr->second >= threshold) curr->second /= 2;
        ++curr;
    }
}
```

A function that counts the total number of shares you own would need to traverse the entirety of the map in a similar way:

```
int countStocks(const map<string, int>& stocks) {
    int stockCount = 0;
    map<string, int>::const_iterator curr = stocks.begin();
    while (curr != stocks.end()) {
        stockCount += curr->second; ++curr;
    }
    return stockCount;
}
```

Explain why the second function only requires `const` access to the map. [5]

In the first snippet we change the values stored in the map whereas in the second snippet we only need read (or const) access to the map because we do not alter the map elements.

Question 4.1

A string is said to be a *doubloon* if every letter that appears in the string appears exactly twice. For example, the following strings are all doubloons:

`abba, anna, appall, appearer, appeases, arraigning, beriberi, bilabial, boob, caucasus, coco, dada, deed, emmett, hannah, horseshoer, intestines, isis, mama, mimi, murmur, noon, otto, papa, peep, reappear, redder, sees, shanghaiings, toto`

Write a `C++` function called `isDoubloon` that returns `true` if the given string is a doubloon and `false` otherwise. [10]

```
bool isDoubloon(string s){
    map<char,int> charCount;
    for(int i=0; i<s.size(); ++i){
        ++charCount[s[i]];
    };
    for(map<char,int>::iterator itr=charCount.begin(); itr != charCount.e
        if(itr->second != 2) return false;
    };
    return true;
}
```

Question 4.2

Shown below are the first 7 rows of *Pascal's triangle* in which each element (apart from the 1's) is obtained by summing two elements from the row above.

```
1
1     1
1     2     1
1     3     3     1
1     4     6     4     1
1     5     10    10    5     1
1     6     15    20    15    6     1
```

Write a `C++` function, `pascalPrint(int n)`, that given an integer `n > 0` will print the first `n` rows of Pascal's triangle. [10]

```
vector<int> nextRow(vector<int> thisRow){
    vector<int> newRow;
    if (thisRow.size()==0) {
        newRow.push_back(1);
        return newRow;
    };
    newRow.push_back(thisRow[0]);
    for(int i=1; i<thisRow.size(); ++i){
        newRow.push_back(thisRow[i-1]+thisRow[i]);
    };
    newRow.push_back(1);
    return newRow;
}
void pascalPrint(int n){
    vector<int> row;
    while(row.size()<n){
        row = nextRow(row);
        for(int i=0; i<row.size(); ++i){
            cout << row[i] << "\t";
        }
        cout << endl;
    }
}
```

Question 4.3

Given two sets, A and B, the `unMatched` elements are those that occur in A but not in B joined with those that occur in B but not in A.

a) Draw a **Venn diagram** illustrating the `unMatched` elements in two arbitrary sets A and B. [2]

b) Make use of the STL algorithms to construct a `C++ function` that, given two `set<int>` containers, constructs and returns a `set<int>` container holding the `unMatched` elements. [8]

```
// 2 marks for:
// Venn diagram showing unM(A,B) = Union(A,B) - IntSect(A,B)

set<int> unMatched(set<int> a, set<int> b){
    set<int> absd;
    set_symmetric_difference(a.begin(), a.end(),b.begin(),b.end(),
        inserter(absd, absd.begin()));
    // btw: this wont work;
    // set_symmetric_difference(a.begin(), a.end(),b.begin(),b.end(),
    //      absd.begin());
    return absd;
}
```

Question 5.1

Study the following `C++ mystery` function and describe what it is supposed to do. In particular explain the use of STL containers, iterators and algorithms in the design of `mystery`. [10]

```
bool isNotAlphaOrSpace(char ch) {
    return !isalpha(ch) && !isspace(ch);
}

bool mystery(string input) {
    input.erase(remove_if(input.begin(), input.end(), isNotAlphaOrSpace),
        input.end());
    transform(input.begin(),input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    tokens.insert(tokens.begin(), istream_iterator<string>(tokenizer),
        istream_iterator<string>());

    return equal(tokens.begin(), tokens.begin() + tokens.size() / 2,
        tokens.rbegin());
}
```

```
Checks whether or not a string is a word palindrome
eg: "five four five" is one.
```

Question 5.2

Use the STL to construct a `C++` function, `reverseWords`, that takes in a string of words and returns a string with the words in reverse order. [10]

For example:

```
cout << reverseWords("The cat sat on the mat") << endl;
```

should print:

```
mat the on sat cat The
```

```
string reverseWords(string input) {
    stringstream tokenizer(input);
    vector<string> tokens;
    tokens.insert(tokens.begin(), istream_iterator<string>(tokenizer),
        istream_iterator<string>());
    string output;
    while(tokens.size()>0){
        output += tokens.back()+" ";
        tokens.pop_back();
    };
    return output;
}
```

Question 5.3

You have been requested to design an offline `spellChecker` that implements the following:

The dictionary is stored in a file.

The text to check is stored in a file.

Each word of the text should be checked against the dictionary.

Every word in the text not found in the dictionary should be displayed.

You decide to use the STL to accomplish this task. Describe with explanation those parts of the STL suited to the implementation of an offline `spellChecker`. [10]

```
        // read all words in dictionary and store in:
        set<string> dWords;
        string word;
        string inFileName = dictionaryFilePath;
        ifstream ifs(inFileName.c_str());
(2)     while (ifs >> word){
            word=toLowerCase(word);
            dWords.insert(word);
        };
        ifs.close();

(1)     // do the same for words in text and store in:
        set<string> tWords;

        // compute "difference"
        set<string> badWords;
(4)     set_difference(tWords.begin(), tWords.end(),
            dWords.begin(), dWords.end(),
            inserter(badWords, badWords.begin()))

        // copy to output
(2)     copy(badWords.begin(), badWords.end(),
            ostream_iterator<string>(cout, "\n"));

(1)     // note that use of set ensures minimal comparisons
        // and that final output is ordered
```

vector API

Constructor: `vector<T> ()`	`vector<int> myVector;` Constructs an empty vector.
Constructor: `vector<T> (size_type size)`	`vector<int> myVector(10);` Constructs a vector of the specified size where all elements use their default values (for integral types, this is zero).
Constructor: `vector<T> (size_type size, const T& default)`	`vector<string> myVector(5, "blank");` Constructs a vector of the specified size where each element is equal to the specified default value.
`size_type size() const;`	`for(int i = 0; i < myVector.size(); ++i) { ... }` Returns the number of elements in the vector.
`bool empty() const;`	`while(!myVector.empty()) { ... }` Returns whether the vector is empty.
`void clear();`	`myVector.clear();` Erases all the elements in the vector and sets the size to zero.

`T& operator [] (size_type position);`
`const T& operator [] (size_type position) const;`
 	
`T& at(size_type position);`
`const T& at(size_type position) const;`
	`myVector[0] = 100;`
	`int x = myVector[0];`
	`myVector.at(0) = 100;`
	`int x = myVector.at(0);`
	
	Returns a reference to the element at the specified position. The bracket notation [] does not do any bounds checking and has undefined behavior past the end of the data. The `at` member function will throw an exception if you try to access data beyond the end. We will cover exception handling in a later chapter.

`iterator erase(iterator position);`
`iterator erase(iterator start, iterator end);`
	`myVector.erase(myVector.begin());`
	`myVector.erase(startItr, endItr);`
	
	The first version erases the element at the position pointed to by `position`. The second version erases all elements in the range [`startItr`, `endItr`). Note that this does **not** erase the element pointed to by `endItr`. All iterators after the remove point are invalidated. If using this member function on a `deque` (see below), all iterators are invalidated.

`void resize(size_type newSize);`
`void resize(size_type newSize, T fill);`
	`myVector.resize(10);`
	`myVector.resize(10, "default");`
	
	Resizes the vector so that it's guaranteed to be the specified size. In the second version, the `vector` elements are initialized to the value specified by the second parameter. Elements are added to and removed from the end of the vector, so you can't use `resize` to add elements to or remove elements from the start of the vector.

`void push_back();`	`myVector.push_back(100);` Appends an element to the `vector`.

`T& back();`
`const T& back() const;`
	`myVector.back() = 5;`
	`int lastElem = myVector.back();`
	
	Returns a reference to the last element in the `vector`.

`T& front();`
`const T& front() const;`
	`myVector.front() = 0;`
	`int firstElem = myVector.front();`
	
	Returns a reference to the first element in the vector.

`void pop_back();`	`myVector.pop_back();` Removes the last element from the vector.
`iterator begin();` `const_iterator begin() const;`	`vector<int>::iterator itr = myVector.begin();` Returns an iterator that points to the first element in the vector.
`iterator end();` `const_iterator end() const;`	`while(itr != myVector.end());` Returns an iterator to the element *after* the last. The iterator returned by `end` does not point to an element in the vector.

`iterator insert(iterator position,`
`const T& value);`
`void insert(iterator start,`
`size_type numCopies,`
`const T& value);`
	`myVector.insert(myVector.begin() + 4, "Hello");`
	`myVector.insert(myVector.begin(), 2, "Yo!");`
	
	The first version inserts the specified value into the vector, and the second inserts `numCopies` copies of the value into the vector. Both calls invalidate all outstanding iterators for the vector.

map API

Constructor: `map<K, V>()`	`map<int, string> myMap;` Constructs an empty `map`.
Constructor: `map<K, V>(const map<K, V>& other)`	`map<int, string> myOtherMap = myMap;` Constructs a `map` that's a copy of another `map`.
Constructor: `map<K, V>(InputIterator start, InputIterator stop)`	`map<string, int> myMap(myVec.begin(), myVec.end());` Constructs a `map` containing copies of the elements in the range `[start, stop)`. Any duplicates are discarded, and the elements are sorted. Note that this function accepts iterators from any source, but they must be iterators over pairs of keys and values.
`size_type size() const`	`int numEntries = myMap.size();` Returns the number of elements contained in the `map`.
`bool empty() const`	`if(myMap.empty()) { ... }` Returns whether the `map` is empty.
`void clear()`	`myMap.clear();` Removes all elements from the `map`.
`iterator begin()` `const_iterator begin() const`	`map<int>::iterator itr = myMap.begin();` Returns an iterator to the start of the `map`. Remember that iterators iterate over pairs of keys and values.
`iterator end()` `const_iterator end()`	`while(itr != myMap.end()) { ... }` Returns an iterator to the element one past the end of the final element of the `map`.

`pair<iterator, bool>`
`insert(const pair<const K, V>& value)`
`void insert(InputIterator begin,`
`InputIterator end)`
	`myMap.insert(make_pair("STL", 137));`
	`myMap.insert(myVec.begin(), myVec.end());`
	
	The first version inserts the specified key/value pair into the `map`. The return type is a `pair` containing an iterator to the element and a `bool` indicating whether the element was inserted successfully (`true`) or if it already existed (`false`). The second version inserts the specified range of elements into the `map`, ignoring duplicates.

`V& operator[] (const K& key)`	`myMap["STL"] = "is awesome";` Returns the value associated with the specified key, if it exists. If not, a new key/value pair will be created and the value initialized to zero (if it is a primitive type) or the default value (for non-primitive types).

`iterator find(const K& element)` `const_iterator find(const K& element) const`	`if(myMap.find(0) != myMap.end()) { ... }` Returns an iterator to the key/value pair having the specified key if it exists, and `end` otherwise.
`size_type count(const K& item) const`	`if(myMap.count(0)) { ... }` Returns 1 if some key/value pair in the `map` has specified key and 0 otherwise.

`size_type erase(const K& element)`
`void erase(iterator itr);`
`void erase(iterator start,`
`iterator stop);`
	`if(myMap .erase(0)) {...}`
	`myMap.erase(myMap.begin());`
	`myMap.erase(myMap.begin(), myMap.end());`
	
	Removes a key/value pair from the `map`. In the first version, the key/value pair having the specified key is removed if found, and the function returns 1 if a pair was removed and 0 otherwise. The second version removes the element pointed to by `itr`. The final version erases elements in the range [`start`, `stop`).

`iterator lower_bound(const K& value)`	`itr = myMap.lower_bound(5);` Returns an iterator to the first key/value pair whose key is greater than or equal to the specified value. This function is useful for obtaining iterators to a range of elements, especially in conjunction with `upper_bound`.
`iterator upper_bound(const K& value)`	`itr = myMap.upper_bound(100);` Returns an iterator to the first key/value pair whose key is greater than the specified value. Because this element must be strictly greater than the specified value, you can iterate over a range until the iterator is equal to `upper_bound` to obtain all elements less than or equal to the parameter.

STL algorithms

`Type accumulate(InputItr start, InputItr stop, Type value)`	Returns the sum of the elements in the range [`start`, `stop`) plus the value of `value`.
`bool binary_search(RandomItr start, RandomItr stop, const Type& value)`	Performs binary search on the sorted range specified by [`start`, `stop`) and returns whether it finds the element `value`. If the elements are sorted using a special comparison function, you must specify the function as the final parameter.
`OutItr copy(InputItr start, InputItr stop, OutItr outputStart)`	Copies the elements in the range [`start`, `stop`) into the output range starting at `outputStart`. `copy` returns an iterator to one past the end of the range written to.
`size_t count(InputItr start, InputItr end, const Type& value)`	Returns the number of elements in the range [`start`, `stop`) equal to `value`.
`size_t count_if(InputItr start, InputItr end, PredicateFunction fn)`	Returns the number of elements in the range [`start`, `stop`) for which `fn` returns true. Useful for determining how many elements have a certain property.
`bool equal(InputItr start1, InputItr stop1, InputItr start2)`	Returns whether elements contained in the range defined by [`start1`, `stop1`) and the range beginning with start2 are `equal`. If you have a special comparison function to compare two elements, you can specify it as the final parameter.
`pair<RandomItr, RandomItr> equal_range(RandomItr start, RandomItr stop, const Type& value)`	Returns two iterators as a `pair` that defines the sub-range of elements in the sorted range [`start`, `stop`) that are equal to `value`. In other words, every element in the range defined by the returned iterators is equal to `value`. You can specify a special comparison function as a final parameter.
`void fill(ForwardItr start, ForwardItr stop, const Type& value)`	Sets every element in the range [`start`, `stop`) to `value`.
`void fill_n(ForwardItr start, size_t num, const Type& value)`	Sets the first `num` elements, starting at `start`, to `value`.
`InputItr find(InputItr start, InputItr stop, const Type& value)`	Returns an iterator to the first element in [`start`, `stop`) that is equal to `value`, or `stop` if the value isn't found. The range doesn't need to be sorted.

`InputItr find_if(InputItr start, InputItr stop, PredicateFunc fn)`	Returns an iterator to the first element in [`start`, `stop`) for which `fn` is true, or `stop` otherwise.
`Function for_each(InputItr start, InputItr stop, Function fn)`	Calls the function `fn` on each element in the range [`start`, `stop`).
`void generate(ForwardItr start, ForwardItr stop, Generator fn);`	Calls the zero-parameter function `fn` once for each element in the range [`start`, `stop`), storing the return values in the range.
`void generate_n(OutputItr start, size_t n, Generator fn);`	Calls the zero-parameter function `fn` `n` times, storing the results in the range beginning with `start`.
`bool includes(InputItr start1, InputItr stop1, InputItr start2, InputItr stop2)`	Returns whether every element in the sorted range [`start2`, `stop2`) is also in [`start1`, `stop1`). If you need to use a special comparison function, you can specify it as the final parameter.
`Type inner_product(InputItr start1, InputItr stop1, InputItr start2, Type initialValue)`	Computes the inner product of the values in the range [start1, stop1) and [start2, start2 + (stop1 – start1)). The inner product is the value $\sum_{i=1}^{n} a_i b_i + initialValue$, where a_i and b_i denote the ith elements of the first and second range.
`bool lexicographical_compare(InputItr s1, InputItr s2, InputItr t1, InputItr t2)`	Returns whether the range of elements defined by [`s1`, `s2`) is lexicographically less than [`t1`, `t2`); that is, if the first range precedes the second in a "dictionary ordering."
`InputItr lower_bound(InputItr start, InputItr stop, const Type& elem)`	Returns an iterator to the first element greater than or equal to the element `elem` in the sorted range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.
`InputItr max_element(InputItr start, InputItr stop)`	Returns an iterator to the largest value in the range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.
`InputItr min_element(InputItr start, InputItr stop)`	Returns an iterator to the smallest value in the range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.

`void random_shuffle(RandomItr start, RandomItr stop)`	Randomly reorders the elements in the range [`start`, `stop`).
`ForwardItr remove(ForwardItr start, ForwardItr stop, const Type& value)`	Removes all elements in the range [`start`, `stop`) that are equal to `value`. This function will **not** remove elements from a container. To shrink the container, use the container's `erase` function to erase all values in the range [`retValue`, `end()`), where `retValue` is the return value of `remove`.
`ForwardItr remove_if(ForwardItr start, ForwardItr stop, PredicateFunc fn)`	Removes all elements in the range [`start`, `stop`) for which `fn` returns true. See `remove` for information about how to actually remove elements from the container.
`void replace(ForwardItr start, ForwardItr stop, const Type& toReplace, const Type& replaceWith)`	Replaces all values in the range [`start`, `stop`) that are equal to `toReplace` with `replaceWith`.
`void replace_if(ForwardItr start, ForwardItr stop, PredicateFunction fn, const Type& with)`	Replaces all elements in the range [`start`, `stop`) for which `fn` returns true with the value `with`.
`ForwardItr rotate(ForwardItr start, ForwardItr middle, ForwardItr stop)`	Rotates the elements of the container such that the sequence [`middle`, `stop`) is at the front and the range [`start`, `middle`) goes from the new middle to the end. `rotate` returns an iterator to the new position of `start`.
`ForwardItr search(ForwardItr start1, ForwardItr stop1, ForwardItr start2, ForwardItr stop2)`	Returns whether the sequence [`start2`, `stop2`) is a subsequence of the range [`start1`, `stop1`). To compare elements by a special comparison function, specify it as a final parameter.

`InputItr set_difference(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in the sorted range [`start1`, `stop1`) but not in the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`InputItr set_intersection(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in both the sorted range [`start1`, `stop1`) and the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`InputItr set_union(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in either the sorted range [`start1`, `stop1`) or in the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`InputItr set_symmetric_difference(`
`  InputItr start1,`
`  InputItr stop1,`
`  InputItr start2,`
`  InputItr stop2,`
`  OutItr dest)`
	Stores all elements that are in the sorted range [`start1`, `stop1`) or in the sorted range [`start2`, `stop2`), but not both, in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.

`void swap(Value& one, Value& two)`	Swaps the values of `one` and `two`.

`ForwardItr`
`swap_ranges(ForwardItr start1,`
`  ForwardItr stop1,`
`  ForwardItr start2)`
	Swaps each element in the range [`start1`, `stop1`) with the correspond elements in the range starting with `start2`.

`OutputItr transform(InputItr start,`
`  InputItr stop,`
`  OutputItr dest,`
`  Function fn)`
	Applies the function `fn` to all of the elements in the range [`start`, `stop`) and stores the result in the range beginning with `dest`. The return value is an iterator one past the end of the last value written.

`RandomItr`
`  upper_bound(RandomItr start,`
`  RandomItr stop,`
`  const Type& val)`
	Returns an iterator to the first element in the sorted range [`start`, `stop`) that is strictly greater than the value `val`. If you need to specify a special comparison function, you can do so as the final parameter.