OBJECT ORIENTED PROGRAMMING

`Notes by:` `Syeda Samaha Batool Rizvi`

▼ WEEK 1:

```
/*also known as Object*/class students //entity
{
int std_id= 1234;//attributes
void study()//behaviour
{
}
};
//single occurence of the class -> instance ->object
//therefore Object != object
```

steps to translate scenario into a program

1. identify all entitties
2. identify all attributes and behaviours
3. shortlist all relevant entities

Entities in university

▼ staff

1. teachers

2. workers
3. management

▼ students

1. id
2. batch number

▼ degrees

1. name
2. stds in each degree
3. completion years

▼ society

1. president name
2. events
3. budget

```
class car{
string color;
int capacity;
string odel;
void race()
{
}
void brake()
{
}
};
int main(){
car car1;
car1.color="blue";
}
```

▼ WEEK 2:

OBJECT

anything for which we want to save information

Something tangible

something that can be captured intellectually

an OBJECT has:

- attributes
- behaviors

Information hiding:

The user can’t see the information but can perform it.

Advantages of information hiding

- simplicity
- prevent stealing
- prevent accidental changes

Abstraction:

abstraction is same as information hiding.

it means hiding unnecessary details

we achieve abstraction as a result of encapsulation

```
class car
{
//public:
string color;
void accelerate()//bold writing i s encapsulation
{
cour << "accelerate";
}//italic is information hiding
void brake()
{
};
int main()
{
car ob1;
ob1.accelerate();
}
```

Pillars of OOP:

1. encapsulation
2. abstraction
3. inheritance
4. polymorphism
5. code reuse

💡 standard practice: variables → private | functions → public

```
class student{
float cgpa;
public:
void setgpa(float par)//setter || mutat
or
{if(par <0 || par >4.0)
cout<<"gpa is not valid";
}
else{
cgpa=par;
}
float getdata()//getter || accesor
{
return cgpa;
}
};
int main(){
 student samaha;
samaha.setdata(3.43)
cout<< samaha.getdata();
}
```

```
class bank_acc{
string acc_no;
float balance;
string password;
public:
void deposit(float a, string p)
if (p.equals(pass))
{balance+=a;}
}
void withdraw(float a, string p)
{
if(p.equals(pass){
}
};
int main(){
banc_acc acc1;
acc1.deposit(10, abc);
acc1.withdraw(5, abc);
```

STRUCT	CLASS
members are public by default	members are private by default
struct are value type (stack)	classes are reference type(heap)
struct can not be abstract	classes can be abstract

ARE SETTERS AND GETTERS BAD?

```
class A{
int x;
float y;
public:
void setx(int a){
x=a;}
int getx(){
return x;}
void sety(float b){
y=b;}
float gety(){
return y;}
A (int p )
{
x=p;}
int main(){
A ob;
cout<<ob.getx();
```

```
A ob1(5);
}
```

💡 for each variable, we need separate setter and getters another problem is that we cant call every variable in a single function. will have to pass extra arguments and indexing will take space

💡 so the solution for this problem of getters and setters is **constructor.** parameters should be different for every constructor

CONSTRUCTORS

- always are named same as class name
- no return types
- action which I have to do while constructing an object are done in constructor

parameter signatures for constructors

- order of parameter
- number of parameters
- types of parameters

💡 A(int p, float x) ≠ A(float x, int p)

▼ WEEK 3

CONSTRUCTORS

types of constructors:

▼ compiler provided default constructor

```
class A
{
int x;
public:
A()
```

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

▼ user-defined default constructor

```
class A
{
int x;
public:
A()
{
cout <<"my constructor";
}
};
int main()
{
A ob;
}
```

▼ parameterized constructor

```
class A
{
int x;
public:
A()
{
x=1;
}
A(int p)
{
x=p;
}
};
int main()
{
A ob;//1
A ob2(10);//x=p
}
```

▼ copy constructor

```
class B
{
float var;
};

class A
```

```
{
public:
A( B par)
{
par.f();
B b;
}
};
int main()
{
A a(b);
B classes
```

💡 basically, we are connecting calsses when we are supposed to use constructor of class B into class A

💡 order of constructors do not matter

```
class student
{
string name;
string ID;
public:
student()
{
cin>> name;
academics ob;
ID=ob.generate();
}
student(string name, string ID)
{
this->name=name;
this-> ID=ID;//when class nanme and parameter names are same we use this->
}
};
class academics
{
public:
string generate()
{
//id generate logic
}
};
int main()
{
student s1;
student s2("ali");
student s3("ahmed", 1234);
}
```

💡 for initialization → constructors
for modification → getter/setter

POLYMORPHISM:

- ONE OF THE PRINCIPLE OF OOP
- Same name but different behavior
- what we are exactly doing here is: “constructor overloading”

DESTRUCTOR

- Name same as class name
- normally destructors are used for the cleanup purpose
- action which i have to do while destroying an object are done in destructors
- we do not call destructors by our selves
- destructors are automatically called by compiler
- when we are using DMA we have to make destructors

```
class mycar{
string var;
public:
mycar(string var)
{
this-> var=var;
}
~mycar()
{
cout<<"destroying object:"<<var;
}};
int main(){
mycar o1("first");//construting order 1  destroying order 3
mycar o2("second");//constructing order 2  destroying order 2
mycar o3("third");//constructing order 3  destroying order 1
}
```

```
mycar o1("first");
void f()
{
mycar o2("second");
mycar o3("third");
}
int main()
{ mycar o4("fourth");
mycar o5("fifth");
}
```

```
/*destroying order
third
second
fifth
fourth
first
*/
```

POINTERS

new → runtime memory allocation
delete → free runtime memory

```
myclass
{
int *var;
public:
myclass( int value)
{
var= new int;
*var=value;
}
myclass()
{
delete var;
}
};
int main()
{
myclass o1(5);
}
```

COPY CONSTRUCTORS

```
class test{
int var;
public:
test()
{this -> var=0;
}
test(int var)
{
this->var=var;
}
void setvar(int var){
this-> var=var;}
int getvar(){
return var;
}
};
int main(
```

```
{
test t1(5);
test t2=t1;
//or test t2(t1);
t1.setvar(6);
}
//compiler provided copy constructor
```

SHALLOW COPY

all objects share same memory

if we modify the value of object 2 the value of object 1 will be changed automatically

```
class test{
int *var;
public:
test()
{var=new int;
*var = 0;
}
test(int var)
{
this->var=new int;
*this> var=var;
}
void setvar(int var){
*this-> var=var;}
int getvar(){
return *var;
}
};
inseparate{
test t1(5)allocatedt1;
//or test t2(t1);
t1.setvar(6);
}
//shaloow copy
```

deep copy

separate memory is allocated for each object

changing of value of object 2 will not affect the valule of object 1

without pointers there is deep copy in compiler provided default copy constructor

```
class test{
int *var;
public:
test()
{var=new int;
*var = 0;
}
```

```
test(int var)
{
this->var=new int;
*this> var=var;
}
void setvar(int var){
*this-> var=var;}
int getvar(){
return *var;
}
test (const test & o)
{
this->var=new int;
this-> var=*o.var
}
};
int main(
{
test t1(5);
test t2=t1;
//or test t2(t1);
t1.setvar(6);
}
//userdefinede copy constructor deep copy
```

▼ WEEK 4

Pointers with constants

💡 Constants are not changed throughout the program.
Declaration and initialization of the constant must be at the time of declaration

```
const int a;//not posssible
const int a=5;//possible
```

▼ Non-constant pointers and Non-constant data:

```
int a=5;
int b=16;
int *ptr=&a;
    ptr=&b;
ptr=15;
```

▼ Non-constant pointer and constant data:

```
int a=5;
int b=16;
const int *ptr=&a;
          ptr=&b;
          ptr=15;//not possible
```

▼ Constant pointer and Non-constant data:

```
int a=5;
int b=16;
int *const ptr=&a;
          ptr=&b;//not possible
          ptr=15;
```

▼ Constant pointer and constant data:

```
int a=5;
int b=16;
const int *const ptr=&a;
          ptr=&b;//not possible
          ptr=15;//not possible
```

Theoretically, the 4th case is best according to the principle of least privilege because it is most restricted
constant pointer and constant data

Application of constant in classes

▼ constant as member:

```
class A{
const int y=5;
const int x;//not possible
x=5;//not possible declaration and initialization at the same time
public:
//member functions
};
```

▼ constant parameter:

```
class A{
int y;
public:
void f(const int p, int x)//const parameters are those which can not be changed or
```

```
 modified within a function
{
p=5;//not possible
cout<<p;
}
```

▼ constant function:

💡 there is no concept of constant function in global functions. else it won't work

```
class A{
int y;
public:
void f()const//can not change variables of classes. only read the values. eg: getters
{
y=5;//not possible
cout<<y;
}
```

▼ constant object:

```
class A{
int y;
public:
void f()const
{
y=5;//not possible
cout<<y;
}
void f2()
{
y=5;
cout<<y;
}
};
int main(){
A ob1;//normal objects can call both constant and non constant function
ob1.f();
ob1.f2();
const A ob2;//constant object can call only  constant function
ob2.f();
ob2.f2();//not possible
```

▼ WEEK 5

```
class employee{
const int emp_id;
mutable string name;//mutable give us an extra flexibility that we can edit a selective
```

```
variable in a constant function
float salary;
public:
employee(){
}
//this is a member initialization list that is only reserved for constructors
employee (string n, int e):emp_id(e), name(n)//with this we can give value to constant
 at runtime
{
}
void f() const
{
name="ali";//a constant function can modify the valu eof mutable variable
salary=5000;//not possible
}
};
int main(){
employee e1("ali", 54000);
employee e2("ahmed", 700000);
return 0;
}
```

Static

```
class employee{
string e_name;//instance variable(non-static)
static int count_of emp;//static variable
float salary;
public:
employee(string e, float s): e_name(e), salary(s)
{
}
void increment(){
count_of_emp ++;
}
};int employee::count_of_emp=0;
//:: means scope resolution operation
int main(){
employee e1("ali", 200.5);
employee e2("ahmed", 5.44);
employee e3("abbas", 55.5);
e1.increement();//coount_of_emp=1
e2.increement();//count_of_emp=2
e3.increement();//count_of_emp=3
employee::count_of_employee=5;//preferable syntax
}
```

💡 common data for all objects → static variable
common behavior for all objects → static function

💡 static variable→ can be modified
constant→ can not be modified

```
class student{
public:
string name;
string id;
staic string campus;
void f1(){
name="Ali";
campus="lhr";
f2();//can use static function in instance function
}
static void f2(){
campus="lhr";
name="ali";//cannot use instance member in static function
//for above not possible case
//make a temporary object to use an instance member in static
student temp(){
temp.name="ali";
temp.f1();
}
f2(student ob)//make parameter
{
o.name="ali";
o.f1;
}
}
};string student::campus;

int main(){
student s1;
student s2;
s1.campus="khi";
student :: campus="isb";
student :: f2();
student :: f2(s2);
s1.f2(s1);
```

Inline functions

💡 it is an advice not a command
this concept is not in classes because member function in classes are by default inline

```
inline int square(int a){//we put inline before a global function in order to give advi
ce to the compiler whether it wants to make it inline or not
return a*a;
```

```
}
int main()
{
int i=5;
cout<<square(i);//25
}
```

Drawbacks

- code repetition

cases where compiler will not make inline at any cost

- when the function has loops
- when the function is recursive
- when function has switch or goto
- when function has a static variable
- when function has a return type other than void and it does not return a value

UML (universal markup language)

> documentation of code for non-technical user

Class diagram

-	private
+	public
#	protected

there is no need to represent pointers in class diagram

```
class student{
int id;
public:
string name;
protected:
string address;
public:
student(){
}
student (int x){
}
void f1(int a){
```

```
}
int f2(){
}
};
```

student
-ID:int +name:string #address:string
+f1(int):void +f2 ():int +student() +stident(int)

Array of objects

▼ compile-time array using the default constructor

```
class A{
int x;
public:
a(){
x=1;
}
A(int x){
this->x=x;
}
};
int main(){
A arr[2];
}
```

▼ compile-time array using parameterized constructor

```
class A{
int x;
public:
a(){
x=1;
}
A(int x){
this->x=x;
}
};
int main(){
A arr[]={A(5), A(4)};
}
```

▼ runtime array using default constructor

```
class A{
int x;
public:
```

```
a(){
x=1;
}
A(int x){
this->x=x;
}
};
int main(){
A *arr;
f(arr);
}
void f(A *arr)
{
arr=new A[5];
}
```

▼ runtime array using parameterized constructor

```
class A{
int x;
public:
a(){
x=1;
}
A(int x){
this->x=x;
}
};
int main(){
A *arr;
f(arr);
}
void f(A *arr)
{
arr=new A[5];
for(int i=0;i<5;i++)
{
arr[i]=A(i);
}
}
```

▼ WEEK 7

INHERITANCE

- HAS A → ownership relationship
- IS A → extension relationship

  EXAMPLE:

  employee is a person

  employee has a name

```
//parent class
class person{
string name;
public:
person(){
cout<<"In person";
}
person(string name){
this->name=name;
}
void showname(){
cout<<name;
}
};
//child class or derived class
class employee: public person{
string emp_id;//specialized attribute
public:
employee(){
}
employee(string n){
name=n;
}
};
int main(){
person p1("ahmed");
employee e1("ali");
p1.showname();//ahmed
e1.showname();//ali
}
```

```
//parent class
class person{
string name;
public:
person(){
cout<<"In person";
}
person(string name){
this->name=name;
}
void showname(){
cout<<name;
}
};
//child class or derived class
class employee: public person{
string emp_id;
public:
employee(){
}
employee(string n):person(n)//if we want to call parameterized constructor we will use
 member initialization list
{
name=n;
}
};
```

```
int main(){
person p1("ahmed");
employee e1("ali");
p1.showname();//ahmed
e1.showname();//in person ali
}
```

parent class	child class
common function and attribute	specific and unique attributes

TYPES OF INHERITANCE

- Based on relationship
  - single

```
class A{
};
class B: public A{
};
```

  - multilevel

```
class A{
};
class B: public A{
};
class C: public B{
};
```

  - multiple

```
class A{
};
class B: public A{
};
class C: public A, public B{
};
```

  - hierarchal (tree)

```
class A{
};
class B: public A{
};
class C: public A{
};
```

▼ hybrid

```
class A{
};
class B: public A{
};
class C: public B{
};
class D: public A, public B{
};
class E: public D{
};
```

▼ Based on access

▼ public

```
class A{
public:
int pubvar;
private:
int privar;
protected:
int provar;
public:
A(){
pubvar=10;
privar=20;
provar=30;
}
};
class B: public A{
public:
void f2(){
cout<<pubvar;
cout<<privar;
cout<<provar;
}
};
class C: public B{
public:
void f3(){
cout<<pubvar;//1
cout<<privar;//2
cout<<provar;//3
}
int main(){
B o1;
o1.f2();
/* 1 (10)
  2 (x)no output
3 (30) */
C o2;
o2.f3();
/* 1 (10)
```

```
  2 (x)no output
3 (30) */
o2.pubvar=5; //possible
o2.provar=5;//not possible
}
```

▼ private

```
class A{
public:
int pubvar;
private:
int privar;
protected:
int provar;
public:
A(){
pubvar=10;
privar=20;
provar=30;
}
};
class B: private A{
public:
void f2(){
cout<<pubvar;
cout<<privar;
cout<<provar;
}
};
class C: public B{
public:
void f3(){
cout<<pubvar;//1
cout<<privar;//2
cout<<provar;//3
}
int main(){
B o1;
o1.f2();
/* 1 (10)
  2 (x)no output
3 (30) */
C o2;
o2.f3();
/* 1 (x) no output
  2 (x)no output
3 (x) no output */
}
```

▼ protected

```
class A{
public:
int pubvar;
private:
int privar;
protected:
int provar;
public:
A(){
pubvar=10;
privar=20;
provar=30;
}
};
class B: protected A{
public:
void f2(){
cout<<pubvar;
cout<<privar;
cout<<provar;
}
};
class C: public B{
public:
void f3(){
cout<<pubvar;//1
cout<<privar;//2
cout<<provar;//3
}
int main(){
B o1;
o1.f2();
/* 1 (10)
  2 (x)no output
3 (30) */
C o2;
o2.f3();
/* 1 (x) no output
  2 (x)no output
3 (x) no output */
o2.pubvar=5; // not possible
o2.provar=5;//not possible
}
```

💡 Protected are by definition accessible by only immediate child class

- public variable of parent are protected variable of child
- protected variable of parent are protected variable of child

▼ WEEK 8

polymorphism

Same name but different behavior

▼ constructor overloading

```
class myclass{
int x;
public:
myclass(){
cout<<"default";
}
myclass(int i)
{
cout<<i;
}
};
int main(){
myclass obj1;//default
myclass obj2(3);//3
```

▼ function overloading

```
class myclass{
int x;
public:
void f(int i, float f){
cout<<"f1";
}
void f(){
cout<<"f2";
}
void f2(){
cout<<"f2";
}
void f(float f, int i){
cout<<"f3";
}
int f(char c){
cout<<"f4";
}
void f(int i){
cout<<"f5";
}
void f(int i=0, float f)//i=0 is default parameter
{
cout<<i;
}

};
int main(){
myclass obj1;
obj1.f(5, 2.0f);//f1
obj1.f('k');//f4
obj1.f();//f2
obj1.f2(5);//no function
```

```
obj1.f2();//f2
obj1.f(2.86f);//i
```

```
class myclass{
int x;
public:
void f(int i, float f){
cout<<"f1";
}
void f(){
cout<<"f2";
}
void f2(){
cout<<"f2";
}
void f(float f, int i){
cout<<"f3";
}
//int f(char c){
//cout<<"f4";
//}
void f(int i){
cout<<"f5";
}
void f(int i=0, float f){
cout<<i;
}

};
int main(){
myclass obj1;
obj1.f(5, 2.0f);//f1
obj1.f('k');//f5
/*integer and characateat are compatible
int will rad ascii value of character*/
```

function overloading is done within a function of the same class, not other classes. but it has same scope in global functions

▼ function overriding

```
class student{
int x;
public:
void showid(){
cout<<"k11 1234";
}
};
class cs_student:public student{
public:
void showid(){
cout<<"cs- k11- 1234";
```

```
}
};
int main(){
cs_student s1;
s1.showid();//cs_k11-1234
}
```

```
class student{
int x;
public:
void showid(){
cout<<"k11 1234";
}
};
class cs_student:public student{
public:
//void showid(){
//cout<<"cs- k11- 1234";
//}
};
int main(){
cs_student s1;
s1.showid();//k111234
}
```

```
class student{
int x;
public:
void showid(){
cout<<"k11 1234";
}
};
class cs_student:public student{
public:
void showid(){
cout<<"cs- k11- 1234";
}
};
int main(){
cs_student s1;
s1.showid();//cs_k11-1234
student s2;
s1=s2;//reassignment within a class
}
```

```
class student{
int x;
public:
void showid(){
cout<<"k11 1234";
}
};
class cs_student:public student{
public:
```

```
void showid(){
cout<<"cs- k11- 1234";
}
};
int main(){
cs_student s1;
student *o;
o=&s1;
o->showid();//k111234
//early binding
}
```

```
class student{
int x;
public:
virtual void showid(){
cout<<"k111234";
}
};
class cs_student:public student{
public:
void showid(){
cout<<"cs- k11- 1234";
}
};
int main(){
cs_student s1;
student *o;
o=&s1;
o->showid();//cs-k11-1234
//late binding
}
```

💡 function overriding is same as function over loading but it need inheritance

💡 same name and same parameter between parent-child class parent child class are always compatible they can do implicit typecasting

▼ operator overloading

in week 10

💡 name of parameter and return type has no concern with polymorphism

▼ WEEK 9

PROBLEMS WITH MULTIPLE INHERITANCE

multiple inheritance concept is exclusively in c++
we make classes to encapsulate the functions and attributes of a single entity
there is no static constructor in c++ because constructors have different instances but static is shared between instances

```
//normal multiple inheritance
class A{
public:
void show()//overriden
{cout<<"A";}
};
class B: public A{
void show()//overriding
{
cout<<"B";}
};
class C: public A{
void show()//overriding
{
cout<<"C";}
};
class D: public B, public C{
void show()//overriding
{
cout<<"D";}
};
int main(){
A o;
o.show();
B o1;
o1.show();//B
c o2;
o2.show();//C
D o3;
o3.show();//D
}
```

```
//normal multiple inheritance
class A{
public:
void show()//overriden
{cout<<"A";}
};
class B: public A{
void show()//overriding
```

```
{
cout<<"B";}
};
class C: public A{
void show()//overriding
{
cout<<"C";}
};
class D: public B, public C{
//void show()//overriding
//{
//cout<<"D";}
};
int main(){
A o;
o.show();
B o1;
o1.show();//B
c o2;
o2.show();//C
D o3;
o3.show();//error ambigous call
}
```

```
//normal multiple inheritance
class A{
public:
void show()//overriden
{cout<<"A";}
};
class B: public A{
void show()//overriding
//{
//cout<<"B";}
//};
class C: public A{
void show()//overriding
//{
//cout<<"C";}
//};
class D: public B, public C{
//void show()//overriding
//{
//cout<<"D";}
};
int main(){
A o;
o.show();
B o1;
o1.show();//A
c o2;
o2.show();//A
D o3;
o3.show();//error ambigous call
}
```

💡 such error is called diamond problem in orader to solve this we move towards virtual inheritance

virtual inheritance

```
class A{
public:
void show()//overriden
{cout<<"A";}
};
class B:  virtual public A{
void show()//overriding
//{
//cout<<"B";}
};
class C: virtual public A{
void show()//overriding
{
cout<<"C";}
};
class D: public B, public C{
void show()//overriding
{
cout<<"D";}
};
int main(){
A o;
o.show();
D o3;
o3.show();//C when B is comment
// if not comment then again error
}
```

```
class A{
public:
void show()//overriden
{cout<<"A";}
};
class B:  virtual public A{
void show()//overriding
//{
//cout<<"B";}
};
class C: virtual public A{
void show()//overriding
//{
//cout<<"C";}
};
class D: public B, public C{
void show()//overriding
{
```

```
cout<<"D";}
};
int main(){
A o;
o.show();
D o3;
o3.show();//A
}
```

💡 while using superclass we do not use the concept of inheritance in it at all

CHAINED METHOD CALLS

```
class test{
int x, y;
test(){
x=0;}
test & setx(int a){
x=a;
retrun this;}
test & sety(int b){
y=b;
return this;}
};
int main(){
test ob;
ob.setx(5);
ob.sety(3);
//or simply
ob.setx(5).sety(3);
/*chain of function calls
change return type from void to class name and retrun "this"
*/
```

operator overloading

💡 another type of polymorphism
defining a customized behavior of an operator is operator overloading

```
class vector{
int x;
int y;
public:
vector(){}
vector(int x, int y){
```

```
this->x=x;
this->y=y;
}
vector operator+(vector o)
{
vector temp;
temp.x=x+o.x;
temp.y=o.y;
return temp;
}
};
int main (){
vector o1(3,5);
vector o2(5,4);
vector o3=o1+o2;
//o1=calling object
//o2=argument
//or we can write it as
o3=o1.operator+(o2);
}
```

▼ WEEK 10

operator overloading

cont...

```
class vector{
int x;
int y;
public:
vector(){}
vector(int x, int y){
this->x=x;
this->y=y;
}
vector operator+(vector o)//vector =vector
{
vector temp;
temp.x=x+o.x;
temp.y=o.y;
return temp;
}
vector operator+(int val)//vector +scalar
{
vector temp;
temp.x=x+val;
temp.y=y+val;
return temp;}

/*not cahnge the value of operator permenantly*/
void operator++()//pre increament
{
++x;
++y;}
void operator ++(int/*only for identification*/)//post increament
```

```
{
x++;
y++;
}
};
int main (){
vector o1(3,5);
vector o2(5,4);
vector o3=o1+o2;
//o1=calling object
//o2=argument
//or we can write it as
o3=o1.operator+(o2);
o3=v1+3;
o3=v1=v2+3
v2++;
++v1;
}
```

why do we need to create a no parameter constructor?
to make temporary objects

negation: + → -, - → +

```
class vector{
int x;
int y;
public:
vector(){}
vector(int x, int y){
this->x=x;
this->y=y;
}
vector& operator -()//zero parameter indicates negation
{
x=x*-1;
y=y*-1;
}

};
int main (){
vector o1(3,5);
vector o2(5,4);
-v1;
cout<<v2;//polymorpihsm in objects
    //insertion
//object of ostream
cout<<v2<<v1;
    //insertion
//to overcome this a global operator overload is introduced inorder to reolve this issu
e
}
//global function
ostream & operator <<(ostream &o, vector v){
```

```
return o;//cutomize string ;
o<<"("<<"v.x"<<","<<v.y<<".";
// 0 -> copy and cout
return o;}
/*syntax
friend ostream & operator<<(ostream , vector)
```

💡 when dos an overloaded operator call?
At least one object is an operand

💡 you cant do static overloading
when the operator overload function is a member function, it must always be non-static

💡 operator overload can not change the arity of an operator
arity → number of operands

we can not overload the operators when

- ternary, if else

friend function

```
class vector{
int x, y;
public:
vector(){}
vector (int x, int y ){
this->x=x;
this->y=y;}
friend void f();//friend function can call any variable of class irrespective of privat
e/ protected
friend class test;//friend class
friend vector operator+(int , vector);
};
//global function
void f(){
vector v;
cout<<v.x;
cout<<v.y;}
vector operator+(int val, vector o){
vector temp;
temp.x=o.x+val;
temp.y=o.y+val;
retrun temp;}
int main(){
```

```
vector v1(2,5);
vector v2=3+v1;
}
```

▼ WEEK 11

pure virtual functions (abstract functions)

💡 there is no such keyword as" abstract" in c++. therefore, we use pure virtual functions

```
//normally paren tclss is incomplete/ abstract
class employee//abstract class
{
public:
virtual void work()=0
/*specifies that this function has no implementation but an only declaration
this function is used as a blueprint*/
/*virtual*/void signin(){
}
};
class pilot: public employee{
public:
work()
{
cout<<"flies plane";
}//concrete function
};
class technician: public employee{
public:
void work(){
cout<<"identifies fault";
}//concrete function
};
int main(){
employee e;//not possible
employee * e;//possible
pilot p;
e= &p;
//upcasting
//late binding in this case
e= new technician();
e->work()//late binding
e->signin();//early binding bcz non virtual function
//in case virtual is present with signin();
e-> signin()//late binding but error bcz no function in object class
}
```

💡 in an abstract class non-virtual function must be concrete or in other words concrete function must not be virtual

💡 making an abstract class → now you cannot make instance/object of abstract class
or
an abstract class can not be instantiated
At least one abstract function makes the class abstract

💡 concrete function → which has some implementation of code in it

🖊️ reasoning questions

▼ WEEK 13

GENERICS

```
// NORMAL CODE
void print(int a){
cout<<"a";
}
void print2(char a){
cout<<a;
}
void print3(string a){
cout<<a;
}
int main(){
print(5);
print2('k');
print3("string");
}
//same implementation of code in functions but different names
```

solution?

for such codes we make generics or template functions

```
template<class T> void print(T a)//here instead of class T , keyword "type name" can al
so be used
```

```
{
cout<<a;
}
template<class T> void print(T a, T b)//both parameters must be of same type
{
cout<<a<<b;
}
template<class T1, class T2> void print(T1 a, T2 b)//two different types of parameters
{
cout<<a<<b;
}
template<class T1> void print(T1 a)
{
T1 var=a;//variable of tempelate type
return var;
}
int main(){
print(5);
print('K');d
print("string");
}
```

💡 Generic functions can take any type of keyword

```
class myclass{
  public:
  template<class T1, class T2>void func(T1 a, T2 b)
    {
    cout<<a<<b;
    }//f1
  template<class T1>void func(T1 a, int b)
    {
    cout<<a<<b;
    }//f2
  template<class T1>void func(T1 par)
    {
    }//f3
void func(int a)
{
}//explicit specialization
//it can be written as
template<>void func<int>(int p)
{
}//new style syntax
};
int main(){
func(10);//f4
func("c++")//f3
func( 'K', 3)//f2
}
```

💡 always the preference is given to concrete parameters over generic

💡 function overloading can be done in generic functions only on the basis of the number of parameters only

```
//normal class
class list{
int arr[10];
public:
void insertelement(int val, int index){
arr[index]=val;
}
void remeove element(int index){
arr[index]=0;
}
};
```

```
//generic class
template<class T>class list{
T arr[10];
public:
void insertelement(T val, int index){
arr[index]=val;
}
void removeelement(int index){
arr[index]=0;
}};
int main(){
list <int> ilist;//integer
ilist.insertelement(5, 3);
list <char> clist;//character list
list <char *> slist;//string list
list<student> oblist;//class as a parameter
}
```

💡 if we want to use any function of student class in list class we have to provide checks and type of student then

```
template<class T1, class T2>class list{
//same class
}
int main(){
list <int ,char> ob1;
}
```

```
template<class T, int size> class list{//here non generic placeholder can only have int
eger compatible value
T arr[size];
//same class
}
int main(){
list<char, 5>abc;//no variable on hardoded valu in 2nd parameter
list <char, 'c'>abcd;//no variable can go, only hardcoded value in 2nd parameter
```

▼ WEEK 14

FILING

- typedef
- static_cast
- fstream

```
int main(){
typedef int i;//alias-> make a shorter and user friendly name
i var=5;
typedef float f;
f varf= 30f;
}
```

```
int main(){
char * word ="hello";
cout<<static_cast<void *> word;//address not value
//for above specific line typecast char* to void * in order to get address
cout<<word;//hello
```

▼ STREAM

▼ IOSTREAM

▼ ISTREAM

cin

▼ OSTREAM

cout

▼ FSTREAM

▼ IFSTREAM

read

▼ OFSTREAM

write

reading from the file

- get() //char
- get(c) // char
- read(char*, int) //bulk text

> 💡 for conditional reading → get
> for bulk reading → read

writing to the file

- put() //char
- write(char*, int) //bulk text

random access functions

▼ seek

▼ peek

▼ ignore

```
#include<iostream>
#include<fstream>
int main(){
string path="D:\\oop\\files\\test.txt";
//ouput block
ofstream o(path);
o.put('T');//write single character
char * inputText="this is a string";
o.write(inputText, 7);//write bulk text
o.close();//close every file opened when the use is finished in order to prevent file f
rom coruption
//input block-> to read from file
ifstream i(path);
char c;
c=i.get();//read single character
cout<<c;
char* output= new char[10];
i.read(output, 7);//read bulk text
for(int i=0;i<7;i++){
cout<<output[i];//print the readed text on console
}
char ct;
while(!i.eof()){
i.get(ct);
```

```
if(!(ct=='x'){
i.get(c);
cout<<c;
}
}
//default case
/*
while(!i.eof()){
i.get(c);
cout<<c;
}
*/
while(!i.eof()){
i.get(c);
if(!i.eof())
{
cout<<c;}
}
i.close();
}
```

▼ WEEK 15

DEFAULT BEHAVIOUR

→write pointer at begining

→read pointer at begining

→ overwrite mode

> To change this default behaviour
> we use random access
> →seekg
> →peekg

writing an object to a file

```
#include<iostream>
#include<fstream>
#inlcude<string>
using namespace std;
class employee{
char* name;
int age;
public:
employee(){
}
employee(char*n,int a){
this->name=n;
this->age=a;
```

```
}
void display(){
}
};
int main(){
employee e1("ahsan", 43);
employee e2("ali", 30);
ofstream os("myfile.txt",ios::app);//open in append mode
os.write((char*)&e1,sizeof(e1));
os.write((char*)&e2,sizeof(e2));
os.close();
//reading
employee e;
ifstream is("myfile.txt");
is.read((char*)&e,sizeof(e));//sizeof(e1)->size of employee class, same for all objects
is.close();
}
```

RANDOM ACCESS

```
#inlcude<iostream>
#include<fstream>
#include<string>
using namespace std;
int main(){
string path="D:\\oop\\files\\test.bin";
char* input="this is my input text";
ofstream o(path);
o.write(input,strlen(input));//overwrite
o.peekp(5,ios::beg);//set pointer at begining
o.peekp(-4, ios::end);//set pointer at -4 spaces from end
o.seekp(-4,ios::cur);//set pointer at -4 spaces fom current location
o.write(input,strlen(input));
o.close();
char* output=new char[10];
ifstream i(path);
i.read(output, 10);
for(int i=0;i<10;i++){
cout<<output[i];
}
i.seekg(4,ios::beg)//read fom 4 space from begining
i.seekg(4,ios::cur);
char c=i.get(c);
cout<<c;
i.read(output, 10);
for(int i=0;i<10;i++){
cout<<output[i];
}
i.close();
}
```

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

```
#include<string.h>

class Myclass
{
    int x;
    public:
    Myclass(){}
    Myclass(int x){this -> x = x}
    void display()
    {
        cout << x << endl;
    }
};


int main()
{
    string path = "C:\\UNIPRACTICEFiles\\Hello.txt";
    char* input = "This is our string";

    ofstream o(path);
    o.seekp(3); // skips 3 index from current position
    o.seekp(2, ios::beg); // skips 2 from beginning
    o.write(input, strlen(input));
    o.close();

    ifstream i(path);
    char* output = new char[10];
    char c = i.get(); //cursor is now on index 1 of the file
    i.seekg(4); //skipping for positions //we can add the second arguement to guide it
 from where will it skip the 4 indexes from
    //i.seekg(-4,ios::end);
    i.read(output,5); //reads 5 characters from where the cursor is right now
    for(int i=0; i<5; i++) cout << output[i];
    i.close();
}


//Now using objects with random access function in filing:

#include<iostream>
#include<fstream>
using namespace std;
#include<string.h>

class Myclass
{
    int x;
    public:
    Myclass(){}
    Myclass(int x){this -> x = x}
    void display()
    {
        cout << x << endl;
    }
};


int main()
{
    Myclass o1(5);
```

```
    Myclass o2(10);
    Myclass o3(0);
    string path = "C:\\UNIPRACTICEFiles\\Hello.txt";
    ofstream o(path);

    o.write((char*)&o1, sizeof(o1));
    o.write((char*)&o2, sizeof(o2));
    o.close();

    //for reading:

    ifstream i(path);
    i.seekg(sizeof(o1), ios::beg); //skipping all bytes of the first object
    i.read((char*)&o3, sizeof(o3));
    o3.display();
    i.close();
}
```

EXCEPTION HANDLING

Run time anomalies or abnormal conditions

→ results in crashing the program

most common exceptions

- divide by zero
- Bad memory allocation
- Using an out of range index
- Stack overflow

> TO handle these exceptions we use
> ”TRY”, “CATCH” and “THROW” (there is one more which is not commonly used is “FINALLY”)

```
try {//eception code
//ideally less code should be in the try block
}
catch(exception ob.type){//code to handle the exception
}
//other catch blocks if any
catch(...){//default catch block..must be in last
}
```

▼ WEEK 16

▼ revision

abstraction:

hiding unnecessary information

does not want anyone to change the details, only modification is allowed

data hiding:

hiding the variables

code hiding

hiding any code of a function for some purpose

> to modify or read private variables we make setter and getters

binary operator in operator overloading, only single parameter will b e used and that parameter will be right side operand

`operator overload kay ander jo code likhtay hain wo hamari personal requirments hain`

`matlab operator + main hum substract bhi kar saktay hain`

`operator overloading is not available in java and c#`

`jab bhi kisi equation ka aik bhi argument object ho to wo overloaded operator dhoondta hai`

`there is no use of inheritance if there is no  new implementation of functions or variables in the child class`