Standard Template Library

Using https://community.topcoder.com/tc?module=Static&d1=features&d2=082803 as a template

Standard Template Library = STL

The STL is a collection of containers and simple algorithms. A container is something that holds several elements of the same type.

The STL header files required in this crash course are **vector**, **set**, **map** and **algorithm**. We also use **iostream** and string (the latter is not part of the STL, even though it in many ways have the same functionality). All classes are declared in the **namespace std**, so having the following lines at the beginning of your code is a good idea:

```
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <map>
#include <algorithm>

using namespace std;
```

Vector

A vector corresponds to a 1-dimensional array in C. A vector is declared like this:

```
vector<int> a;        // Declare a vector of integers
vector<MyStruct> b;   // Declare a vector of MyStruct
```

The vectors declared above both initially contain zero elements.
This is no problem, since vectors (like all the STL containers) can grow dynamically by inserting elements.
It's also possible to initialize a vector with elements and also give those a specific value:

```
vector<int> c(50,1); // Declare a vector of integers with 50 elements, all set to 1
```

For structs and classes, the initial value should be a constructor. If the initial value is omitted, the default constructor will be used (in case of simple types like int, double etc, the initial value is 0).

Elements in a vector are accessed the same way as elements in an array, using the [] operator. Elements can also be accessed using at:

```
cout << c[5] << endl;
cout << c.at(5) << endl;
```

The difference between the two is that at will raise an exception if you try to access an element outside the vector while the [] operator won't. Extending the size of a vector by appending an element to the end is done with the push_back method:

```
vector<int> a(10,1);
a.push_back(2);       // Element 0 to 9 in a is now 1, and element 10 is 2.
```

Inserting an element in a vector is done using insert(position,value) or insert(position, size, value)

```
vector<int> a(10,1);
it = std::vector<int>::iterator it;

it = myvector.begin();
it = a.insert(it,200);       // insert 200 in front
a.insert(it+3,300);          // insert 300 before the 4th element
```

Resizing a vector adds zero elements:

```
vector<int> a(10,1);
a.resize(15);          // Element 0 to 9 in a is now 1, and element 10 to 14 is 0.
```

Other useful methods in the vector class are: (T is the type of each element)

```
size_t size();         // Returns the number of elements in the vector
void pop_back();       // Removes the last element
T back();              // Returns the value of the last element
void clear();          // Essentially the same thing as resize(0)
```

Looping over elements

Looping over all elements in a vector can be done by looping over all indexes, or using iterators:

```
for(vector<int>::iterator i=a.begin();i!=a.end();++i)
{
  cout << *i << endl;
}
```

or less cryptic:

```
vector<int> a(10,1);
vector<int>::iterator i;
for(i=a.begin();i!=a.end();++i)
{
  cout << *i << endl;
}
```

or unsophisticated:

```
for(int i=0;i<a.size();++i)
{
  cout << a[i] << endl;
}
```

Sorting

A vector can of course be sorted: (also works on arrays)

```
sort(&a[0],&a[N]);   // N = no elements in the vector (i.e. a.size())

sort(a.begin(),a.end()); // using iterators
```

This will sort all elements in a in the default order. For built-in types, the default order is ascending. For structs and classes, you can define the default order by defining how the less-than operator should work:

```
bool operator<(const MyStruct &a, const MyStruct &b)
{
  // Return true if A<B, false if A>=B
}
```

The two parameters to sort specify the range in the vector to be sorted, by pointing to the first element and the element after the last one. So sort(&a[0],&a[N]) sorts elements 0,1,...,N-1. This way of specifying a range is used throughout all the STL, and is very practical.

Since it's most common you want to perform an operation on the whole vector, the start and endpoint have special names: **begin()** and **end()**. So instead of **sort(&a[0],&a[N])** one can do **sort(a.begin(),a.end())**. Actually, **&a[0]** and **a.begin()** are not in general interchangeable as the former is a pointer and the latter an iterator.

Code example: write a code that counts to 10 and back then insert zeroes into every even element

Pair

A tiny class that is part of the STL is pair, which basically just contains (as the name implies) two data members - first and second.

```
vector< pair<int,int> > a; // It's essential that there is a space between > and >!
//vector<pair<int,int>> a; // leads to an error because compiler is confused about >>
```

However, it's practical to use, especially when sorting a vector of pairs.
The members are, respectively, used as the first and second sort key, which means that both data types used must have a less-than operator defined.
For instance, if you want to sort coordinates first according to their x value and then their y value, then we can use the pair class like this:

```
int N,x,y;
vector< pair<int,int> > a; // It's essential that there is a space between > and >!
cin >> N;
for(int i=0;i<N;++i) {
  cin >> x >> y;
  a.push_back(make_pair(x,y)); // make_pair creates a pair<int,int> object
}
sort(a.begin(),a.end()); // Sorts first according to the x-coord, then the y-coord
```

Note the use of make_pair (also part of the STL). Without it, we would have to type something like

```
a.push_back(pair<int,int>(x,y));
```

Set

One of the most fundamental ways to store information is to have a set of objects. Defining a set of integers is done with

```
set<int> a;
```

To add, remove and check for single elements in a set:

```
a.insert(7);          // Insert integer 7 in the set
a.erase(5);           // Remove integer 5 (if it exist) from the set
if (a.find(7)!=a.end())
  ; // Integer 7 exists in the set
else
  ; // Integer 7 does not exist
cout << a.size() << endl; // Print the number of elements in the set a
```

Other common set operations includes finding the union, intersection and difference between two sets.

```
set<int> a,b,un,in,di;
..
..
set_union(a.begin(),a.end(),b.begin(),b.end(),insert_iterator<set<int> >(un,un.begin()));
set_intersection(a.begin(),a.end(),b.begin(),b.end(),insert_iterator<set<int> >(in,in.begin()))
set_difference(a.begin(),a.end(),b.begin(),b.end(),insert_iterator<set<int> >(di,di.begin()));
```

It's necessary to use the insert_iterator thingy which insert all elements in the union (or intersection or difference) of the two sets a and b into set c. If c is not empty, you might want to clear the set first using the clear() method.
To allow for fast operations, the internal representation of a set is a balanced binary tree. This means that the type of the values must be sortable, i.e. having a less-than operator defined (see above). Thus all elements in a set are always sorted, which can be quite practical.

Convert from vector to set

One can initialize a set from a vector:

```
set<int> b(a.begin(),a.end()); // a is a vector<int>
```

Going the other way is also possible:

```
vector<int> c(b.begin(),b.end()); // b is a set<int>
```

Doing these two operations after each other effectively sorts a vector of integers and removes all duplicate elements!

map

It's often desirable to have a 1-1 relation between two data types.
One member in the relation is the key element, the lookup value.
The other is the data value, retrieved by doing a lookup.

For example, we might want to map names of cities (stored as strings) to serial number (integers):

```
map<string,int> a;
//Now, map can in some ways be used exactly as a vector:
a["New York"]=7;
a["Los Angeles"]=8;
a["Boston"]=10;
a["Los Angeles"]=3;
cout << a["Los Angeles"] << endl;     // Prints 3
cout << a["San Francisco"] << endl;   // Prints 0
```

If no value is given to a key, it receives the default constructor (0 for simple types).
The map class can be thought of as a set of pairs (where first is the key value and second the data value) with the [] operator declared in a handy way, except that only the key value needs a less-than operator defined.

Looping over elements

Looping over all elements in a vector can be done by looping over all indexes,
but that's not possible in a set or a map.
Instead one can use the following method using iterators

```
set<int> a;
map<string,string> b;
..
..
for(set<int>::iterator i=a.begin();i!=a.end();++i)
  cout << *i << endl;
for(map<string,string>::iterator i=b.begin();i!=b.end();++i)
  cout << i->first << " => " << i->second << endl;
```

Note that it's necessary to use **i!=a.end()** and not **i<a.end()** as the elements don't necessary have to be stored in order in the memory.

We can here see that **i** is not a pointer but an iterator: if i was a pointer, **++i** would cause *i* to point to the next element in the memory, while with an **iterator** it will point to the next element in the container **(which does not necessarily lie immediately after the previous element in the memory).**

Looping over elements 2

Instead of explicitly creating the for-loop, one can use a function from the algorithm library: **for_each**

```
void doit(const int &t) { cout << t << endl; }
..
..
set<int> a; // Can be a vector or map as well
for_each(a.begin(),a.end(),doit);
```

That is, for all elements in the range **[a.begin(),a.end())**, call the function doit.
The parameter in doit must be a const reference of the data type in the container.

Code example: take you vector from before,

- create a set
- create a map
- test whether you can insert duplicate elements
- print your set and map