C++ Advanced

- Harsh Gupta

Goal

To understand:

- Range of Datatypes
- Setprecision
- Functions
- Headers and Namespaces
- Fast I/O
- Basic C++ Template for CP

Range of Datatypes

2^{34} (5)

int @ = 5

int ✓✓
long long ✓✓
double ✓✓

- **int:** (-2^{31}) to (2^{31-1}) INT_MIN INT_MAX
  - 2^{31} is a bit higher than 2×10^9
- **long long:** (-2^{63}) to (2^{63-1}) LLONG_MIN LLONG_MAX
  - 2^{63} is a bit higher than $9*10^{18}$
- **float / double / long double:** 7 digits / 15 digits / 18 digits

NOTE: **(INT_MAX + 1)** leads us back to **(INT_MIN)** value.

long long int = long long

15.$\boxed{643}$

int → -2^{31} to $2^{31}-1$

int $a = 2^{32}$

2^{32}

INT_MIN

INT_MAX

$a_i \le 2 \times 10^9$

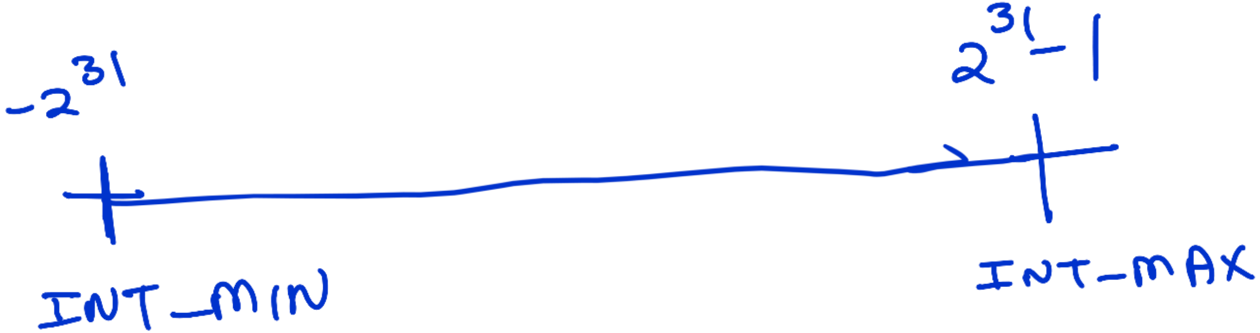
int

$2 \times 10^9 < 2^{31}$

int

$\boxed{\text{int}}$

$$\underbrace{2^{31}-1} - \underbrace{(-2^{31})} + \underbrace{1}_{0} = 2^{32}$$



# of numbers you can represent using int data type $= 2^{32}$

_ _ _ _ _ _ _ _

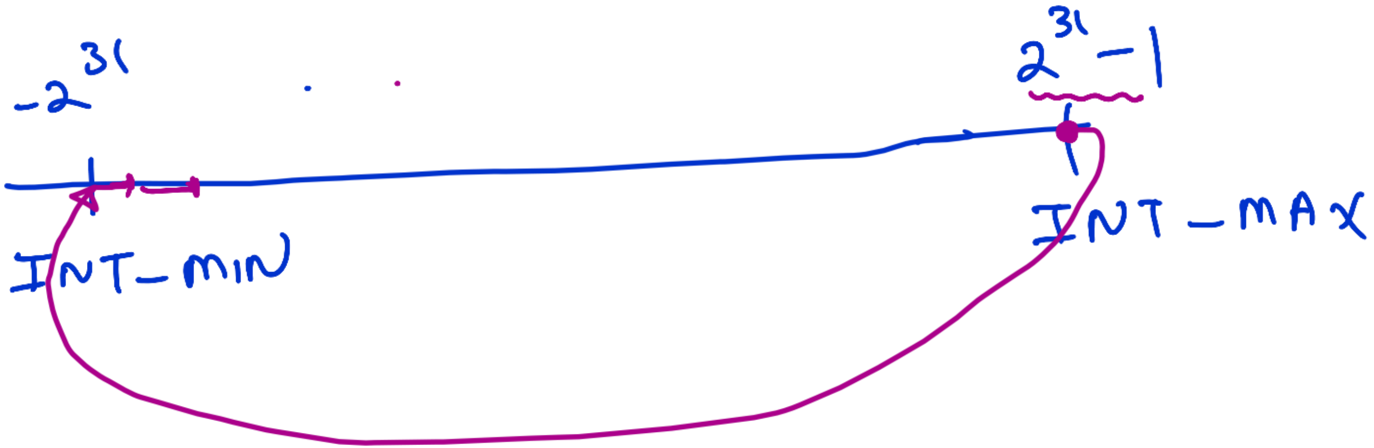
32 bits

2x 2 x

_ _ _ _ 0/1 0/1 0/1

2

32 bits



-2^{31}
$2^{31}-1$
INT_MIN
INT_MAX

Setprecision

Setprecision() method in C++ is an inbuilt method that is used to manipulate floating-point values. It is used to **set the precision of the floating-point numbers after the decimal.**

This function also helps us avoid a **common mistake of printing big numbers** with floating-point data types.

Let us look how?

Setprecision

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

int main() {
    long long bigNumber = 1234567890123456789; // A very large integer
    double floatingPoint = bigNumber;          // Assigning to double

    cout << "Original big number (integer): " << bigNumber << endl;
    cout << "Stored in double without setprecision: " << floatingPoint << endl;
    // 1.23457e+18

    // Print the number with high precision for comparison
    cout << fixed << setprecision(10); //
    cout << "Stored in double with setprecision: " << floatingPoint << endl;
    // 1234567890123456768.0000000000

    return 0;
}
```

Functions

Functions are **reusable blocks of code** that can be run whenever called.

They can take in parameters (input) and return a value (output).

Syntax:

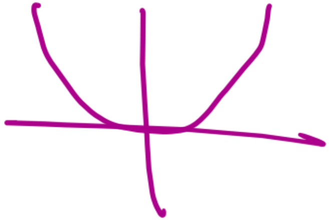
```
return_type name(d1 param1, d2 param2, …) {
    // result must be same as return_type
    return result;
}
```

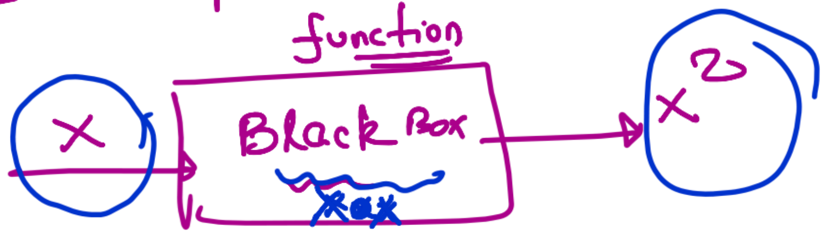
functions → 11, 12

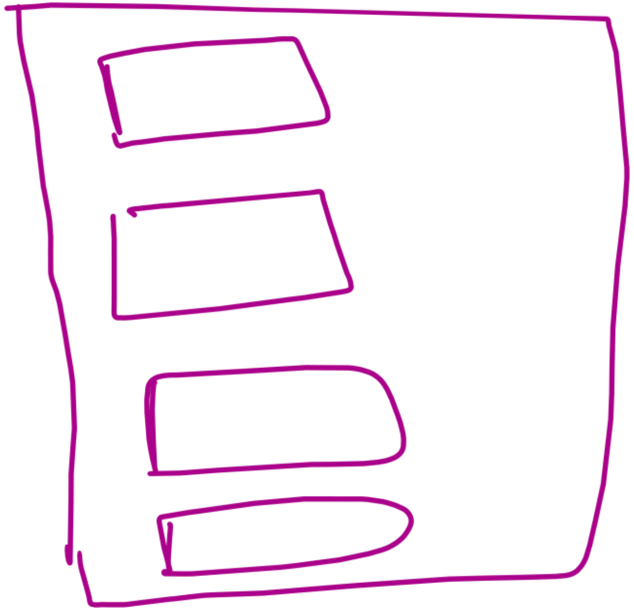
$$\boxed{f(x) = x^2}$$

input

output







x, y

$f(2,3)$ $f(6,2)$ $f(x,y) = \boxed{x \cdot y + 2 \cdot x + y + 40 + x^2}$

$f(4,5)$

Functions

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

int add(int a, int b) {
    int sum = a + b;
    return sum; // This line returns the sum of a and b to the caller.
}

int main() {
    // Function call: We call the add function and pass 3 and 5 as arguments.
    int result = add(3, 5);

    // Display the result
    cout << "Sum: " << result << endl;

    return 0;
}
```

double/ int/ string/char func_name (parameters){

return ;

}

}

Functions

- The **order of parameters passed** is important while function calling.

- Be sure to check and **return value of the required type only.**

- Meaningful function names are encouraged.
    - **Example: addSum, findMax**

Header Files

Header files store C++ variables, functions, etc. to be shared with multiple files.

- **Pre-Existing Header Files:** Files provided by the compiler for a variety of purposes. Example: <bits/stdc++.h>
- **User-defined Header Files:** Files written by the user.
  Can be used for templates, or to make code less complex. Not common in CP.

Syntax:
```
#include <filename>
```

Namespace

A namespace is a **scope of the program** that can store various useful functions and variables.

Two ways to use namespaces:

- Use scope resolution operator “::” (double colon) to use the values inside the namespace.
- Type `using namespace name;` at the start of the file.

Namespaces are used to avoid conflicting names.

Namespace

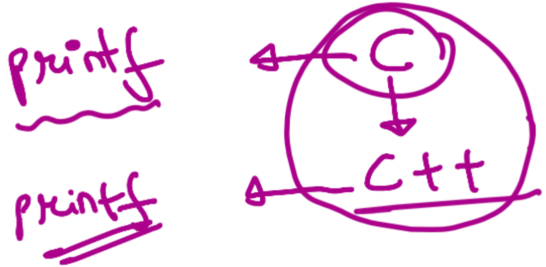
It is clearly obvious that using the **second type of declaration method** for namespaces is better for the context of CP.

```
using namespace name;
```

- Saves time, to write futile code of scope resolution “::” at all places.
- Cleans up our code, easier to debug.

Fast I/O

```
ios::sync_with_stdio(false);
```



Removes sync between “cout” (in C++) and “printf” (in C).

This is to remove the synchronization of I/Os from C and C++ world. If you synchronize, then you have a guarantee that the orders of all I/Os is exactly what you expect, **but that slows down your execution time.**

Let us see an example of this.

Fast I/O

Output:

1

2

3

4

This is expected since we have sync between **“cout”** & **“printf”.**

```cpp
// ios::sync_with_stdio(false);
cout << 1 << '\n';
printf("2\n");
cout << 3 << '\n';
printf("4\n");
```

extra

overload

of maintaining sync b/w cout and printf

∴ It takes some extra time

Fast I/O

Output:

(cursor blinks for input)
Enter number

This is not expected, but happens since we don't have sync between **"cin"** & **"cout"**, giving us **faster I/Os.**

```
cin.tie(NULL);
int a;
cout << "Enter number" << '\n';
cin >> a;
```

Fast I/O

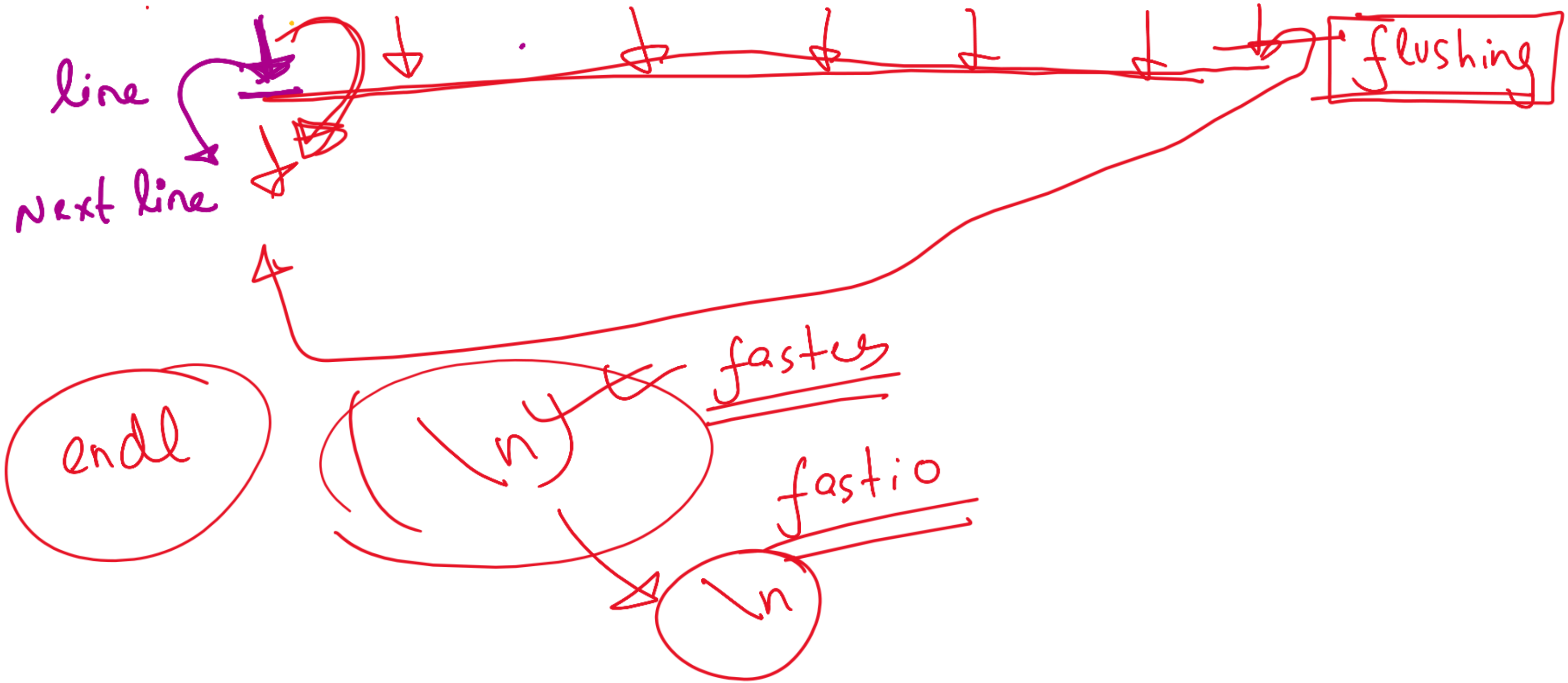
```
endl vs ‘\n’
```

cout << endl inserts a new line and flushes the stream (output buffer), whereas **cout << “\n”** just inserts a new line.

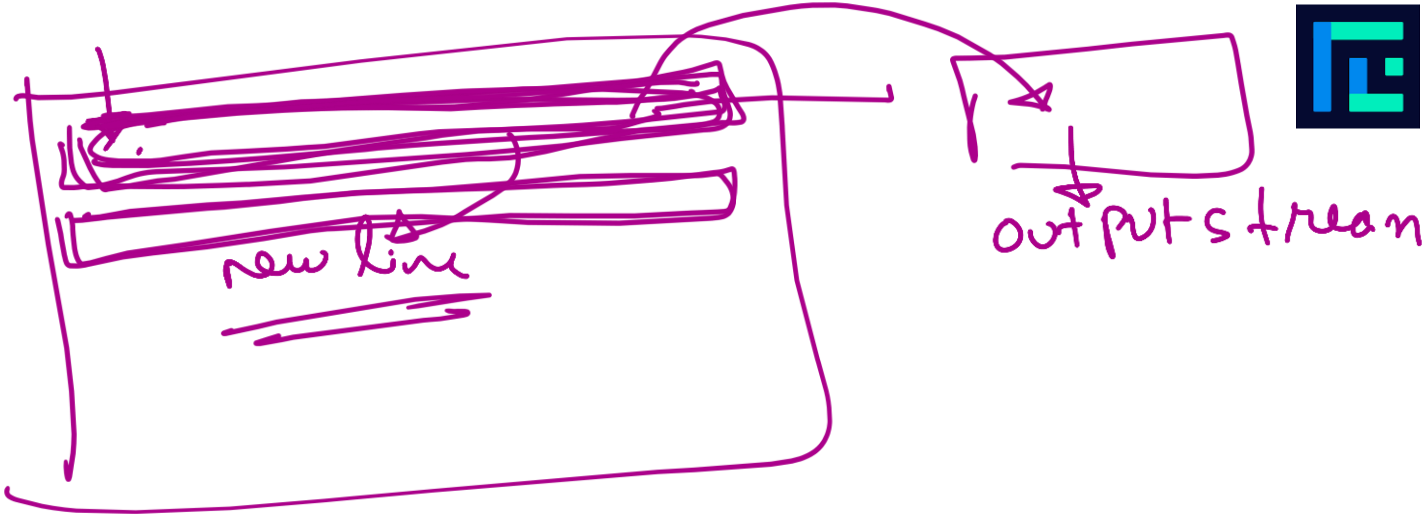
Flushing the buffer in C++ means **clearing the output buffer by forcing its contents to be written** to the output stream **immediately.**

NOTE: When using fastio, use ‘\n’ rather than endl. Let us see why?

endl, '\n' → task is some



\n



\n is faster than endl

Fast I/O

When we run a simple loop of 1000 iterations, we see the output as:

Time taken by function: 7267 microseconds

```cpp
// Get starting timepoint
auto start = high_resolution_clock::now();

for(int i=0; i<1000; i++)
{
    cout << "" << endl;
}

// Get ending timepoint
auto stop = high_resolution_clock::now();

auto duration = duration_cast<microseconds>(stop - start);

cout << "Time taken by function: "
     << duration.count() << " microseconds" << endl;

```

Fast I/O

However, with '\n', we get the following output:

Time taken by function: 67 microseconds

Magic Right?

```cpp
// Get starting timepoint
auto start = high_resolution_clock::now();

for(int i=0; i<1000; i++)
{
    cout << "" << '\n';
}

// Get ending timepoint
auto stop = high_resolution_clock::now();

auto duration = duration_cast<microseconds>(stop - start);

cout << "Time taken by function: "
     << duration.count() << " microseconds" << endl;

```

Basic C++ Template for CP

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

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    int t;
    cin >> t;
    while (t--)
    {
        // code here
    }
    return 0;
}
```