

UNIT-2

CLASSES AND OBJECTS

- The most important feature in C++ is class
- Bjarne Stroustrup → initially gave the name 'C with classes'.
- Class is a user-defined datatype and it is also called abstract Data Type

Specifying a Class

- A class is a way to bind the data and its associative functions.
- It allows the data and functions to be hidden, if necessary for external use.
- A class specification has 2 parts:

- 1) Class declaration
- 2) Member function definition

- 1) The class declaration describes the type and scope of its members
 - 2) The class function definition describes how the class' function are implemented.
- The general form of a class declaration is:

```
class class_name
```

```
{
```

```
private:
```

```
variable declaration;
```

```
function declaration;
```

```
public:
```

```
variable declaration;
```

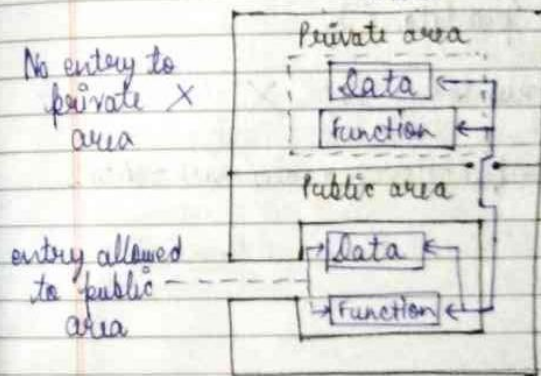
```
function declaration;
```

```
};
```

Function definition & }

Data Hiding in classes

CLASS



Example-

```
class item
```

```
{
```

```
private:
```

```
int number;
```

```
float cost;
```

```
public:
```

```
void getdata (int a, float b);
```

```
void putdata (void);
```

```
};
```

* If scope is not defined then by default it is taken as private.

* Private functions can be accessed only by public functions

* Public functions can be called by objects.

→ Accessing class members / functions

Create an object

example -

Item 0;

0. getdata(5, 1000.5);

0. putdata();

0. number = 10; X

Q- Can private member functions access each other's function?

Static Data Members

→ A data member of a class can be qualified as static.
→ The properties of a static member variable are similar to that of a C static member variable.

→ A static member variable has special characteristics. These are:

→ It is initialised to 0 when the first object of the class is created.

• No other initialisation is permitted.

→ Only one copy of the member is created for the entire class and is shared by all the objects of that class no matter how many objects are created.

→ It is visible only inside the class but its lifetime is only inside the ~~class~~ entire program.

→ Static variables are normally used to maintain values common to the entire class.

• Forexample - a static data member can be used as a counter that records the occurrences of all the objects.

Include <iostream>
using namespace std;

class item {

static int count;

int number;

public:

void getdata(int a)

{

number = a;

count++;

}

```
void getcount (void)
{
    cout << count << endl;
    count << count << "\n";
}
```

```
}
```

```
int item :: count;
```

```
int main()
{
```

```
    item a, b, c;
```

```
    a.getcount();
```

```
    b.getcount();
```

```
    c.getcount();
```

```
    a.getdata(100);
```

```
    b.getdata(200);
```

```
    c.getdata(300);
```

```
    cout << "after reading data" << endl;
```

```
    a.getcount();
```

```
    b.getcount();
```

```
    c.getcount();
```

```
    return 0;
```

OUTPUT :

```
count : 0
```

```
count : 0
```

```
count : 0
```

after reading data

```
count : 3
```

```
count : 3
```

```
count : 3
```

Static Member Functions

→ Like static member variable we have these

→ A member function declared static has following properties.

1) can have access to only static data members of same class

2) Static member can be called using the class name (not by object)

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

```
class test
```

```
{
```

```
    int code;
```

```
    static int count; // static data member
```

```
    public :
```

```
    void setcode (void)
```

```
{
```

```
        code = ++count;
```

```
}
```

```
    void showcode (void)
```

```
{
```

```
        cout << "obj. number" << code << endl;
```

```
}
```

/static member fx "

```
static void showcount (void)
```

```
{
```

```
    cout << "count : " << count << endl;
```

```
}
```

```
int test :: count;
```

```
int main()
```

```
{
```

```
    test t1, t2;
```

```
    t1.setcode();
```

```
    t2.setcode();
```

```
    test::showcount();
```

// static member function call

```

test t3;
t3.test setcode();
test::showcount();
t1.showcode();
t2.showcode();
t3.showcode();
return 0;

```

OUTPUT:

```

count : 2
count : 3
object number : 1
object number : 2
object number : 3

```

Constructor

- A constructor is a special member function whose task is to initialise the objects of its class.
- It is special because its name is the same as class' name.
- The constructor is invoked whenever the object of its associated class is created.
- It is called constructor because it constructs the values of data members of class.
- A constructor is declared and defined as follows:

```

class item {

```

```

    int x, y;

```

```

public:

```

```

    item(); // Declaration of constructor
}

```

```

item::item() // Constructor definition outside class
{

```

```

    x = 0;

```

```

    y = 0;
}

```

- When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialised automatically. For example, the declaration:

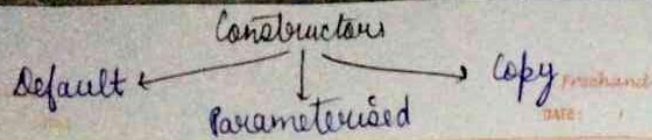

```
item int i; // object created
```

not only creates an object of type item but also initialises its data members → x, y to 0.

There is no need to write any statement to invoke the constructor function (as we do with the normal member functions)

① Default Constructors

- A constructor that accepts no parameters.



→ If no such constructor is defined, then the compiler does.

class A; A ();

→ Therefore a statement such as A a; invokes the default constructor of the compiler to create the object a.

→ The constructor function has some special characteristics

- (1) Declared in public section.
- (2) Invoked automatically when the objects are created.
- (3) They do not have any return type not even void.
- (4) They can't be inherited.
- (5) They can't be virtual.
- (6) We can't refer to their addresses.
- (7) An object with a constructor can't be used as a member of union.
- (8) They make implicit calls to the operators 'new' and 'delete' when memory allocation is required.

(2) Parameterised Constructor

→ The constructor integer defined below initializes the data members of all objects to 0.

⇒ integer :: integer () { m=0, n=0; }

→ However, to initialise the data members of different objects with different values, C++ permits to achieve this objective by passing arguments to the constructor function when the

object is created.

→ The constructors that can take arguments is called parameterised constructor

class integer

{
int m, n;
public;

integer (int x, int y);
// Parameterised constructor declaration

}
integer :: integer (int x, int y)
// Definition

{
m = x;

n = y;
}

⇒ Constructor calling

By calling constructor explicitly

integer int1 = integer (0, 100);

By calling constructor implicitly

integer int1 (0, 100);

(3) Copy constructor

- It takes an object (reference) as argument.
- It copies all the values of data members of passed object to ~~the~~ another object which makes the call.

⇒ `integer (integer &);` // declaration

$\Rightarrow \text{integer} :: \text{integer} \text{ (integer \& i)}$ // Definition

```

    integer I1(2,3);
    => integer I2(I1); // calling

```

The statement invokes copy constructor which copies the values of I_1 to I_2 .
It sets the value of every data element of I_2 to the value of the corresponding data elements of I_1 .

Destructive

A destructor as the name suggests is used to destroy the objects created by constructor.

Like a constructor, the destructor is a member function whose name is same of the class but it preceded by a tilde. (~)

The destructor of the class integer can be defined as shown below:

$$\sim \text{integer } ()$$

- (1) It never takes any argument nor it returns any value.
- (2) It will be invoked implicitly by the compiler upon the exit from the program.
- (3) To clean up storage that is no longer in use. It is a good practice to declare destructors in a program since it releases memory space for future use.

FRIEND FUNCTION

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

class. sample {

```
int a;
```

Int b;

public

```
void setvalue () {a=25; b=40;}
```

friend float mean (sample s)
acceleration

Definition

float mean (sample s)

return float (s.a + s.b) / 2.0;

```
int main() {
```

sample x ; || object x

n -th value ()

Count < "Mean value" = "mean(x)";

return 0;

Calling

- DATE: _____
- To make an outside function 'friend function'
 - declare the function with keyword 'friend' inside class
 - The function is defined like normal function. i.e. without scope operator '::' and without keyword 'friend'.
 - It can be declared as friend in any no. of classes

Properties of friend function:

- It is not ⁱⁿ the scope of the class in which it has been declared
- It cannot be called using object of that class
- It is invoked like a normal function
- It cannot access data members directly, so it uses object name and dot operator (e.g. A.x)
- It can be declared either in public or private part.
- Usually, it has objects as arguments

* Member functions of one class can be friend of another class.

↳ So, they are defined using '::'

```
class X {
```

```
    int fun1(); // member function of X
```

```
};
```

```
class Y {
```

```
    friend int X::fun1(); // fun1() of X is friend of Y
```

```
};
```

Freehand
DATE: / /

→ So `fun1()` is a member of class X and friend of class Y.

We can declare all members of one class as friend functions of another class → the class is called friend class
class Z {

friend class X; // all member functions of X are friend of Z.
};

Passing objects as arguments

→ This can be done in two ways:

- Pass by value
 - Copy of entire object is passed to function
 - changes are not reflected in object used to call
- Pass by reference
 - Only address of object is transferred to function
 - changes are reflected in object used to call

ex- class Time {
 int hours, minutes;
 public:
 void sum (Time t1, Time t2);
 // Pass by value
 void subtract (Time & t);
 // Pass by reference
 void display ();
};
void Time :: sum (Time t1, Time t2) {
 minutes = t1.minutes + t2.minutes;
 hours = minutes / 60;
 minutes = minutes % 60;
 hours = hours + t1.hours + t2.hours;
 t1.minutes = t2.minutes; // to check if it reflects
};
void Time :: subtract (Time & t) {
 t.minutes = minutes - 5;
 t.hours = hours - 5;
};

int main () {

Time t1 (5, 20);

Time t2 (14, 40);

Time t3;

t3.sum (t1, t2); // Pass by value

t1.display (); // 5:20 } remains same

t2.display (); // 14:40 } (no change)

t3.display (); // 19:00

t1.subtract (t2); // Pass by reference

t1.display (); // 5:20 } changed t2

t2.display (); // 0:15

return 0;

→ t3.sum (t1, t2);
 t3 invokes the function sum. Hence we use
 statement like:
 minutes = t1.minutes + t2.minutes
 ↓
 t1 + t2

Returning objects

We can also return objects :

Suppose in previous program:
 there is a function:

friend Time sum (Time t1, Time t2);
 ↪ Declaration

Definition: time sum (time t1, time t2)

```
time t3;  
t3.hh = t1.hh + t2.hh;  
t3.mm = t1.hm + t2.hm;  
return (t3);  
}
```

return t3 object

Calling:

```
time t3;  
t3 = sum(t1, t2);
```

stores returned
t3 in t3

returns t3

Array of objects

We can have array of type class that stores objects.

ex -

```
class employee {  
    char name [30];  
    float age;  
public:  
    void getdata();  
    void putdata();  
};
```

This class employee (user-defined datatype) can be used to create objects.
So we can array like:

```
employee manager [3];
```

manager is an array containing 3 objects:

manager [0], manager [1], manager [2]

So these objects can individually access data members:

manager [0].putdata();

name		} manager [0]
age		
name		} manager [1]
age		
name		} manager [2]
age		

manager [3] ← storage of object array

Pointers to Objects

Suppose there is a class 'item' with

we can create:

```
item *ptr;
```

→ Pointer of type item.

So we initialise
it as:

```
item x;  
item *ptr = &x;
```

where item has declaration:

```

class item {
    int code;
    float price;
    public:
        void getdata();
        void putdata();
}

```

we can access public members using an object or pointer.

statements:

```

n.getdata();
n.putdata();

```

] for normal objects

or same as:

```

ptr -> getdata();
ptr -> putdata();

```

] for pointer objects

also $(*ptr)$ is alias of n
(dereferencing)

So, we can also write:

```

(*ptr).getdata();
(*ptr).putdata();

```

this pointer

→ to represent an object that invokes a member function.

→ it is a pointer that points to the object for

which ^{this} ~~this~~ function was called

Ex-

A.max();
will set pointer this to address of object A.

→ It acts as an implicit argument to all the member functions because it is automatically passed to a member function when it is called.

ex-

```

class ABC {
    int a;
}

```

};

'a' variable can be used inside a member function as:

a = 123;

and also as:

this -> a = 123;

→ It can be used to return object it points to:

return *this

→ will return the object that invoked the function.

Operator Overloading

- The mechanism of giving special meanings to an operator is known as operator overloading
- We cannot overload the following functions:
 - class member access operators (`., : *`)
 - scope resolution operator (`::`)
 - size operator (`sizeof`)
 - conditional operator (`? :`)
- The grammatical rules such as number of operands, precedence and associativity are not lost.

⇒ Defining operator overloading → special function = 'operator'

General form: definition outside class

```
return_type classname :: operator op (arguments)
{
    function body
}
```

op → operator being overloaded
return type + type of value returned by specified operation

ex- vector operator + (vector);
 vector operator - ();
 int operator == (vector);
 friend vector operator + (vector, vector);

Declaration
 Friendhand
 ops

Operator Overloading

Unary
 ex - -, +, ++, --
 one operand
 → using member func
 → using friend func

Binary
 ex - +, -, /, *
 two operands
 → using member func
 → using friend func

→ vector is a datatype of a class and may represent both magnitude and direction as in physics and mathematics, a series of points/elements.

→ The process of overloading involves the following steps:

- (1) Create a class that defines a datatype i.e. used in overloading operation.
- (2) Declares the operator function:

operator op (); in the public part

- It is always declared in public section
- They are non static
- It can be done using friend function or member function.

- (3) Defines the operator function to implement the required operation.

→ Overloaded operator function can be invoked

by expressions which as op x or x op for unary operators and x op y for binary operators.

I# Overloading unary operators

- (1) Using member functions → It takes no argument as it is invoked using object implicitly (which is due to this pointer)

Unary minus

int x, y, z;

public:

void getdata (int a, int b, int c);

void display ();

void operator - (); //overloading unary minus

};

void space :: operator - () //definition

x = -x;

y = -y;

z = -z;

}

int main () {

space s;

s.getdata (10, -20, 30);

cout << "s:";

s.display();

-s; // activates operator -

cout << "s:";

s.display();

return 0;

Output:

S: 10 -20 30

S: -10 20 -30

→ It does not return any value/object. Hence
 $s2 = -s1;$ X (Invalid)

② Using friend functions → It takes one argument as it is not a member function and hence we need to pass object explicitly.

Declaration: friend void operator - (space &);

Definition: void operator - (space &s) {
 $s.x = -s.x;$
 $s.y = -s.y;$
 $s.z = -s.z;$
}

Calling: same space s;
 $-s;$

→ Here argument is passed by reference because we want changes to be reflected in passed object.

II Overloading binary operators

① Using member functions → It takes one argument

Binary plus (+)

```
class complex {
    float x, y;
public:
    complex(float real, float imag) {
        x = real;
        y = imag;
    }
}
```

Declaration

complex operator + (complex);
void display();

Definition:
complex complex :: operator + (complex c) {
 complex temp; // temporary object
 temp.x = x + c.x;
 temp.y = y + c.y;
 return (temp);
}

int main() {

complex c1, c2, c3;

c1 = complex(2.5, 3.5);

c2 = complex(1.6, 2.7);

c3 = c1 + c2; // Calling
cout << "c3 = "; c3.display();
return 0;

Output:

c3 = 4.1 + j6.2

→ Here, the operator function is:
 • It receives only one complex type argument explicitly
 • Return complex type value
 • Member function

c3 = c1 + c2; is same as:
c3 = c1.operator + (c2);

∴ c1 data members are accessed directly & c2 data members are accessed using dot operation ~~for they are~~ (c2 is passed as argument)

Left hand operator

- so, left hand operand → invokes operator function & right hand operand → passed as argument
- we can also return without use of temp :

return complex((x+c.x), (y+c.y));
// using constructor

- This will create a temporary object with no name & returns the contents for copying into an object.

2) Using friend function → It takes two arguments explicitly.

Declaration:

friend complex operator + (complex, complex);

Definition:

```
complex operator + (complex a, complex b)
{
    return complex(a.x + b.x, a.y + b.y);
}
```

Calling:

C3 = C1 + C2;
which is same as:

C3 = operator + (C1, C2);

* Situation where friend function is used rather than member function:

- ex- we use two different operands for a binary operator :
one is object and other is built in data type

A = B + 2 → can work for both member & friend fⁿ
objects

But,

A = 2 + B → will not work for member fⁿ

because left hand operator is responsible for invoking member function. Hence it must be object.

∴ We use friend function because it does not need an object to invoke it.

Rules for overloading:

- 1) Only existing operators can be overloaded
- 2) Overloaded operator must have at least one operand that is of user defined type
- 3) We cannot change the basic meaning of an operator
- 4) Overloaded operators follow syntax rules of original operators
- 5) Some operators can't be overloaded [::, *, ::, sizeof, ?:]
- 6) Some operators can't be overloaded using friend functions [=], [()], [()], [→] → don't member access
function call subscript

Freehand

DATE: / /

7) Binary operators such as $+$, $-$, $*$, $/$ must explicitly return a value. They must not change their own arguments