IOI Training 2017 - Week 1
The C++ Standard Template Library

Vernon Gutierrez

March 6, 2017

1 Introduction

This week, we will learn about the C++ Standard Template Library, or STL for short. The STL contains some useful common data structures and algorithms that you can use in your programs, so that you don't have to spend time implementing these yourself, and therefore make your code shorter, and therefore solve your problems faster. The STL is technically not part of the C++ language, but as they are available almost everywhere C++ is available, for the most part, you can think of them as if they were part of the language.

According to Wikipedia, "a **library** is a collection of sources of information and similar resources, made accessible to a defined community for reference or borrowing." You have probably borrowed books from a library sometime in your life. In programming, you can think of a **library** as a collection of code that is made available to all users of a particular programming language (AKA you), so that these users (AKA you) can borrow or *include* these pieces of code in their programs. So that you don't have to write them yourself. So that you don't re-invent the wheel. Normally, these collections are code for common functionality that is common enough to be useful in *lots* of programs, but not common enough to be useful in *all* programs, so they are not made part of the language, but have to be included separately. Like toys with no batteries included, C++ is! (You can't even read and print stuff without using the C++ standard library `iostream`.)

Speaking of which, you've already used the C++ standard library without even realizing it. If you've done `#include <iostream>` and `using namespace std;` and `cin` and `cout`, you were using the C++ standard library. In general, to include functions from a C++ library, you write `#include <name-of-library>` at the beginning of your C++ file.

"**Standard**" just means that a bunch of guys met together, wearing suits, and decided that some library features are so useful, they mandated that all implementers of C++ who wish to be respected must include them with every installation of C++. What is a programming language implementer, you ask? They are, for example, people who write `g++`, and in general, everyone who writes programs that compile C++ programs. *Standard* means you can count on these features to be already installed when you install a good C++ compiler on your computer. There are also *third-party* libraries, which are also quite useful but not as widely useful, so they are not included with every C++ installation, and you need to install these manually if you want to use them. One example is the GNU Multiple Precision Arithmetic Library, which lets you perform arithmetic on numbers larger than 2^{64}, among other things. But we don't have third-party libraries in most programming contest settings, especially the IOI, so we will not talk about or use them. If you want to delve deeper into what a programming language really is and why we do this weird business of separating the language itself from libraries, check this YouTube playlist out.

"**Template**" here means the particular part of the C++ standard library that includes data structures and algorithms, which are most useful for programming contests. For all C++ programs actually, since data structures and algorithms are the bread and butter (or *kanin* and *toyo*) of programs. The nice thing about the C++ STL is that all of its features can be easily used through a common pattern called an *iterator*, which you will learn more about below. For a competitive programmer (AKA you), practically, this means

that code using the STL look very similar to each other, even if they are using different features of the STL. Hence *less* memorizing. These patterns are not the reason why STL is called "template." The real reason is a bit too technical for our purposes, and has something to do with the template feature of the C++ language. You can check this video (and the entire playlist) out if you want to delve deeper into templates (and the STL).

2 Mathematical Functions

The `cmath` library contains a bunch of useful mathematical functions, as commonly found in a scientific calculator, and more. When you `#include <cmath>`, it is like magically converting your standard C++ calculator into a scientific C++ calculator, so that you can do more fun stuff. The `cmath` library is technically a C library and not part of the C++ STL, but that's not too important.

You will almost never need `cmath` for the IOI, but it's good to check it out and see what it contains, just so you know it's there in case you need it. Also, you might use it in the future.

3 Pairs

3.1 Theory

There are many situations wherein you would want to define and use ordered pairs. Rather than creating arrays with two elements, it is sometimes cleaner to use a `pair` object. Also, there are cases where you might want to group together two items of *different types*. An array, which is all about grouping together items of the *same type*, is not quite the right concept for such a grouping. The C++ Standard Library contains a built-in `pair` object for your convenience.

3.2 How to Use C++ Built-in Pairs

Using `pair` is easy. Just `#include <utility>` to make it available. The type of a pair variable is `pair<T1, T2>`, where `T1` is the type of the first member of the pair, and `T2` is the type of the second member of the pair. To make a pair, just use the function `make_pair` (you don't say). To get the first member of a pair, apply `.first` on the pair. You can probably guess how to get the second member of a pair.

Sample code:

```
#include <iostream>
#include <utility>
#include <string>
using namespace std;

int main() {
    pair<int, int> lattice_point = make_pair(1, -1);
    cout << lattice_point.first << " " << lattice_point.second << endl; // 1 -1

    lattice_point.first = 2;
    cout << lattice_point.first << " " << lattice_point.second << endl; // 2 -1

    pair<string, int> name_age = make_pair("Aldrich", 20);
    cout << name_age.first << ", " << name_age.second << endl; // Aldrich, 20
```

```
    pair<pair<int, int>, int> nested_pair = make_pair(make_pair(1, 2), 3);
    cout << nested_pair.first.first << " " << nested_pair.first.second << endl; // 1 2
    cout << nested_pair.second << endl; // 3
    return 0;
}
```

3.3 How They're Implemented in C++

Under the hood, C++ pairs are actually structs. If you want to group items into a pair and give meaningful names to each member (not just `first` and `second`), you would create your own `struct` instead. Also, if you wanted to group together three or more items, nesting pairs, as we did above, gets messy and hard to read and debug. For these cases, you would use a `struct` or `tuple` instead. But, for many cases where you need pairs of items, using `pair` is the most convenient choice. The benefits of using pairs will become clearer later with graph algorithms. Thinking about pairs will also help you understand the `map` data structure below.

3.4 Gotchas

Modify the code above to directly print `lattice_point`. That is, try `cout << lattice_point << endl;`. What happens? `std::pairs`'s can't be directly fed into `cout` or read from `cin`.

Create a nested pair where the second element is a pair, instead of the first one:

```
pair<int, pair<int, int>> nested_pair;
```

What happens? In the currently most widely-used version of C++, C++03, this throws a compile error. This is due to a design flaw in C++03 that treats all instances of `>>` in the code as a `>>` (e.g. used for `cin`) operator. If you are using C++03, to properly disambiguate the angle brackets used for types and the angle brackets used for the `>>` operator, you must put a space between the two closing brackets:

```
pair<int, pair<int, int> > nested_pair;
```

This is fixed in newer versions of C++ starting from C++11. If you didn't encounter this error, then great! Your compiler uses C++11 (or C++14) by default. If you did, then you should either add the space, or explicitly tell your compiler to use C++11 instead:

```
g++ -std=c++11 program.cpp
```

Or, if you have an even newer compiler, use C++14: instead:

```
g++ -std=c++14 program.cpp
```

The same gotcha applies to all the different data structures below. Most modern programming contest environments, including the IOI, have compilers that fully support C++11. So, you might want to make it a habit to always compile with `-std=c++11`. C++14 is not yet as widely supported, but in a few years, you should probably change that habit to use C++14 instead.

4 Vectors

4.1 Theory

Arrays are great, but they require you to specify the maximum size in advance. In cases where you cannot predict this maximum size, you would need a list that allows you to keep adding as many elements as you need, by changing its size on demand. One way this is done is through a linked list, which you saw in Week

0. But a linked list has a serious disadvantage: retrieving elements from the middle of the list requires $O(n)$ time. In technical terms, a linked list does not have efficient *random access*.

Another way is through dynamic arrays. The idea is, just initially allocate space for some small number of elements. Whenever you need more space, create a new array that is bigger than your old array, and copy all the elements from the old array to the new array. To save space, after a lot of elements, create a new array that is smaller than the old array, and again copy all the elements from the old array to the new array. If you want to see this in greater detail, check this YouTube playlist out.

After going through this entire lesson, you may want to check out this video and this video, to really understand why this is fast even though it seems like we are doing $O(n)$ work per operation to copy items.

C++ has a built-in dynamic array, called `vector`. Like arrays, you can have vectors of any type. You can have vectors of integers, vectors of pairs, even vectors of vectors and vectors of vectors of vectors of vectors. Though in those last two cases, it's probably easier to work with multi-dimensional arrays instead.

4.2 How to Use C++ Built-in Vectors

Most of you are probably already familiar with vectors, if not check this out. Aside from `vector`, the video also discusses a bunch of other sequence containers. But in competitive programming settings, only `vector` and `deque` are widely used.

4.3 How They're Implemented in C++

Behind the scenes, a `vector` grows and shrinks by dynamically allocating new arrays of a new size, and copying the old array into the new array. For this reason, they are slower than regular arrays, but the speed difference is usually not significant enough to cause performance problems and TLE's. A `vector` also keeps track of its size so that it can be queried in $O(1)$.

4.4 Gotchas

Because `vectors` grow and shrink through dynamic allocation, there are annoying rare cases when this causes memory problems. Especially if you are solving a problem that requires processing multiple test cases, take care to cleanup the vector after you are done using it. No, you don't use `myvector.clear()`. Instead you need to do `vector<type>().swap(myvector)`. The reason is too technical. Just trust us for now.

5 Stacks, Queues, and Doubly-Ended Queues

5.1 Theory

Stacks, queues, and doubly-ended queues are limited versions of linked lists and vectors, where you can only access, insert, and delete at one or both ends of an array. Why would you ever want a more limited version of a data structure? One reason is that it is conceptually clearer that your intention is to just operate on the ends of the list, rather than randomly accessing elements in the middle. You also avoid bugs that may occur if accidentally do access elements in the middle, when your intention is to only access elements from the ends. There is also a very beautiful connection between these restricted data structures and graph algorithms, which you will see later.

Check this video out to see how stacks, queues, and doubly-ended queues work.

5.2 How to Use C++ Built-in Stacks, Queues, and Doubly-ended Queues

If you watched the video about `vector`'s, `deque`'s, and other sequence containers from the previous section, it should be very easy to understand the following example programs for `stack` and for `queue`.

Doubly-ended queues can be used in two ways. This and this should make that clear. These two ways are not mutually exclusive. You can insert, access, and delete from either end at any time.

5.3 How They're Implemented in C++

Stacks, queues, and doubly-ended queues in C++ are generally implemented using dynamic arrays rather than linked lists.

5.4 Gotchas

The same gotcha for `vector` applies to `stack`, `queue`, and `deque`.

Don't forget to check first if the data structure actually has elements in it before popping.

6 Priority Queues

6.1 Theory

Priority queues are a generalization of queues, where in each dequeue operation, instead of accessing or removing the the element that has been in the queue for the longest time, we access or remove the element with the highest priority. The priority rule can be anything we like. It can be highest value first, or lowest value first, or some other rule. It is quite easy to implement this with arrays or vectors, so that either insertion or access and deletion takes $O(n)$ time. But if we needed to do lots of operations, this is too slow. A heap data structure allows us to perform each operation in $O(\lg n)$ time. Watch this or this to learn how heaps achieve this.

6.2 How to Use C++ Built-in Priority Queues

If you understand how to use stacks and queues, then using priority queues is fairly straightforward. This sample program should be easy to understand. Note that you must `#include <queue>`, and not `<priority_queue>`.

By default, `priority_queue` prioritizes elements by highest value first.

6.3 Defining Priority

"Highest value" makes sense for numbers, but what does it mean for strings and other kinds of data? For strings and vectors, lexicographic order is the default rule. For pairs, the first elements are compared first. If they are tied, then the second elements are compared.

What about for `struct`'s and objects that you define yourself? C++ knows nothing about your custom objects and how they should be ordered, so it asks you to specify the rules yourself. There are several ways to specify these rules, and Ashar Fuadi, competitive programmer from University of Indonesia, has a nice blog post about it.

6.4 How They're Implemented in C++

Under the hood, `priority_queue`'s are implemented using binary heaps.

6.5 Gotchas

The order in which two equal elements are retrieved from a priority queue is not specified. Any one of them can come before the other.

Don't forget to check first if the data structure actually has elements in it before popping.

7 Sets and Maps

7.1 Theory

Vectors and lists are *indexed* collections of items. If you didn't care about the positions of items, but instead would like to check for the existence of items or *keys* really quickly, then you would use a set instead. A set allows you to store a collection of items, and quickly determine if your collection contains a certain key or not.

Maps are similar to sets, but in addition to storing just keys, you can also store a *value* associated with each key. When you go look for a key, the map will also tell you what the value associated with the key is, if the key exists in the map. You can think of a map as a generalization of an array, where instead of associating integers from 0 to $n-1$ to objects, you are associating characters, strings, or any arbitrary member of the key type to objects. Like sets, maps are able to quickly check for the existence of a certain key and retrieve values associated with that certain key, though not quite as quickly as an array can retrieve values associated with certain integers.

Maps can also be used as *sparse arrays*, where you can store n items in "positions" 0 to $N-1$, using only $O(n)$ space rather than $O(N)$ space. If $n << N$, this is a significant saving and can spell the difference between feasible and infeasible solutions.

In order to support insertion, deletion, and lookup of keys in $O(\lg n)$ time per operation, sets and maps are typically implemented using binary search trees. Check this video out to learn about them.

7.2 How to Use C++ Built-in Sets and Maps

See this video for an overview of sets and maps. For more details, check out TopCoder tutorials for `set` and `map`

7.3 How They're Implemented in C++

Under the hood, a `set` is a binary search tree of keys of the specified type, while a `map` is a binary search tree of pairs, where the first element of each pair is a member of the key type, and the second element is a member of the value type.

Since `set`'s and `map`'s are implemented using binary search trees, you can actually do another interesting thing with them: finding the key nearest to a given query key, in $O(\lg n)$ time. There are two variants for this: `upper_bound` and `lower_bound`. Here is the reference for set. Map works similarly.

7.4 Gotchas

If you use custom `struct`'s or objects as keys to your sets and maps, you need to specify a custom comparison function, like you would for `priority_queue`. Only very rarely would you actually need this. But precisely because you only do it very rarely, it's very easy to forget. Maybe the lesson on graphs and graph search will remind you about this again.

8 Iterators

As we mentioned earlier in this document, the nice thing about the C++ STL is that there is a common way to use all the data structures above, and to use them with all the algorithms below. That way is through what is called an *iterator*. An iterator is like a pointer, but fancier.

See this video to learn how they work.

9 Sorting

9.1 Theory

This is probably not new to you. You're already probably convinced of the usefulness of sorting, and may have in fact already used the C++ STL `sort` function before. You've probably also heard that sorting takes $O(n \lg n)$ time, but you're not sure why. If you're curious and want to know why, watch this video.

9.2 How to Use C++ Built-in Sorting Methods

Check this video out to see all the various ways you can sort using the C++ STL. You can again define your own sorting rule, like you would for priority queues, sets, and maps, if you wanted to override the default ordering, or if you were using custom `struct`'s or objects.

10 Permutations

10.1 Theory

When you want to do a brute-force solution, you sometimes need to check all possible permutations of a list of items. While it is possible to write your own short function that does this, it is not such a good idea, especially under contest pressure. It is better to use a built-in function so that you don't spend time implementing and debugging your own permutation generator.

10.2 How to Use C++ Built-in Permutation Generators

The sample program here should be easy enough to understand. Again, if you are permuting custom-defined objects, you also need to specify a custom comparison function.

11 Miscellaneous Helper Functions in the C++ STL

For programming contests, `sort` and `next_permutation` are the most useful functions of the algorithms library. Another set of important functions are `binary_search`, `lower_bound`, and `upper_bound`, which implement binary search on sorted sequences. Binary search is deceptively simple to implement on your own, but under contest pressure, even the most experienced coders can sometimes implement them incorrectly and waste a few minutes trying to debug their binary search implementation. I therefore recommend reading the C++ reference for them and learning how to use them, to give you a slight advantage in programming contests. They are covered in the first few minutes of this video. There are also a bunch of other functions that are not as useful, in the sense that you will still be able to write solutions quickly without them, but they are good to know and they can spell the difference between writing code within 5 minutes versus writing code within 4 minutes. They are the following, in order of usefulness: `min`, `max`, `fill_n`, `fill`, `copy`, `reverse`, `count`, `count_if`, `find`, `find_if`, `for_each`. Check the complete list of available functions in the algorithms library.

12 Miscellaneous Tips

Be very careful in naming your variables when you are `using namespace std` and the C++ STL. If you include the above libraries in your program, there is a chance that you might name your variable using one of the names already used by the above libraries. For example, if you `#include <algorithm>` and you are `using namespace std`, you never want to name your variables `min`, `max`, or `count`, because functions with these names exist in the algorithms library. Your variable name and `std`'s name will clash, and you will get a weird compile error. To avoid having to deal with this, you can either stop `using namespace std`, or stick to a naming convention that is weird enough to ensure that your names never clash with `std`'s names.

TopCoder has an excellent tutorial on using the C++ STL for competitive programming, here and here. If you need more information and examples, I recommend checking them out.

13 Challenges (To Be Submitted)

Try out the problems below. Use the C++ STL to make your code as short as possible.

Codeforces 493B - Vasya and Wrestling

UVa 11995 - I Can Guess the Data Structure!

Codeforces 44A - Indian Summer

Codeforces 390A - Inna and Alarm Clock

UVa 11849 - CD

Codeforces 501B - Misha and Changing Handles

UVa 10226 - Hardwood Species

Codeforces 474B - Worms

Codeforces 405A - Gravity Flip

UVa 146 - ID Codes

Codeforces 431B - Shower Line

14 Extra Challenges (Optional)

UVa 732 - Anagrams by Stack

UVa 10901 - Ferry Loading III

UVa 11034 - Ferry Loading IV

UVa 1203 - Argus

UVa 11286 - Conformity

UVa 11136 - Hoax or what