



Notes Made By

- RITI Kumari

OOP in C++

OOP



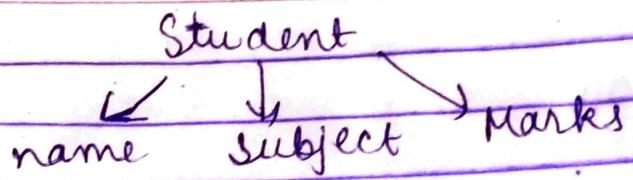
A way to build a software

class

A datatype with
function's

Object

A variable of
class



Class Complex {

private : int real;
int imag;

// can't be accessed outside
the class

public :

void print()

{

cout << real << " + i " << imag << endl;

}

Complex (int re, int im) // constructors

real = re;

imag = im;

};

b;

int main()

{

Complex C1(10, 15);

C1.print();

return 0;

}.

Abstraction - Hiding internal details.

Encapsulation - Putting data & functions in same class.

Inheritance - Don't need to rewrite the code again & again.

Polymorphism → One function ^{name}, diffⁿ ~~same~~ functionalities

- ① Run-time (dynamic)
- ② Compile time (static)

Constructors & Destructors



- 1) Same name as class name
- 2) No return type
- 3) Called when class is created

Class Point {

private:

int x, y;

public:

Point() {

x=0;

y=0;

}

Point (int x1, int y1) {

x=x1;

y=y1;

y

```
void print() {
    cout << x << " " << y << endl;
}
```

```
int main() {
    Point p1, p2(10, 20);
    p1.print();
    p2.print();
    Point *ptr = new Point(5, 10);
    ptr->print();
    return 0;
}
```

We access class members using . but for pointer
we use (\rightarrow)

```
// initializer list
Point : x(0), y(0) {
}
Point (int x1, int y1) : x(x1), y(y1) { }
```

```
class Test {
```

```
public:
```

```
Test() {cout << "Default";}
```

```
Test(int x) {cout << "Parameterized" << x;}
```

```
}
```

```
class Main {
```

```
Test t;
```

```
public:
```

```
Main()
```

```
{
```

```
t = Test(10);
```

```
}
```

```
};
```

O/p: Parameterized

Copy constructors - Special objects which are called when you create an object from existing object.

```
class Test {
```

```
int *ptr;
```

```
public:
```

```
Test(int x)
```

```
{
```

```
ptr = new int(x);
```

```
}
```

Test t₂; X not copy
t₂ = t₁

```
void set(int x) {  
    *ptr = intx;  
}
```

```
Test (const Test &t) {  
    int val = *(t.ptr);  
    ptr = new int(val);  
}
```

```
void set (int x) {  
    *ptr = x; }
```

```
void print() {  
    cout << *ptr << " ";  
}
```

O/P : 10 20.

Compiler defines its own copy constructor.
Dynamic memory allocation is there. Copy
constructor called at this time is a shallow
copy constructor. So to use dynamic memory
allocation we use deep copy.

t₁.ptr = 0x2004 t₂.ptr = 0x2064

10

0x2004

10

0x2064

→ delete
(we write own des
while creating
dynamic memory)

Destructor - Called when an object is destroyed
(out of scope) (No parameter, no return type)

~Test()

{

cout << "destructor called" << endl;

class Test {

public:

Test() {

cout << "con called" << endl;

~Test() {

cout << "des called" << endl;

}

```
int main() {  
    Test t1;  
    Test t2;  
    return 0;  
}
```

O/P:

con called

des called

con called

des called.

when we have multiple objects in same scope they
are destructed in reverse order of their creation

Cons 10

Cons 20

Des 20

Des 10

This pointer
↓

Object to which we are calling.

```
class Point {  
    int x, y;  
}
```

```
public:  
    point (int x, int y)  
    {
```

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

```
}
```

x, y

$\boxed{x=x}$
 $y=y$.

```
int main() {  
    point p1(10, 20);  
    point p2(5, 5);  
}
```

Changing of function

```
class Point
```

```
{
```

```
    int x, int y;
```

```
public:
```

```
    point (int x, int y){
```

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

```
}
```

Point & setX(int x)

{
 this->x = x;
 return *this;
}

point & setY(int y)

this->y = y;
return *this;

}

int main()

{
 Point p1(10,10);

[p1.setX(5)].setY(5);

return 0;

}

this pointer is a constant pointer

Static Member in C++

class Player {

public:
 static int count;
 Player() { count++; }
 ~Player() { count--; }
};

int Player::count = 0.

int main()

{

 Player p1;

cout << Player::count << " "; // 1

{ Player p2; cout << Player::count << " "; } // 2

, cout << Player::count << " "; // 1

Static variable

(are shared among
all objects &
are created once
in a class)

(defined outside
the class)

Static members = class members

Static Member function



When you have static member, we use static member fun to modify static data members.

Class Player {

private:

 static int count;

public:

 player () { count++; }

 ~player () { count--; }

 static int getCount() { return count; }.

}

int Player::count = 0; // It is better to initialize it outside the class

int main()

 Player p1, p2;

 cout << Player::getCount();

 return 0;

}

$$O_p = 2.$$

- 1) static funcⁿ can access only static data members.
Non static members are not allowed.
- 2) They don't have any this pointer as they are class members only.

Inheritance And Virtual function in C++

→ protected access specifier

, Inheritance Syntax.

public : accessible anywhere outside the class.

private : not accessible outside the class.

protected: not accessible outside the class except sub
Class (Inherited class).

Class Person {

protected:

String name;

int id;

};

Class Student : public person {

private:

int marks;

public:

void print() {

cout << name << id << marks;

is able to access name

dirctly as it is public

is able to access id

dirctly as it is public

is able to access marks

Constructor:

Person()

Student()

Destructor

Student()

Person()

	Base	Derived
private	✓	X
public	✓	Might be (public, protected, private)
protected	✓	✓

Class Student : public person

Class Student : ~~public~~ person.

public/protected member in base class become ~~private~~.

Class Student : protected person.

Everything except the private members becomes protected.

~~Inheritance Access~~

public → protected & public of base class remain as they are.

private → protected & public of base class become private.

~~protected → protected & public of base class become
protected~~

~~virtual function → At runtime, it is decided which funcⁿ
to be called.~~

~~A base class type reference or pointer can refer to a
derived class object~~

class Base { }

class Derived : public Base { } ;

int main()

{

Base *b = new Derived();

Derived d;

Base & b = d;

return 0;

}

Class Base {

public:

virtual void print() {
 cout << "Base";
}

Class Derived : public Base {

public:

void print

{
 cout << " Derived";
}

function overriding

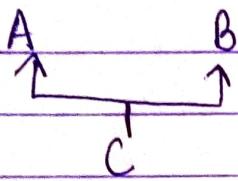
How do virtual funcⁿ' works

They maintain a vptr & vtable
(object) (class)

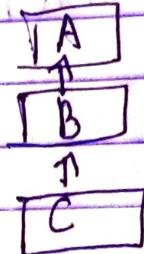
1) They reduce code complexities

Multiple inheritance

When a class inherits from 2 or more classes it's called multiple inheritance



multiple



multilevel.

Class A { }

Class B { } Order.

Class C : public A, public B { } ;

int main () {

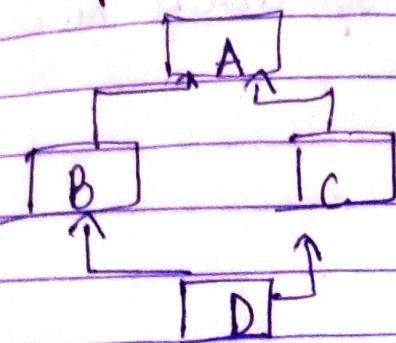
 Cobj;

 return 0;

}

%
A const
B const
C const

Java doesn't allow multiple inheritance due to diamond problem.



Diamond problem

Class A {

```
public:  
    int x;  
};
```

Class B : public A { } ;

Class C : public A { } ;

Class D : public B, public C { } ;

→ A constructor called two times

int main() {

```
    D d;
```

```
    cout << d.x;
```

```
    return 0;
```

f.

Solⁿ: for C++ is we use virtual keyword.

class B : virtual public A { } ,

class C : virtual public A { } ;

class D : public B, public C { } ;

Java doesn't support operator overloading

Operator overloading

If we want to use operators for our own classes

```
Class Complex {
```

```
private:
```

```
int real, img;
```

```
public:
```

```
Complex(int r=0, int i=0) : real(r), img(i) {}.
```

```
Complex operator+(Complex &obj)
```

```
{
```

```
Complex res;
```

```
res.real = real + obj.real;
```

```
res.imag = img + real obj.img;
```

```
return res;
```

```
}.
```

```
void print()
```

```
{ cout << real << " + i " << img << endl; }
```

```
}
```

```
int main()
```

```
{
```

```
Complex c1(10, 5), c2(2, 4);
```

```
Complex c3 = c1 + c2; // c1.operator+(c2)
```

```
c3.print();
```

```
return 0;
```

```
}
```

Operators which can't be overloaded

• , ::, ?:) and size of

* While operator overloading there is no change in precedence, associativity

$$c_4 = c_1 + c_2 * c_3 ;$$

| first
second |

→ (by default created)

* If we don't call an assignment operator, compiler calls itself.

Friend Function → Java doesn't support

A class A can specify other class B a friend of it (then class B can access private & protected member of class A)

i) It is against encapsulation.

```
Class A {
```

```
    int x=4;
```

```
    friend Class B; // friend class
```

```
};
```

```
Class B {
```

```
public :
```

```
    void display(A &a)
```

```
{
```

```
    cout << "value of x" << a.x;
```

```
}
```

```
};
```

```
int main()
```

```
{
```

```
    A a;
```

```
    B b;
```

```
    b.display(a);
```

```
    return 0;
```

```
.
```

O/p : value of x : 4

```
Class Box {
```

```
    int l;
```

```
public :
```

```
    Box(): l(0) {}.
```

```
    friend int printlength(Box); // friend func.
```

```
};
```

```
int printLength(Box b)
{
    b.length += 10;
    return b.length;
}
```

```
int main()
{
    Box b;
```

```
cout << printLength(b);
```

%/p 10.

Coerced, Not Mutual, Not Transitive & Not Inherited

Advanced

Exception Handling in C++

```
int average(int arr[], int n)
```

```
{
```

```
    int sum = 0;
```

```
    for (int i = 0; i < n; i++)
```

```
        sum = sum + arr[i]; // When n=0 then
```

```
    sum = sum/n;
```

```
    return sum;
```

```
}
```

we can't use if ($n=0$) because there might be a condⁿ that the avg is -1.

Cases of Exception Handling

- 1) Divide by zero
- 2) No heap memory Available.
- 3) Accessing array elements outside the allowed index range
- 4) Pop from an empty stack.

Try, catch & throw in C++

↓ ↓ ↓
before the used used to throw
block of before an exception
code a block.
 after try

try {

 // The code that may

 // throw exception

}

throw : used to throw an exception

catch : one or more catch blocks are used
to handle exception

Example code:

```
int main()
{
    int x, y;
    cin >> x >> y;
    try {
        if (y == 0)
            throw 0;
        cout << "Result is " << x/y;
    }
}
```

catch (int x)

{

cout << "Divisor is zero";

} return 0;

I/P

x = 10

y = 2

O/P

Result is 5.

x = 10

y = 0

O/P

Division is 0.

```

int main() {
    double x, y;
    cin >> x >> y;

    try {
        if (x == 0.0)
            throw 0;
        if (y == 0.0)
            throw string ("Y is zero");
        if (x+y < 0.0)
            throw (x+y);
    }

```

```

    catch (int e1) { cout << e1; }
    catch (string &e2) { cout << e2; }
    catch (...) { cout << "x+y is less than 0"; }

    catch all (it catches all the datatype exception
               concept datatype
               not present in catch like for double)

```

```

cout << "In Bye . . . ";
between 0;
}

```

I/p 0.0 1.0
 O
 Bye . . .
I/p 1.0 0.0
 y is zero
 Bye . . .

I/p -1.0 -2.0
 x+y is less than zero
 Bye . . .

Ex

```
int avg (int arr[], int n)
{
    if (n == 0)
        throw string ("Array size is zero");
}
```

```
int sum = 0;
for (int i = 0; i < n; i++)
    sum += arr[i];
return sum / n;
}
```

```
int main()
{
    int arr[] = {4, 5, 2};
    int n = 0;
    try {
        int res = avg(arr, n);
        cout << res;
    }
}
```

```
catch (string e) {
    cout << e;
}
cout << "Bye...";
return 0;
}
```

O/P:
Array size is zero
Bye...

int average (int arr[], int n) throw(string)

↓
It is used to specify what all exceptions can be thrown by the function.

Stack Unwinding in C++

↓
If a func " throws an exception & doesn't handle this exception then control goes to the caller & if it doesn't handle then goes to the handler.

void f1 throw(int)
{

cout << "f1 begins \n";

throw 100;

cout << "f1 ends \n";

}

void f2 throw(int)

{

cout << "f2 begins \n";

f1();

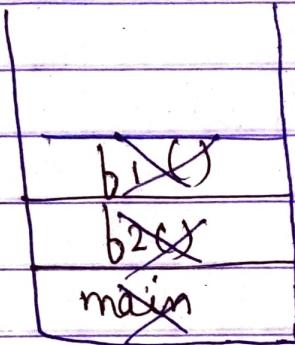
cout << "f2 ends \n";

}

```

int main()
{
    try {
        f2();
    }
    catch (int i)
    {
        cout << "Caught exception\n";
    }
    cout << "Bye ....";
}

```



O/p : f2 Begins
 f1 begins
 Caught Exception
 Bye ..

User Defined Exceptions

```

int avg (int arr[], int n)
{
    if (n == 0) throw ArraySizeZero();
    if (n < 0) throw ArraySizeNegative();
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum / n;
}

```

```

int main()
{
    int n;
    cin >>n;
    int * arr = new int[n];
    for(int i = 0; i < n; ++i)
        cin >>arr[i];
}

try {
    int avg = avg(arr, n);
    cout << avg;
}

catch (ArraySizeZeroException & e1)
{
    cout << "Array size is zero" ;
}
catch (ArraySizeNegativeException & e2)
{
    cout << "Array size is negative" ;
}

return 0;
}

```

* It is not allowed in Java to throw primitive type as exception.

Exception class in C++ → from new operator

All Standard Library exceptions like bad_alloc,
 bad_cast etc inherit from exception class
 directly or indirectly.

exception class has an what function.

```
class MyException : public exception
```

```
{
```

```
    virtual const char* what() const throw();
```

```
    { return "Exception occurred\n"; }
```

```
}
```

```
};
```

```
int main()
```

```
{ try {
```

```
    throw MyException();
```

```
}
```

```
catch (exception e) {
```

```
    cout << e.what();
```

```
}
```

```
}.
```

Function pointers

Stack	→ for func ⁿ calls (local data of a func ⁿ)
Heap	→ for dynamically allocated memory.
Data	→ to store static, global variables
Text	→ instructions or executable codes (a func ⁿ pointer stays here)

Q. why do we need function pointer?

- Stores address of a function (set of instructions)
- like normal pointers, we can pass a "func" pointer to other ~~pointer~~ functions
- Used in qsort(), sort(), foreach() or any other place where we wish to provide functionality as a parameter.
- Used to implement virtual function.

Example:

```
void fun() {  
    cout << "Hello";  
}
```

```
int main () {  
    void (*fun_ptr)() = &fun; // if  
    fun_ptr(); // (* fun_ptr)();  
    return 0;  
}
```

auto fun_ptr = fun;

it is optional
as function name gives the
address of func.

```
int fun (int x, int y) {  
    return (x+y); }
```

```
int main() {
```

```
    auto fun_ptr = fun;
```

```
    cout << fun_ptr(10, 20); // int (*fun_ptr)(int, int) = fun;
```

```
    return 0;
```

```
}
```

Passing functions as parameters

```
int add ( int x, int y) { return (x+y); }
```

```
int multiply (int x, int y) { return x*y; }
```

```
int compute (int x, int y, int (*fun_ptr)(int, int))  
{ return fun_ptr(x, y); }
```

```
}
```

```
int main()
```

```
{
```

```
    cout << compute(10, 20, add) << "\n";
```

```
    cout << compute(10, 20, multiply);
```

```
    return 0;
```

```
}
```

O/p

30

200

Sort funcn' in array.

```
bool compare (int x, int y) {  
    return abs(x) < abs(y); }
```

```
int main() {
```

```
    int arr[] = {2, 4, -1, 8, -9};  
    int n = sizeof(arr)/sizeof(arr[0]);
```

```
    sort (arr, arr+n, compare);
```

```
    for (int i=0; i<n; i++) {
```

```
        cout << arr[i] << " ";
```

```
    }  
}
```

O/p: -1 2 4 8 -9

(Canonymis)

Lambda expression

You don't have to create a separate function

```
int main() {
```

```
    int arr = {-20, 10, -30, 5};
```

```
    sort (arr, arr+n, [int a, int b] {return abs(a) < abs(b); })
```

```

for (auto x : arr)
    cout << x << " ";
return 0;
}

```

O/p 5 10 -20 -30

[] () → ↑ { }
 ↑ ↑
 capture parameter ↑
 list function body

↑ return type

OR .

```

auto mycmp = [ ] (int a, int b) {
    return abs(a) < abs(b);
};

```

```

sort (arr, arr+n, mycmp);

```

Example 1: for_each()

↓
 b/w 2 operators we can do a provided
 functionality.

```

int main() {
    vector<int> v { 10, 20, 30 };
    for_each (v.begin(), v.end(), [ ](int& x) { x = x * 2; });
    for_each (v.begin(), v.end(), [ ](int& x) { cout << x << " "; });
    return 0;
}

```

O/p: 3.

Example 2: count_if → (count no. greater than a no)

```
int main() {
    vector<int> v {10, 5, 3, 20, 100};
    int res = count_if(v.begin(), v.end(),
                       [](int x) {return x >= 10; });
    cout << res << "in";
    return 0;
}
```

Example 3: find_if() → iterator to the first element that satisfies the condn.

```
int main() {
    vector<int> v {100, 20, 4, 200, 1};
    auto it = find_if(v.begin(), v.end(),
                      [](int x) {return x < 10; });
    cout << *it;
    return 0;
}
```

O/P : 4

Example 4: accumulate()
accumulate by default does sum of elements.

```
int main() {  
    vector<int> v { 10, 2, 4, 20, 1 };  
    int res = accumulate(v.begin(), v.end(), 0);  
    cout << res << "\n";  
    res = accumulate(v.begin(), v.end(), 1,  
                      [](int x, int y) { return x * y; });  
  
    cout << res;  
    return 0;  
}
```

O/p : 35 $0 + 10 + 2 + 4 + 20 + 1 = 35$
 1600 $1 * 10 * 2 * 4 * 20 * 1 = 1600$

Capture list in lambda expression

- [] : Nothing
- [=] : Everything by value
- [&] : Everything by ~~value~~ reference
- [=, &x] : Everything by value & x by reference
- [&, x] : Everything by reference & x by value

Static & global variables are always captured

```

int main() {
    int x=5, y=10;
    auto lambda_expre = [=] (int a) {
        x = x+a;
        y = y+a;
    };
    lambda_expre(20);
    cout << x << " " << y << endl;
    return 0;
}.

```

O/p = 25, 30

Some exception

```

auto lambda_expre = [=] (int a) mutable {
    x = x+a;
    y = y+a;
}.

```

without the
use of mutable
these
values
can't
be modified

O/p : 5, 10

Smart pointers in C++
(wrap a pointer to a class object)

In normal pointers there is a memory leak.

```
void fun()
```

```
{
```

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

```
}
```

```
int main()
```

```
{
```

```
    while (true) {
```

```
        fun();
```

```
}
```

```
}
```

```
.
```

```
}
```

Crashed.

```
class SP {
```

```
    int *ptr;
```

// wraps an integer to pointer

```
public :
```

```
    SP(int *p = NULL) { ptr = p; }.
```

```
    ~SP() { delete ptr; }
```

```
    int & operator*() { return *ptr; }.
```

```
};
```

```
int main()
```

```
{
```

```
    SP sp(new int());
```

O/P = 20

```
    *sp = 20;
```

```
    cout << *sp;
```

```
    return 0;
```

```
}
```

Here, in the example we don't get the value as
destructor called.

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

```
    x=a;
```

```
    y=b;
```

```
    cout << " constructor called \n";
```

```
}
```

```
~Test()  
{
```

```
    cout << " Destructor called \n";
```

```
}
```

```
};
```

```
int main()
```

```
{  
    cout << "main begins \n";  
    }
```

```
    Test *p = new Test(10, 20);
```

```
{
```

```
    cout << "main ends";
```

```
    return 0;
```

```
}
```

O/p: Main begins

constructor called

Main ends

so we would overload \rightarrow and $*$ operator.

```
class SP {  
    Test *ptr;  
public:  
    SP(Test *p = NULL) { ptr = p; }  
    ~SP() { delete ptr; }
```

```
    Test & operator*() { return *ptr; }  
    Test * operator->() { return ptr; }  
};
```

```
int main()  
{  
    cout << "main begin";  
    SP sp(new Test(10, 20));  
    cout << "Main ends";  
    return 0;  
}
```

O/P

Main begins
constructor called
Destructor called
Main ends
Objects of SP goes out of
scope it is called

Template Smart Pointer

Template <class T>

Class SP

{

T *ptr;

public:

SP(T *p = NULL) { ptr = p; }

~SP() { delete ptr; }

4)

T & operator *() { return *ptr; }

T *operator ->() { return ptr; }

};

int main()

{

SP<int> sp(new int());

*sp = 20;

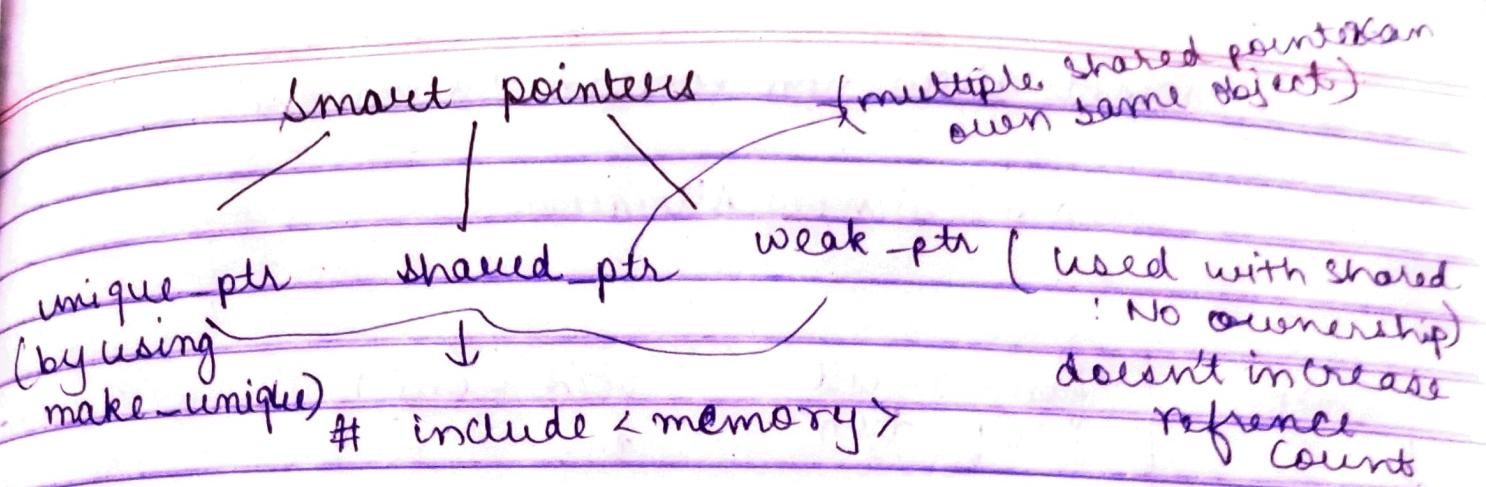
cout << *sp;

return 0;

};

O/P: 20

Since it deallocates the memory sometimes pointers which are deallocated is called which may lead to compiler error.



Errors in C++

- 1) Syntax Error → like putting Semicolon
- 2) Semantic Error → statement doesn't makes a sense
($16 = x$)
- 3) linker error. → If funcn. definition is not present which is called.
- 4) Runtime Errors → when program is running ($x/y, y=0$)
- 5) Logical errors. → $x = a + b/2$ (forgot to put brackets)

Compilation process

preprocess directives → compiled → binary code

O/P ← an executable ← linking of the func file