Module-4: Strings and Math

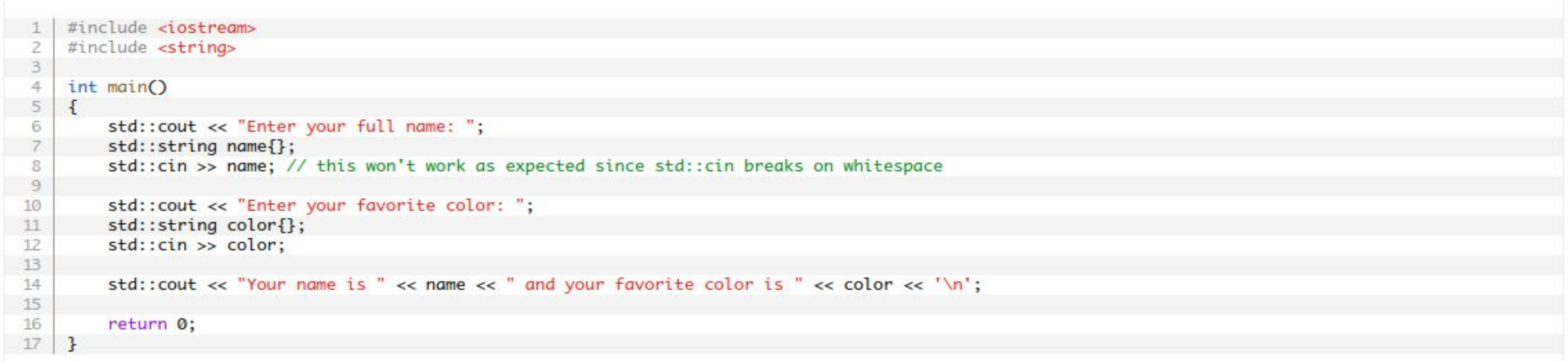
CRUx Bootstrap

4.A: String Datatype

The easiest way to work with strings and string objects in C++ is via the std::string type,which lives in the <string> header.

Just like normal variables, you can initialize or assign values to std::string objects as you would expect.

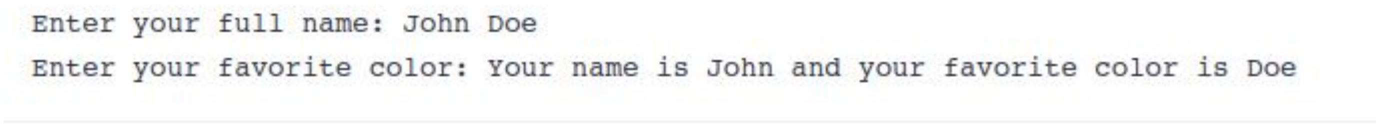
Using string with cin may yield come unexpected surprises!



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

int main()
{
    std::cout << "Enter your full name: ";
    std::string name{};
    std::cin >> name; // this won't work as expected since std::cin breaks on whitespace

    std::cout << "Enter your favorite color: ";
    std::string color{};
    std::cin >> color;

    std::cout << "Your name is " << name << " and your favorite color is " << color << '\n';

    return 0;
}
```

4.A: String Datatype

Here's the result from the sample program:



```
Enter your full name: John Doe
Enter your favorite color: Your name is John and your favorite color is Doe
```

Hmmm,that isn't right! It turns out that when using operator >> to extract a string from cin>>, it only returns characters up to the first whitespace it encounters.

If we want to find out the size of a string s, the syntax is pretty straightforward, we can use s.length(), it returns the total number of characters in s.

4.1: Codeforces 1791A: Codeforces Checking

4.2: Codeforces 1146A: Love ‘A’

4.3: Codeforces 855A: Tom Riddle’s Diary

4.B: Subarrays, subsequences and substrings

As most of you might already know, an array is a collection/list of elements of the same datatype.So by definition a string is an array of characters.

There are different jargons that you will come across while dealing with arrays like subarrays,subsequences and substrings. We will now discuss the difference between the different types of structures:

1. Subarray

- Definition: A contiguous part of an array.
- Properties: Can be as small as a single element or as large as the entire array.
- Example: For arr[] = {1, 2, 3}, possible subarrays are {1}, {2}, {3}, {1, 2}, {2, 3}, {1, 2, 3}.

2. Subsequence

- **Definition**: A sequence derived by deleting some or no elements from an array without changing the order of the remaining elements.
- **Properties**: Not necessarily contiguous.
- **Example**: For arr[] = {1, 2, 3}, possible subsequences are {}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}.

4.B: Subarrays, subsequences and substrings

3. Permutation

- **Definition**: An arrangement of all elements of an array in a specific order.
- **Properties**: Involves reordering.
- **Example**: For `arr[] = {1, 2, 3}`, permutations are `{1, 2, 3}`, `{1, 3, 2}`, `{2, 1, 3}`, `{2, 3, 1}`, `{3, 1, 2}`, `{3, 2, 1}`.

4. Substring

- **Definition**: A contiguous sequence of characters within a string.
- **Properties**: Specific to strings.
- **Example**: For `str = "abc"`, substrings are `"a"`, `"b"`, `"c"`, `"ab"`, `"bc"`, `"abc"`.

5. Set

- **Definition**: A collection of distinct elements.
- **Properties**: No duplicates, order is not guaranteed.
- **Example**: `set<int> s = {1, 2, 3}`

4.4: Codeforces 1689B: Mystic Permutation

4.5: Codeforces 1714B: Remove Prefix

4.6: Codeforces 1703B: ICPC Balloons

4.C: Prime numbers

Prime numbers are frequently used in competitive programming due to their importance in various algorithms and problem-solving techniques.

How do you check whether a number is prime or not?

Primality Test

Sometimes, you are asked to generate prime numbers till n or check if n numbers are prime or not. if you use the above algorithm you will get a time complexity of O(n*sqrt(n)) .

So, how do you generate n primes?

Sieve of Eratosthenes

Here they have just printed it out. But you can store it set or vector according to the problem.

4.7: Codeforces 831A: Factorise N+M

4.8: Codeforces 102267B: Primes

4.9: Codeforces 615 C: Product of Three Numbers

4.D: Binary exponentiation

Binary exponentiation is a trick which allows to calculate a^n using only **O(log n)** multiplications (**instead of O(n)** multiplications required by the naive approach).

The idea of binary exponentiation is, that we split the work using the **binary representation** of the exponent.

For example let's take 3^13 , 13 in binary is **1101**.

$$3^{13} = 3^{1101_2} = 3^8 \cdot 3^4 \cdot 3^1$$

Now , we only need to know 3^1 , 3^2 , 3^4, 3^8 .This we can calculate easily with as each element is just the square of the previous.

$$3^1 = 3$$
$$3^2 = \left(3^1\right)^2 = 3^2 = 9$$
$$3^4 = \left(3^2\right)^2 = 9^2 = 81$$
$$3^8 = \left(3^4\right)^2 = 81^2 = 6561$$

So to get the final answer for 3^13, we only need to multiply three of them (skipping because the corresponding bit in is not set 1101): 6561 * 9 * 3.

The **final complexity** of this algorithm is: **O(log(n))** we have to compute log(n) powers of a , and then have to do at most log(n) multiplications to get the final answer from them.

```
long long binpow(long long a, long long b) {
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a;
        a = a * a;
        b >>= 1;
    }
    return res;
}
```

b >>= 1 . bit shifts b once to the right. (1011 becomes 101)

b & 1 checks if the 0th bit is set or not .
(1001 & 1) becomes 1
(1010 & 1) becomes 0

4.E: finding GCD and LCM

GCD (Greatest Common Divisor) problems can vary in complexity and application, but they generally involve finding the greatest common divisor of one or more integers or applying GCD properties to solve a specific problem.

finding GCD efficiently (Euclidean algorithm)

LCM of two numbers is the smallest number which can be divided by both numbers.

Finding LCM is very easy if you know the GCD and the following property.

a x b = LCM(a, b) * GCD (a, b)

LCM(a, b) = (a x b) / GCD(a, b)

4.F: Modular arithmetic

Sometimes , solutions to problems are so big that the don’t even fit into the 8 bytes of **long long** and overflow**.**

In the kinds of problems you are usually given a modulo such as 10^9 +7 (1000000007).

There are various rules of modular arithmetic that you would need to follow when doing operations on number that can overflow.

Modular Addition:

(a + b) mod m = ((a mod m) + (b mod m)) mod m

Let's say you have to sum all elements of an array

```
vector<int> v ;

int sum = 0 , mod =1000000007;

for( int i: v) {

   sum += i ;

}
```

Instead of this you would be adding like,

```
   sum = (sum+i)%m ;
```

Modular multiplication:

(a x b) mod m = ((a mod m) x (b mod m)) mod m

Let's say you have to multiply all elements of an array

```
vector<int> v ;

int prod = 0 , mod =1000000007;

for( int i: v) {

   prod *= i ;

}
```

Instead of this you would be multiplying like,

```
   prod = (prod*i)%m ;
```

Modular division does not work the same way. Check this article for division and modular exponentiation: Modular arithmetic.

4.G PnC, series and sequences, stuff from JEE

Understanding concepts from PnC, series and sequences, as covered in the JEE syllabus, can be incredibly useful in competitive coding. PnC provides simple and fast formulas to arrange and count items, which is very useful for problems that involve creating combinations or calculating probabilities. Series and sequences come in handy when you need to recognize patterns or find the sum of terms of something like an AP/GP. By mastering these mathematical ideas, you can solve coding challenges more effectively, using logical and analytical thinking.

4.10: Codeforces 1204A: BowWow and the Timetable

4.11: Codeforces 1642C: Division by Two and Permutation

4.12: Codeforces 1352B: Same Parity Summands