Algorithm Tutorials

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

Archive
Normal view
Discuss this article
Write for TopCoder

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 vect

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) { // Corre
            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
#define 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.

Coming soon: Part II: C++ STL Advanced Uses