Power up C++ with the Standard Template Library: Part I

By DmitryKorolev
TopCoder Member

Perhaps you are already using C++ as your main programming language to solve TopCoder problems. This means that you have already used STL in a simple way, because arrays and strings are passed to your function as STL objects. You may have noticed, though, that many coders manage to write their code much more quickly and concisely than you.

Or perhaps you are not a C++ programmer, but want to become one because of the great functionality of this language and its libraries (and, maybe, because of the very short solutions you've read in TopCoder practice rooms and competitions).

Regardless of where you're coming from, this article can help. In it, we will review some of the powerful features of the Standard Template Library (STL) – a great tool that, sometimes, can save you a lot of time in an algorithm competition.

The simplest way to get familiar with STL is to begin from its containers.

Containers

Any time you need to operate with many elements you require some kind of container. In native C (not C++) there was only one type of container: the array.

The problem is not that arrays are limited (though, for example, it's impossible to determine the size of array at runtime). Instead, the main problem is that many problems require a container with greater functionality.

For example, we may need one or more of the following operations:

- Add some string to a container.
- Remove a string from a container.
- Determine whether a string is present in the container.
- Return a number of distinct elements in a container.
- Iterate through a container and get a list of added strings in some order.

Of course, one can implement this functionality in an ordinal array. But the trivial implementation would be very inefficient. You can create the tree- of hash- structure to solve it in a faster way, but think a bit: does the implementation of such a container depend on elements we are going to store? Do we have to re-implement the module to make it functional, for example, for points on a plane but not strings?

If not, we can develop the interface for such a container once, and then use everywhere for data of any type. That, in short, is the idea of STL containers.

Before we begin

When the program is using STL, it should #include the appropriate standard headers. For most containers the title of standard header matches the name of the container, and no extension is required. For example, if you are going to use stack, just add the following line at the beginning of your program:

```
#include <stack>
```

Container types (and algorithms, functors and all STL as well) are defined not in global namespace, but in special namespace called "std." Add the following line after your includes and before the code begin:

```
using namespace std;
```

Another important thing to remember is that the type of a container is the template parameter. Template parameters are specified with the ‘<’/’>’ "brackets" in code. For example:

```
vector<int> N;
```

When making nested constructions, make sure that the "brackets" are not directly following one another – leave a blank between them.

```
vector< vector<int> > CorrectDefinition;

vector<vector<int>> WrongDefinition; // Wrong: compiler may be confused by 'operator >>'
```

Vector

The simplest STL container is vector. Vector is just an array with extended functionality. By the way, vector is the only container that is backward-compatible to native C code – this means that vector actually IS the array, but with some additional features.

```
 vector<int> v(10);

 for(int i = 0; i < 10; i++) {

      v[i] = (i+1)*(i+1);

 }

 for(int i = 9; i > 0; i--) {

      v[i] -= v[i-1];

 }
```

Actually, when you type

```
 vector<int> v;
```

the empty vector is created. Be careful with constructions like this:

```
 vector<int> v[10];
```

Here we declare 'v' as an array of 10 vector<int>’s, which are initially empty. In most cases, this is not that we want. Use parentheses instead of brackets here. The most frequently used feature of vector is that it can report its size.

```
 int elements_count = v.size();
```

Two remarks: first, size() is unsigned, which may sometimes cause problems. Accordingly, I usually define macros, something like sz(C) that returns size of C as ordinal signed int. Second, it’s not a good practice to compare v.size() to zero if you want to know whether the container is empty. You're better off using empty() function:

```
 bool is_nonempty_notgood = (v.size() >= 0); // Try to avoid this

 bool is_nonempty_ok = !v.empty();
```

This is because not all the containers can report their size in O(1), and you definitely should not require counting all elements in a double-linked list just to ensure that it contains at least one.

Another very popular function to use in vector is push_back. Push_back adds an element to the end of vector, increasing its size by one. Consider the following example:

```
vector<int> v;
for(int i = 1; i < 1000000; i *= 2) {
      v.push_back(i);
}
int elements_count = v.size();
```

Don’t worry about memory allocation -- vector will not allocate just one element each time. Instead, vector allocates more memory then it actually needs when adding new elements with push_back. The only thing you should worry about is memory usage, but at TopCoder this may not matter. (More on vector’s memory policy later.)

When you need to resize vector, use the resize() function:

```
vector<int> v(20);
for(int i = 0; i < 20; i++) {
      v[i] = i+1;
}
v.resize(25);
for(int i = 20; i < 25; i++) {
      v[i] = i*2;
}
```

The resize() function makes vector contain the required number of elements. If you require less elements than vector already contain, the last ones will be deleted. If you ask vector to grow, it will enlarge its size and fill the newly created elements with zeroes.

Note that if you use push_back() after resize(), it will add elements AFTER the newly allocated size, but not INTO it. In the example above the size of the resulting vector is 25, while if we use push_back() in a second loop, it would be 30.

```
vector<int> v(20);
for(int i = 0; i < 20; i++) {
      v[i] = i+1;
}
v.resize(25);
for(int i = 20; i < 25; i++) {
      v.push_back(i*2); // Writes to elements with indices [25..30), not [20..25) ! <
}
```

To clear a vector use clear() member function. This function makes vector to contain 0 elements. It does not make elements zeroes -- watch out -- it completely erases the container.

There are many ways to initialize vector. You may create vector from another vector:

```
vector<int> v1;
// ...
```

```
vector<int> v2 = v1;

vector<int> v3(v1);
```

The initialization of v2 and v3 in the example above are exactly the same.

If you want to create a vector of specific size, use the following constructor:

```
vector<int> Data(1000);
```

In the example above, the data will contain 1,000 zeroes after creation. Remember to use parentheses, not brackets. If you want vector to be initialized with something else, write it in such manner:

```
vector<string> names(20, "Unknown");
```

Remember that you can create vectors of any type.

Multidimensional arrays are very important. The simplest way to create the two-dimensional array via vector is to create a vector of vectors.

```
vector< vector<int> > Matrix;
```

It should be clear to you now how to create the two-dimensional vector of given size:

```
int N, N;

// ...

vector< vector<int> > Matrix(N, vector<int>(M, -1));
```

Here we create a matrix of size N*M and fill it with -1.

The simplest way to add data to vector is to use push_back(). But what if we want to add data somewhere other than the end? There is the insert() member function for this purpose. And there is also the erase() member function to erase elements, as well. But first we need to say a few words about iterators.

You should remember one more very important thing: When vector is passed as a parameter to some function, a copy of vector is actually created. It may take a lot of time and memory to create new vectors when they are not really needed. Actually, it's hard to find a task where the copying of vector is REALLY needed when passing it as a parameter. So, you should never write:

```
void some_function(vector<int> v) { // Never do it unless you're sure what you do!

    // ...

}
```

Instead, use the following construction:

```
void some_function(const vector<int>& v) { // OK

    // ...

}
```

If you are going to change the contents of vector in the function, just omit the 'const' modifier.

```
int modify_vector(vector<int>& v) { // Correct

    V[0]++;
```

```
}
```

Pairs

Before we come to iterators, let me say a few words about pairs. Pairs are widely used in STL. Simple problems, like TopCoder SRM 250 and easy 500-point problems, usually require some simple data structure that fits well with pair. STL std::pair is just a pair of elements. The simplest form would be the following:

```
template<typename T1, typename T2> struct pair {
      T1 first;
      T2 second;
};
```

In general pair<int,int> is a pair of integer values. At a more complex level, pair<string, pair<int, int> > is a pair of string and two integers. In the second case, the usage may be like this:

```
pair<string, pair<int,int> > P;
string s = P.first; // extract string
int x = P.second.first; // extract first int
int y = P.second.second; // extract second int
```

The great advantage of pairs is that they have built-in operations to compare themselves. Pairs are compared first-to-second element. If the first elements are not equal, the result will be based on the comparison of the first elements only; the second elements will be compared only if the first ones are equal. The array (or vector) of pairs can easily be sorted by STL internal functions.

For example, if you want to sort the array of integer points so that they form a polygon, it's a good idea to put them to the vector< pair<double, pair<int,int> >, where each element of vector is { polar angle, { x, y } }. One call to the STL sorting function will give you the desired order of points.

Pairs are also widely used in associative containers, which we will speak about later in this article.

Iterators

What are iterators? In STL iterators are the most general way to access data in containers. Consider the simple problem: Reverse the array A of N int's. Let's begin from a C-like solution:

```
void reverse_array_simple(int *A, int N) {
      int first = 0, last = N-1; // First and last indices of elements to be swapped
      While(first < last) { // Loop while there is something to swap
            swap(A[first], A[last]); // swap(a,b) is the standard STL function
            first++; // Move first index forward
            last--; // Move last index back
      }
}
```

This code should be clear to you. It's pretty easy to rewrite it in terms of pointers:

```
void reverse_array(int *A, int N) {
      int *first = A, *last = A+N-1;
      while(first < last) {
```

```
            Swap(*first, *last);
            first++;
            last--;
        }
    }
```

Look at this code, at its main loop. It uses only four distinct operations on pointers 'first' and 'last':

- compare pointers (first < last),
- get value by pointer (*first, *last),
- increment pointer, and
- decrement pointer

Now imagine that you are facing the second problem: Reverse the contents of a double-linked list, or a part of it. The first code, which uses indexing, will definitely not work. At least, it will not work in time, because it's impossible to get element by index in a double-linked list in O(1), only in O(N), so the whole algorithm will work in O(N^2). Errr...

But look: the second code can work for ANY pointer-like object. The only restriction is that that object can perform the operations described above: take value (unary *), comparison (<), and increment/decrement (++/--). Objects with these properties that are associated with containers are called iterators. Any STL container may be traversed by means of an iterator. Although not often needed for vector, it's very important for other container types.

So, what do we have? An object with syntax very much like a pointer. The following operations are defined for iterators:

- get value of an iterator, int x = *it;
- increment and decrement iterators it1++, it2--;
- compare iterators by '!=' and by '<'
- add an immediate to iterator it += 20; <=> shift 20 elements forward
- get the distance between iterators, int n = it2-it1;

But instead of pointers, iterators provide much greater functionality. Not only can they operate on any container, they may also perform, for example, range checking and profiling of container usage.

And the main advantage of iterators, of course, is that they greatly increase the reuse of code: your own algorithms, based on iterators, will work on a wide range of containers, and your own containers, which provide iterators, may be passed to a wide range of standard functions.

Not all types of iterators provide all the potential functionality. In fact, there are so-called "normal iterators" and "random access iterators". Simply put, normal iterators may be compared with '==' and '!=', and they may also be incremented and decremented. They may not be subtracted and we can not add a value to the normal iterator. Basically, it's impossible to implement the described operations in O(1) for all container types. In spite of this, the function that reverses array should look like this:

```
template<typename T> void reverse_array(T *first, T *last) {
    if(first != last) {
        while(true) {
            swap(*first, *last);
            first++;
            if(first == last) {
                break;
            }
            last--;
            if(first == last) {
                break;
```

```
                }
            }
        }
    }
```

The main difference between this code and the previous one is that we don't use the "<" comparison on iterators, just the "==" one. Again, don't panic if you are surprised by the function prototype: template is just a way to declare a function, which works on any appropriate parameter types. This function should work perfectly on pointers to any object types and with all normal iterators.

Let's return to the STL. STL algorithms always use two iterators, called "begin" and "end." The end iterator is pointing not to the last object, however, but to the first invalid object, or the object directly following the last one. It's often very convenient.

Each STL container has member functions begin() and end() that return the begin and end iterators for that container.

Based on these principles, c.begin() == c.end() if and only if c is empty, and c.end() – c.begin() will always be equal to c.size(). (The last sentence is valid in cases when iterators can be subtracted, i.e. begin() and end() return random access iterators, which is not true for all kinds of containers. See the prior example of the double-linked list.)

The STL-compliant reverse function should be written as follows:

```
template<typename T> void reverse_array_stl_compliant(T *begin, T *end) {
    // We should at first decrement 'end'
    // But only for non-empty range
    if(begin != end)
    {
        end--;
        if(begin != end) {
            while(true) {
                swap(*begin, *end);
                begin++;
                If(begin == end) {
                    break;
                }
                end--;
                if(begin == end) {
                    break;
                }
            }
        }
    }
}
```

Note that this function does the same thing as the standard function std::reverse(T begin, T end) that can be found in algorithms module (#include <algorithm>).

In addition, any object with enough functionality can be passed as an iterator to STL algorithms and functions. That is where the power of templates comes in! See the following examples:

```
vector<int> v;

// ...

vector<int> v2(v);

vector<int> v3(v.begin(), v.end()); // v3 equals to v2



int data[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 };

vector<int> primes(data, data+(sizeof(data) / sizeof(data[0])));
```

The last line performs a construction of vector from an ordinal C array. The term 'data' without index is treated as a pointer to the beginning of the array. The term 'data + N' points to N-th element, so, when N if the size of array, 'data + N' points to first element not in array, so 'data + length of data' can be treated as end iterator for array 'data'. The expression 'sizeof(data)/sizeof(data[0])' returns the size of the array data, but only in a few cases, so don't use it anywhere except in such constructions. (C programmers will agree with me!)

Furthermore, we can even use the following constructions:

```
vector<int> v;

// ...

vector<int> v2(v.begin(), v.begin() + (v.size()/2));
```

It creates the vector v2 that is equal to the first half of vector v.

Here is an example of reverse() function:

```
int data[10] = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };

reverse(data+2, data+6); // the range { 5, 7, 9, 11 } is now { 11, 9, 7, 5 };
```

Each container also has the rbegin()/rend() functions, which return reverse iterators. Reverse iterators are used to traverse the container in backward order. Thus:

```
vector<int> v;

vector<int> v2(v.rbegin()+(v.size()/2), v.rend());
```

will create v2 with first half of v, ordered back-to-front.

To create an iterator object, we must specify its type. The type of iterator can be constructed by a type of container by appending "::iterator", "::const_iterator", "::reverse_iterator" or "::const_reverse_iterator" to it. Thus, vector can be traversed in the following way:

```
vector<int> v;



// ...



// Traverse all container, from begin() to end()
```

```
for(vector<int>::iterator it = v.begin(); it != v.end(); it++) {

      *it++; // Increment the value iterator is pointing to

}
```

I recommend you use '!=' instead of '<', and 'empty()' instead of 'size() != 0' -- for some container types, it's just very inefficient to determine which of the iterators precedes another.

Now you know of STL algorithm reverse(). Many STL algorithms are declared in the same way: they get a pair of iterators – the beginning and end of a range – and return an iterator.

The find() algorithm looks for appropriate elements in an interval. If the element is found, the iterator pointing to the first occurrence of the element is returned. Otherwise, the return value equals the end of interval. See the code:

```
vector<int> v;

for(int i = 1; i < 100; i++) {

      v.push_back(i*i);

}



if(find(v.begin(), v.end(), 49) != v.end()) {

      // ...

}
```

To get the index of element found, one should subtract the beginning iterator from the result of find():

```
int i = (find(v.begin(), v.end(), 49) - v.begin();

if(i < v.size()) {

      // ...

}
```

Remember to #include <algorithm> in your source when using STL algorithms.

The min_element and max_element algorithms return an iterator to the respective element. To get the value of min/max element, like in find(), use *min_element(...) or *max_element(...), to get index in array subtract the begin iterator of a container or range:

```
int data[5] = { 1, 5, 2, 4, 3 };

vector<int> X(data, data+5);

int v1 = *max_element(X.begin(), X.end()); // Returns value of max element in vector

int i1 = min_element(X.begin(), X.end()) - X.begin; // Returns index of min element in vector



int v2 = *max_element(data, data+5); // Returns value of max element in array

int i3 = min_element(data, data+5) - data; // Returns index of min element in array
```

Now you may see that the useful macros would be:

```
#define all(c) c.begin(), c.end()
```

Don’t put the whole right-hand side of these macros into parentheses -- that would be wrong!

Another good algorithm is sort(). It's very easy to use. Consider the following examples:

```
vector<int> X;

// ...

sort(X.begin(), X.end()); // Sort array in ascending order
sort(all(X)); // Sort array in ascending order, use our #define
sort(X.rbegin(), X.rend()); // Sort array in descending order using with reverse iterators
```

Compiling STL Programs

One thing worth pointing out here is STL error messages. As the STL is distributed in sources, and it becomes necessary for compilers to build efficient executables, one of STL's habits is unreadable error messages.

For example, if you pass a vector<int> as a const reference parameter (as you should do) to some function:

```
void f(const vector<int>& v) {
    for(
        vector<int>::iterator it = v.begin(); // hm... where’s the error?..
        // ...
    // ...
}
```

The error here is that you are trying to create the non-const iterator from a const object with the begin() member function (though identifying that error can be harder than actually correcting it). The right code looks like this:

```
void f(const vector<int>& v) {
    int r = 0;
    // Traverse the vector using const_iterator
    for(vector<int>::const_iterator it = v.begin(); it != v.end(); it++) {
        r += (*it)*(*it);
    }
    return r;
}
```

In spite of this, let me tell about very important feature of GNU C++ called 'typeof'. This operator is replaced to the type of an expression during the compilation. Consider the following example:

```
typeof(a+b) x = (a+b);
```

This will create the variable x of type matching the type of (a+b) expression. Beware that typeof(v.size()) is unsigned for any STL container type. But the most important application of typeof for TopCoder is traversing a container. Consider the following macros:

```
#define tr(container, it) \
    for(typeof(container.begin()) it = container.begin(); it != container.end(); it++)
```

By using these macros we can traverse every kind of container, not only vector. This will produce const_iterator for const object and normal iterator for non-const object, and you will never get an error here.

```
void f(const vector<int>& v) {
    int r = 0;
    tr(v, it) {
        r += (*it)*(*it);
    }
    return r;
}
```

Note: I did not put additional parentheses on the #define line in order to improve its readability. See this article below for more correct #define statements that you can experiment with in practice rooms.

Traversing macros is not really necessary for vectors, but it's very convenient for more complex data types, where indexing is not supported and iterators are the only way to access data. We will speak about this later in this article.

Data manipulation in vector

One can insert an element to vector by using the insert() function:

```
vector<int> v;
// ...
v.insert(1, 42); // Insert value 42 after the first
```

All elements from second (index 1) to the last will be shifted right one element to leave a place for a new element. If you are planning to add many elements, it's not good to do many shifts – you're better off calling insert() one time. So, insert() has an interval form:

```
vector<int> v;
vector<int> v2;

// ..

// Shift all elements from second to last to the appropriate number of elements.
// Then copy the contents of v2 into v.
v.insert(1, all(v2));
```

Vector also has a member function erase, which has two forms. Guess what they are:

```
erase(iterator);
```

```
erase(begin iterator, end iterator);
```

At first case, single element of vector is deleted. At second case, the interval, specified by two iterators, is erased from vector.

The insert/erase technique is common, but not identical for all STL containers.

String

There is a special container to manipulate with strings. The string container has a few differences from vector<char>. Most of the differences come down to string manipulation functions and memory management policy.

String has a substring function without iterators, just indices:

```
string s = "hello";

string

      s1 = s.substr(0, 3), // "hel"

      s2 = s.substr(1, 3), // "ell"

      s3 = s.substr(0, s.length()-1), "hell"

      s4 = s.substr(1); // "ello"
```

Beware of (s.length()-1) on empty string because s.length() is unsigned and unsigned(0) – 1 is definitely not what you are expecting!

Set

It's always hard to decide which kind of container to describe first – set or map. My opinion is that, if the reader has a basic knowledge of algorithms, beginning from 'set' should be easier to understand.

Consider we need a container with the following features:

- add an element, but do not allow duples [duplicates?]
- remove elements
- get count of elements (distinct elements)
- check whether elements are present in set

This is quite a frequently used task. STL provides the special container for it – set. Set can add, remove and check the presence of particular element in O(log N), where N is the count of objects in the set. While adding elements to set, the duples [duplicates?] are discarded. A count of the elements in the set, N, is returned in O(1). We will speak of the algorithmic implementation of set and map later -- for now, let's investigate its interface:

```
set<int> s;



for(int i = 1; i <= 100; i++) {

      s.insert(i); // Insert 100 elements, [1..100]

}



s.insert(42); // does nothing, 42 already exists in set



for(int i = 2; i <= 100; i += 2) {

      s.erase(i); // Erase even values

}
```

```
int n = int(s.size()); // n will be 50
```

The push_back() member may not be used with set. It make sense: since the order of elements in set does not matter, push_back() is not applicable here.

Since set is not a linear container, it’s impossible to take the element in set by index. Therefore, the only way to traverse the elements of set is to use iterators.

```
// Calculate the sum of elements in set

set<int> S;

// ...

int r = 0;

for(set<int>::const_iterator it = S.begin(); it != S.end(); it++) {

      r += *it;

}
```

It's more elegant to use traversing macros here. Why? Imagine you have a set< pair<string, pair< int, vector<int> > >. How to traverse it? Write down the iterator type name? Oh, no. Use our traverse macros instead.

```
set< pair<string, pair< int, vector<int> > > SS;

int total = 0;

tr(SS, it) {

      total += it->second.first;

}
```

Notice the 'it->second.first' syntax. Since 'it' is an iterator, we need to take an object from 'it' before operating. So, the correct syntax would be '(*it).second.first'. However, it’s easier to write 'something->' than '(*something)'. The full explanation will be quite long –just remember that, for iterators, both syntaxes are allowed.

To determine whether some element is present in set use 'find()' member function. Don’t be confused, though: there are several 'find()' ’s in STL. There is a global algorithm 'find()', which takes two iterators, element, and works for O(N). It is possible to use it for searching for element in set, but why use an O(N) algorithm while there exists an O(log N) one? While searching in set and map (and also in multiset/multimap, hash_map/hash_set, etc.) do not use global find – instead, use member function 'set::find()'. As 'ordinal' find, set::find will return an iterator, either to the element found, or to 'end()'. So, the element presence check looks like this:

```
set<int> s;

// ...

if(s.find(42) != s.end()) {

      // 42 presents in set

}

else {

      // 42 not presents in set

}
```

Another algorithm that works for O(log N) while called as member function is count. Some people think that

```
if(s.count(42) != 0) {
      // …
}
```

or even

```
if(s.count(42)) {
      // …
}
```

is easier to write. Personally, I don't think so. Using count() in set/map is nonsense: the element either presents or not. As for me, I prefer to use the following two macros:

```
#define present(container, element) (container.find(element) != container.end())
#define cpresent(container, element) (find(all(container),element) != container.end())
```

(Remember that all(c) stands for "c.begin(), c.end()")

Here, 'present()' returns whether the element presents in the container with member function 'find()' (i.e. set/map, etc.) while 'cpresent' is for vector.

To erase an element from set use the erase() function.

```
set<int> s;
// …
s.insert(54);
s.erase(29);
```

The erase() function also has the interval form:

```
set<int> s;
// ..

set<int>::iterator it1, it2;
it1 = s.find(10);
it2 = s.find(100);
// Will work if it1 and it2 are valid iterators, i.e. values 10 and 100 present in set.
s.erase(it1, it2); // Note that 10 will be deleted, but 100 will remain in the container
```

Set has an interval constructor:

```
int data[5] = { 5, 1, 4, 2, 3 };
set<int> S(data, data+5);
```

It gives us a simple way to get rid of duplicates in vector, and sort it:

```
vector<int> v;

// …

set<int> s(all(v));

vector<int> v2(all(s));
```

Here 'v2' will contain the same elements as 'v' but sorted in ascending order and with duplicates removed.

Any comparable elements can be stored in set. This will be described later.

Map

There are two explanation of map. The simple explanation is the following:

```
map<string, int> M;

M["Top"] = 1;

M["Coder"] = 2;

M["SRM"] = 10;



int x = M["Top"] + M["Coder"];



if(M.find("SRM") != M.end()) {

      M.erase(M.find("SRM")); // or even M.erase("SRM")

}
```

Very simple, isn't it?

Actually map is very much like set, except it contains not just values but pairs <key, value>. Map ensures that at most one pair with specific key exists. Another quite pleasant thing is that map has operator [] defined.

Traversing map is easy with our 'tr()' macros. Notice that iterator will be an std::pair of key and value. So, to get the value use it->second. The example follows:

```
map<string, int> M;

// …

int r = 0;

tr(M, it) {

      r += it->second;

}
```

Don't change the key of map element by iterator, because it may break the integrity of map internal data structure (see below).

There is one important difference between map::find() and map::operator []. While map::find() will never change the contents of map, operator [] will create an element if it does not exist. In some cases this could be very convenient, but it's definitly a bad idea to use operator [] many times in a loop, when you do not want to add new elements. That's why operator [] may not be used if map is passed as a const reference parameter to some function:

```
void f(const map<string, int>& M) {
      if(M["the meaning"] == 42) { // Error! Cannot use [] on const map objects!
      }
      if(M.find("the meaning") != M.end() && M.find("the meaning")->second == 42) { // Correct
           cout << "Don't Panic!" << endl;
      }
}
```

Notice on Map and Set

Internally map and set are almost always stored as red-black trees. We do not need to worry about the internal structure, the thing to remember is that the elements of map and set are always sorted in ascending order while traversing these containers. And that's why it's strongly not recommended to change the key value while traversing map or set: If you make the modification that breaks the order, it will lead to improper functionality of container's algorithms, at least.

But the fact that the elements of map and set are always ordered can be practically used while solving TopCoder problems.

Another important thing is that operators ++ and -- are defined on iterators in map and set. Thus, if the value 42 presents in set, and it's not the first and the last one, than the following code will work:

```
set<int> S;
// ...
set<int>::iterator it = S.find(42);
set<int>::iterator it1 = it, it2 = it;
it1--;
it2++;
int a = *it1, b = *it2;
```

Here 'a' will contain the first neighbor of 42 to the left and 'b' the first one to the right.

More on algorithms

It's time to speak about algorithms a bit more deeply. Most algorithms are declared in the #include <algorithm> standard header. At first, STL provides three very simple algorithms: min(a,b), max(a,b), swap(a,b). Here min(a,b) and max(a,b) returns the minimum and maximum of two elements, while swap(a,b) swaps two elements.

Algorithm sort() is also widely used. The call to sort(begin, end) sorts an interval in ascending order. Notice that sort() requires random access iterators, so it will not work on all containers. However, you probably won't ever call sort() on set, which is already ordered.

You've already heard of algorithm find(). The call to find(begin, end, element) returns the iterator where 'element' first occurs, or end if the element is not found. Instead of find(...), count(begin, end, element) returns the number of occurrences of an element in a container or a part of a container. Remember that set and map have the member functions find() and count(), which works in O(log N), while std::find() and std::count() take O(N).

Other useful algorithms are next_permutation() and prev_permutation(). Let's speak about next_permutation. The call to next_permutation(begin, end) makes the interval [begin, end) hold the next permutation of the same elements, or returns false if the current permutation is the last one. Accordingly, next_permutation makes many tasks quite easy. If you want to check all permutations, just write:

```
vector<int> v;

for(int i = 0; i < 10; i++) {
      v.push_back(i);
}
```

```
do {
	Solve(..., v);
} while(next_permutation(all(v));
```

Don’t forget to ensure that the elements in a container are sorted before your first call to next_permutation(...). Their initial state should form the very first permutation; otherwise, some permutations will not be checked.

String Streams

You often need to do some string processing/input/output. C++ provides two interesting objects for it: 'istringstream' and 'ostringstream'. They are both declared in #include <sstream>.

Object istringstream allows you to read from a string like you do from a standard input. It's better to view source:

```
void f(const string& s) {

	// Construct an object to parse strings
	istringstream is(s);

	// Vector to store data
	vector<int> v;

	// Read integer while possible and add it to the vector
	int tmp;
	while(is >> tmp) {
		v.push_back(tmp);
	}
}
```

The ostringstream object is used to do formatting output. Here is the code:

```
string f(const vector<int>& v) {

	// Constucvt an object to do formatted output
	ostringstream os;

	// Copy all elements from vector<int> to string stream as text
	tr(v, it) {
		os << ' ' << *it;
	}
```

```
        // Get string from string stream
        string s = os.str();

        // Remove first space character
        if(!s.empty()) { // Beware of empty string here
            s = s.substr(1);
        }

        return s;
    }
```

Summary

To go on with STL, I would like to summarize the list of templates to be used. This will simplify the reading of code samples and, I hope, improve your TopCoder skills. The short list of templates and macros follows:

```
typedef vector<int> vi;
typedef vector<vi> vvi;
typedef pair<int,int> ii;
#define sz(a) int((a).size())
#define pb push_back
#defile all(c) (c).begin(),(c).end()
#define tr(c,i) for(typeof((c).begin() i = (c).begin(); i != (c).end(); i++)
#define present(c,x) ((c).find(x) != (c).end())
#define cpresent(c,x) (find(all(c),x) != (c).end())
```

The container vector<int> is here because it's really very popular. Actually, I found it convenient to have short aliases to many containers (especially for vector<string>, vector<ii>, vector< pair<double, ii> >). But this list only includes the macros that are required to understand the following text.

Another note to keep in mind: When a token from the left-hand side of #define appears in the right-hand side, it should be placed in braces to avoid many nontrivial problems.

[What Is Not Included in STL]

Creating Vector from Map

As you already know, map actually contains pairs of element. So you can write it in like this:

```
map<string, int> M;

// ...

vector< pair<string, int> > V(all(M)); // remember all(c) stands for

(c).begin(),(c).end()
```

Now vector will contain the same elements as map. Of course, vector will be sorted, as is map. This feature may be useful if you are not planning to change elements in map any more but want to use indices of elements in a way that is impossible in map.

Copying data between containers

Let's take a look at the copy(...) algorithm. The prototype is the following:

```
copy(from_begin, from_end, to_begin);
```

This algorithm copies elements from the first interval to the second one. The second interval should have enough space available. See the following code:

```
vector<int> v1;

vector<int> v2;

// ...

// Now copy v2 to the end of v1

v1.resize(v1.size() + v2.size());

// Ensure v1 have enough space

copy(all(v2), v1.end() - v2.size());

// Copy v2 elements right after v1 ones
```

Another good feature to use in conjunction with copy is inserters. I will not describe it here due to limited space but look at the code:

```
vector<int> v;

// ...

set<int> s;

// add some elements to set

copy(all(v), inserter(s));
```

The last line means:

```
tr(v, it) {
```

```
// remember traversing macros from Part I

      s.insert(*it);

 }

```

But why use our own macros (which work only in gcc) when there is a standard function? It's a good STL practice to use standard algorithms like copy, because it will be easy to others to understand your code.

To insert elemements to vector with push_back use back_inserter, or front_inserter is available for deque container. And in some cases it is useful to remember that the first two arguments for 'copy' may be not only begin/end, but also rbegin/rend, which copy data in reverse order.

Merging lists

Another common task is to operate with sorted lists of elements. Imagine you have two lists of elements -- A and B, both ordered. You want to get a new list from these two. There are four common operations here:

- 'union' the lists, R = A+B
- intersect the lists, R = A*B
- set difference, R = A*(~B) or R = A-B
- set symmetric difference, R = A XOR B

STL provides four algorithms for these tasks: set_union(...), set_intersection(...), set_difference(...) and set_symmetric_difference(...). They all have the same calling conventions, so let's look at set_intersection. A free-styled prototype would look like this:

```
end_result = set_intersection(begin1, end1, begin2, end2, begin_result);
```

Here [begin1,end1) and [begin2,end2) are the input lists. The 'begin_result' is the iterator from where the result will be written. But the size of the result is unknown, so this function returns the end iterator of output (which determines how many elements are in the result). See the example for usage details:

```
int data1[] = { 1, 2, 5, 6, 8, 9, 10 };

int data2[] = { 0, 2, 3, 4, 7, 8, 10 };

vector<int> v1(data1, data1+sizeof(data1)/sizeof(data1[0]));

vector<int> v2(data2, data2+sizeof(data2)/sizeof(data2[0]));

vector<int> tmp(max(v1.size(), v2.size());

vector<int> res = vector<int> (tmp.begin(), set_intersection(all(v1), all(v2), tmp.begin());
```

Look at the last line. We construct a new vector named 'res'. It is constructed via interval constructor, and the beginning of the interval will be the beginning of tmp. The end of the interval is the result of the set_intersection algorithm. This algorithm will intersect v1 and v2 and write the result to the output iterator, starting from 'tmp.begin()'. Its return value will actually be the end of the interval that forms the resulting dataset.

One comment that might help you understand it better: If you would like to just get the number of elements in set intersection, use int cnt = set_intersection(all(v1), all(v2), tmp.begin()) – tmp.begin();

Actually, I would never use a construction like ' vector<int> tmp'. I don't think it's a good idea to allocate memory for each set_*** algorithm invoking. Instead, I define the global or static variable of appropriate type and enough size. See below:

```
set<int> s1, s2;

for(int i = 0; i < 500; i++) {
```

```
        s1.insert(i*(i+1) % 1000);

        s2.insert(i*i*i % 1000);

}

static int temp[5000]; // greater than we need

vector<int> res = vi(temp, set_symmetric_difference(all(s1), all(s2), temp));

int cnt = set_symmetric_difference(all(s1), all(s2), temp) - temp;
```

Here 'res' will contain the symmetric difference of the input datasets.

Remember, input datasets need to be sorted to use these algorithms. So, another important thing to remember is that, because sets are always ordered, we can use set-s (and even map-s, if you are not scared by pairs) as parameters for these algorithms.

These algorithms work in single pass, in O(N1+N2), when N1 and N2 are sizes of input datasets.

Calculating Algorithms

Yet another interesting algorithm is accumulate(...). If called for a vector of int-s and third parameter zero, accumulate(...) will return the sum of elements in vector:

```
vector<int> v;

// ...

int sum = accumulate(all(v), 0);
```

The result of accumulate() call always has the type of its third argument. So, if you are not sure that the sum fits in integer, specify the third parameter's type directly:

```
vector<int> v;

// ...

long long sum = accumulate(all(v), (long long)0);
```

Accumulate can even calculate the product of values. The fourth parameter holds the predicate to use in calculations. So, if you want the product:

```
vector<int> v;

// ...

double product = accumulate(all(v), double(1), multiplies<double>());

// don't forget to start with 1 !
```

Another interesting algorithm is inner_product(...). It calculates the scalar product of two intervals. For example:

```
vector<int> v1;

vector<int> v2;

for(int i = 0; i < 3; i++) {
```

```
	v1.push_back(10-i);

	v2.push_back(i+1);

}

int r = inner_product(all(v1), v2.begin(), 0);
```

'r' will hold (v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2]), or (10*1+9*2+8*3), which is 52.

As for 'accumulate' the type of return value for inner_product is defined by the last parameter. The last parameter is the initial value for the result. So, you may use inner_product for the hyperplane object in multidimensional space: just write inner_product(all(normal), point.begin(), -shift).

It should be clear to you now that inner_product requires only increment operation from iterators, so queues and sets can also be used as parameters. Convolution filter, for calculating the nontrivial median value, could look like this:

```
set<int> values_ordered_data(all(data));

int n = sz(data); // int n = int(data.size());

vector<int> convolution_kernel(n);

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

     convolution_kernel[i] = (i+1)*(n-i);

}

double result = double(inner_product(all(ordered_data), convolution_kernel.begin(), 0)) /
accumulate(all(convolution_kernel), 0);
```

Of course, this code is just an example -- practically speaking, it would be faster to copy values to another vector and sort it.

It's also possible to write a construction like this:

```
vector<int> v;

// ...

int r = inner_product(all(v), v.rbegin(), 0);

```

This will evaluate V[0]*V[N-1] + V[1]+V[N-2] + ... + V[N-1]*V[0] where N is the number of elements in 'v'.

Nontrivial Sorting

Actually, sort(...) uses the same technique as all STL:

- all comparison is based on 'operator <'

This means that you only need to override 'operator <'. Sample code follows:

```
struct fraction {

	int n, d; // (n/d)

	// ...

	bool operator < (const fraction& f) const {

		if(false) {

			return (double(n)/d) < (double(f.n)/f.d);
```

```
                // Try to avoid this, you're the TopCoder!
            }
            else {
                return n*f.d < f.n*d;
            }
        }
};

// ...

vector<fraction> v;

// ...

sort(all(v));
```

In cases of nontrivial fields, your object should have default and copy constructor (and, maybe, assignment operator -- but this comment is not for TopCoders).

Remember the prototype of 'operator <' : return type bool, const modifier, parameter const reference.

Another possibility is to create the comparison functor. Special comparison predicate may be passed to the sort(...) algorithm as a third parameter. Example: sort points (that are pair<double,double>) by polar angle.

```
typedef pair<double, double> dd;

const double epsilon = 1e-6;

struct sort_by_polar_angle {
        dd center;
        // Constuctor of any type
        // Just find and store the center
        template<typename T> sort_by_polar_angle(T b, T e) {
            int count = 0;
            center = dd(0,0);
            while(b != e) {
                        center.first += b->first;
```

```
                        center.second += b->second;
                    b++;
                count++;
            }
                    double k = count ? (1.0/count) : 0;
            center.first *= k;
                    center.second *= k;
        }
 // Compare two points, return true if the first one is earlier
 // than the second one looking by polar angle
 // Remember, that when writing comparator, you should
 // override not 'operator <' but 'operator ()'
        bool operator () (const dd& a, const dd& b) const {
            double p1 = atan2(a.second-center.second, a.first-center.first);
            double p2 = atan2(b.second-center.second, b.first-center.first);
            return p1 + epsilon < p2;
        }
 };

 // ...

 vector<dd> points;

 // ...

        sort(all(points), sort_by_polar_angle(all(points)));
```

This code example is complex enough, but it does demonstrate the abilities of STL. I should point out that, in this sample, all code will be inlined during compilation, so it's actually really fast.

Also remember that 'operator <' should always return false for equal objects. It's very important – for the reason why, see the next section.

Using your own objects in Maps and Sets

Elements in set and map are ordered. It's the general rule. So, if you want to enable using of your objects in set or map you should make them comparable. You already know the rule of comparisons in STL:

| * all comparison is based on 'operator <'

Again, you should understand it in this way: "I only need to implement operator < for objects to be stored in set/map."

Imagine you are going to make the 'struct point' (or 'class point'). We want to intersect some line segments and make a set of intersection points (sound familiar?). Due to finite computer precision, some points will be the same while their coordinates differ a bit. That's what you should write:

```
const double epsilon = 1e-7;


struct point {
        double x, y;

        // ...


        // Declare operator < taking precision into account
        bool operator < (const point& p) const {
                if(x < p.x - epsilon) return true;
                if(x > p.x + epsilon) return false;
                if(y < p.y - epsilon) return true;
                if(y > p.y + epsilon) return false;
                return false;
        }
 };
```

Now you can use set<point> or map<point, string>, for example, to look up whether some point is already present in the list of intersections. An even more advanced approach: use map<point, vector<int> > and list the list of indices of segments that intersect at this point.

It's an interesting concept that for STL 'equal' does not mean 'the same', but we will not delve into it here.

Memory management in Vectors

As has been said, vector does not reallocate memory on each push_back(). Indeed, when push_back() is invoked, vector really allocates more memory than is needed for one additional element. Most STL implementations of vector double in size when push_back() is invoked and memory is not allocated. This may not be good in practical purposes, because your program may eat up twice as much memory as you need. There are two easy ways to deal with it, and one complex way to solve it.

The first approach is to use the reserve() member function of vector. This function orders vector to allocate additional memory. Vector will not enlarge on push_back() operations until the size specified by reserve() will be reached.

Consider the following example. You have a vector of 1,000 elements and its allocated size is 1024. You are going to add 50 elements to it. If you call push_back() 50 times, the allocated size of vector will be 2048 after this operation. But if you write

```
v.reserve(1050);
```

before the series of push_back(), vector will have an allocated size of exactly 1050 elements.

If you are a rapid user of push_back(), then reserve() is your friend.

By the way, it’s a good pattern to use v.reserve() followed by copy(…, back_inserter(v)) for vectors.

Another situation: after some manipulations with vector you have decided that no more adding will occur to it. How do you get rid of the potential allocation of additional memory? The solution follows:

```
vector<int> v;
```

```
// ...

vector<int>(all(v)).swap(v);
```

This construction means the following: create a temporary vector with the same content as v, and then swap this temporary vector with 'v'. After the swap the original oversized v will be disposed. But, most likely, you won't need this during SRMs.

The proper and complex solution is to develop your own allocator for the vector, but that's definitely not a topic for a TopCoder STL tutorial.

Implementing real algorithms with STL

Armed with STL, let's go on to the most interesting part of this tutorial: how to implement real algorithms efficiently.

Depth-first search (DFS)

I will not explain the theory of DFS here – instead, read this section of **gladius**'s Introduction to Graphs and Data Structures tutorial – but I will show you how STL can help.

At first, imagine we have an undirected graph. The simplest way to store a graph in STL is to use the lists of vertices adjacent to each vertex. This leads to the vector< vector<int> > W structure, where W[i] is a list of vertices adjacent to i. Let's verify our graph is connected via DFS:

```
/*

Reminder from Part 1:

typedef vector<int> vi;

typedef vector<vi> vvi;

*/



 int N; // number of vertices

 vvi W; // graph

 vi V; // V is a visited flag

 

 void dfs(int i) {

        if(!V[i]) {

             V[i] = true;

             for_each(all(W[i]), dfs);

        }

 }

 

 bool check_graph_connected_dfs() {

        int start_vertex = 0;

        V = vi(N, false);

        dfs(start_vertex);

        return (find(all(V), 0) == V.end());

 }
```

That's all. STL algorithm 'for_each' calls the specified function, 'dfs', for each element in range. In check_graph_connected() function we first make the Visited array (of correct size and filled with zeroes). After DFS we have either visited all vertices, or not – this is easy to determine by searching for at least one zero in V, by means of a single call to find().

Notice on for_each: the last argument of this algorithm can be almost anything that "can be called like a function". It may be not only global function, but also adapters, standard algorithms, and even member functions. In the last case, you will need mem_fun or mem_fun_ref adapters, but we will not touch on those now.

One note on this code: I don't recommend the use of vector<bool>. Although in this particular case it's quite safe, you're better off not to use it. Use the predefined 'vi' (vector<int>). It's quite OK to assign true and false to int's in vi. Of course, it requires 8*sizeof(int)=8*4=32 times more memory, but it works well in most cases and is quite fast on TopCoder.

A word on other container types and their usage

Vector is so popular because it's the simplest array container. In most cases you only require the functionality of an array from vector – but, sometimes, you may need a more advanced container.

It is not good practice to begin investigating the full functionality of some STL container during the heat of a Single Round Match. If you are not familiar with the container you are about to use, you'd be better off using vector or map/set. For example, stack can always be implemented via vector, and it's much faster to act this way if you don't remember the syntax of stack container.

STL provides the following containers: list, stack, queue, deque, priority_queue. I've found list and deque quite useless in SRMs (except, probably, for very special tasks based on these containers). But queue and priority_queue are worth saying a few words about.

Queue

Queue is a data type that has three operations, all in O(1) amortized: add an element to front (to "head") remove an element from back (from "tail") get the first unfetched element ("tail") In other words, queue is the FIFO buffer.

Breadth-first search (BFS)

Again, if you are not familiar with the BFS algorithm, please refer back to this TopCoder tutorial first. Queue is very convenient to use in BFS, as shown below:

```
/*
Graph is considered to be stored as adjacent vertices list.
Also we considered graph undirected.

vvi is vector< vector<int> >
W[v] is the list of vertices adjacent to v
*/

 int N; // number of vertices
 vvi W; // lists of adjacent vertices


 bool check_graph_connected_bfs() {
       int start_vertex = 0;
       vi V(N, false);
       queue<int> Q;
       Q.push(start_vertex);
       V[start_vertex] = true;
       while(!Q.empty()) {
```

```
            int i = Q.front();
            // get the tail element from queue
            Q.pop();
            tr(W[i], it) {
                if(!V[*it]) {
                    V[*it] = true;
                    Q.push(*it);
                }
            }
        }
       return (find(all(V), 0) == V.end());
 }
```

More precisely, queue supports front(), back(), push() (== push_back()), pop (== pop_front()). If you also need push_front() and pop_back(), use deque. Deque provides the listed operations in O(1) amortized.

There is an interesting application of queue and map when implementing a shortest path search via BFS in a complex graph. Imagine that we have the graph, vertices of which are referenced by some complex object, like:

```
pair< pair<int,int>, pair< string, vector< pair<int, int> > > >


 (this case is quite usual: complex data structure may define the position in
 some game, Rubik's cube situation, etc…)
```

Consider we know that the path we are looking for is quite short, and the total number of positions is also small. If all edges of this graph have the same length of 1, we could use BFS to find a way in this graph. A section of pseudo-code follows:

```
// Some very hard data structure

typedef pair< pair<int,int>, pair< string, vector< pair<int, int> > > > POS;

// ...

int find_shortest_path_length(POS start, POS finish) {

     map<POS, int> D;
     // shortest path length to this position
     queue<POS> Q;
```

```
    D[start] = 0; // start from here
    Q.push(start);

    while(!Q.empty()) {
        POS current = Q.front();
        // Peek the front element
        Q.pop(); // remove it from queue

        int current_length = D[current];


        if(current == finish) {
            return D[current];
            // shortest path is found, return its length
        }


        tr(all possible paths from 'current', it) {
            if(!D.count(*it)) {
            // same as if(D.find(*it) == D.end), see Part I
                // This location was not visited yet
                D[*it] = current_length + 1;
            }
        }
    }

    // Path was not found
    return -1;
}


// ...
```

If the edges have different lengths, however, BFS will not work. We should use Dijkstra instead. It's possible to implement such a Dijkstra via priority_queue -- see below.

Priority_Queue

Priority queue is the binary heap. It's the data structure, that can perform three operations:

- push any element (push)
- view top element (top)
- pop top element (pop)

For the application of STL's priority_queue see the TrainRobber problem from SRM 307.

Dijkstra

In the last part of this tutorial I'll describe how to efficiently implement Dijktra's algorithm in sparse graph using STL containers. Please look through this tutorial for information on Dijkstra's algoritm.

Consider we have a weighted directed graph that is stored as vector< vector< pair<int,int> > > G, where

- G.size() is the number of vertices in our graph
- G[i].size() is the number of vertices directly reachable from vertex with index i
- G[i][j].first is the index of j-th vertex reachable from vertex i
- G[i][j].second is the length of the edge heading from vertex i to vertex G[i][j].first

We assume this, as defined in the following two code snippets:

```
typedef pair<int,int> ii;

typedef vector<ii> vii;

typedef vector<vii> vvii;
```

Dijstra via priority_queue

Many thanks to **misof** for spending the time to explain to me why the complexity of this algorithm is good despite not removing deprecated entries from the queue.

```
      vi D(N, 987654321);

      // distance from start vertex to each vertex



      priority_queue<ii,vector<ii>, greater<ii> > Q;

      // priority_queue with reverse comparison operator,

      // so top() will return the least distance

      // initialize the start vertex, suppose it's zero

      D[0] = 0;

      Q.push(ii(0,0));



      // iterate while queue is not empty

      while(!Q.empty()) {



            // fetch the nearest element

            ii top = Q.top();

            Q.pop();
```

```
            // v is vertex index, d is the distance
            int v = top.second, d = top.first;

            // this check is very important
            // we analyze each vertex only once
            // the other occurrences of it on queue (added earlier)
            // will have greater distance
            if(d <= D[v]) {
                  // iterate through all outcoming edges from v
                  tr(G[v], it) {
                        int v2 = it->first, cost = it->second;
                        if(D[v2] > D[v] + cost) {
                              // update distance if possible
                              D[v2] = D[v] + cost;
                              // add the vertex to queue
                              Q.push(ii(D[v2], v2));

                        }
                  }
            }
      }
```

I will not comment on the algorithm itself in this tutorial, but you should notice the priority_queue object definition. Normally, priority_queue<ii> will work, but the top() member function will return the largest element, not the smallest. Yes, one of the easy solutions I often use is just to store not distance but (-distance) in the first element of a pair. But if you want to implement it in the “proper” way, you need to reverse the comparison operation of priority_queue to reverse one. Comparison function is the third template parameter of priority_queue while the second paramerer is the storage type for container. So, you should write priority_queue<ii, vector<ii>, greater<ii> >.

Dijkstra via set

Petr gave me this idea when I asked him about efficient Dijkstra implementation in C#. While implementing Dijkstra we use the priority_queue to add elements to the “vertices being analyzed” queue in O(logN) and fetch in O(log N). But there is a container besides priority_queue that can provide us with this functionality -- it’s ‘set’! I’ve experimented a lot and found that the performance of Dijkstra based on priority_queue and set is the same.

So, here’s the code:

```
      vi D(N, 987654321);

      // start vertex
      set<ii> Q;
      D[0] = 0;
```

```
	Q.insert(ii(0,0));

	while(!Q.empty()) {

		// again, fetch the closest to start element
		// from "queue" organized via set
		ii top = *Q.begin();
		Q.erase(Q.begin());
		int v = top.second, d = top.first;

		// here we do not need to check whether the distance
		// is perfect, because new vertices will always
		// add up in proper way in this implementation

		tr(G[v], it) {
			int v2 = it->first, cost = it->second;
			if(D[v2] > D[v] + cost) {
				// this operation can not be done with priority_queue,
				// because it does not support DECREASE_KEY
				if(D[v2] != 987654321) {
					Q.erase(Q.find(ii(D[v2],v2)));
				}
				D[v2] = D[v] + cost;
				Q.insert(ii(D[v2], v2));
			}
		}
	}
```

One more important thing: STL's priority_queue does not support the DECREASE_KEY operation. If you will need this operation, 'set' may be your best bet.

I've spent a lot of time to understand why the code that removes elements from queue (with set) works as fast as the first one.

These two implementations have the same complexity and work in the same time. Also, I've set up practical experiments and the performance is exactly the same (the difference is about ~%0.1 of time).

As for me, I prefer to implement Dijkstra via 'set' because with 'set' the logic is simpler to understand, and we don't need to remember about 'greater<int>' predicate overriding.

What is not included in STL

If you have made it this far in the tutorial, I hope you have seen that STL is a very powerful tool, especially for TopCoder SRMs. But before you embrace

STL wholeheartedly, keep in mind what is NOT included in it.

First, STL does not have BigInteger-s. If a task in an SRM calls for huge calculations, especially multiplication and division, you have three options:

- use a pre-written template
- use Java, if you know it well
- say “Well, it was definitely not my SRM!”

I would recommend option number one.

Nearly the same issue arises with the geometry library. STL does not have geometry support, so you have those same three options again.

The last thing – and sometimes a very annoying thing – is that STL does not have a built-in string splitting function. This is especially annoying, given that this function is included in the default template for C++ in the ExampleBuilder plugin! But actually I’ve found that the use of istringstream(s) in trivial cases and sscanf(s.c_str(), …) in complex cases is sufficient.

Those caveats aside, though, I hope you have found this tutorial useful, and I hope you find the STL a useful addition to your use of C++. Best of luck to you in the Arena!

Note from the author: In both parts of this tutorial I recommend the use of some templates to minimize the time required to implement something. I must say that this suggestion should always be up to the coder. Aside from whether templates are a good or bad tactic for SRMs, in everyday life they can become annoying for other people who are trying to understand your code. While I did rely on them for some time, ultimately I reached the decision to stop. I encourage you to weigh the pros and cons of templates and to consider this decision for yourself.