Non OOP features of C++

→ Reference variable and call by Reference

→ In Line functions

→ ~~Default~~ Default values in function parameters

→ Scope resolution operator

→ Function over loading

→ new and delete operator.

Reference variable is an alias name given to a memory location. No new memory space is created.

'&' indicates a reference variable.

NOT the address of the variable.

∴ int & r1 = a ;

r1 is the reference variable created or alias name given to a.

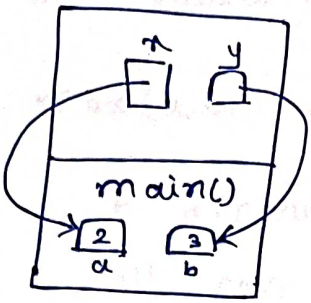
```
int main() {
    int a = 2;
    int & r1 = a;
    cout << a;
    cout << r1;
}
```

output

2

2

x and y in formal parameters are a and b themselves. They are alias name given to a and b as x and y. No new memory locations are created. Function works on actual parameters themselves.



same memory block

In C++,

a function call statement can be used preceding the assignment operator.

But in C, it is invalid.

```
max (a, b)  = 1000;
```

is NOT valid in C language

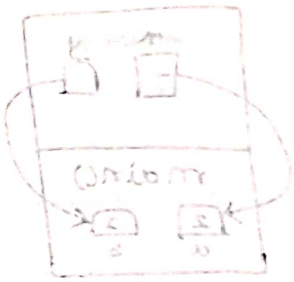
but

```
max (a, b)  = 1000;
```

is possible in C++.

```
int max (int x, in y) {
    if (x > y) {
        return x;
    }
    else {
        return y;
    }
}
```

```
int main(){
    int a = 10, b = 20;
    max(a,b) = 1000;
    cout << a << b;
}
```



→ Error is given

∵ max (10, 20) = 20

But 20 ≠ 1000;

NOT possible

∵ 20 ≠ 1000

It is a constant literal

NOT variable.

Here, call by value is used.

a new memory block is created

for x and y for a and b.

Photocopy of a is sent to x and

photocopy of b is sent to y.

∴ new memory is created

But, if memory location of

max itself is passed.

b value is changed.

As the return type of max is int &, the function returns reference to x and y (and NOT the values). Then a function call such as max(a, b) yeilds a reference to either a or b depending on their vallues. So the function call can now apper on L.H.S. of operator.

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

int main () {
        int a = 10, b = 20;
        max (a, b) = 1000;
        Cout << a << b;
        return 0;
}
```

Output

10, 1000

Scope Resolution operator :: var

```
int a = 1;                    // global variable
int main(){
    int a = 2; {
        int a = 3;
        cout << "::a" << ::a << endl;
        cout << "a=" << a << endl;
    }
    cout << a << endl;
    cout << "::a" << ::a << endl;
}
```

Output

::a = 1

a = 3

a = 2

::a = 1

Scope variable operator ALWAYS access and prints the global variable.

The scope of a variable extends from the point of its declaration till the end of the block containing the declaration ({ .... }).

A variable declared inside a block is said to be local to that block.

In C, the global version of a variable can NOT be accessed from within the inner block, C++ resolves this problem by introducing a new operator Scope Resolution operator (:: var). It allows access to the global version of a variable.

```
int a = 1;
int main (){
    a = 2;{
        a = 3
        cout << :: a << endl;
        cout << a << endl;
    }
    cout << a << endl;
    cout << :: a << endl;
}
```

Output

3
3
3
3

Value of global variable changes.

Default values in Function parameters

```
int sum(int a=0, int b=0, int c=0){
    return (a+b+c);
}

int main(){
    cout << sum(1,2,3);   → 1+2+3 = 6
    cout << sum(1,2);     → 1+2+0 = 3
    cout << sum(1);       → 1+0+0 = 1
    cout << sum();        → 0+0+0 = 0
}
```

Supply of default values must be supplied from right to left in function signature and in function calling, actual values or actual parameters must be supplied from left to right.

~~int sum~~

```
int sum( int a, int b, int c=0)
```

is possible

```
int sum( int a, int b=0, int c=0)
```

is valid.

But

```
int sum( int a=0, int b, int c)
```

is NOT valid and not possible

∵ parameter default values are supplied from right to left.

```
void print ( int = 5, char = '+');
int main () {
        print ();
        print (10);
        print (15, 'A');
        print ('A');
    // 'A' is converted into
    // its ASCII value
    // ∴ it is int 67
}
void print ( int a, char ch) {
    for (int i = 0; i < a; i++){
        cout << ch << endl;
    }
}
```

→ Here the char type is converted or type casted into int type and ASCII value of 'A' is used i.e.. 67.

A default argument is checked for the tie at the time declaration and evaluated at the time of call. only the trailing arguments can have default value, and so we must have default values from right to Left, We can NOT provide a default value to a particular argument in the middle of an argument list. If a function uses default argument, calls to it need not include all the arguments shown in the declaration.

Function overloading

Poly morphism → many forms

Polymorphism

- Compile Time Polymorphism (Early Binding) (CTP)
- Run Time polymorphism (Late Binding) (RTP)

```
void sum(int a, int b){                    // f1
    cout << a+b;
}
void sum (double a, double b){             // f2
    cout << a+b;
}
void sum (double a, int b){                // f3
    cout << a+b;
}
int main (){
    sum (1, 2);          // f1 is called
    sum (1.5, 3.6);      // f2 is called
    sum (1.5, 2);        // f3 is called
}
```

Compiler decides which function to call and binds arguments to function before compilation

∴ It is Early Binding or Compile Time polymorphism.

Output

3

5.1

Inline function

Body of the function will get copied and pasted inside the main function.

∴ Both memory and time is reduced.

compiler will NOT stop the compilation of main function to move to sum function and return back to main function, and compilation will smoothly continued without any extra space or time loss.

Inline keyword is just a request to the compiler. sometimes the compiler will ignore the request and compile or run the function as a normal function if it is too long.

Some situation while inline keyword is ignored by compiler:

→ for function returning values

→ for recursive function

→ if a function contains a loop. a switch or a goto statement.

→ if a function contains static variables

→ for a function not returning a value but having a return statement.

```
inline void sum (int a, int b) {
    cout << a + b;
}

int main () {
    int a = 10, b = 20;
    sum (a, b);
    return 0;
}
```

New - Delete

used for Dynamic memory allocation.

Malloc and calloc can also be used, but <stdlib.h> must be included as a header file.

new operator returns a typecasted pointer, whereas malloc only returns a void pointer that needs to be type casted.

But typecasting is already done by new operator.

malloc, calloc are functions,

But new is an operator that may be overwritten.

new initialises the allocated memory blocks with 0 (just as calloc does).

new also returns a pointer but it is already typecasted to return the exact type of pointer required.

```
int * p;
p = new int[10];
```

creates 10 blocked int Array. 10 block of memory each of the size of integer is created.

```
int* p = new int(2);
```


```
*p = 2.
p = 1000
```

```
delete (p);
```

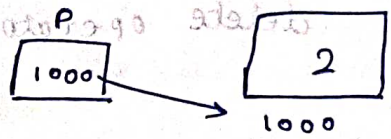
or

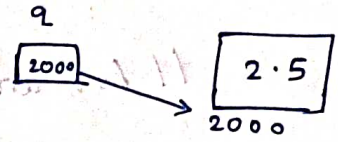
```
delete [] p;
```

delete is an operator for an array stored as p.

free is a function

```
int * p;
float * q;

p = new int;
q = new float;
*p = 2;
*q = 2.5;
```






```
int * p;
p = new int;
*p = 2;
```

} `int * p = new int (2);`

same as

```
float * q;
q = new float;
*q = 2.5
```

} `float * q = new float (2.5);`

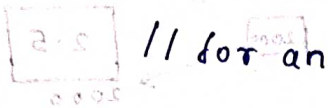
`int *p = new int [10]`

New operator creates a memory space for an array of 10 integers. P[0] will represent the first element.

P pointer stores the base address of the array i.e.. address of P[0].

When a data objed is no longer needed. it is destroyed the memory space for reuse, deleted by delete operator as

delete [] p;



// for an array stored by p

or delete (p);

// for single memory p.

scanf replaced by cin >>

printf replaced by cout <<.

Object

Real world active entity.

object is created for a class.

Class is passive entity. NO extra memory is created, until an object is defined

Class is an user defined data types.

It is same as structure.

class → key word.

Just like structure, class has members inside it.

```
class Point {
    int a;
    int b;
    void set_a(int x){
        a = x;
    }
    void set_b(int y){
        b = x;
    }
    void display {
        cout << "a = " << a << "b = " << b;
    } };
```

// member variables of class.

called attributes of a class.

functions can be defined inside class.

No extra memory space created.

But if object is created, memory is allocated.

In class, all members and functions are private.

In structure, all members or functions are public.

∴ Member variables of a class can NOT be used outside the body of the class.

Only difference b/w class and structure is

class is private. (by default).

But structure is public.

∴ public: must be specified before the fundions.

But, after public: is declared, if some attributes is to be made private. private: keyword must be declared.

∴ Class supports Data Abstraction Property

```
class Point {
    int a;
    int b;
public:
    void set_a (int x){
        a = x;
    }
    Void set_b (int x){
        b = x;
    }
    Void disp (){
        cout << a <<b;
    }
};
int main (){
    Point p;
    p.set_a(10);
    p.set_b(20);
    p.display();
}
```

→ 30

If function body are to be written outside class body, scope resolution operator must be used.

```
class Point {
    int a;
    int b;
public:
    void set_a (int);
    void set_b(int);
    void disp ();
};

void Point :: set_a (int x) {
    a = x;
}
void Point :: set_b (int y) {
    b = y;
}
void Point :: display () {
    cout << a << b;
}
```

this operator

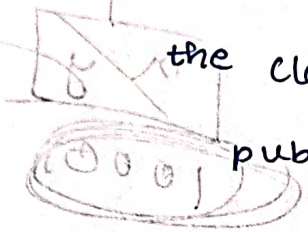
In C++, there are 3 types of access specifier.

→ private

→ public

→ protected

private variables are accessable inside the class body.



public variables can be accessed throughout the program.

'this' is a pointer. It references to the current object of the class.

It refers to that object in the class that is being used inside the function.

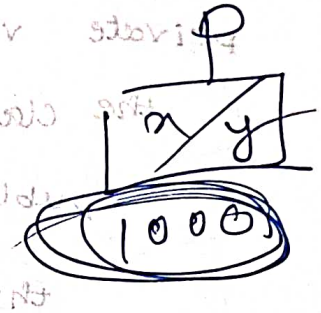
```
class Point{
    int x, y;
    public:
    void set_xy (int x, int y) {
        this -> x = x;
        this -> y = y;
    }
    void show (){
        cout << x << y
    }
};
```

```
this -> x = x ;
this -> y = y ;
```

can also be written as

```
(* this). x = x;
(* this) . y = y;
```

```
int main (){
    Point p;
    p.set_xy (2, 3);
    p.show ();
}
```



this = 1000

this operator is access able ONLY within the class functions.

It can NOT be called inside main.

this points to the current object inside class.

(the variables declared inside class),

⟹ 'this' operator returns the address of the memory allocation of the object.

this returns the address of the object itself.

```
int main (){
        Point *P[5];
        for (int i=0; i<5; i++){
                P[i] = new Point();
        }
        for (int i=0; i<5; i++){
                P[i] -> set_xy (i, i+1);
                P[i] -> show();
        }
}
```

Array of pointers is created.

If a normal array is to be created, in array of objects.

```
int main (){
        Point P[5];
        for (int i=0; i<5; i++){
                P[i].set_xy (i+1, i+2);
        }
        for (int i=0; i<5; i++){
                P[i].show ();
        }
}
```

```
class Point {
    int x, y;
  public:
Point& void set_x (int x){
        a = x;
        return *this;
    }
Point& void set_b (int b){
        b = y;
        return *this
    }
    void show () {
        cout << a << b;
    }
};
int main() {
    Point ob;
    ob. set_x(5) .show();
    ob. set_y(6) .show();
    (ob. set_x(5)) .set_y(6) . show();
}
```

This is Cascading function call.

```
int main (){
    Point * ob;
    ob = new Point ();
    ob -> set_x (5);
    (*ob). set_y (6);
}
```

int a = 3;

struct Point

{ int a; }

;

struct Point P;

```
Class Point {
    int a, b ;
    public :
    void set_a_b (int x, int y) {
        a = x ;
        b = y ;
    }
    void show () {
        cout << a ;
        cout << b ;
    }
};

int main () {
    Point P1 ;
    P1.set_a_b (1, 2) ;
    P1.show () ;
    return 0 ;
}
```

Using Constructor

```
class Point {
    int a, b;
    public:
        Point (int x, int y) {          // constructor
            a = x;
            b = y;
        }
        Void show () {
            cout << a << b;
        }
};

int main () {
    point P1 (1, 2);
    P1.show ();
    return 0;
}
```

Constructor has

→ no return type

→ has the same name as the class itself.
same name of function as class

→ written in public type.

constructor is a type of function. ∴ Just like functions, constructors can also be overloaded.

constructors can also be overloaded.

```
class Point {
    int a, b;
    public:
        Point (int x, int y) {
            a = x;
            b = y;
        }
        Point (int x) {
            a = b = x;          // valid statement in C++
        }
};
```

Similarly, default values are also possible in constructors.

```
class Point { int a, b;
  public:
    Point (int x = 0, int y = 0) {
        a = x;
        b = y;
    }
};
```

Default values also needs to be supplied from right to left.

```
class Point {
    int a, b;
    public:
        Point (int x, int y=0) {      // f1
            a = x;
            b = y;
        }
        Point (int z = 0) {           // f2
            a = b = z;
        }
};
int main () {
    Point P1 (5);        → error is shown
                           ∵ both the functions are possible to be binded.
    Point P2 (10,20);    → a = 10
                           → b = 20
    Point P3;            → a = b = 0
    return 0;                f2 is binded.
}
```

In P3, f2 is getting binded.

In P2, f1 is getting binded.

In P1, both f1 and f2 can be binded. ∴ error is shown

```
class Point {
    int a, b;
    public:
        point (int x) {
            a = b = x;
        }
};
int main () {
    Point P1 (1);
    Point P2 = P1;
    P2.show();
}
```

Compiler supplies default copy constructor P2. where P1 gets copied to P2.

Valid statement.

Default copy constructor P2 is provided where objects of P1 gets copied to P2.

But to modify or write a copy constructor

```
class point {
    int a, b;
public:
    point (int x) {
        a = b = x;
    }
    point (point & obj) {
        a = obj.a;
        b = obj.b;
    }
};
int main () {
    point P1 (1);
    Point P2 (P1);
    P2.show()
    return 0;
}
```

→ Copy constructor always takes a call by reference.

Call by refference is to be used since, if call by value is used, infinite loop take place that causes error.

Destructor

destructor has NO parameters and ~~[illegible]~~ it can NOT be overloaded.

the memory allocated by constructor gets destroyed by the destructor.

No need to explicity call.

once program ends. automatically destructor gets ~~[illegible]~~ called.

```
~Point () {
    cout << "Destructor ";
};
```

opposite of constructor.

```
class sum {
    int a, b, c ;
    public:
        sum (int x , int y , int z = 0) {
            a = x;  b = y;  c = z;
        }
        int sum_cal () { return a+b+c; }
        ~~int~~ void sum_cal () {
                    cout << a+b+c; }
};
```

```
int main (){
    sum s1 (1, 2);
    sum s2 (2, 3, 5);
    cout << s1.sum - cal ();
}
```

Error is displayed.

NO two functions that are same w.r.t. parameters and name can ever be overloaded depending on the return type alone.

If functions have same parameters and same name, they can never be overloaded depending on their return type.

→ A constant variable can NEVER be initialised within the constructor body.

constructor body can NEVER have a constant variable.

one way to initialise constant variable is

```
class sample {
    int a;
    const int b;
    public:
        sample (int x) : b(x) {
            a = 2 * x;
        }
        void show () const {
            cout << a << b;
        }
};
// constant keyword const is added to a
// function to ensure no member variable
// can be modified (like a++;) inside the
// body of function.
int main () {
    sample ob (10);
    ob.show ();
    return 0;
}
```

```
Void show () {
    a++;
    b++;
    cout << a << b;
}
```

→ possible ∵ a is NOT constant

→ NOT possible, ∵ b is a constant variable.

A constant variable can NEVER be modified.

`b++;` shows error ∵ it is constant

```
Void show() const {
    a++;
    b++;
    cout << a << b;
}
```

NOT possible

∵ constant function can NEVER modify a member variable.

```
void show (){
    a++;
    cout << a << b;
}
```

possible

Constant objects

```
Class sample {
        int a;
        const int b;
public:
        sample (int x) : b(x) {
                a = x;
        }
        void change () {        // non constant function
                a++;
        }
        void show() const {
                Cout << a << b;
        }
};

int main (){
        const sample ob(10)
        ob.show ();
        // ob. change ();  gives immediate error
        return 0;
}
```

for a constant object, ONLY constant member functions are possible to call.

A member variable of a constant object can NEVER be changed.

constant object member variables are fixed.

∴ only constant member functions of the class can be called from constant object.

now, if a member variable is changed from constant const to mutable, then the value of the mutable member variable can be changed within constant member functions.

```
class Sample {
    int a;
    mutable int b;
public:
    Sample (int x) : b(x) {
        a = x; }
    void change(){ a++; }
const void show () { cout << a<<b;
        b++; } };
```

But.

here,

NON constant objects are to be made.

constant objects can NOT be made

Constant functions

→ It specifies that the function is a read-only function which is NOT allowed to modify an object.

→ If a constant member function attains to change the data, the compiler reports an error. So it is illegal to declare a constant member function that modifies a data member.

Note **

→ Non constant member methodes can only be invoked using non constant objects only

→ constant member member methodes can be invoked on constant as well as non constant objects.

→ Mutable Data members can be changed even by the constant member functions.

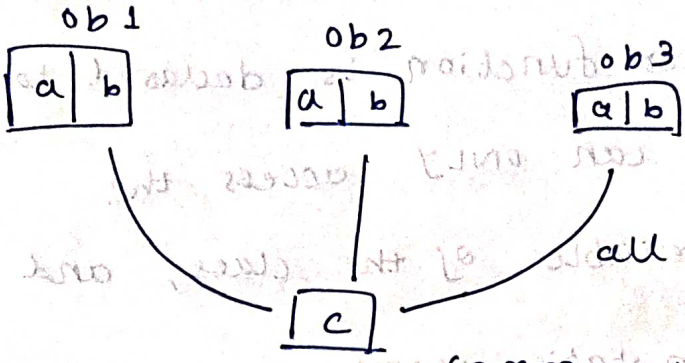
```
class sample {
    int a;
    mutable int b;
    public :
        sample (int x) : b(x) {
            a = x;
        }
        void show () const {
            cout << a << b;
            b + +;
        }
};
```

This is possible if b is mutable only.

If a member variable is declared to be static variable, then it is NOT an individual property of the class. It is a shared property that is created in a common location that is accessible to all the objects of that class.

only one memory is allocated to the static variable from where all the objects of that class access the object.

```
class sample {
    int a; int b;
    static int c;
    public:
        sample (int x) {
            a = x;
            b = 2 * x;
        }
        void show() {
            cout << a << b << c;
        }
};  int sample :: s = 1;
int main () {
    sample ob1(5);
    sample ob2(3) ; ob3(4);
    ob1.show() ;
    ob2.show();
    ob3.show() ;
}
```



all objects of the class shares the obiect.

common location created for static variable c

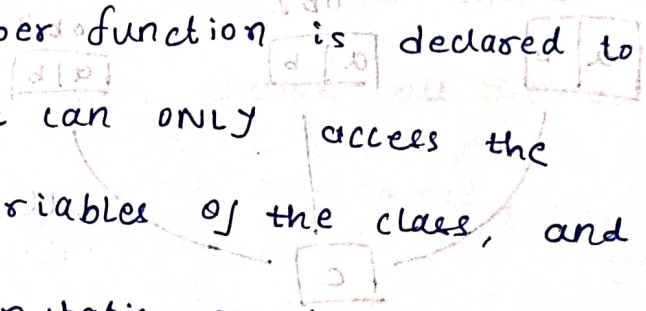
→ if a member function modifies or changes the value of constant variable, then the modified change will be reflected in every upcomming operations performed after the change i.e.. the change will be reflected in all the variables.

The change is showed in all the object variables.

The memory of the static variable is created only once, after the class is declared. Memory allocation is created and a value is entered after the class by

```
int sample :: s = 1;
```

⇒ If a member function is declared to be static, it can ONLY access the static variables of the class, and NOT the non static member functions of the class.



non static member functions can access all the member variables,

but static function can access only the static variables.

```
Class sample {
    int a, b;
    static int s;
    public:
        static void s-show() {
            cout << s;
        }
        void show() {
            cout << a << b;
            cout << s;
        }
};
int sample :: s = 1;
```

```
int main () {
    sample ob ;
        ob . s_ show () ;
        sample :: s_ show();
        ob. show ();
}
```

Object need NOT be created to show the static variable, by sample :: s_show();

sample :: s_show (); ⟶ 1

∵ As long as objects are NOT created, NO memory is allocated for objects of the class.

But even if object is NOT created, shared memory is already allocated for static variable just after the ending of class definition.

This is why static member function can NOT access static member variable.

```
class sample {
    int a;
    static int s;
    public :
        sample ( int a = 0){
            this -> a = a ;
            s = 1;
        }
        static void s_show (){
            cout << s ;
        }
        void show (){
            cout << a << s <<endl;
        }
};
int sample :: s ;          // here, memory of the static
int main (){               //variable is created.
    sample :: s_show (); ⟶ 1
    cout << sample :: s; ⟶ ERROR
                          ∵ s is a private
                            member variable.
    Sample ob(10);
    ob.show (); ⟶ 10, 1
```

∵ s is private
∴ cout can NOT be used.

If s was a public member variable, this line was possible.

NO static member function can access the 'this' pointer nor can it access a non static function inside it.

But a non static member function can access 'this →' pointer. also it can access a static function inside it.

'this →' operator can NOT be accessed within a static member function.

'this →' pointer always access a non static member variable.

But when static member function is called, it may happen that object has not been created, inside main.

This ways, static member variables have memory allocation but non static variables may not have memory allocated, and thus, 'this →' can NOT access the non static member variables.

∴ error is given.

∴ A constructor can be called inside another constructor as

```
sample (int x, int y) {
        a = x;
        b = y;
        sample (x+y);
        // OR  this(x+y)
        // OR  this -> sample (x+y);
}
sample (int z=0) {
        s = z;
}
class Sample {
    mutable int s;
    const int c;
    public:
        sample (int x) : c(x) {
            s = x;
        }
        void show() const {
            s++;
            cout << c << s; }
        void change () {
            s++;
            //c++; -> ERROR } };
```

for a constant object. ONLY the constant member functions may be called.

as the member variables of a constant object, must NOT change.

but non constant object can access all member functions.

```
class A {
    int a;
    public:
        A (int a = 0) {
            this -> a = a;
        }
        int get_a () {
            return a;
        }
};

class B {
    int b;
    public:
    B (int b = 0){
        this -> b =
    }
    int get_b (){
        return
    }
};

int main () {
    int x;
    A ob1 (5);
    B ob2 (6);
    x = ob1. get_a () + ob2. get_b ();
    cout << x << endl;
}
```

→ 11

```
// To swap values
class A {
    int a;
public:
    A(int a = 0) {
        this -> a = a;
    }
    int get_a () {
        return a;
    }
    int set_a (int t2) {
        a = t2;
    }
};
class B {
    int b;
public:
    B(int b = 0) {
        this -> b = b;
    }
    int get_b () {
        return b;
    }
    int set_b (int t1) {
        b = t1;
    }
};
```

```
int main () {
    A ob1 (5);
    B ob2 (6);
    int t1 = ob1.get_a () ;
    int t2 = ob2.get_b() ;
    ob1. set_a (t2);
    ob2. set_b (t1);
}
```

If a non-member function friend function is declared within the class of one or more classes, then it can access the private members of those classes. Acts as a bridge blw those classes. It is NOT a member function. but it can access private elements.

Definition of friend function is written after the ending of all the classes, but it is declared within within the classes.

It is called inside main as a global function.

```
class B;
class A {
    int a;
public:
    A(int a) {
        this->a = a;
    }
    friend void swap (A &, B &);
};
class B {
    int b;
public:
    B(int b) {
        this->b = b;
    }
    friend void swap (A &, B &);
};
void swap (A &ob1, B &ob2) {
    int t;
    t = ob1.a;
    ob1.a = ob2.b;
    ob2.b = t;
}
```

// done so that, when B & is
// called incide class A in the
// declaration of friend function.
// compiler do NOT report error
// as B is user defined class.
// It is called foreword
// declaration.

// A & and B & is the
// type of parameter declaration
// :. swap function takes in
// reference as
// A & ob1, B & ob2 ;

// major property of friend function is
// that, it hampers
// the data
// abstraction
// property.

```
int main () {
    A ob1(5);
    B ob2(6);
    swap ( ob1, ob2 );
    ob1. show ();
    ob2. show();
}
```

friend function hampers data abstraction property.

```
class dist {
    int feet;
    int inches;
    public:
    dist (int f, int i){
        feet = f;
        inches = i;
    }
    dist add ( dist ob) {
        int i = inches + ob.inches ;
        int f = (feet + ob.feet) + (i % 12);
        if (i > 12) {
            i = i - 12;
        }
        return dist (f, i);
    }
};
```

```
int main() {
    dist d3 = add (d1);
    d3.show();
}
```

Here, ∵ add function takes in ONLY one type of object class, (d1), ∴ it is member function.

friend function can be as add (d1, d2).

But, ∵ only one object is used,

∴ add is member function.

If friend function is to be used.

```
class dist {
    int feet, inches;
    public:
        dist (int f, int i) {
            feet = f; inches = i;
        }
        friend dist add (dist, dist);
};
dist add (dist ob1, dist ob2) {
    // int i = ob1.inches + ob1.inches;
        int f = ob1.feet + ob2.feet;
        int i = ob1.inches + ob2.inches;
        return dist (f, i);
}
```

```
class A {
    int a;
    public:
        A(int = 0);          // declaration of
        A (int, int);        // constructors
        void show ();
};
A :: A(int x) {              // definition of
    a = x;                   // constructor outside
}                            // class.
A:: A (int x, int y) {
    a = x + y;
}
void A :: show () {
    cout << a;
}
int main() {
    A ob1(5);
    A ob2(10, 20);
    ob1.show();
    ob2.show();
}
```

```
class A {
    int a, b;
    public:
        A (int a, int b) {
            this -> a = a;
            this -> b = b;
        }
        friend class B;
    // class B entirely is delared to be friend.
};

class B {
    public:
        void f1 (A ob) {
            cout << ob.a;
        }
        void f2 (A ob) {
            cout << ob.b;
        }
        void f3 (A ob) {
            cout << ob.a + ob.b;
        }
};
```

// If B class was NOT declared as a friend of A, inside B class, private members of another class can NEVER be accessed by a member of that class.

If B was NOT a friend class

then

```
void f1 (A ob) {        → valid statement
    Cout << Ob.a;}      → Invalid statement
```

provides immediate error

A private member of one class can NEVER be accessed inside another non friend class, even by an object of earlier function.

```
int main () {
    A ob (1, 2);
    B obb;
    obb.f1 (ob);
    obb.f2 (ob);
    obb.f3 (ob);
}

class A {
    int a;
public:
    A (int x = 0) {
        a = x;
    }
friend void f1 (A, B); };
```

Declaring friend can be done to a single function.

```
class A {
    int a;
    public:
    A (int x = 0) {
        a = x;
    }
    friend void f1 (A, B);
    int get_a ( ) [
        return a;
    }
};

class B {
    int b;
    public:
    B (int y = 0) {
        b = y;
    }
    friend void f1 (A, B);
    void f2 (A ob) {
        cout << ob. get_a ();
    ]
};
```

```
void f1 (A oba, B obb){
    int t = oba.a + obb.b;
    cout << t;
}
```

// friend function can be delared both inside private as well as inside public part.

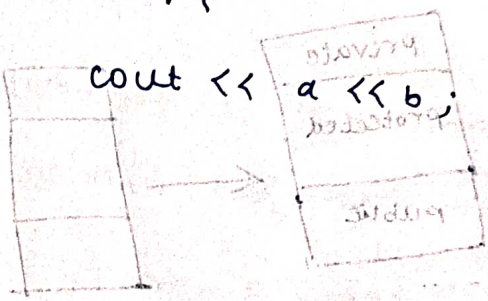
```
class A{
    int a;
    friend class B;
    public:
    // code
};
```

is also possible.

∵ friend is NOT a member function.

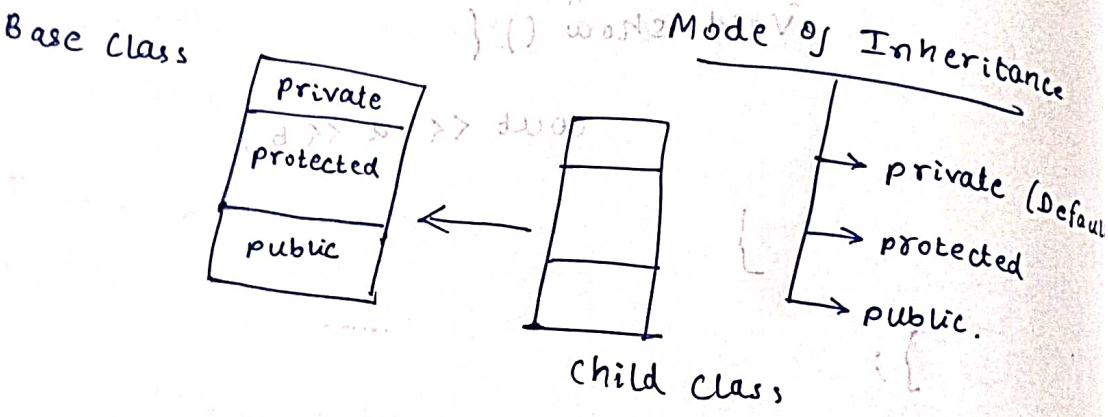
∴ friend function can be declared inside private.

```
class A {
    int a, b;
  public:
    A & change () {
        a++;
        b++;
        return *this;
    }
    void show () {
        cout << a << b;
    }
};

int main () {
    A ob;
    ob.change().show();   // cascading function calls.
}
```



Inheritance

The property of oop for inheriting a base class features and use it to create another child class/ derived class.



If inheritance of a class is done in public mode all the features of the base class will be inherited in the Derived class. but the private data members can NOT be directly accessible within the Derived Class body.

In Public mode inheritance. public members of Base class comes under public part of Child class and protected members of Base class comes under protected part of Child Class.

If inheritance of Base class is done in Base class, the protected and private members of the Base class comes under private part of the child class.

private members can NOT be accessible outside the class Body.

protected members of the Base class are accessible within the child class Body but the Private members are absolutely NOT accessible outside the class definition.

If inheritance of Base class is done in protected mode, both the private and private members of Base class is accessible by / comes under the protected part of the child class.

Types of Inheritance

→ single inheritance (there is single class)

→ multi level inheritance. (child class again sets inherited into another Grand child class).

→ multiple inheritance (More than one base class.

→ Hierarchical Inheritance.

(Multiple child classes from a single Base class).

→ Hybrid Inheritance.

(upper part is hierarchical, lower part is Multiple).

Single Inheritance

```
Class B {
    int a;
protected:
    int b;
public:
    int c;
};

class C : public B {
    int p;
protected:
    int q;
public:
    int r;
};

int main () {
    C ob;
    ob.a = 1;      → Error. ∵ private member
    ob.b = 2;      → Error. ∵ protected member
    ob.c = 3;      → c = 3
    ob.p = 4;      → Error  private
    ob.q = 5;      → Error  protected
    ob.r = 6;      → r = 6
```

protected members of one class can only be accessible within the protected part of the child class.

```
class C : protected B {
    int p;
protected:
    int q;
public:
    C() : B(100)
    {
        b = 20;
        p = 30;
        q = 40;
    }
    void show() {
        cout << p << q;
        B::show();
        // function
        // overriding is used.
    }

int main() {
    C ob;
    ob.show();
}
```

```
class B {
int a;
protected:
    int b;
public:
    B(int n)
    {
        a = 10;
    }
    void show() {
        cout << a << b;
    }
};
```

Constructor is called from Base class to child class.

But Destructor is called from child class to the Base class.

Multiple inheritance

If both the Base function has the same function with the same signature, error is obtained.

```
Class B1{
    public:
        void fun(){
            cout << "B1::f";
        }
};
Class B2{
    public:
        void fun(){
            cout << "B2::f";
        }
};
// NOT function overriding :. B1 and B2 are
// not inherited and are independent.
```

```
class C : public B1, public B2 {
public:
    void fun() {
        cout << "...";
        B1::fun(); B2::fun();
    }
};
// function overriding is used.

int main() {
    C ob1;
    ob1.B1::fun();
    ob1.B2::fun()
    ob1.fun();
}
```

```
class B {
    int a;
protected:
    int b;
public:
    B(int x) {
        a = x;
    }
virtual void show() {
    cout << a;
}
};
```

```
class C : public B {
    int p;
protected:
    int q;
public:
    C(int x) : B(x) {
        p = x;
        b = q = x + 1;
    }
    void show() {
        cout << b << p << q;
        B::show()
    }
};
```

```
int main (){
    C ob(5);
    ob. show();
    B * ptr = new C;
    ptr -> show();
}
```

```
int main()
{ B *ptr = new C;
  ptr -> fun();
  C ob;
  ob.fun();
  ob.B::fun()
}
```

```
class B
{ public:
  void fun()
  { cout << "B::fun";
  }
};
```

```
class C: public B
{ public:
  void fun()
  { cout << "C::fun";
    B::fun();
  }
};
```

```
class B {
public:
    void fun () {
        cout << "B: fun";
    }
};

class C : public B {
public:
    void fun () {
        cout << "C: fun";
    }
};

int main () {
    C ob;
    ob.fun ();          ⟶  C: fun
    ob.B::fun();        ⟶  B: fun
}
```

```
int main () {
    B* ptr [3];
    C ob1;
    C ob2;
    ptr [0] = &ob1;
    ptr [1] = &ob2;
    ptr [2] = new C;
    for (int i=0; i<3; i++) {
        ptr[i] -> fun ();
    }
}
```

// Pointer is of base class B, so the fun function of // base class is called.

⟶ B: fun
B: fun
B: fun

Runtime polymorphism is NOT obtained.
∵ Pointer is of base class type, ∴ fun() of base is called.

But : if virtual key word is used before fun() function of class B, the child class function is called using Base class pointer

```
class B {
public:
virtual void fun () {
    cout << "B : fun ";
}
};

class C : public B {
public:
void fun() {
    cout << "C : fun ";
}
};

int main() {
    B* ptr[3];
    C ob1;
    C ob2;
ptr[0] = &ob1;
    ptr[1] = &ob2;
    ptr[3] = new C;
    for (int i=0; i<3; i++) {
    ptr[i] -> fun();
}}
```

// virtual is used.
// child class is used.

→ C : fun
C : fun

NOTE: The virtual functions are called according to the type of object pointed or referred, not according to the type of the pointer or reference. Therefore virtual functions are resolved at runtime — the desicion is taken later. This is known as Runtime Polymorphism.

```
class B1 {
public:
    B1(int x) {
        cout << "B1 parameter ";
    }
    ~B1() {
        cout << "B1: dest ";
    }
};
class B2 {
public:
    B2(int y) {
        cout << "B2 parameter";
    }
    ~B2() {
        cout << "B2: dest";
    }
};
```

```
class C : public B1, public B2 {
public:
    C() : B2(5), B1(2) {
        cout << "C parmeter ";
    }
    ~C() {
        cout << "C dest ";
    }
};

int main() {
    C ob;
}
```

Output

// The order in which inheritence is done after the class header "class C" is the order in which constructor is used and opposite to which destructor is called.

B1 parameter
B2 parameter
C parameter
C dest
B2 dest
B1 dest

```
class B1 {
public:
void fun() {
    cout << "B1: fun";
}
};

class B2 {
public:
void fun() {
    cout << "B2: fun";
}
};

class C : public B2, public B1 {
public:
void fun() {
    cout << "C: fun";
}
};   // function overriding is used.

int main() {
    C ob;
    ob.fun();      --> C: fun
    ob.B1::fun();  --> B1: fun
    ob.B2::fun();  --> B2: fun
}
```

```
class B {
    public:
    void fun() {
        cout << "B1 : fun()";
    }
};
class C : public B {
    public:
    void fun() {
        cout << "B2: fun";
    }
};
class GC : public C {
    public:
    void fun() {
        cout << "GC : fun";
    }
};
int main(){
    GC ob;
    ob.fun();  ⟶  GC : fun
    C * ptr;
    ptr = &ob;
    ptr -> fun();  ⟶  C : fun.
```

```
B * ptr1 ;

ptr1 = & ob;
ptr1 → fun () ; ⟶ B 1 : fun ()
```

if class C is made

```
class C : public B {
    public:
        void fun () {
            cout << "B 2 : fun" ;
        }
} ;

int main () {   C ob;
    B * ptr = & ob;
    C * ptr1 = & ob;
    ptr → fun() ; ⟶ B 1 : fun ()
    ptr1 → fun() ; ⟶ B 2 : fun
}
```

if C is NOT made virtual

and B is made virtual, then

if is written `B* ptr = &ob ; ptr → fun();`

always prints G c : fun

if C is NOT virtual, But B is virtual.

```
class B {
    public :
    virtual void fun () = 0;        // Pure virtual function
                                    // do NOT have any
    virtual void show() {           // function body.
        cout << "B: show";          // all other class must
    }                               // have to override this
                                    // function.
};
// If pure virtual function is used, then class is
class C1 : public B {               // called abstract
    public:                         // class.
    void fun() {
        cout << "C1: fun()";
    }
};

class C2 : public B {
    public:
    void fun() {
        cout << "C2 : fun";
    }
};
```

```
int main () {

    B * ptr [2];

    C1 ob1 ;

    C2 ob2 ;

    ptr [0] = & ob1 ;

    ptr [1] = & ob2 ;

    for ( int i=0; i<2; i++ ) {

        ptr [i] → fun () ;

    }

}
```

→ C 1 : fun ()
C 2 : fun

```
class B {
public:
    virtual void fun () = 0;
};

class C : public B {
    void fun () {
        cout << "C : fun ";
    }
};

int main () {
    B * ptr = new C;
    ptr → fun () ;
}
```

→ C : fun

```
class B {
public:
    B() {
        cout << " B const ";
    }
    ~B() {
        cout << " B dest ";
    }
};

class C : public B {
public:
    C() {
        cout << " C const ";
    }
    ~C() {
        cout << " C dest ";
    }
};

int main() {
    B* ptr = new C;
    delete ptr;
}
```

→ B const
C const
B dest

If virtual destructor is used in Base class.

```
class B {
    public:
        B() {
            cout << "B const";
        }
        virtual ~B() {
            cout << "B dest";
        }
};

int main() {
    B* ptr = new C;
    delete ptr;
}
```

→

B const
C const
C dest
B dest

But if pure virtual destructor is used, the definition of the destructor MUST be provided after the ending of the class.

```
class B {
public:
    B() {
        cout << "B const ";
    }
// virtual ~B() = 0;
    virtual ~B() = 0;
};
B :: ~B() {
    cout << "B: pure dest";
}
```

```
class B {
public:
virtual void fun() = 0;
};
```

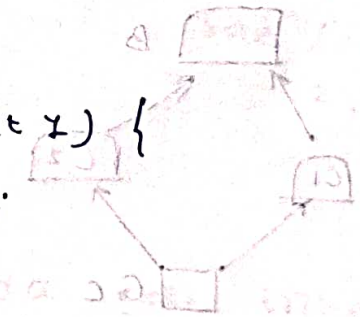
```
class C : pubic B {
public:
void fun() {
cout <<"C:fu
};
```

// class ~~derived~~ G C : public

```
class G C : public C {

};

int main () {
    G C ob;
}
```

→ C : fun {

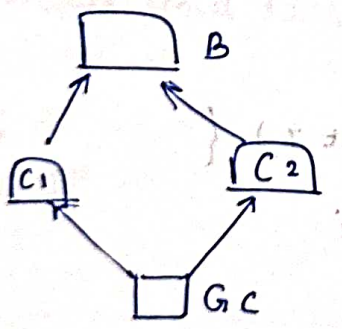
```
class B {
    int a;
public:
    void set_a (int x) {
        a = x;
    }
};

Class C: public B {
        int b;
    protected :
        int c ;
    public :
        void set_b(int y) {
            b = y;
        }
};
```



// GC :

```
Class GC: public C{
    int d;
    public:
        void set_c (int p, int q){
            c = p;
            d = q;
        }
};
int main (){
    GC ob;
    ob.set_c (10, 20);
    ob.set_a(5);
    ob.set_b (100);
}
```

Hybrid



```
class B {
    protected :
        int data ;
};

class C1 : public B {
};

class C2 : public B {
};

class GC : public C1, public C2 {
    public:
        void set (int x) {
            data = x;
        }
};

int main () {
    GC ob;
    ob.set (10);
}
```

⟶ ERROR is shown.

// Here, the data comes in two copies inside GC, one from C1 and one from C2.

In GC, two copies of data gets inherited to GC, one via path of C1 and other via C2.

∴ Ambigious situation is obtained in data = x; line.

same is true for member functions.

But if inheritance is done in virtual mode, then ONLY one copy of data member variable or one copy of member function is shared by the many child class.

```
class B {
public :
    B() {
        cout << " B const ";
    }
};

class C1 : virtual public B {
public :
    C1() {
        cout << " C1 const ";
    }
};
```

```
class C2 : virtual public B {
    public :
        C2() {
            cout << " C2 const " ;
        }
};
class GC : public C1, public C2 {
    public :
        GC() {
            cout << " GC const " ;
        }
};
```

output

B const
C1 const
C2 const
GC const.

If virtual is NOT used in both inheritances.
output is

B const
C1 const] path 1
B const
C2 const] path 2
GC const.

If virtual is used, the default constructor of base is searched. If base has No default constructor, error is reported.

```
class Base {
public:
    virtual void fun() = 0;
};

class C : public Base {
public:
    void fun() {
        cout << "C: fun";
    }
};

int main() {
    Base* p = new Child;
    p->fun();   ⟶  C: fun
}
```

```
class Base {
public:
    virtual void fun ();
    virtual ~Base () {
        cout << "B: dest";
    }
};
void Base::fun() {
    cout << "B: fun";
}
class Child : public Base {
public:
    void fun () {
        cout << "C: fun";
    }
    ~Child(){
        cout << "C: dest";
    }
};
int main () {
    Base *pb = new child;
    pb -> fun();          ⟶  C: dest
    delete(pb);               B: dest.
}
```

Abstract class can be made by using either a pure virtual function or a pure virtual destructor, not by making a constructor pure virtual.

If Base destructor body is NOT provided for a pure virtual destructor after the class definition, then linger error is provided.

But body of a pure virtual function may or may NOT be provided in Base Class.

BUT, definition of pure virtual function MUST be provided in the child class.

```
class B {
public:
    virtual void fun() = 0;
    virtual ~B() = 0;
};
B :: ~B() {
    cout << "B : dest ";
}
```

```
class C: public B {
public:
    void fun () {
        cout << "C: fun";
    }
    ~C() {
        cout << "C: dest";
    }
};

int main () {
    B *p = new C;
    p -> fun ();
    delete p;
}
```

→ for a normal virtual function if it NOT defined in class definition, then it must be defined after the class definition by "::" operator.

→ A pure virtual function is a function with No body in the class. There is No need for the Base class version for that particular function, only use the functions of the derived class functions. A pure virtual function is always assigned with a zero. The "= 0" sign has nothing to do with assignment. The value 0 is NOT assigned to anything. The =0 syntax simply tells the compiler that a function is pure i.e.. it has NO body.

Compiler
↓
Linker
↓
Execution

```
class B {
public:
    B();
    B(int);
    virtual ~B() = 0;
    virtual void fun() = 0;
    virtual void show;
};
```

```
B :: B() {
    cout << "B: def count";
}
B :: B(int x) {
    cout << "B: p" << x;
}
B :: ~B() {
    cout << "B dest";
}
void B :: show() {
    cout << "B:show";
}

class C : public B {
    public:
        void fun() {
            cout << "C: fun";
        }
        ~C();
};
C :: ~C() {
    cout << "C:dest";
}
```

```
int main () {
        B * ptr = new C;
        ptr → fun;          → C : fun
        (*ptr). show ();    → B : show
        delete ptr;         → C : dest
                               B : dest
        C ob;
        ob.show;            → B : show
}

class B {
public:
    B () {
        cout << "B : default";
    }
    B(int x) {
        cout << "B : p";
    }
    int data () {
        return 5;
    }
};
```

```
class C1 : public B {
public:
    C1(int x) : B(x+1) {
        cout << "C1: parameter ";
    }
};

class C2 : public B {
public:
    C2(int x) : B(x+2) {
        cout << "C2 : p";
    }
};

class GC : public C1, public C2 {
public:
    GC(int x) : C1(x), C2(x+1) {
        cout << "GC : p";
    }
};

int main() {
    GC ob(5);
    cout << ob.data;
}
```

→ B : p, C1 : p. B : p. C2 : p, GC : p

→ error

when child 1 and child 2 are derived from Base each inherits a copy of base — this copy is called a subobject. Each of the two subobjects contains its own copy of base class. when the grand child class is created by inheriting child 1 and child 2, it can NOT understand which of the copies it will access. This is where compiler reports an ambiguity.

The use of the key word virtual in these two classes child 1 and child 2, causes them to share a single common subobject of the base class. ∴ There is only one copy of the base class member variable data, there is No ambiguity when it is referred to in the Grand Child class object.

Function Overriding

Suprose both the base class and the child class have a member function with same name and arguments (number and type of arguments are same). If created an object of the derived class and call the

member function which exists in both the classes base and child, the member function of child class is invoked and member function of base class is ignored.

```
class B {
    protected:
        int x;
    public:
        B(int y) {
            x = y;
        }
    virtual void show () {
            cout << x;
        }
};

class C1 : public B {
    public:
        C1 (int z) {
            x = z;
        }
        void show () {
            cout << x;
        }
};

class C2 : public C1 {
    public:
        C2 (int w)
            : C1(w)
        {
            x = w;
        }
        void show () {
            cout << x;
        }
};
```

```
int main ( ){

    B *p = new C1(5);
    p->show();          ⟶  C1:show

    B* p = new C2(5);
    p->show();          ⟶  C2:show.

}
```