

# STANDARD TEMPLATE LIBRARY IN C++

SIDDHANT CHAUDHARY

ABSTRACT. This is a reference sheet for the STL in C++.

## CONTENTS

0.1. Iterators .....	1
0.2. Vector .....	2
0.3. Queue .....	2
0.4. Deque .....	2
0.5. Set and Multiset .....	2
0.6. Maps and Unordered Maps .....	3
0.7. Priority Queue .....	4
0.8. Bitsets .....	4
1. Strings in C++ .....	5
1.1. scanf and printf with strings .....	5
1.2. Comparing strings .....	5
1.3. scanf return values are useful .....	5
1.4. memcpy and memset with strings .....	6
1.5. String parsing with scanf .....	6
2. Random Number Generation .....	6
2.1. Reference blog .....	6
2.2. Initialising the Mersenne Twister Generator in C++ .....	7
2.3. The uniform integer distribution .....	7
2.4. Random shuffle .....	7

**0.1. Iterators.** In all STL classes, we use *iterators* instead of *pointers*. Iterators are used to point to elements in an STL class; for instance, if  $v$  is a vector, then the iterator  $v.begin()$  points to the first element of the vector  $v$ . Iterators work just like pointers; so the first element of the vector  $v$  will be  $*(v.begin())$ . To store the iterator in a variable, we can do something like `stl_class::iterator p`. For instance, in maps of the type `map <int , int>`, we can do the following.

```
1 map <int , int> mymap;  
2 map <int , int>::iterator p; p = mymap.begin();  
3 cout << *(p).first; //prints the key of the first element in the map
```

A similar line can be used to declare iterators for other STL classes.

0.2. **Vector.** Consider vector `<int> v`. Then the following are defined.

- (1) `v.assign(no_of_elements , val);` //initialisation of `v`
- (2) `v.size();` //returns the size
- (3) `v.empty();` // returns whether `v` is empty
- (4) `v.front();` // first element
- (5) `v.back();` // last element
- (6) `v.push_back(x);` // push `x` in `v`
- (7) `v.pop_back();` // pop last element
- (8) `v.begin();` // iterator to beginning. This is an iterator
- (9) `v.end();` // iterator to end. NOT the last element. This is an iterator
- (10) `v.rbegin();` // reverse iterator to beginning, i.e iterator to last element
- (11) `v.rend();` // reverse iterator to beginning. NOT the first element

**Remark 0.0.1.** Consider the iterators `v.end()` and `v.rend()`. In C++, the last element of the vector `v` will be succeeded by a *theoretical last element*. So, the element `*(v.end())` will actually *not* be the last element of `v`; instead, the last element will be `*(v.end() - 1)`. A similar thing holds for the reverse iterator.

0.3. **Queue.** Consider queue `<int> myqueue`. Then the following are defined.

- (1) `myqueue.empty()`
- (2) `myqueue.size()`
- (3) `myqueue.front()`
- (4) `myqueue.back()`
- (5) `myqueue.push(x)`
- (6) `myqueue.pop()`

0.4. **Deque.** Consider deque `<int> mydeque`. Then the following are defined;

- (1) `mydeque.assign(no_of_elements , val);` //initialisation of `mydeque`
- (2) `mydeque.size();` //returns the size
- (3) `mydeque.empty();`
- (4) `mydeque.front();`
- (5) `mydeque.back();`
- (6) `mydeque.push_back(x);`
- (7) `mydeque.pop_back();`
- (8) `mydeque.push_front(x);`
- (9) `mydeque.pop_front();`
- (10) `mydeque.begin();`
- (11) `mydeque.end();`
- (12) `mydeque.rbegin();`
- (13) `mydeque.rend();`

0.5. **Set and Multiset.** These are implemented as balanced BSTs in C++, and their operations take time  $O(\log n)$ . Both of these are ordered structures. The operations for `set` and `multiset` are the same. Consider `set <int> myset`.

- (1) `myset.begin()`
- (2) `myset.end()`
- (3) `myset.rbegin()`
- (4) `myset.rend()`
- (5) `myset.empty()`
- (6) `myset.size()`

- (7) `myset.insert(x)`
- (8) `myset.erase(position)` or `myset.erase(value)`. `position` must be an iterator and this erasing takes constant time. `value` must be a value contained in the set, and this erasing takes logarithmic time.
- (9) `myset.find(value)`. Get the iterator (pointer) where the value `value` is stored inside the set. If `value` is not in the set, simply return `set.end()`. Takes logarithmic time.

If you want to pass a general comparison function to a `set` or `multiset`, then you can do something like this.

```

1  struct my_structure{
2      ll x , y , z;
3  };
4
5  class compare_function{
6      public:
7      bool operator()(const my_structure &a , const my_structure &b
8  ){
9          //sort on the basis of the first coordinate
10         return a.x < b.x;
11     }
12 };
13
14 //initialise the set or multiset like this
15 set <my_structure , compare_function> myset;
```

**0.6. Maps and Unordered Maps.** A set that contains key-value pairs, and the keys must be *distinct*. `map` is an ordered set, while `unordered_map` is not ordered; `map` uses the BST data structure, and hence operations are of the order  $O(\log n)$ . For `unordered_map`, the operations are  $O(1)$  on average, since these use hashing. Let `map <int , int> mymap` be a definition.

- (1) `mymap.begin()`
- (2) `mymap.end()`
- (3) `mymap.rbegin()`
- (4) `mymap.rend()`
- (5) `mymap.empty()`
- (6) `mymap.size()`
- (7) `mymap.insert(pair<int , int>(x , y))` //insert the key-value pair (x , y)
- (8) `mymap.erase(position)` or `myset.erase(key)`. Erasing by position takes constant time, while erasing by key takes logarithmic time.
- (9) `mymap.find(key)`. Get the iterator to the pair with key attribute equal to `key`.

As usual, we can pass a custom key comparison function to a `map`. So you can do something like this.

```

1  struct key_type{
2      ll x , y , z;
3  };
4
```

```

5     class compare_function{
6     public:
7         bool operator()(const key_type &a , const key_type &b){
8             return a.x < b.x;
9         }
10    };
11
12    //initialise the map as follows
13    map <key_type , mapped_type , compare_function> mymap;

```

**0.7. Priority Queue.** Look at this example.

```

1     struct Node {
2         int x , y , value;
3     };
4     class compare{
5     public:
6         int operator()(Node &a, Node &b){
7             return a.value > b.value;
8         }
9     };
10    priority_queue <Node , vector<Node> , compare> myqueue;
11    Node mynode;
12    myqueue.push(Node);
13    myqueue.pop();
14

```

First, only look at the last four lines. We have defined `myqueue` to be a priority queue, which contains values of type `Node`, and the comparison function is `compare`. The class `compare` is just used to say what the comparison function does. You can literally use any comparison function of your choice!

**Remark 0.0.2.** In this case, we will have a min-priority queue. To have a max-priority queue, just invert the inequality in the `compare` class.

The following are defined.

- (1) `myqueue.empty()`
- (2) `myqueue.size()`
- (3) `myqueue.top()` // top element
- (4) `myqueue.push(x)` // `x` must be of type `Node`
- (5) `myqueue.pop()` // pop the top element

**0.8. Bitsets.** We will use the following for reference (this is also given in the main document).

Declare a bitset like this: `bitset <size> mybitset`. Given two bitsets `a` and `b`, the following are defined.

- (1) `a.count()` //the number of set bits in `a`
- (2) `a[i]` //access the  $i^{\text{th}}$  bit
- (3) `a.size()` //size of the bitset
- (4) `a.test(pos)` //true if bit at position `pos` is 1, false otherwise
- (5) `a.any()` //test if any bit is set
- (6) `a.none()` //test if no bit is set
- (7) `a.all()` //test if all bits are set

```

(8) a.set() // set all the bits
(9) a.set(pos)// set the bit at position pos
(10) a.set(pos , val) //set bit at position pos to val
(11) a.reset() //reset all the bits, i.e all the bits to 0
(12) a.reset(pos) // reset the bit at position pos
(13) a.flip() //flip all the bits
(14) a.flip(pos) //flip the bit at position pos

```

The good thing about bitsets is that you can apply the usual bitwise operations to them. We can also construct a bitset from integers or strings, as follows.

```

1  bitset<100> a(17);
2  bitset<100> b("1010");

```

## 1. STRINGS IN C++

These are specially defined in the STL. See the [documentation](#) for more information.

**1.1. scanf and printf with strings.** The functions `scanf` and `printf` do not support any C++ class. In particular, we cannot take input using this function to a `string` object. For example, the following code will not make sense.

```

1  string temp;
2  scanf("%s" , &temp); //will not work

```

Instead, the following can be used.

```

1  char temp1[100];
2  scanf("%s" , &temp1);
3  string temp = temp1; //this works

```

**1.2. Comparing strings.** NEVER try to compare two strings using the `==` operator. For instance, both of the below snippets are not encouraged.

```

1  char temp1[100]; char temp2[100];
2  if (temp1 == temp2){ \\don't do this
3      ...
4  }

```

```

1  string temp1; string temp2;
2  if (temp1 == temp2){ \\don't do this
3      ...
4  }

```

Instead, convert the given strings to a `string` object, and then use the `std::string.compare` function.

**1.3. scanf return values are useful.** The `scanf` function returns the number of items of the argument list successfully filled (see <https://www.cplusplus.com/reference/cstdio/scanf/>). This means the return value of `scanf` can be very useful in parsing input. For instance, consider the problem [UVa230](#), in which we have to consider a specific input format. In order to input the whole library of books, we can do something as follows.

```

1 //taking the library books
2 char title[81], author[81];
3
4 while(scanf("%s by %s", title, author) == 2){
5     string tit = title, auth = author;
6     //do something
7 }

```

1.4. **memcpy and memset with strings.** Using `memcpy` and `memset` with strings is very useful. For example, do something like this.

```

1 char my_string[80];
2 memset(my_string, 'a', 80); //my_string becomes aaaa..aa (80 a's)
3 char another_string[80];
4 memcpy(another_string, my_string, 80) //copying my_string to
   another_string

```

1.5. **String parsing with scanf.** In many problems, especially ICPC problems, one has to parse the input in a very specific way. To do this, `scanf` comes in handy, and the `cout` way of input is highly discouraged (as it will make parsing difficult to code). The basic format specifier of `scanf` is as follows.

`%[*][width][length]specifier`

specifier can take the following values.

- (1) `d`: integer.
- (2) `x`: hexadecimal integer.
- (3) `f`: floating point number.
- (4) `c`: character.
- (5) `s`: any number of non-whitespace characters.
- (6) `p`: pointer.
- (7) `[characters]`: any number of characters specified within the brackets.
- (8) `[^characters]`: any number of characters *not within* the characters specified within the brackets.
- (9) `%`: a single `%` sign.

Here are the meaning of the optional specifiers.

- (1) The optional specifier `*` indicates that the data is to be read but ignored.
- (2) `width` specifies the maximum number of characters to be read in the current reading operation.
- (3) `length` alters the expected type of the storage pointed by the corresponding argument. `length` has to be one of `hh`, `h`, `l`, `ll`, `j`, `z`, `t`, `L`.

Some cool examples of string parsing are given below.

```

1 scanf("%s\\n"); //reads a string without whitespace within
   quotation marks
2 scanf("\\[^\\]\\n"); //reads a string (with white space possibly)
   without the " character within quotation marks
3 scanf("[^\\n]\\n"); //reads a line, without the \\n character

```

## 2. RANDOM NUMBER GENERATION

2.1. **Reference blog.** Check out this reference blog to know more: <https://codeforces.com/blog/entry/61587>.

**2.2. Initialising the Mersenne Twister Generator in C++.** To initialise the generator, we will use the current system time as the seed (using a fixed seed is not recommended for competitive programming, as it might results in hacks and someone challenging your code).

```
1 mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
```

Use `mt19937_64` to generate 64-bit numbers.

**2.3. The uniform integer distribution.** To generate a random integer in a given range  $[a, b]$  uniformly, you can do the following.

```
1 <type> x = uniform_int_distribution<type>(a , b)(rng)
```

Here `type` must be an integer type (`int`, `long long int` etc). `rng` is the generator that we initialised before.

**2.4. Random shuffle.** To randomly shuffle a vector, you can do something like this.

```
1 vector <long long int> v;  
2 shuffle(v.begin(), v.end(), rng);
```