Data members

→ The class data members are declared in the same way as variables are declared.

→ In general, all data members of a class are made private to that class. This is part of the way that encapsulation is achieved.

→ If the data members are declared in the public section of the class they will be accessed using member access operator, dot (.)

Syntax:

objectName.dataMember.

- objectName → User defined object Name
- . → Member Access operator
- dataMember → data member of a class.

Member Functions

Functions that are declared within a class are called member functions.

↳ declared inside the class

→ Syntax for accessing member functions in a class

objectName.functionName(ActualArguments)

- objectName → User defined object name
- . → Member access operator
- functionName → Name of the member function
- (ActualArguments) → Arguments list to the function

C++ Classes and Objects.

08.00 → Class in C++ is the building block that leads to OOP.

→ Class is a user-defined data type that

09.00 has data members and member functions.

10.00 → Data members are the data variables and member functions are the functions used

11.00 to manipulate these variables together, these data members and member functions

12 00 → define the properties and behaviours of the objects in a class

01.00

02.00

03.00 E.g.→ Consider class of Car.

Shares diff. names & brand but all of them will share some common properties like 4 wheels, Speed Limit, Mileage range, etc.

04.00 Data Members

05.00 So here Car is class and wheels, speed limits, & mileage are their properties.

06.00

EVE

Notes → So Class → user-defined type that has → data members → data variables

+

→ member functions → functions used to manipulate these variables together,

So class Car

{

data members { speed limit
mileage.

member functions → applying brakes
→ increasing speed, etc.

An Object is an instance of a Class.

When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated).

Defining class and Declaring objects

class (Keyword) ClassName (user-defined name)
{
Access specifier; // can be private, public or protected
Data members; // variables to be used
Member functions() // Methods to access data members
{ }
};
// class name ends with a semicolon

Declaring Objects

Why do we need to create objects?

-> When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax

ClassName ObjectName;

-> Accessing data members and member functions;

The data members and member functions of the class can be accessed using the dot('.') operator with the object.

For ex-> obj -> object Name.

printName()-> member function

obj.printName()

-> The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object.

↳ Accessing a data member depends solely on the access control of that data member. This access control is given by access modifiers in C++

public <- private -> protected

Sunday 24

Gratitude is the memory of the heart

```
using namespace std;
class Student {
    // Access specifier
public:
    // Data Members
    string studentname;
    // Member Function()
void printname() { cout << "Student name is:" <<
                studentname; }

};

int main()
{
    // Declare an object of class student
    Student obj1;
    // accessing data member
    obj1.studentname = "XYZ";
    // accessing member function
    obj1.printname();
    return 0;
}
```

Student name is: XYZ

Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition.

Great causes are never tried on their merits

To define a member function outside the class definition we have to use the scope resolution :: operator along with the class name and function name.

C++

```
// C++ program to demonstrate function declaration
// outside class.

using namespace std;
class Student
{
  public:
   string studentName;
   int id;
  //printname is not defined inside class
    definition
   void printname( );

   //printid is defined inside class definition
  void printid()
    {
     cout << "Student id is : " << id;
    }
};

// Definition of print name using scope resolution
   operator ::
```

```
void Student::printname()
{
    cout << "studentname is: " << studentName;
}
int main() {
    Student obj1;
    obj1.studentName = "xyz";
    obj1.id = 15;

    // call printname()
    obj1.printname();
    cout << endl;

    // call printid()
    obj1.printid();
    return 0;
}
```

O/P: studentName is: xyz
studentid is: 15

Note: → All the member functions defined inside the class definition are by default inline, but we can also make any non-class function inline by using the keyword inline with them.
→ Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.

Note:- Declaring a friend function is a way to give private access to a non-member function.

Great genius has the shortest biographies

OCTOBER NOVEMBER DECEMBER

Constructors

- Constructors are special class members which are called by the compiler every time an object of that class is instantiated.
- constructor have the same name as the class and may be defined inside or outside class definition. There are 3 types of constructors
  - → Default Constructors
  - → Parameterized constructors
  - → Copy constructors

C++ program to demonstrate constructor.

```
using namespace std;
class Student
{
public:
int id;

//Default Constructor
Student()
  {
  cout<< " Default constructor called "<<endl;
    id=-1

//Parameterized Constructor
Student (int x)
  {
  cout<< "Parameterized Constructor called "<<endl;
```

```
        id=x;
    }
};

int main() {
    // obj1 will call Default constructor
    Student obj1;
    cout<<" student id is: "<<obj1.id<<endl;

    //obj2 will call Parameterized Constructor
    Student obj2(21);
    cout<<" student id is: "<<obj2.id<<endl;
    return 0;
}
```

O/P :- Default Constructor called
Student id is: -1
Parameterized Constructor called
Student id is: 21.

→(1) A Copy Constructor creates a new object, which is an exact copy of the existing object. The compiler provides a default copy constructor to all the classes.

Syntax:

```
class-name (class-name &) { }
```

Eg→

```
Sample (Sample &t)
{
    id=t.id;
}
```

(t → object)

(2) // copy constructor takes a reference to an object of the same class as an argument.

→ In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a copy constructor

He who laughs last, Laughs longest

* A destructor takes no arguments and has no return type (not even void).
* One cannot take its address.
* One cannot declare destructors as const, volatile
* One can declare " " virtual or pure virtual
* Destructors are used to deallocate memory and other cleanup for an object and its members when the object is destroyed.

30 SATURDAY
Sep 2023 Week 39 / 273-092

SEPTEMBER 2023
S M T W T F S S M T W T F S
1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30

Destructor

It is another special member function that is called by the compiler when the scope of the object ends.

```
class Student
{
public:
int id;

//Definition for Destructor
~Student()
{
cout << "Destructor called for id: " << id << endl;
}
};
int main()
{
Student obj1;
obj1.id = 7;
int i=0;
while (i < 5)
{
Student obj2;
obj2.id = i;
i++;
} // Scope for obj2 ends here

return 0;
} // Scope for obj1 ends here
```

O/P: Destructor called for id: 0
" " " 1
" " " 2
" " " 3
" " " 4
" " " 7

~~Also give 115 page no. destructor example~~

Sunday 01

Notes

09.00 " " " : 4
" " " : 7

10.00

Not (Rare known concept) → Fact.

11.00

Q:- Why doe we give semicolons at the end of class?

12.00

01.00 Data Characteristics of Copy Constructor:-

(1.) The copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.

02.00

03.00 (2) Copy constructor takes a reference to an object of the same class as an argument.

04.00
```
Sample (sample &t)
{
  id = t.id;
}
```

05.00

06.00 (3) The process of initializing members of an object through a copy constructor is known as copy initialization.

EVE (4) It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member-by-member copy basis.

Notes (5). The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.

Copy Constructor & Destructor

```
Class Car
{
    Private :
        int speed;
        int rpm;
        boolean start;

    Public :
        Car ( )                                  // default Constructor
        {
            this.speed = 0;
            this.rpm = 0;
            this.start = false;
        }
                                                 // Parameter Constructor
        Car (int speed, int rpm, boolean start)
        {
            this.speed = speed;
            this.rpm = rpm;
            this.start = start;
        }

        void startCar ( )
        {
            this.start = true;
        }
        void stopCar ( )
        {
            this.start = false;
        }
        void increaseSpeedandRpm (int speed, int rpm)
        {
            this.rpm = rpm;
```

"this" represent current object
Ex - if C1.object
Call a function
then this object
will represent
C1.

```
        this.speed = speed;
    }

    Void decreaseSpeedAndRpm(int speed, int rpm)
    {
            this.rpm = rpm;
            this.speed = speed;
    }

    ~Car()                    //destructor
    {
        cout<< "destructor called" << endl;
    }

    Car(Car &c)               // copy Constructor
    {
        this.speed = c.speed;
        this.rpm = c.rpm;
        this.start = c.start;
    }
};
Void main()
{
Line 1    Car C1, C2(10, 1000, true);
Line 2    C1.startCar();
Line 3    C1.increaseSpeedAndRpm(20, 1200);
Line 4    C2.increaseSpeedAndRpm(25, 1400);
Line 5    Car C3(C1);
Line 6    C3.increaseSpeedAndRpm(25, 1500);
```

Line 7 C_1.decreaseSpeedAndRpm(0, 0);

Line 8 C_2.decreaseSpeedAndRpm(0, 0);

Line 9 C_3.decreaseSpeedAndRpm(0, 0);

Line 10 C_1.stopCar();

line 11 C_2.stopCar();

line 12 C_3.stopCar();

line 13 return 0;

}

this.rpm = this.rpm + rpm

Main Method

{

Line 1 → C_1

speed	0
rpm	0
start	false

C_2

speed	10
rpm	1000
start	true

Line 2 → C_1 will start the Car. and, C.start = true

Line 3 → C_1 will increase speed and rpm

C_1.speed = 20;

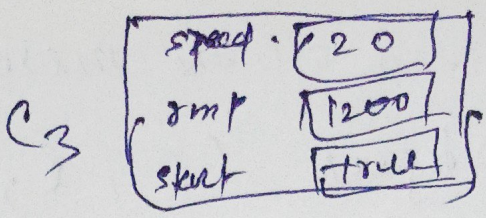
C_1.rpm = 1200;

Line 4 → C_2 will increase speed and rpm

C_2.speed = 25

C_2.rpm = 1400

Line 5 →



Line 6 → C_3 will increase the speed and rpm

$C_3.speed = 25;$

$C_3.rpm = 1500;$

Line 7 → C_1 will decrease speed and Rpm;

$C_1.speed = 0;$

$C_1.rpm = 0;$

Line 8 → C_2 will decrease Speed and Rpm

$C_2.speed = 0;$

$C_2.rpm = 0;$

line 9 → C3 will decrease Speed And Rpm

$C_3.speed = 0;$

$C_3.rpm = 0;$

line 10 → C_1 will stop car

start = false

line 11 → C2 will stop car

start = false

line 12 → C3 will stop car

start = false

line 13 → will return 0;

then destructor will call for C_1, C_2 and C_3

and clean all the memory allocated for object C_1, C_2 and C_3

1 ~~Point~~ and will print

destructor called

1 destructor called

1 destructor called

Difference between Constructor and destructor

	Constructor	Destructor
1	Constructor is used to initialize the object and it allocate the memory	Where as destructor is used to destroy the instance or object and it will deallocate ~~or~~ free the memory
2	Constructor can either arguments or without arguments	destructor can't have any arguments
3	A Constructor will called when a object is created	A destructor will call when scope of a object will end
4	Constructor can be overloaded	Destructor can't overloaded

5	The Constructor is declared with Same name as Class name	The Destructor is declared with Same name as class name with (~) tilde Symbol
6)	Constructor can be copy Constructor	While there, there is no copy destructor
7)	Constructor calls often Called in successive order	Destructor are often called in reverse order

ex

```
class Z
{
    public:
        Z( )
        {
            cout << "Constructor Called" << endl;
        }
        ~Z( )
        {
            cout << "Destructor Called" << endl;
        }
};
```

```
int main ( )
{
    Z z1, z2 ;
}
```

output-

```
Constructor Called
Constructor Called
Destructor Called
Destructor Called
```

```
cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

return 0;
}
```

O/P

p1.x = 10, p1.y = 15
p2.x = 10, p2.y = 15

* Encapsulation in C++

↳ In C++ it is defined as the wrapping up of data and information in a single unit.

↳ In OOP, Encapsulation is defined as binding together the data and the functions that manipulate them.

Eg => A real-life example of encapsulation,

company has → diff. sections like → accounts section, finance section, sales section, etc.

Now, a situation may arise when for some reason an official from the finance section needs all the data about sales in a particular month.

In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officers in the sales section and then request him to give the particular data.

This is what Encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".

Two important property of Encapsulation.

1. **Data Protection**: Encapsulation protects the internal state of an object by keeping its data member private. Access to and modification of these data members is restricted to the class's public methods, ensuring controlled and secure data manipulation.

2. **Information Hiding**: Encapsulation hides the internal implementation details of a class from external code. Only the public interface of the class is accessible, providing abstraction and simplifying the usage of the class while allowing the internal implementation to be modified without impacting external code.

Sunday 08

For example, if we give input, and output should be half of input.

```
Eg-> #include <iostream>
     using namespace std;
     class temp{
         int a;
         int b;
```

Notes

```
public:
int solve (int input) {
    a = input;
    b = a/2;
    return b;
  }
};
int main () {
  int n;
  cin >> n;
  temp half;
  int ans = half.solve (n);
  cout << ans << endl;
}
```

Features of Encapsulation:

1. We can not access any function from the class directly. We need an object to access that function i.e using the member variables of that class.
2. The function which we are making inside the class must use only member variables, only then it is called encapsulation.
3. If we don't make a function inside the class which is using the member variable of the class then we don't call it encapsulation.
4. Encapsulation improves readability, maintainability and security by grouping data and methods together.
5. It helps to control the modification of our data members.

Encapsulation in C++

Methods | variables
Class

Encapsulation also leads to data abstraction
Using encapsulation also hides the data,

Ex - program explained in the other notebook.

Encapsulation

Data Protection

```
class Student
{
    private:
        int rollNumber; // Data Protection

    public:
        int age; // This is not Data Protection
        void display()
        {
            cout << "roll Number: " << rollNumber;
        }
};                          // Information Hiding

void main()
{
    Student s;
    s.display();
    s.rollNumber = 1;   // error because is private member
    s.age = 24;         // No error because its public member
    return 0;
}
```

Encapsulation

Encapsulation

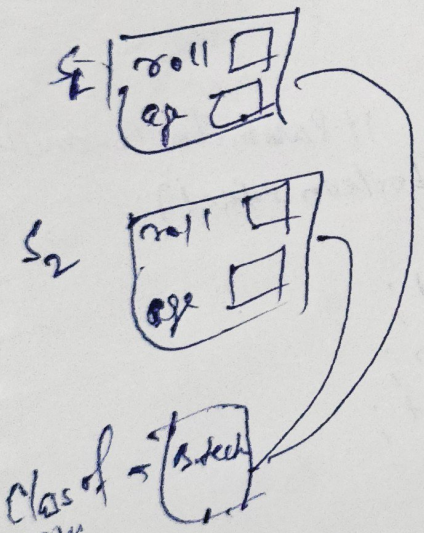
```
Class Student
{
    private:
        int rollNumber;
        int age;

    Public:
        student(int r, int a)
        {
            rollNumber = r;
            age = a;
        }
        void display()
        {
            cout << "roll: " << rollNumber;
            cout << "age: " << age;
        }
};

int main()
{
    student s1(1, 24), s2(2, 25);
    s1.display();
    s1.age = 25;   // it will give error because it is a private member.
    return 0;
}
```

Not:- S_1 don't know the information of S_2 and vice versa that is called information hiding.

Static Member function & data member

```
class Student
{
    private:
        int roll;
        int age;
        static string classofstu;

    Public:
        static string classofstu; // static data member
        Student (int r, int age)
        {
            roll = r;
            age = age;
        }
        void display ()
        {
            Cout << "Roll: " << roll;
            Cout << "Age: " << age;
            Cout << "Class: " << classofstu;
        }
};
```

```
static void print( )        // static Member function
{
    cout << " Roll: " << roll;     // error
    cout << " Age : " << age;      // error
    cout << " Class: " << class of stu;  // No error
}

string student:: class of stu = " B.tech ";  // initialize the variable
                  ↳ scope resulation operator

int main( )
{
    student S1(1, 24), S2(2, 25);

    S1.display( );    // No error
    S2.display( );    // No error

    S1.print( );      // No error
    S2.print( );      // No error

    student::print( );    // No error

    student::display( );  // error

    return 0;
}
```



Only can call static function using class Name

Can't call non-static function using class Name.

Characteristics of an Object

All individual objects Possess three basic characteristics

1) Identity
2) state
3) behaviour

→ **Identity**

Each objects' name or identity, i's Unique and distinct from other objects

Note:- name of a object

→ **State**

It refers to the Properties of an object for Example. values of variables in the object & Contain data can added, Changed or deleted.

Note:- All data member value can change, added, or Delete

→ **Behavior** (functionality)

refer to actions that the object can take

Note. All member function of class

Dynamic Memory Allocation & Deallocation

(Garbage collections)

~~Difference between~~

How is it different from memory allocated to normal variables

Static Memory Allocation

```
int a;
char str[10];
Car c1;
```

Memory is automatically allocated and deallocated

Dynamic Memory allocation

```
int *p = new int[10];
Car *c1 = new Car;
int *x = NULL; // not allocate memory
x = new int; // Allocate memory
```

Memory will allocate dynamically at run time.

memory deallocation need to be done manually

How is memory allocated / deallocated

- In C Uses the malloc() and calloc() function to allocate memory dynamically and free() to allocate
- In C++ all support malloc() and calloc() free() function but also has two operators

"new" to allocate memory "delete" to de-allocate the memory

Syntax —

Pointer-Variable = new data-type ;

example —

int *p = NULL ; →

p = new int ; →

or

int *p = new int ; →

Car *c1 = NULL ; →

c1 = new Car ; →

or

Car *c1 = new Car ;

or

Car *c1 = new Car(10, 1200) ;

Full Example —

```
class car
{
  int speed;
public:
  car() {
     this -> speed = 0;
  }
  car (int speed) {
    this -> speed = speed;
  }
  void print() {
      cout << this -> speed << endl;
  }
};
int main()
{
  int* x = new (nothrow) int(10); // allocate the memory
  cout << "Value of x: " << *x << endl;
  delete x; // deallocate the memory
  int* y = new (nothrow) int; // if no memory available
  if (!y) {                   // it will assign by NULL
   cout << "Memory allocation failed" << endl;
  }
  else
  {
   *y = 30;
   cout << "Value of y: " << *y << endl;
   delete y;
  }
```

For pointer object we must use (→) arrow in place of (.) dot

if no memory available then it give error and program will stop

```
Car* var1 = new Car;   // inOrder to access pointer variable we did "this ->"
 //OR
// Car var1 = new Car();
 var1 -> print(); // it will print 0
 Car* var = new Car(25);
 var -> print();  // it will print 25
  delete var1, var; // deallocate these memory for both object.
  return 0;
}
```

Dynamic Constructor.

- The constructor which allocates a block of memory that can be accessed by the objects at run time is known as dynamic constructor.
- In simple words, a Dynamic constructor is used to dynamically initialize the objects, that is memory is allocated at run time.

- Dynamic Constructor in C++

When allocation of memory is done dynamically using dynamic memory allocator new in a constructor, it is known as dynamic constructor. By using this, we can dynamically initialize the objects.

Ex-1:

```
class Student{
    const char* s;
public:
    // default constructor
    Student()
    {
        // allocating memory at run time
        s= new char[10];
        s= "student";
    }
    void display() { cout << s << endl; }
};
```

```
int main ()
{
    student obj;
    obj display ();
}
```

O/P : student

Explanation: In this, we point data member of type char which is allocated memory dynamically by new operator and when we create dynamic memory within the constructor of class that is known as dynamic constructor.

Ex-2:

```
class Student {
    int* s;
public:
    // default constructor
    student ()
    {
        //allocating memory at run time
        // and initializing
        s = new int[3] {1, 2, 3};
        for (int i = 0; i < 3; i++) {
            cout << s[i] << " ";
        }
        cout << endl;
    }
};
```

Notes

```
int main()
{
    // five objects will be created
    // for each object
    // default constructor would be called
    and memory will be allocated to
    array dynamically
    Student* ptr = new Student[5];
}
```

Output

123
123
123
123
123

Explanation: In this program we have created array of object dynamically. The first object is ptr[0], second is ptr[1] and so on. For each object creation default constructor is called and for each object memory is allocated to pointer type variable by new operator.

Ex-3:

```
class students{
    int* s;
public:
    // default constructor
    students()
    {
        // allocating memory at run time
        s = new int;
```

```
    *s = 0;
}
//parameterized constructor
Students(int x)
{
  s = new int;
  *s = x;
}
void display (0)
{
 cout << *s << endl;
}
//Deallocate the dynamically allocated memory
~Students()
  {
    delete s;
  }
};
int main()
{
  // default constructor would be called
   students obj1 = Students();
    obj1.display();

  // parameterized constructor would be called
   Students obj2 = students(7);
    obj2.display();
}
```

output :→ 0
7

He who sups with the Devil should have a long spoon

NOVEMBER
DECEMBER

Explanation: In this integer type pointer variable is declared in class which is assigned memory dynamically when the constructor is called. When we create object obj1, the default constructor is called and memory is assigned dynamically to pointer type variable and initialized with value 0. And similarly when obj2 is created, parameterized constructor is called and memory is assigned dynamically.

08.00

09.00

10.00

11.00

12.00

Type Conversion in C++

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. Implicit Type Conversion: also known as 'automatic type conversion'.
   - Done by the compiler on its own, without any external trigger from the user.
   - Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
   - All the data types of the variables are upgraded to the data type of the variable with largest data type.

     bool → char → short int → int → unsigned int → long → unsigned → long long → float → double → long double.

   - It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Notes

Example of Type Implicit Conversion:

```
// An example of implicit conversion
#include <iostream>
using namespace std;
```

```
int main()
{
    int x=10;  //integer x
    char y='a'; //character c
    //y implicitly converted to int. ASCII
    // value of 'a' is 97
    x=x+y;
    //x is implicitly converted to float
    float z=x+1.0;

    cout<<"x= "<<x<<endl
        <<"y= "<<y<<endl
        <<"z= "<<z<<endl;
    return 0;
}
```

Output:

x=107
y=a
z=108.

Explicit Type Conversion:

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

- Converting by assignment: This is done by explicitly defining the required type in front of

the expression in parenthesis. This can be also considered as forceful casting.

Syntax:

(type) expression

where type indicates the data type to which the final result is converted.

Ex:-

```
// C++ program to demonstrate explicit type casting
#include <iostream>
using namespace std;
int main()
{
    double x = 1.2;
    // Explicit conversion from double to int
    int sum = (int) x + 1;
    cout << "sum = " << sum;
    return 0;
}
```

O/P: sum = 2

- Conversion using Cast operator: A Cast operator is an unary operator which forces one data type to be converted into another data type.

C++ supports 4 types of casting:

1. Static Cast
2. Dynamic Cast
3. Const Cast.
4. Reinterpret Cast.

Ex:-

```
int main()
{
  float f = 3.5;
  Using cast operator
  int b = static_cast<int>(f);
  cout << b;
}
```

Output:
3

Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of diff. data types.

Operator Overloading in C++

Operator is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.

→ In C++, we can make operators work for user defined classes.

→ This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

→ For ex:- we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

→ Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers etc.

```
Ex:- int a;
     float b, sum;
     sum = a + b;
```

Here, variables "a" & "b" are of types "int" and "float", which are built-in data types. Hence the addition operator '+' can easily add the contents of "a" and "b". This is because the addition operator "+" is predefined to add variables of built-in data type only.

NOVEMBER

DECEMBER

Sunday 29

Ex:- // C++ program to demonstrate the working logic behind Operator Overloading.

```
class A {
    statements;
};

int main()
{
    A a1, a2, a3;
    a3 = a1 + a2;
    return 0;
}
```

In above program we've taken 3 variables of type "class A". Here, we're trying to add two objects of type class A (which is user-defined) using the "+" operator. This is not allowed because the addition operator "+" is predefined to operate only on built-in data types. Since here "class A" is user-defined type so the compiler generates error. Thus, this is where the concept of "operator Overloading" comes in.

Now if the user wants to make the operator "+" add two class objects, the user has to redefine the meaning of the "+" operator such that it adds two class objects. This is done by using the concept of "operator Overloading". So the main idea behind "operator overloading" is to use C++ operators with class variables or class objects.

Redefining does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

08.00

09.00

Complex-number = x + iy

real part / imaginary part

imaginary number

10.00

11.00

12.00 Ex→

```
class Complex {
  private:
    int real, imag;
  public:
    Complex (int r=0, int i=0)
    {
      real = r;
      imag = i;
    }
```

01.00

02.00

03.00

04.00

05.00 //This is automatically called when '+' is used with between two Complex objects

06.00

```
Complex operator+ (Complex const& obj)
{
  Complex res;
  res.real = real + obj.real;
  res.imag = imag + obj.imag;
  return res;
}
  void print() { cout << real << " +i" << imag << '\n'; }
};
```

EVE

redefining

Notes

```
int main()
{
    Complex c1(10, 5), c2(2,4);
    Complex c3 = c1+c2;
    c3.print();
}
```

Output : 12 + i9

Q:- What is the difference between operator functions and normal functions?

Operator functions are the same as normal functions. The only differences are, that the name of an operator function is always the "operator" keyword followed by the symbol of the operator, and operator functions are called when the corresponding operator is used.

```
class Complex {
    private:
        int real, imag;
    public:
        Complex (int r=0, int i=0)
        {real=r;
         imag=i;
        }

void print()
{ cout << real << " +i" << imag << endl; }
```

```
// The global operator function is made friend of
this class so that it can access private
    members
friend Complex operator+ (Complex const& c1,
                          Complex const& c2);
};

Complex operator+ (Complex const & c1, Complex
                   const & c2)
{
    return Complex (c1.real + c2.real, c1.imag
                    + c2.imag);
}
int main()
{
    Complex c1(10,5), c2(2,4);
    Complex c3
        = c1
          + c2; // An example call to "operator+"
    c3.print();
    return 0;
}
```

Output: 12 + i9

- Can we overload all operators?

→ Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

Every favor you do to a rich or a poor is charity.

sizeof
typeid
Scope resolution (::)
Class member access operators (. (dot), .
Ternary or conditional (?:)

- **Operators that can be overloaded**

We can overload
- Unary operators
- Binary operators
- Special operators ([], () etc.)

But among, there are some operators that cannot be overloaded. They are
- Scope resolution operator (::)
- Member selection operator
- Member selection through * Pointer to a member variable
- conditional operator (? :)
- Sizeof operator sizeof( )

Operators that can be overloaded	Examples
Binary Arithmetic	+, -, *, /, %
Unary Arithmetic	+, -, ++, --
Assignment	=, +=, *=, /=, -=, %=
Bitwise	&, |, <<, >>, ~, ^
De-referencing	(->)

To overcome evil with good is good, to resist evil by evil is evil.

- Dynamic memory allocation, De-allocation — New, delete

08.00 - Subscript — [ ]

- Function call — ( )

09.00 - Logical — &, ||, !

- Relational — >, <, ==, <=, >=

10.00

Why can't the above-stated operators be overloaded?

11.00 1. sizeof - This returns the size of the object or datatype entered as the operand. This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a

01.00 pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using

02.00 overloading would cause a fundamental part of the language to collapse.

03.00

2. typeid: This provides a CPP program with the

04.00 ability to recover the actually derived type of the object referred to by a pointer or reference. For this operator

05.00 the whole point is to uniquely identify a type. If we want to make a user-defined type 'look' like another type;

06.00 polymorphism can be used but the meaning of the typeid operator must remain unaltered, or else serious issues could arise.

Sunday 05

EVE

→ typeid operator is to check the type of the variable.

Notes

Ex:- int main() {
int i, j;
char c;
// Get the type info using typeid operator

```
const type_info& ti1= typeid(i);
"       "      " ti2= typeid(j);
"       "      " ti3 = typeid(c);

// check if both types are same
if (ti1 == ti2)
cout << " i and j are of " << "similar type" << endl;
cout << "i and j are of " << "different type" << endl;

// check if both types are same
if (ti2 == ti3)
cout << " j and c are of " << "similar type" << endl;
cout << " j and c are of " << "different type" << endl;

return 0;
}
```

O/P: i and j are of similar type
j and c are of different type

3. Scope resolution (::): This helps identify and specify the context to which an identifier refers by specifying a namespace. It is completely evaluated at runtime and works on names rather than values. The operands of scope resolution are note expressions with data-types and CPP has no syntax for capturing them if it were overloaded. So it is syntatically impossible to overload this operator.

4. Class member access operators (.(dot), .* (pointer to member operator)): The importance and implicit use

of class member access operators can be understood through the following example:

Ex: 08.00 // C++ program to demonstrate operator overloading using ~~Ex~~ ~~class~~ dot operator.

09.00

```
class ComplexNumber {
10.00 private:
        int real;
11.00   int imaginary;
      public:
12.00   ComplexNumber (int real, int imaginary)
        {
01.00     this -> real = real;
          this -> imaginary = imaginary;
02.00   }
      void print() { cout << real << " + i" << imaginary; }
03.00 ComplexNumber operator+ (ComplexNumber c2)
        {
04.00     ComplexNumber c3 (0, 0);
          c3.real = this -> real + c2.real;
05.00     c3.imaginary = this -> imaginary + c2.imaginary;
          return c3;
06.00   }
      };
EVE   int main()
      {
        ComplexNumber c1 (3, 5);
        ComplexNumber c2 (2, 4);
        ComplexNumber c3 = c1 + c2;
        c3.print();
        return 0;
      }
```

N Notes

The best richness is the richness of the soul.

Output : 5+i9

Explanation: The statement Complex Number c3=c1+c2; is internally translated as Complex Number c3 = c1.operator+(c2); in order to invoke the operator function. The argument c1 is implicitly passed using the '.' operator. The next statement also makes use of the dot operator to access the member function print and pass c3 as an argument.

Besides, these operators also work on names and not values and there is no provision (syntactically) to overload them.

5. Ternary or conditional (?:): The ternary or conditional operator is a shorthand representation of an if-else statement. In the operator, the true/false expressions are only evaluated on the basis of the truth value of the conditional expression.

conditional statement ? expression1 (if statement is TRUE) : expression2 (else)

A function overloading the ternary operator for a class say ABC using the definition.

ABC operator?:(bool condition, ABC trueExpr, ABC falseExpr);

Notes: would not be able to guarantee that only one of the expressions was evaluated. Thus, the ternary operator cannot be overloaded.

* Important points about operator overloading

(1) For operator overloading to a work, at least one of the operands must be a user-defined class object.

(2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases.

(3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

Example:

```
# include <iostream>
using namespace std;
class Fraction {
private:
    int num, den;
public:
    Fraction (int n, int d)
    {
        num = n;
        den = d;
    }
// Conversion operator: return float value of fraction
    Operator float() const
    {
        return float(num)/float(den);
    }
};
```

```
int main()
{
    Fraction f(2,5)
    float val = f;
    cout << val << '\n';
    return 0;
}
```

O/P : 0.4

Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.

(4) Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

Ex. →

```
class Point{
    private:
        int x, y;
    public:
        Point(int i=0, int j=0)
        {
            x=i;
            y=j;
        }
        void print()
        {
            cout<<"x= "<<x<<", y= "<<y<<'\n';
        }
};
```

```
int main()
{
    Point t(20, 20)
    t.print();
    t = 30;                    // Member x of t becomes 30
    t.print();
    return 0;
}
```

O/P:

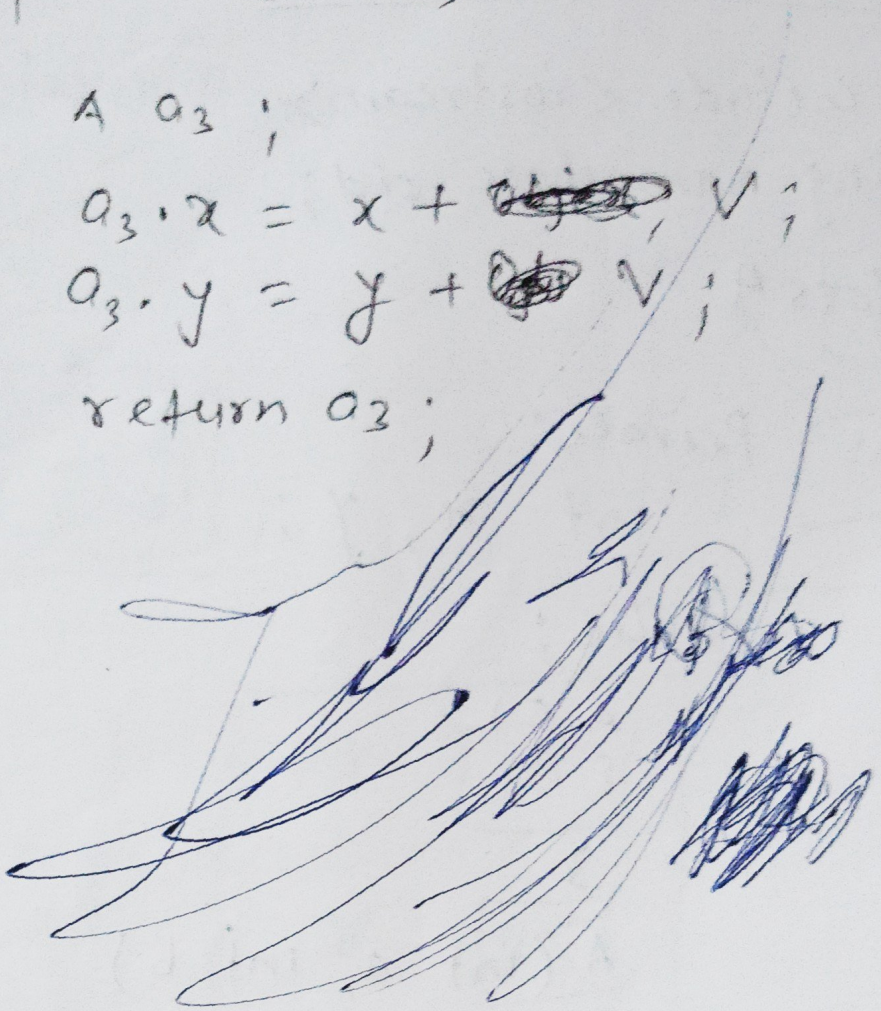
x = 20, y = 20

x = 30, y = 0

Complete Example

```
#include <iostream>
using namespace std;
class A
{
    private:
        int x, y;
    public:
        A ()
        {
        }
        A (int a, int b)
        {
            x = a;
            y = b;
        }
        A operator+(A &obj)
        {
            A a3;
            a3.x = x + obj.x;
            a3.y = y + obj.y;
            return a3
        }
```

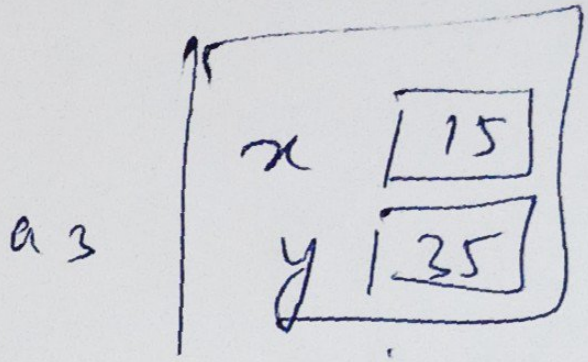
```
A operator+(int v)
{
    A a3;
    a3.x = x + v;
    a3.y = y + v;
    return a3;
}
```

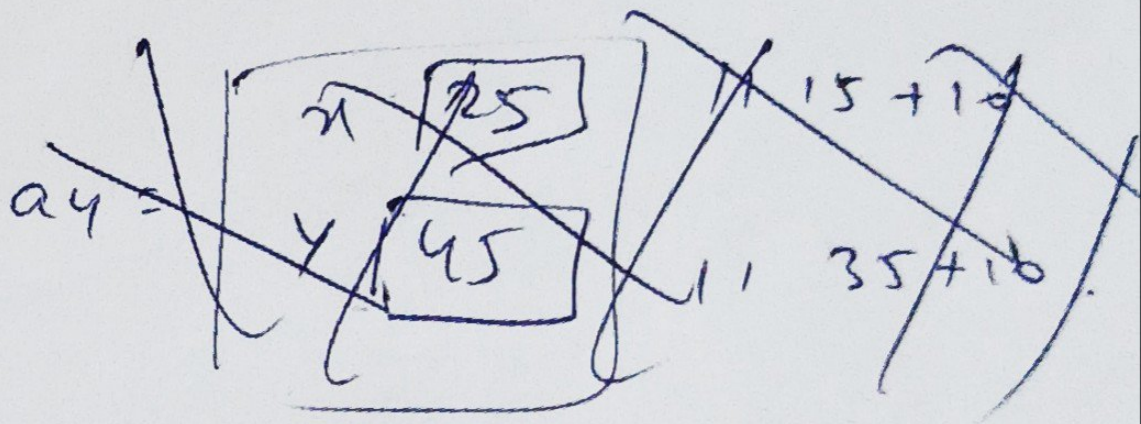


```
Void Print()
{
    cout<<"value of x:"<<x<<endl;
    Cout<<"Value of y:"<<y<<endl;
}
};
int main()
{
    A a1(10,20), a2(5,15);
    A a3 = a1 + a2;
    a3.print();
    A a4 = a1 + 10;
```

a4.print();
return 0;

O/P:





a4 = x 10+10
y 20+10