C++ programming

```
#include <iostream>
using namespace std;
int main() {
    // your code goes here!
    cout << "Hello world";
    return 0;
}
/* 1st program */
```

- `#include <iostream>` → including libraries or header file
- `using namespace std;` → standard functions
- `int main() {` → main function
- `cout << ____ ;` → insertion operator
- `// your code goes here!` → single line command
- `cout << "Hello world";` → printing output (or) displaying
- `return 0;` → EOL
- `}` → EOL
- `/* 1st program */` → multiple line command

Note:-
main()
starting line
of program

* escape sequence:-

\n → next line

\t → space separate or tab separate

\" → To print double quotes in print statement.

\a → producing sound (variable)

Note:- if not "using namespace std;" on head line

```
cout << ____ ;   // not work raise error
std::cout << ____ ;   // work finely
```

must mention if std is imported.

* moving 2nd line:-

```
cout << "\n";   // 1ST method.
cout << "\n" << endl;  // 2nd method
```

* valid insertion operators:

```
cout << "\n" << "\n";
```

" work finely print in one - line"

eg:-

```
cout << "I AM" << 25-7 << "old";
```

op:-

I AM 18 old

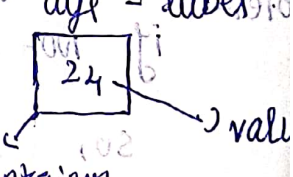
* variables:-

↳ basic container to store data

→* labeling a name to variable is known as identifier

```
int age;  // declaration
age=24;   // initialisation or assign a value
```

Data types <integer>




```
int age=25;  // declaration + initialisation;
```

```
cout << age;   op:- 25
```

↳ display variable

* rules of naming variables :-

-> letters, underscore, and numbers only allowed.

-> case sensitive Abc ≠ abc

-> numbers are not allowed in the starting of the variable.

-> name must not be key word (special meaning word)

-> name not contain space in-between.

---

* declaring multiple variables :-

int num1, num2;

also, int num1 = 1, num2 = 2;

also,

-> num1 = num2 = 9;

---

* constant :-

ex:- const int dob = 2000;

const -> variable can't change. [we can't change]

must initialize in decl

if try raise error as "read only variable"

* data types :- & input :-

int age = 24;

cin >> age;

extraction operator

-> printf(cout << "enter ur age :";

* float = decimal points [5 digit precision]
* double = decimal points [store digit above 5 precision]

ex:-
```
float pi = 3.141516478;
cout << fix pi;
```
OP:-

3.141512
└→ round off to 5 digit precision.

```
double pi = 3.141516478;
cout << fixed << setprecision(10); }
cout << pi;
```
OP:-
3.141516478

To set precision is must to print all digits

must give
```
#include <iomanip>
```
to use setprecision function

* bool - true or false
  - true → 1
  - false → 0

ex:-
```
bool isadult = true;
cout << isadult;
```
OP:-
1
└→ for true

* char - storing single character data type.

```
char gender = 'M';
cout << gender;
```

OP:-
M

```
char gender = 65;
cout << gender;
```

OP:-
A

[ASCII value

---

* string (basic) = "store words para-sentences"

※ must = #include <string> -> header file.

```
string greeting = "Hi Hi Hi";
cout << greeting << endl;
```

OP:-
Hi Hi Hi

were:-

primary data types

- INT - 4 bytes (or 2 bytes for old compilers)
- float - 4 bytes
- double - 8 bytes
- bool - 1 byte
- char - 1 byte

secondary data types.

- string - series of char bytes

Note:- size of (variable_name).

OP:-
size display in bytes

ex:-

```
int age = 24;
float int gpa = 9.56;

cout << size of (age);
cout << size of (gpa);
```

OP:-
4 -> int size
4 -> float size

* Arithmetic operations :- (+, -, *, /, %, ++, --)
↓ ↓
increment decrement

ex:-

```
int main() {
    int a=5;
    int b=6;
    int sum = a+b;
    int diff = a-b;
    int product = a*b;
    cout << "sum :" << sum << endl;
    cout << "diff :" << diff << endl;
    cout << "product :" << product << endl;
```

OP:-

sum : 11
diff : -1
product : 30
~~div : 0.833333~~

for divisision :- [a=5, b=6]

```
if int div = a/b;
if float div = a/b;
```

So,

```
float div = ~~(float)a/b~~;
          = float(a)/b;
```

OP:-
0
0.83333

for modulus :- [give reminder] [a=15, b=6]

```
int modu = a%b;
```

OP:-
3
↳ reminder

* conversion:-

-> int a=6.7; // implicit conversion.

op:-
6

-> also;

float b=6.8;
cout<< int(b); // explicit conversion.

op:-
6

---

* Increment & decrement:

int a=15;
a++;
cout<<a;

op:-
16 // incremented 15 -> 16

also;

a--;
cout<<a;

op:-
15 // decremented 16 -> 15

Types:-

-> pre increment
-> post increment
} same for decrement.

a++;

++a;

* Assignment operator :- [= , += , -= , /= , %=]

```
int n1;
n1 = 25
n1 += 1;   ---> n1 = n1+1
n1 -= 1;   ---> n1 = n1-1
```

Short form — expansion

---

* comparison operator :- [== , != , <= , >= , > , <]

```
int n1 = 9, n2 = 8;
cout << (n1 > n2);   | op:- 1 = "true"
cout << (n1 < n2);   |      0 = "false"
cout << (n1 == n2);  | 0
cout << (n1 != n2);  | 1
```

---

* logical operator :- [AND , OR , NOT]

basic :-

a + b
- a, b → operands
- + → operators

so,

※ operands for logical operator is always (0 or 1) means (true or false)

AND , OR = two operands

NOT = one operands.

Diagram or truth table representation:-

AND

I_1	I_2	O
1	1	1
0	1	0
1	0	0
0	0	0

OR

I_1	I_2	O
1	1	1
0	1	1
1	0	1
0	0	0

NOT

I_1	O
1	0
0	1

Here, AND = '&&', OR = '||', NOT = '!'

```
cout << (true && true);          OP:- 1
cout << (true || true);          1
cout << (! true);                0
```

also,

```
gender = 'M'
age = 24
cout << (gender == 'M' &&
         age >= 18);             OP:- 1

cout << (gender == 'M' &&
         age == 18);             0

cout << (gender == 'M' ||
         age == 18);             1
```

* strings (basics) :-

```
#include <iostream>
#include <string>  -> string header file
using namespace std;

int main(){
    string name = "Croms";
    string lname = "A";
    cout << name + lname;
    cout << name + "xyz" + lname;
```

[concatination]

OP:-

CromsA

Croms xyz A

#method 2 of concatination:

```
cout << name.append(lname);
```

※ [were, name was changed to CromsA]

name = "CromsA"

-) length of string :- [syntax: string.length()]

```
cout << name.length();
cout << name.size();
```

} same result.

-> accessing of strings:

0	1	2	3
L	O	K	I
-4	-3	-2	-1

```
string name = "LOKI";
cout << name[0];
```

OP:-
L

also,

```
cout << name.at(0);
```

OP:-
L

-> updating a strings:-

```
string name = "LOKI";
name[0] = 'M';
```

└→ singl quotes

```
cout << name;
```

OP:-
MOKI

-> getting string from user input:-

```
string name;
cout << "enter ur name : ";
cin >> name;
cout << name;
```

OP:-
Enter ur name : LOKI A
LOKI

Space after ignored

like: [applicable for int]

```
int n1, n2;
cout << "enter two num:";
cin >> n1 >> n2;
cout << n1 << endl << n2;
```

OP:-
enter two: 23 45
23
45

for [string]:

```
string fullname;
cout << "enter name:";
getline(cin, fullname);
cout << fullname;
```

OP:-
enter name: LOKI A
LOKI A

---

* conditional statements:- [syntax: if(condition){ }]

```
int age = 52;
if (age < 18)
{
    cout << "your child" << endl;
}
else if (age > 60)
{
    cout << "your senior citizen" << endl;
}
else
{
    cout << "your are an adult" << endl;
}
```

op:-

your are an adult

nested if statements :- [leap year]

example :-

```
#include <iostream>
using namespace std;
int main() {
    int year;
    cout<<" enter ur year ";
    cin >>year;
    if (year%100 == 0){
        if (year%400 == 0) {
            cout<<" leap year ";
        }
        else {
            cout<<" not leap year ";
        }
    }
    else if (year%4 == 0){
        cout<<" leap year ";
    }
    else {
        cout<<" not leap year ";
    }
    return 0;
}
```

op:
enter year: 2016
leap year

op:
enter year: 1900
not leap year.

* ternary operators :- [if statement in one line]

(condition) ? #statement1 ; #elsestatement1 ;

ex:-

```
int age=14;
(age<18) ? cout<<"child" : cout<<"adult";
```

op:-
child

* switch-case [statement] [alternate of if-else] :-

example:-

```
int day=2;
switch(day){
    case 1:
        cout<<"monday";
        break;
    case 2:
        cout<<"tuesday";
        break;
    case 3:
        cout<<"wednesday";
        break;
```

if break no given in case then, it will fall back and execute each case after satisfy condition.

must give in each case

```
case 4:
    cout<<"friday";
    break;
case 5:
    cout<<"saturday";
    break;
dyault;
    cout<<"sunday";
    break;
```

└ not nuzzzary for dyault "

```
}
return 0
}
```

---

* looping :- [while, do-while, for]

```
-> #include<iostream>          // while-loop
   -uzing namespace std;

   int main(){
       int n=5;
       int i=1;        -> initialisation
       while(i<=n){    -> condition - 2
           cout<<i;<<endl;
           i++;        -> incrementation - 3
       }
```

```
#include<iostream>          // do-while
using namespace std;
int main() {
    int n=5;
    int i=1;          -> initialize -1
    do {
        cout<<i<<endl;
        i++;          -> increment -2
    } while(i<=n);          -> condition -3
    return 0;
}
```

"while-loop"
-> * [entry control loop]

"do-while"
-> * [exit control loop]

---

* for loop :-

syntax:-

```
for(initialize; condition; incrementation)
{
    // # statement to run;
}
```

```
#include <iostream>
using namespace std;
int main(){
    int n=5;
    for(int i=1; i<=n; i++)
    {
        cout<<i<<endl;
    }
    return 0;
}
```

* nested loops :-

```
int main(){
    int n;
    cout<<"enter num:";
    cin>>n;
    for(int i=2; i<n; i++)
    {
        int c=0;
        for(int j=2; j<i; j++)
        {
            if(i%j==0){
                c=1
                break;
            }
        }
        if(c==0){
            cout<<i<<" ";
        }
    }
    return 0;
}
```

* Continue & Break statements:

- Break statement used to Break the loop or iterations of condition.
- continue used to skip the iteration of the loop.

Break:- your nested loop program.

continue:-

```
int main() {
    int n=8;
    for(int i=0; i<=8; i++)
    {
        if(i%2!=0)
            continue;
        else
            cout<<i<<" ";
    }
    return 0;
}
```

OP:-

0 2 4 6 8

math functions :- [import or include <cmath>]

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

int main() {

    int n1=5, n2=7;

    cout << max(n1,n2) << endl;
    cout << min(n1,n2) << endl;

    cout << sqrt(81) << endl;
    cout << pow(9,3) << endl;
    cout << round(5.4) << endl;
    cout << ceil(5.4) << endl;
    cout << floor(5.6) << endl;
    cout << log(2) << endl;

    return 0;
}
```

op:-

```
7
5
9
729
5
6
5
0.693147
```

* arrays :-

- To store similar data in one variables.

syntax:

```
data-type variable[size] = {
                values, ... };
```

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

int main(){
```

```
string name = "goms";
```

//same data

```
string names[5] = {"goms", "siva", "code IO",
                   "logic IO", "undefined" };
string agus[5] = {24, 21, 4, 1, 0};
cout << name << endl;
cout << names << endl;
cout << agus << endl;
```

op:-

goms

0x6ff dd0] memory starting

0x7ff d440] address

so, -> accessing the elements of array by indexes :-

```
cout << names[0] << endl;
cout << agus[0] << endl;
```

op:-

goms - 1st element

24 - 2nd element

so,

-5	-4	-3	-2	-1	} reverse index.
goms	siva	codio	logic io	undefined	

24	21	4	1	0	
0	1	2	3	4	} forward index

Note:-

* size is not manditory if declaration & ~~Initiali~~ initialisation take place in same line:.

* size is manditory if declaration occur without initialisation.

→ printing of arrays :-

```
for (int i=0 ; i<5 ; i++)
 {
   cout<< name[i] << ":" << ages[i]
                  << endl ;
```

OP:-

```
gomz : 24
siv : 21
codio : 4
logicio : 1
undyined : 0
```

→ finding size of arrays:- (or) length of arrays :-

```
int size = sizeof(ages)/ sizeof(ages[0]) ;
cout<< size ;
```

5

ex:-

= 20/4 = 5 = no. of element

Total size / Size of element.

* special for-loop for arrays:-

```
for ( int i : agus )
{
    cout<<i<<endl;
}
```

op:-

24
21
4
1
0

---

* two dimensional & multidimensional array:-

```
int main() {
//2D array or matrix
int arr2[3][3] = { { 1, 2, 3 },
                   { 4, 5, 6 },
                   { 7, 8, 9 } };
```

row column

face or collection of matrix

```
int arr[3][3][2] =
{ { { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 } },
  { { 10, 11, 12 },
    { 13, 14, 15 },
    { 16, 17, 18 } }
};
```

* printing or display 2D array :-

```
int row= sizeof(arr2)/sizeof(arr2[0]);
int col= sizeof(arr2[0])/sizeof(arr2[0][0]);

for(int i=0; i<row; i++)
{
    for(int j=0; j<column; i++)
    {
        cout<<arr2[i][j]<<" ";
    }
    cout<<endl;
}
```

output:-

```
1 2 3
4 5 6
7 8 9
```

} matrix format display

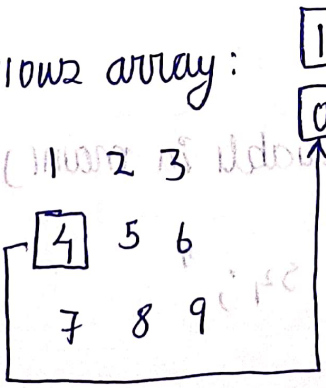
---

* access & update elements in 2D arrays :-

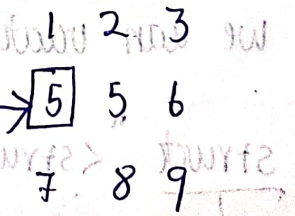
```
arr2[1][0] = 5;
~~cout<<arr2~~ // display array
```

(row, column)

previous array:



updated array:



* structures:

↳ storing a related data is called structures. (or custom data type)

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

Struct ——) can specify struct name ; [mandatory for different struct]

```
{ //structure definition.
    string name;
    int age;
    float CGPA;
} S1, S2; // structure variables.
```

} structure members.

```
int main() {
    S1.name = "Loki";
    S1.age = 18;
    S1.CGPA = 8.6;
    cout << S1.name << " " << S1.age << " "
         << S1.CGPA << endl;
```

also,

we can create structure variable in main()

```
struct <struct_name> S3, S4;
```

[not mandatory]

* array of structures :-

```
<struct_name>  class[0 [30];
```

* Enums :-

→ collection of constant or never change variable.

```
enum Day {
    sunday,     // 0 - index
    monday,     // 1 - index
    Tuesday,    // 2 - index
    wednesday,  // 3 - index.
};

int main() {
    Day today = Tuesday;
    cout << today << endl;        → //output :
                                  2 -> index.
    cout << (today+1) << endl;

    if (today == Day::sunday)     → //output :
                                  3 -> index
        cout << "Holiday" << endl;
```

* POINTER & reference :

↳ alias name

```
int main() {
    int agi = 24;
    int &aliasagi = agi;
    cout << aliasagi << endl;
```

→ reference of agi variable [not copy]

address or reference address

output: 24

Difference b+w copy & alias :-

```
int agi = 24;
int &aliasagi = agi;
int copyagi = agi;
agi = 25;
cout << aliasagi << endl;
cout << copyagi << endl;
```

output:
25
24

alias

copy

Note:-

aliasagi is not new variable
it just another name of variable agi

cout << &age << endl; → //output
78H21E5
1) Hexadicimal value which is address of age

So,
* &<variablinam> → not store address of another variable, it access address of another variable.,

---

* POINTERS :-
storing a address of another variable. is called pointers.

```
int main() {
    int age=24;
    int* Ptr=",   //declaring of pointer
                  ) capable of store address
    Ptr = &age;
    cout << "Ptr" << Ptr << endl;
```

//output
78H21E5
↓
address of age

```
    cout << "value" << *Ptr << endl;
```

//output
24
↓
value

```
    *Ptr = 56;
    cout << age << endl;
```

//output
56

change value by pointer.

* Dynamic memory allocation :- (run time memory allocation)

```
int main() {
    int agi = 24;   // complie time memory allocation
    int *ptr;
    ptr = nuw int;  // Run time allocation
    *ptr = 24;
    cout << * ptr;
```

#output
24

nuw - keyword
which allocate memory size of int;

-> creating array by dynamic memory allocation :-

```
int main() {
    int num;
    cout << "enter num of students :";
    cin >> num;
    int *arr = nuw int[num];
    for (int i=0; i<num; i++)
    {
        cin >> arr[i];
    }
    for (int i=0; i<num; i++)
    {
        cout << arr[i] << endl;
    }
    return 0;
}
```

note :-
memory are stored in RAM

* Functions: (code reusability)
  - declaration *
  - definition *
  - calling *

```
#include<iostream>
#include<string>
using namespace std;
void add5(int a);----->// declaration
int main(){
    add5(15);---->// calling.
    return 0;
}
void add5(int a){
    cout<<a+5;
}
```

(nothing input) [nothing return]

Arguments

parameter

->// definition

Note :-
for return any value
we must to use specific datatype
instead of void like int <func name>
if return value is int

* Default arguments & call by reference :

```
void add5 (int n=5)
{
        cout << n+5 << endl;
}

int main() {
        add(15); ——> // O/P:- 20
        add(); ——> // O/P:- 10 -) default =5
```

also,

```
void add5 (string grut, int n=2)
{
        cout << grut << " " << n+5;
}

int main() {
        add5("Hi", 6); ——> // O/P:- Hi 11
        add5("Hello"); ——> // op:- Hello 7
```

---

if :-

```
void add5(int n=2, string grut)   ✗
```

// error because default argument must give in last of the parameter

* call by value & call by reference:-

[call by value]

```
void add5(int num)
{
    num += 5;
    cout << num << endl;
}

int main(){
    int num=6;
    add5(num);
    cout << num << endl;
    return 0;
}
```

```
//output
11
6
```

were, num is not changed

[call by reference]

if void add5(int &num)

Then,

were, num value changed.

```
//output
11
11
```

✳

were, if array was pass it reflect the main() array without reference because array was created by reference address.

```
void array(int arr[])
{
    arr[0] = 89;
}
int main(){
    int arr[] = {1,2,3,4};
    array(arr);
    cout<<arr[0]<<endl;
    return 0
}
```

//output
89

arr[0] was changed in main().

---

* Function overloading :-

```
int sum(int a, int b)
{
    return a+b;
}
int sum(int a, int b, int c)
{
    return a+b+c;
}
double sum(double a, double b)
{
    return a+b;
}
```

```
int main() {
    cout << sum(2,3) << endl;
    cout << sum(2,3,6) << endl;
    cout << sum(3.5, 3.5) << endl;
    return 0;
}
```

// output:-

5

11

7.0

* if we give another:-

```
int sum (int a, int b) {
    return 3*a+b;
}
```

it raise error known as function overloading

because, were, sum with int a & int b are exist so, it raise error. we must give different data type as parameter or different function name.

Scope of variables:-

```
#include <iostream>
using namespace std;
int a = 56; // Global variable.
int print() {
    int a=6 // local variable
    cout<<a<<endl;
}
int main() {
    print();
    cout<<a<<endl;
    return 0;
}
```

#output
6
56

if we comment a=6 in print()

```
//int a=6
```

So,

#output
56
56

priority

local variable ✓ global variable

Local variable:- It's a variable which declare in the specific function or block of code and use only in that block code.

Global variable:- It's declared begging of the code which can accessable by every function, main() or entire program.

* Recursion :- [calling a function in the same function which work like loop concept].

example:

```
int fact(int n) {
    if
    return n * fact(n-1);
}
```

flow:

5 x fact(4) = 120

24 = 4 x fact(3)

3 x fact(2) = 6

2 = 2 * fact(1)

1 => return 1

*
```
int fact(int n) {
    if(n==1) {
        return 1;
    return n*fact(n-1);
}
```

base case - condition

function calling

Class & objects :

Oops = Object oriented programming
- functions (methods)
- act on data

under object (real world objects)

main purpose - DRY [Don't Repeat yourself]

most common terms :

class = blueprints / templates
object = instance

class generally contain attributed & methods (functions) also, access specifier.

use capz letter in first for naming class (better)

```
* Declaration :
  #include <string>
  #include <iostream>
  using namespace std;

  class Student {
      public :  ——> access specifier
          int roll_no;
          string name;
          void display ( );
  };
```

class function / method declaration

```
int main() {
      Student s1;
      s1. roll_no = 11;
      s1. name = "Loki"
}     s1. display ( );
```

for class function

specify method with respect class.

```
void student::display(){
        cout << "Roll_no: " << roll_no << "name:" <<
                                    name << endl;
}
```

→ definition of method of class in outside of the class.

※ "::" scope resolution operator

* if we add another object in main program

```
student s2;
s1.roll_no = 12;
s1.name = "YUJI";
s2.display();
s1.display();
```

>>> output:

```
Roll_no: 11  name : LOKI
Roll_no: 12  name : YUJI
```

* Constructors and destructors:

→ - special function
- called when object is created
- name same as class
- it doesn't have return type.

↓

Three types:
- default constructor
- parameterized constructor
- copy constructor

* self_created_constructor:

```
class Car {
    public:
        string brand;
        double engine;
        Car(){
            cout << " Constructor for car " << endl;
        }
};

int main(){
    Car swift;
    swift.brand = "Swift";
    swift.engine = 1.2;
}
```

This is self created constructor which is created by you. but if u not create constructor There will be a default constructor exist but can't

output:

```
Constructor for car
```

* self_created : parameterised_constructor :

```
class Car {
    public:
        string brand;
        double engine;
        Car(string b, double e){
            brand = b;
            engine = e
            cout << constructor for car << endl;
        }
};
```

```
int main() {
    Car Swift("Swift", 1.2);
    Cout << swift.brand << swift.engine << endl;
}
```

* Copy Constructor

```
int main() {
    Car swift("swift", 1.2);
    Car copy = swift;
    Cout << copy.brand << endl;
}
```

→ Copy the object swift
so, when copy created the copy constructor will execute.

* Destructor:

```
~Car() {
    Cout << "Destructor" << endl;
}
```

→ it doesn't require or not pass the argument like constructor

→ when we discard or when the object became out of scope or use then, it call destructor.

* This keyword:

```
class Car {
    Public:
        string brand;
        double engine;
        Car(string brand, double engine) {
            This -> brand = brand;
            This -> engine = engine;
        }
};
int main() {
    Car Swift("Swift", 1.2);
}
```

"This -> brand" is more like "object.brand"

```
class Base {
    Public:
        Base(int a) {
            cout << "Base value : " << a << endl;
        }
};
class child : public Base {
    Public:
        child(int a, int b) : Base(a) {
            cout << "child value : " << b << endl;
        }
};
int main() {
    child obj(10, 20);
}
```

both constructor was called when the child object was created. & This is Thy way to pass the parameters

Pillors of Oops:

- encapsulation;
- Inheritance;
- Polymorphism;
- Abstraction;

* encapsulation :- (data/Attributes & function/method - single entity - class)
  → (restrict access of the object)

→ public, protected, private
  ↳ default

example:-

```
class student {
    public:
        { declared variable & methods are used by main()
          which access is public
};

class student {
    { declared variable & methods without access specifier were default was "private".
};
```

class student {

protected:

{ inside variable & methods are protected.
They can access by inherited class not even its own object

* Inheritance (types): { inherite or access a variable and method of one class to another class

example:

class Parent {

Public:

int age = 45;

string name = "Loki";

void display() {

cout << name << age << endl;

};

class student { " student : public parent

Public:

int age = 50;

string name = "Loki_child";

} ← access specifier

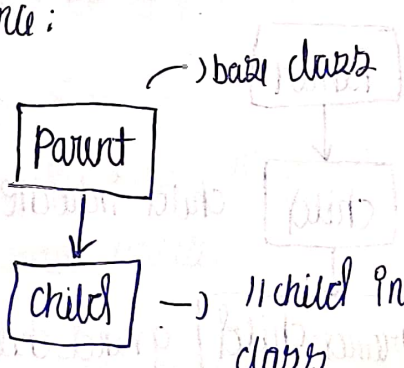
```
void displays() {
        cout << "This is child" << endl;
    }
};
int main() {
    parent P;
    cout << P.display(); ;  ⟶ 45 Lokl
    Student S;
    S.display();  ⟶ 45 Lokl ..
```

Because student inherite the base class parent

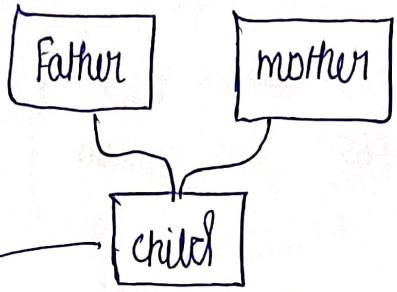
-) Types:

- Single inheritance
- multiple inheritance
- multilevel inheritance
- hierarchical inheritance
- hybrid inheritance

* Single inheritance:



* multiple inheritence :



-> class child : Father, mother

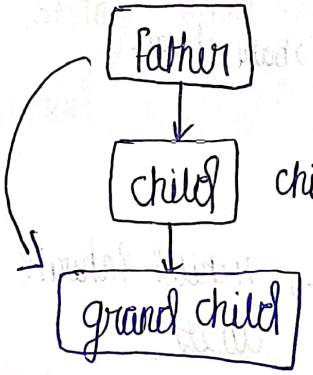
(child inherite both father & mother class)

* if "name" variable is present in both father & mother class & u try to access a name variable in child class. Then it raise. "error ambigous variable"

so, u must specify which class variable.

ex: "Father : : name"

* multilevel inheritance :

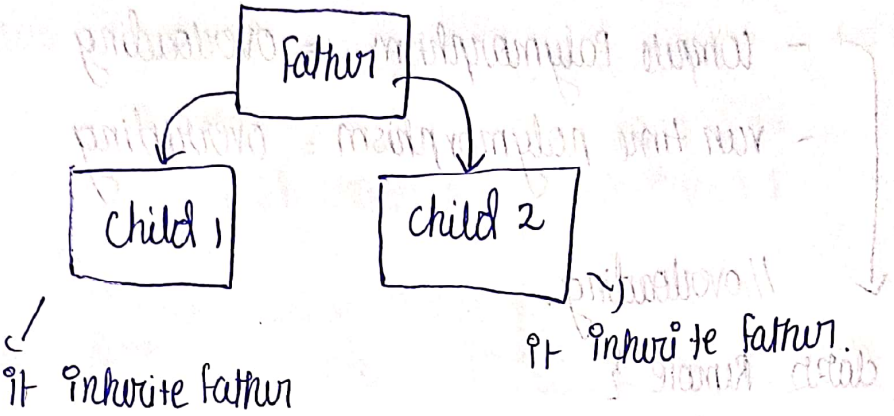


child inherite father

grand_child inherite child

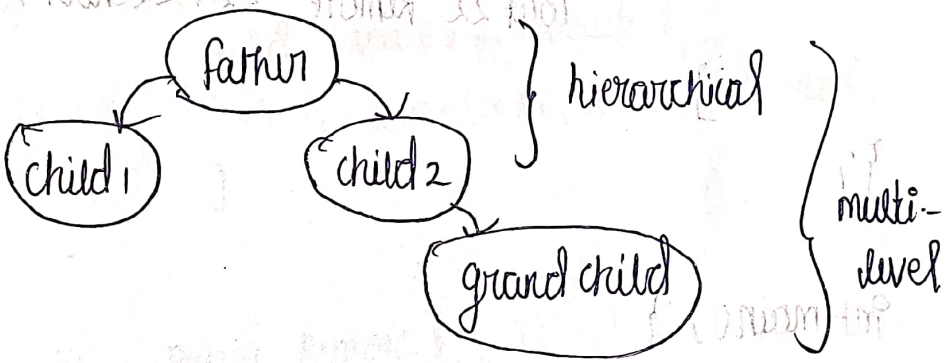
so, it can also access the father

* Hierarchical inheritance:



* hybrid inheritance:

↳ combination of multiple types of inheritance.



* why base class constructor call even child inherited as private:

because, construction is about memory + object setup, NOT access control

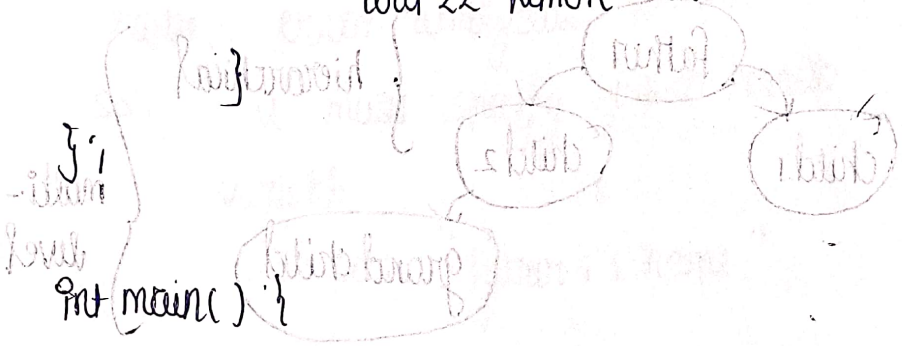
Before c++ can build child part, it must build the base part.

* Polymorphism - (many forms)

- compile Polymorphism = overloading
- run time polymorphism = overriding

// overloading.

```
class Remote {
        Public
        void show () {
                cout << " Remote " << endl;
        }
        void show (int i) {
                cout << " Remote " << i << endl;
        }
};
int main() {
        Remote r;
        r.show();  ---> Remote
        r.show(5); ---> Remote 5
        return;
}
```



// overriding.

```
class Remote {
    Public:
        void show {
```

add distruction:

```
~AC() {
    ...;
}
~TV() {
    ...;
}
```

```
class Remote {
    Public:
        virtual void show() = 0;
};

class AC : Public Remote {
    Public:
        void show() override {
            cout << "AC Remote " << endl;
        }
};

class TV : Public Remote {
    Public:
        void show() override {
            cout << "TV Remote " << endl;
        }
};

int main() {
    Remote *r = new AC();
    r->show();
    r = new TV();
    r->show();
    delete r;
}
```

Abstraction:-

```
class Shape {        // abstract class
    Public
    virtual void area() = 0;
};
class Rectangle : public Shape {
    Public:
    int width;
    int length;
    Rectangle (int width, int length) {
        This->width = width;
        This->length = length;
    }
    void area() override {
        cout << "Rectangle area : " << length *
                width << endl;
    }
};
class Circle : public Shape {
    Public:
    int radius;
    Circle (int radius) {
        This-> radius = radius;
    }
    void area() override {
        cout << "circle area : " << 3.14 *
                radius * radius << endl;
    }
};
```

```
int main() {
    Rectangle r(15,20);
    Circle c(10);
    Shape *s1;
    s1 = &r;
    s1->area();
    s1 = &c;
    s1->area();
    return 0;
}
```

out:

Rectangle area: 300

Circle area: 314

* abstract class must have one pure virtual function"

Problem 2:

1.
```
class Vehicle {
    Public:
        string brand;
        int speed;
        Vechicle(int speed, string brand) {
            This -> speed = speed;
            This -> brand = brand;
        }
};

class Car : public Vechicle {
    Public:
        string fuel_type;
        Car(int speed, string brand,
            string fuel_type):
            Vechicle(speed, brand) {
            This -> fuel_type = fuel_type
        }
        void display() {
            cout << " Brand : " << brand <<
            "\n Speed : " << speed <<
            "\n Fuel Type : " <<
            fuel_type << endl;
        }
};
```

```
int main() {
        cout << "TASK 1 :\n" << endl;
        Car C1 (80, "swift", "petrol");
        C1. display();
        return 0;
}
```

output:

Brand: swift
Speed: 80
Fuel Type: petrol

2°

```
class Animal {
        public:
            void sound() {
                cout << "Animal make sound !" <<
                                        endl;
            }
};
class Dog : public Animal {
        public:
            void sound() {
                cout << " Dog barks ! " << endl;
            }
};
class Cat: public Animal {
        public :
            void sound() {
                cout << " Cat meow ! " << endl;
            }
};
```

```
int main() {
    Dog D1;
    Cat C1;
    D1.Sound();
    C1.Sound();
    return 0;
}
```

Output :

Dog barks !
Cat meow !

3.

```
class Employee {
    protected:
    string name;
    int salary;
    public:
    Employee (string name, int salary) {
            This -> name = name;
            This -> salary = salary;
    }
    virtual void calculateBonus() = 0;
};

class manager : public Employee {
    Public:
    Manager (string name, int salary) :
            Cout << "manager object " << endl;
    }
```

```
    void calculatebonus() {
        cout << name << "Bonus : " << salary * 0.2 << endl;
    }
};

class Developer : public Employee {
    Public:
    Developer( string name, int salary ) : employee
        (name, salary) {
        cout << "developer objects " << endl;
    }
    void calculate bonus() {
        cout << name << " Bonus : " << salary * 0.1
        << endl;
    }
};

int main() {
    vector <Employee *> employee;
    employee.push_back ( new Manager ("LOKI", 500000))
    employee.push_back ( new Developer( "ram", 700000))
    for ( Employee * e : employee ) {
        e -> calculate bonus();
    }
    return 0;
}
```

output:

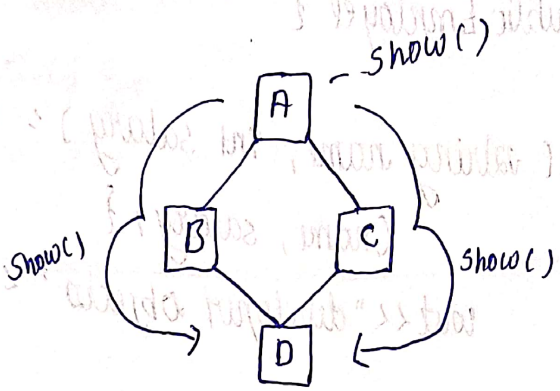
Manager object
developer object
loki Bonus : 10000
ram Bonus : 70000

* Diamond problem:

consider A, B, C, D are classes

→ * B & C inherite the class A

→ * D inherite both B & C.



if the A class contain a function show() and other B & C doesn't have function show(). Then,

if object of D try to use show()

Then, it gets confused because, it can only know there is two show() one from B & other C. it raise error as "ambigous function show()"

This is called Diamond problem.

So, To solve this we will use virtual while inherite A in both B & C

example:

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

using namespace std;

class A {
      public :
      void show() {
            cout << "show function " << endl;
      }
};
class B : virtual public A {          // inherite A as
                                          virtual class
};
class C : virtual public A {          // inherite C as
                                          virtual class.
};
class D : public B, public C {
};

int main() {
      D obj;
      obj.show();
      return 0;
}
```

output:

show function

* TV is a remote -> inheritence relationship

Book has Author -> we can relate without inheritence

example:

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

class Author {
    protected:
    int id;
    string name;
    Public:
    void displayAuthor() {
        cout << "Author's name: " << name << endl;
    }
    Author (int id, string name) {
        This -> id = id;
        This -> name = name;
    }
};

class Book {
    public:
    int book_id;
    string book_id name;
    Author * aumor;        // one class object inside in other class
```

```
    void displayBook() {
        cout << "Book's Name: " << book_name << endl;
        author -> displayAuthor();
    }

    Book(int book_id, string book_name, Author *author) {
        This -> book_id = book_id;
        This -> book_name = book_name;
        This -> author = author;
    }
};

int main() {
    Author a1(1, "JK Rowling");
    Book P1(1, "Philisophar stone", &a1);
    Book P2(2, "chamber of secret", &a2);
    P1.displayBook();
    return 0;
}
```

class object pointer.

class object address

Output:

```
Book's name : Philisophar stone
Author's name : JK Rowling
```

* Exception or Error handling

-> try

-> Throw

-> catch

Basic syntax:

```
try {
    // risky code
}
catch ( type variable ) {
    // handle error
}
```

example 1:

```
# include <iostream>
using namespace std;
int main() {
    int a=10, b=0;
    try {
        if (b==0) {
            Throw string (" Division by error");
        }
        else {
            cout << a/b;
        }
    }
    catch ( string e ) {
        cout << e << endl;
    }
    return 0;
}
```

if u give in normal " " we send as c string

use string( )

How works:

1. code inside try runs.
2. if error occurs → Throw
3. control immediately jumps to catch
4. Remaining try block is skipped.

* multiple catch box:

```
try {
        Throw 10;
}
catch ( int e ) {
        cout << "Integer error " << endl;
}
catch (double e ) {
        cout << "float or double error " << endl;
}
```

* Catch all Exceptions.

```
catch( . . . ) {
        cout << " unknown error occured ! " ;
}
```

* Standard Exceptions in C++

(# include <exception>)

(,) Standard exception library

-) common ones :

std :: exception
std :: runtime_error
std :: out of range
std :: invalid_argument
std :: bad_alloc

(Learn it later don't forget) :x: