

- ▶ [Competitions](#)
- ▶ [TopCoder Networks](#)
- ▶ [Events](#)
- ▶ [Statistics](#)
- ▼ [Tutorials](#)

[Overview](#)[Algorithm Tutorials](#)[Software Tutorials](#)[Marathon Tutorials](#)[Wiki](#)[Forums](#)[Surveys](#)[My TopCoder](#)[Help Center](#)[▶ About TopCoder](#)

Member Search:

Handle:  [Go](#)[Advanced Search](#)

## Algorithm Tutorials

### Power up C++ with the Standard Template Library:



#### Part II: Advanced Uses

By [DmitryKorolev](#)*TopCoder Member*[Archive](#)[Printable view](#)[Discuss this article](#)[Write for TopCoder](#)

In this tutorial we will use some macros and typedefs from [Part I](#) of the tutorial.

[Creating Vector from Map](#)[Copying Data Between Containers](#)[Merging Lists](#)[Calculating Algorithms](#)[Nontrivial Sorting](#)[Using Your Own Objects in Maps and Sets](#)[Memory Management in Vectors](#)[Implementing Real Algorithms with STL](#)[Depth-first Search \(DFS\)](#)[A word on other container types and their usage](#)[Queue](#)[Breadth-first Search \(BFS\)](#)[Priority Queue](#)[Dijkstra](#)[Dijkstra priority queue](#)[Dijkstra by set](#)[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 elements 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,  $R = A*(\sim B)$  or  $R = A-B$
- set symmetric difference,  $R = A \text{ XOR } B$

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  $V[0]*V[N-1] + V[1]*V[N-2] + \dots + V[N-1]*V[0]$  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;
    // Constructor 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  $8 * \text{sizeof(int)} = 8 * 4 = 32$  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 Dijkstra's algorithm in sparse graph using STL containers. Please look through [this tutorial](#) for information on Dijkstra's algorithm.

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;
```

## Dijkstra 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 parameter 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  $O(\log N)$  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 `istream(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.

[Home](#) | [About TopCoder](#) | [Press Room](#) | [Contact Us](#) | [Careers](#) | [Privacy](#) | [Terms](#)  
[Competitions](#) | [Cockpit](#)

Copyright TopCoder, Inc. 2001-2014