# C++

- **Basic structure in C++**

```
#include <iostream>
int main() {
std::cout << "Hello World";
return 0;
}
```

or

```
#include <iostream>
using namespace std;
int main() {
cout << "Hello World";
return 0;
}
```

* iostream → Header file for input and output
* << → Insertion operator (with cout)
  >> → Extraction operator (with cin)
* cout, cin, endl, ... are functions defined in namespace std, if `using namespace std` is not declared, we must put `std::` before each function defined in namespace std.
* Multiple use of << in single statement ⇒ Operator Overloading

- **Data types**

* int, float, char, bool (true(1), false(0)).

| Data type | Size (bytes) | Range |
|---|---|---|
| short int | 2 | -32,768 – 32,767 |
| unsigned short int | 2 | 0 – 65,535 |
| int | 4 | -2147483648 – +2147483647 |
| unsigned int | 4 | 0 – 4294967295 |
| long int | 4 | -2147483648 – +2147483647 |
| unsigned long int | 4 | 0 – 4294967295 |
| long long int | 8 | $-2^{63}$ to $+(2^{63}-1)$ |
| unsigned long long int | 8 | 0 to 18,446,744,073,709,551,615 |

DATE: PAGE:

| | | |
|---|---|---|
| unsigned char | 1 | 0 to 255 |
| char | 1 | -128 to 127 |
| float | 4 | |
| double | 8 | |
| long double | 10 | |

- **Header files and operators**

Two types of header files

1) System header files
2) User defined header files

As we have seen in C-preprocessor in C

#include <> → System defined

#include " " → User defined

**Operators**

1) Arithmetic operators ( +, -, *, /, %, ++, -- )
2) Assignment operators ( =, +=, -=, *=, %=, &=, |=, ^=, >>=, <<= )
3) Comparision operators ( ==, !=, >, <, >=, <= )
4) Logical operators ( &&, ||, ! )
5) Bitwise operators ( &, |, >>, <<, ~, ^ )

- **Scope resolution operator ( :: )**

If a variable is declared globally and the same variable is also declared locally in a function, if we print that variable inside a function, local variable is given first priority, so the local variable is printed. So, if we want to print global variable, use **scope resolution operator ( :: )**

DATE: PAGE:

```
#include <iostream>
using namespace std;
int x=10;
int main(){
int x=5;
cout<<x<<::x<<endl;
return 0;}
```

prints 5 (local variable) prints 10 (Global variable)

- The default prototype of a decimal number is double and its size is definitely 8. So if we want to manipulate it, we use a suffix f or F for float and l or L for long double.

sizeof(30.1) → 8
sizeof(30.1f) → 4
sizeof(30.1L) → 12

} Called literals
Float, double & long double literals

- Reference variables

Reference variable is another name for an already existing variable (it is just like using typedef for a variable);

```
int x=10;
int &y=x;
cout<<x<<endl;
cout<<y<<endl;
```

x = 10, addr → 64456
y → x
This means x & y will be storing the same address
o/p → 10 10

Here & is not an address of operator, it is just a symbol to represent a variable as reference var

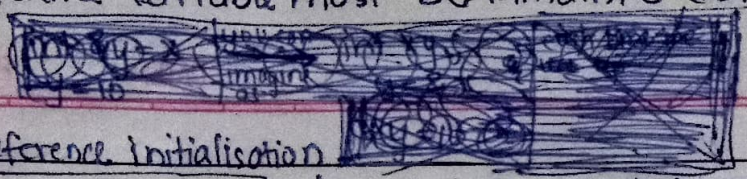
It is just like y points to x

So if you change 'y', 'x' will also change and vice versa.

Both ref var & actual var points to same address

So it is an internal pointer

* Reference variable is an internal pointer.
* Reference variable must be initialised during declaration




* Reference Initialisation

- **Typecasting** — to an object cannot be changed in future to another object

Here (float) a/b <u>same</u> float (a/b)

- **Constants in C++**

We cannot reinitialise a variable if we use 'const' prefic before prototype of variable

```
const int a=10; → Means, a cannot be altered
a = 15; → invalid, shows error.
```

- **Manipulators**

Operators which are used to format output (console I/O)

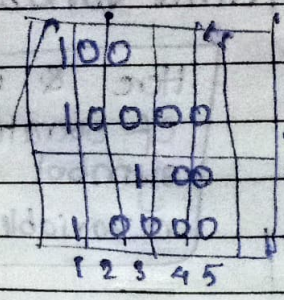
Most commonly used → endl, setw

endl requires <iostream> ; setw Requires <iomanip>

endl → end line
setw( ) → set width

```
cout << 100 <<endl;
cout << 10000 << endl;
cout << 100<< setw(6) <<endl;
cout << 100<< setw(5) <<endl;
```

O/P →




without setw()

with setw()

setw(5) → just like %5d in C

→ Usually variables in function are cleared after function

## Operator Precedence

It is the Heiranchy of operator

| Precedence | operators | | Associativity |
|---|---|---|---|
| 1 | :: | Scope resolution | L-R |
| 2 | a++, a-- | post increment operator | |
| | type(), type{} | Functional cast | R-L |
| | a() | Function call | |
| | a[] | subscript | |
| | ., ->, | Member access | |
| 3 | ++a, --a | Pre increment operator | L-R |
| | +a, -a | unary + & - | |
| | !, ~ | Logical NOT, Bitwise NOT | |
| | *a | value at / indirection / dereference | |
| | &a | address of | |
| | sizeof() | | |
| | co_await | await expression | |
| | new, new[] | Dynamic memory allocation | |
| | delete, delete[] | Deallocation | |
| 4 | .*, ->* | Pointer to member | L-R |
| 5 | *, /, % | | |
| 6 | +, - | | |
| 7 | >>, << | Bitwise right & left shift | |
| 8 | <=> | Three way comparision | |
| 9 | <, <= | | |
| | >, >= | | |
| 10 | ==, != | | |
| 11 | & | Bitwise AND | |
| 12 | ^ | Bitwise XOR | |
| 13 | \| | Bitwise OR | |
| 14 | && | Logical AND | |

DATE: PAGE:

| | | | |
|---|---|---|---|
| 15 | \|\| | Logical OR | |
| 16 | a ? b : c | Ternary/conditional operator | R-L |
| | throw | throw operator | |
| | co_yield | yield expression | |
| | = | | |
| | +=, -= | | |
| | *=, /=, %= | Bitwise compound operators | |
| | <<=, >>= | | |
| | &=, ^=, \|= | | |

- **Control Structures**

1) Sequence structures
2) Selection structures
3) Loop structures

1. **Sequence structure** → The statements get executed one after other from top to bottom (without if/else, loop etc).
2. **Selection structures** → Program with conditional operators, (if, else ...) where selected portions run
3. **Loop structure** → With loop

- **endl v/s \n**

cout << endl ; → Insert a new line & flushes the stream
cout << "\n"; → Only inserts a newline

cout << endl; equivalent to cout << "\n" << flush;

DATE: PAGE:

* endl is a manipulator while \n is a character
* endl doesnot occupy any memory while \n occupies memory

- **Swap values**

* In C, swapping values in a function was only possible using pointers i.e passing variables by call by reference
* But in C++, we can do the same using REFERENCE VARIABLES

```
#include<iostream>
void swap(int &a, int &b)
{   int temp=a;
    a=b;
    b=temp;
}
int main(){
 int x=10, y=15;
 swap(x,y);
 cout<<"x:"<<x<<endl;
 cout<<"y:"<<y<<endl;
 return 0;
}
```

Since reference variables are internal pointers (both variables points to same memory location i.e share same memory) changing 'a' would be same as changing 'x'

- **Return by reference**

Say we want to return 'a' in the above program, we would use function return type as `int &max(int &a,int &b)`

```
int &max(int &a, int &b)
{ if(a>b)
   return a;
  else
   return b;
}
```

```
int main(){
 int x=, y=;
 ;
 max(x,y) = -1
 ;
```

Here x would be assigned -1 if a is returned
y would be assigned -1 if b is returned

BY SHIVKARAMAN

DATE: PAGE:

- **Inline functions**

In a normal function call, when program executes the function call instruction, the CPU stores the memory address of the instruction following the function call, copies the arguements of the function on the stack and transfers control to the called function. The CPU then executes the function code, stores the function return value in predefined memory location/registers and returns control to calling function. This can become overhead when the execution time of called function is very lesser than the switching time from calling function to called function.

Runs program faster

In this case we use INLINE FUNCTION. When the inline function is called whole code of the inline function gets inserted or substituted at point inline function call. Inline function's are not useful for functions which have long codes because here the execution time of function would be larger than switching time.

Declaration → We just use a keyword 'inline' before the function prototype

Eg → inline int sum( );

- We wont use inline functions with
  1) Large function
  2) Recursions
  3) While using static variable

- **Static variable**

The variables whose lifetime is till the program terminates. Memory is allocated to it till lifetime of program

```
void demo() {
Static int count=0;
count++;
}
int main(){
.
.
}
```

STATIC VARIABLES ARE INITIALISED ONLY ONCE. AND THE VARIABLE RETAINS THE VALUE IF ITS ALTERED LATER OR RETAINS THE INITIALISED VALUE

It is default that the life-time of a variable in a function is only till the function is executed (local scope). But when we declare a static variable, memory is allocated for it till the lifetime of the whole program.

In first function call, count = 0
count = 1 (count++)
In 2nd function call, count will not be initialised to 0 again, count will be 1 because it is a static variable

- Default Arguements

```
int sum(int a, int b, int c=10, int d=15)
{ return(a+b+c+d);
}
int main(){
cout<< sum(10,15)<<endl;
cout<< sum(10,15,20)<<endl;
cout<< sum(10,15,20,25)<<endl;
return 0
}
```

O/P
50
60
70

A default argument is a value provided in a function declaration that is automatically assigned if the calling function doesn't provide a value for argument with default value

DATE: PAGE:

- **Function Overloading**

Two or more functions having same name, but having different number of parameters

```
#include <iostream>
using namespace std;
int sum(int a, int b){
    return (a+b);
}
int sum(int a, int b, int c){
    return (a+b+c);
}
int main(){
int x=10, y=15, z=1;
printf("%d", sum(x,y));
printf("%d", sum(x,y,z));
return 0;
}
```

Called FUNCTION POLYMORPHISM IN OOPS

called when 2 arguements are passed

called when 3 orguements are passed

- **Dynamic memory allocation in C++**

| | memory allocation | memory deallocation |
|---|---|---|
| C | malloc() | new |
| | calloc() | |
| C++ | free() | delete |

- **New**

syntax

```
ptr = new data-type;
```

Eg → `ptr = new int;`

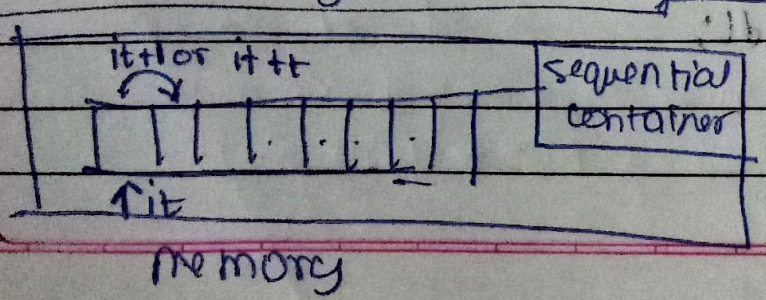
`ptr = new float`

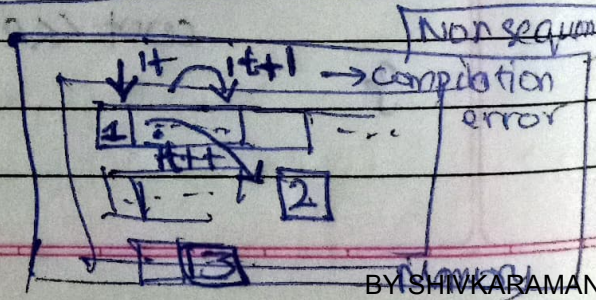
# Iterators

- They are pointer like objects pointing to specific location of containers.
- .begin() → first block
- .end() → block next to last block
- The difference b/w it++ and it=it+1
  - it++ → points to next iterator
  - it=it+1 → moves to next location

In vectors and other sequential containers
it++ <=> it=it+1
Because they are sequential

In other containers like sets and maps.
They are not sequential
so it=it+1 → Compilation err



## Auto keyword and range based loops

```
for (int it = v.begin(); it != v.end(); it++)
    cout << *it << " ";
```

data type of vector

```
for (int value : v)
    cout << value << " ";
```

→ What happens here is, the loop iterates over the entire range of vector one by one and copies it to value and printed inside loop.

* We can use (int& value : v) instead to avoid copying

* auto → Determines automatically the data-type of variable.

DATE: PAGE:

## Time complexity and constraints

No of itrations in online editors → $10^7 - 10^8$ per second

∴ Solutions the

> No of iterations in our solutions per second must always be $<= 10^7$

eg. N <= 100000
a[i] <= 100000

```
for( i=0; i<n; i++)
    cout<<a[i]<<endl
```

T(n) = O(n)
≈ $10^5$ < $10^7$
NO ERROR

```
for( i=0; i<n; i++)
  for(j=0; j<n; j++)
    cout << a[i] <<endl
```

T(n) = O($n^2$)
≈ $10.10$ > $10^7$
TLE

eg: Constraints:
1 <= T <= 1000
1 <= N <= 1000
1 <= a[i] <= 1000

⇒ Solution can be max of $n^2$ complexity

```
int main (){
    int t;
    cin >> t;   int count=0;
    while (t--){
```

```
int n;
cin >> n;
int sum = 0;
for( int i=0; i<n ; i++){
    int x;
    cin >> x;
    sum += x;   count ++;
}
cout << sum << endl;
}
}
```

$T(n) = O(NT)$

$\approx O(10^6)$ $< 10^7$

Now if we change the constraints as below

$1 <= T <= 10^5$

$1 <= N <= 10^5$

$1 <= a[i] <= 1000$

Sum of N over all test cases is $< 10^7$

Special case

Now $T(N)$ = No of times count was incremented

= cnt

| time | cnt |
|---|---|
| t=0 | 0 |
| t=1 | $n_1$ |
| t=2 | $n_2$ |
| ⋮ | ⋮ |
| $t_N$ | $n_N$ |

## Range of data types

| | |
|---|---|
| Int → | $-10^9$ to $10^9$ |
| long int → | $-10^{12}$ to $10^{12}$ |
| long long int → | $-10^{18}$ to $10^{18}$ |

Though double has a range very much greater than long long int, we avoid using it coz it is not precize.

Eg
```
long long x = 1e10;
cout << x << endl;
```
Output → 10000000000

```
double x = 1e24;
cout << x << endl;
```
O/p → 99999999.... 324.000

↓

Not exactly printing 1e24, prints a number very near to 1e24. → May cause errors due to wrong calculations.

```
int a = 100000;
int b = 100000;
int c = a*b;
```
→ overflow

```
long int d = a*b;
```
→ No overflow but answer wrong because a and b are int, a and b will be multiplied and the final ans will be again converted into 'int' and then stored into d.

```
long int d = a* 1LL *b;
```
→ No overflow correct answer

# Strings

getline function → `getline(cin, str);` → Takes input from entire line

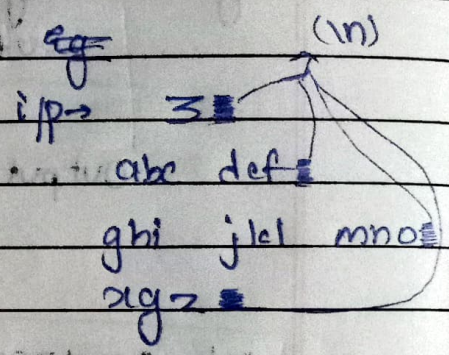
But there is a catch here

cin → It skips all spaces (blanks) and newline characters (\n) before taking input.

getline → It starts taking input from wherever the current position of pointer is.

Eg:

```
int n;
cin>>n;
while(n--){
    string str;
    getline(cin, str);
    cout << str << endl;
}
```




Because, after taking 3 as input, the getline takes \n as the first string.

To avoid this we can use `cin.ignore()` after `cin>>n;` to ignore the '\n'

## Adding a character at end of string

1. `s+= 'char'` → O(size) → Copies string same then adds a 'char' and then stores it.
2. `s.push_back('char');` → O(1)

Best approach → `s.push_back(char)`

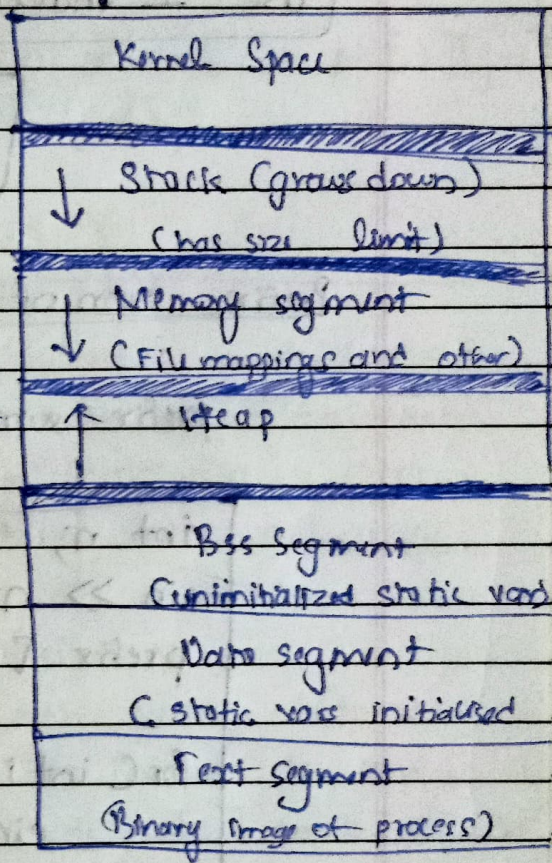
- **Array size limit**

  locally → $10^5$
  globally → $10^7$

While declaring array globally say `arr[n]`, n should be constant ie `const int n;`

**Why ?**

This is because, whenever, a variable is declared locally, it is stored in Stack which has a memory limit.
But when it is declared globally, it is created in Data segment which has no limit.




- **Modular Arithmetic**

1. $(a+b)\%M = ((a\%M) + (b\%M))\%M$
2. $(a*b)\%M = ((a\%M) * (b\%M))\%M$
3. $(a-b)\%M = ((a\%M) - (b\%M) + M)\%M$
4. $(a/b)\%M = ((a\%M) * (b^{-1})\%M)\%M$

$b^{-1}$ → Multiplicative inverse.

WATCH LUV [redacted] VIDEO NO 17

- **Vectors**

* vec.pop_back(); ⟶ O(1)
* vec.push_back(val); ⟶ O(1)
* vec.size() ⟶ O(1)

. You can simply copy vectors just like normal data types but that internally uses a loop to copy element by element

vector<int> v1 = v2; ⟶ O(n)

* vec.erase(val) ⟶ O(n)
  vec.clear() ⟶ O(n)

↳ This is because, deleting one element requires moving other elements to fill up the gap

- **Iterators:**

They are pointer like objects used to point to some entity

> Difference b/w `it++` and `it = it + 1`
> it++ ⟶ Points to next iterator
> it = it + 1 ⟶ Points to next location in memory

In vectors `it++` ⟷ (same as) `it = it + 1` because next iterator itself is the next location (at it has contigious blocks of memory)

In maps and sets → we must use `it++`

Red Black Trees → Self balancing binary tree which always maintain hight $\log n$

DATE: PAGE:

## maps

Maps key → value pair

`map<key, value>` → Values are stored in sorted order of keys.

Internally implimented using RED BLACK TREES

Two ways to init m. map → ① `mp[key] = value`
② `mp.insert({key, value})`

| | | |
|---|---|---|
| `m.insert({key, value})` | → $O(\log n)$ | n → curr size of map |
| `m.erase(key)` | → $O(\log n)$ | |
| `m[key]` or `m.find(key)` | → $O(\log n)$ | |
| `m.empty()`, `m.size()` | → $O(1)$ | |
| `m.clear()` | → $\Theta(n)$ | |

`m.insert({key, value})` → Not always $O(\log n)$.
When something is inserted into red black tree, the element is searched it is ~~always~~ and compared with already existing elements and then gets inserted at correct position.

∴ T(insert) = T(search) + T(comparision)

~~`m.insert({key, value})` → O(log n)~~

∴ `m.insert(S k, V3)` = T(comparision) · log n

↓
Negligible for small input

BY SHIVKARAMAN

eg. map <string, string> mp;

mp["abc"] = "abc"; → size small ∴ O(log n)

mp["abcdefg ------ z"] = "abc" → Here size is large.

∴ TC m.insert() = TC(comparision) · log n

= size · log n

= O(size · log n).

- **Unordered maps**

unordered_map <key, value> → values are stored in random order

Internally implimented using inbuilt HASH TABLES (key is hashed)

m.insert({key, value}) → O(1)

m.erase(key) → O(1)

m.find(key) → O(1)

Rest everything is same.

★★

- Another big difference is you cannot create unordered_map of complex data type of key (such as map <pair<int, int>, value>) because there are no inbuilt hash function to hash the value of complex data type.
- But it will work in normal map because the key is not hashed, it is just compared with others in tree ie it is comparision based insertion.

Valid key data-type → all basic data types (int, float, string...)

- **multimap**

  multimap <key, value> → Everything is same as of maps, but the only difference is you can insert multiple pair<key, value> for same key i.e it allows duplicate values.

- **Sets**

  set <key> → set of values/elements,
  → Stores unique elements in sorted order

  Time complexity → Same as in map

- **unordered set**

  unordered set <key> → Stores unique elements in random order

  Everything same as unordered_map

- **multiset**

  multiset <key> → Allows ~~multiple values~~ insertion of elements with same value i.e allows duplicates.

  s.find(key) → Returns iterator to first occurance of key (if duplicates exist)
  s.erase(iterator) → Deletes element pointed by iterator
  s.erase(key) → Deletes all duplicates.

  ↓
  all duplicates are not deleted.