# C++ String Class

Strings in the C language are internally stored in an array of characters. For C++, particularly for programming competitions, we prefer to use vectors to arrays. Naturally, internally, a string is stored as a vector of characters (type char). This means that the standard way in which we deal with vectors also works with strings, plus some other special functionality that is special to strings. First let's take a look at how we can read information into a string:

```
#include <string>
#include <iostream>

int main() {
    string s;
    cin >> s;
    cout << s << endl;
    return 0;
}
```

Thus, at a bare minimum, to use the string class in C++ we must include string and then we can read a string (sequence of characters) in the usual way.

One important thing to note is that cin naturally tokenizes on spaces, tabs and newlines. Thus, the code above will only read in one token of characters without any spaces in it. Most competitive programming problems don't require reading in strings with spaces, but for ones that do, we'll have to use a different strategy. We'll put that off for a little bit while we look at how to work with strings, beyond reading them in.

## *Standard String Functionality*

To retrieve an element of a string, just use [], exactly like vectors. In addition, you can iterate through a string just like a vector. The following function, for example, returns the first index of the character c in the string s, or -1 if the character c isn't in the string s:

```
int firstOccurrence(string s, char c) {
    for (int i=0; i<s.size(); i++)
        if (s[i] == c)
            return i;
    return -1;
}
```

One important operator that can be used with strings is the plus sign(+). This operator, for strings, is overloaded to represent string concatenation (sticking strings together). Here is a short code snipped with string concatenation:

```
string s, t, res;
cin >> s >> t;
res = s + t;
```

Most commonly, one might continually concatenate into a particular string to build a longer one. Here is a section of code which reads in n strings and concatenates them together (first the integer n is read, followed by the strings):

```
int n;
cin >> n;
string s = "";
for (int i=0; i<n; i++) {
    string tmp;
    cin >> tmp;
    s += tmp;
}
```

A very common task is to compare two strings in lexicographical order. (Actually, we probably want to compare strings in alphabetical order, but the built in function uses lexicographical order.) The string method that compares two strings in lexicographical order is **compare**. Since strings are objects, we can call the method as follows:

```
if (s.compare(t) < 0)
    cout << s << " comes before " << t << endl;
```

C++ is object oriented and strings are objects. Thus, when methods in the string class are called (these are like functions), we put the name of the string we call the method on, followed by a dot, followed by the method name. Here, we are calling compare on s with an input parameter of t. If s comes before t, a negative integer will be returned. If the strings are equal 0 will be returned. If s comes after t, a positive integer will be returned.

Lexicographical ordering is as follows:

Compare corresponding characters' Ascii values (first characters, second characters, etc.) until the first difference is encountered. At that difference, if whichever string has the character that has a smaller Ascii value is considered as going first. For example,

`telephone` comes before
`teleprompter`

The yellow highlight indicates the first mismatching character.

The Ascii values of all the uppercase letters are contiguous, from 'A' = 65 to 'Z' = 90, and the Ascci values of all the lowercase letters are contiguous, from 'a' = 97 to 'z' = 122. In code, one should never actually use these numbers. Rather, the characters themselves, properly evaluate to these numbers. This means that as long as two strings have all lowercase characters or all uppercase characters, the compare method properly compares the strings in alphabetical order.

Let's look at an example problem we can solve with strings and the compare method:

## <u>**String compare Example: Line Them Up**</u>

Here is the Kattis link to the problem:

https://open.kattis.com/problems/lineup

Basically, we must read in a list of names (all uppercase letters), and determine if the list is in increasing order, decreasing order, or neither. We can solve this by reading in the first string, and the looping through and reading the rest. If every comparison with the previous element results in a later string, then the list is increasing. Alternatively, if every comparison with the previous element results in an earlier string, then the list is decreasing.

Let's look at the solution:

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

int main() {

    int n, up = 0, down = 0;
    cin >> n;
    string cur, next;

    // Get the first name.
    cin >> cur;

    // Read through the rest.
    for (int i=0; i<n-1; i++) {

        cin >> next;
        int cmp = cur.compare(next);
        if (cmp < 0)
            up++;
        else if (cmp > 0)
            down++;

        // Update cur.
        cur = next;
    }
    if (up == n-1)
        cout << "INCREASING" << endl;
    else if (down == n-1)
        cout << "DECREASING" << endl;
    else
        cout << "NEITHER" << endl;

    return 0;
}
```

Basically, we always compare the previous string to the next and count the number of the times the comparison goes up or down. Then, if either of our accumulators is n-1, then we output increasing or decreasing. Otherwise, it's neither.

Unfortunately, sometimes strings in input contain spaces or are separated by characters other than spaces. Let us look at how to deal with both of these situations:

*Reading in Strings with Spaces*

In some problems, the input string may have some spaces (say someone's name). In these cases, cin is insuffcent to read in the input because cin naturally tokenizes on spaces. (This means it stops reading once it hits a space.) Thus, we must read in the data one line at a time, and the function that does this is getline. Here is a code snippet of how to read in a whole line of input via getline:

```
string s;
getline(cin, s);
```

The first parameter to getline is where you want to read from. This can be a file or standard input. For competitive programming, this is usually standard input, so cin. The second parameter to getline is the string that you want the input to be read into.

At the end of this statement, the whole line (minus the '\n' character at the end of it) will be stored in s, with spaces and all other characters on the line preceding the '\n' character.

Once you determine that you have to read in whole lines, **READ THE WHOLE INPUT LINE BY LINE!!!** Do not switch between reading lines and reading tokens via cin.

A simple program where getline is necessary is the Kattis problem "The Last Problem". Here is the link:

https://open.kattis.com/problems/thelastproblem

Here, the input is a string, with spaces potentially, and basically, we just have to echo back that string in output, with some other words. Here is the full solution:

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

int main() {

    string s;

    // This is tricky...they say there might be spaces in the input
    // so you have to read in the whole line!
    getline(cin, s);

    // Just echo the string back out in the desired format.
    cout << "Thank you, " << s << ", and farewell!" << endl;
    return 0;
}
```

This is fine and good as long as no "parts" of a whole line need to be parsed, but often times, once you read in a line, you have to "separate out" parts of that line into different pieces of information. Thus, if  line contains separate tokens, each of which are separated by some delimiter (either a space, or a comma, or something of that nature), then we can write a function that takes in the string as a whole line, and returns a vector of strings, representing all of the tokens, delimited by the appropriate character.

For example, consider reading in a time, such as 12:23:16. Here, there are no spaces, but we clearly want to separate out "12", "23" and "16". So, let's look at a function that takes in the string "12:23:16", and the delimiter ':', and returns the vector: ["12", "23", "16"]. This function can be used in any situation similar to this with any input string and any delimiter.

```
vector<string> getTokens(string line, char delim) {

    vector<string> res;

    string cur = "";
    int i = 0;
    while (i < line.size()) {

        // Get to string start.
        while (i<line.size() && line[i] == delim) i++;
        int j = i;

        // Get to string end.
        while (j<line.size() && line[j] != delim) {
            cur += line[j];
            j++;
        }

        // Add this string.
        res.push_back(cur);
        cur = "";
        i = j+1;
    }

    return res;
}
```

Here is the main function that reads in a single string that is a time and echoes back out the three pieces (hours, minutes, seconds):

```
int main() {
    string t;
    cin >> t;
    vector<string> items = getTokens(t, ':');
    cout << items[0] << " hr " << items[1] << " min " << items[2] << " sec" <<
endl;
    return 0;
}
```

Let’s look at the solution to a couple different Kattis Problems that involve strings.

**Kattis Problem: Trik**
In this problem, we are asked to follow which shell is hiding a ball after observing which shells are swapped:

```
using namespace std;
#include <bits/stdc++.h>

int main() {

    // This stores that move a swaps with b, move b swaps with c
    // and move c swaps with a.
    vector<int> moves;
    moves.push_back(1);
    moves.push_back(2);
    moves.push_back(0);

    // Original ordering.
    string orig = "ABC";

    // Read in the swaps that occur.
    string swaps;
    cin >> swaps;

    // Execute the swaps.
    for (int i=0; i<swaps.size(); i++) {

        // Obtain the two positions that are swapping.
        int x = swaps[i] - 'A';
        int y = moves[x];

        // Swap the positions.
        char tmp = orig[x];
        orig[x] = orig[y];
        orig[y] = tmp;
    }

    // Ta da!
    if (orig[0] == 'A')         cout << 1 << endl;
    else if (orig[1] == 'A')    cout << 2 << endl;
    else                        cout << 3 << endl;

    return 0;
}
```

Some key things to notice here: We could store each move A, B, C as a which spot we were swapping with slot 0, 1 and 2 respectively, so moves[0] = 1, moves[1] = 2, and moves[2] = 0, because move A represented swapping position 0 with position 1, and so forth.

Next, when reading in a move, we had to convert the character ‘A’ to 0, which we did by subtracting ‘A’, which naturally subtracts the Ascii value ‘A’, so that for uppercase letters, our new numbers will be 0 to 25, in the order of the alphabet.

## Kattis Problem: Hissing Microphone

This is another simple problem where we are looking to see if an input string has two consecutive lowercase ‘s’ characters. This is easy enough, we can just index into the string.

```
using namespace std;
#include <bits/stdc++.h>

int main() {

    // Get word.
    string word;
    cin >> word;

    // Go slot by slot to see if two consecutive ones store 's'.
    bool hiss = false;
    for (int i=0; i<word.size()-1; i++)
        if (word[i] == 's' && word[i+1] == 's')
            hiss = true;

    // Output accordingly.
    if (hiss)
        cout << "hiss" << endl;
    else
        cout << "no hiss" << endl;

    return 0;
}
```