

## UNIT - 2.

### Inline function:-

One of the objective of using `func` in prog is to save some memory space which becomes appreciable when a `func` is likely to be called many times. However every time a `func` is called it takes a lot of extra time in executing a series of instruction for task such as jumping to the `func`, saving register, pushing arguments into the stack and returning to the calling `func`.

When a `func` is small its overhead of execution time ~~can~~ cannot be minimized. One sol<sup>n</sup> to this problem is

to use macro functions. These are popular in C. The major drawback <sup>with</sup> macro is they aren't nearly functions and therefore the usual error checking doesn't occur during compilation. C++ has a different sol<sup>n</sup> to that problem. <sup>To eliminate</sup> the cost of calls to small function. C++ proposes a new feature called inline function.

The compiler replaces the `func` called with the corresponding `func` code.

inline function header {

function body;

}

If we want to make a function inline or we need to do is to prefix the keyword `inline` to function definition. We should exercise care before making the function inline. The speed benefits of inline function diminished as the function grows in size. At some point the overhead of function call becomes small compare to the execution of the function and the benefit of inline function may be lost.

Usually the functions are made inline when they are small enough to be defined in a one or two lines.

Remember that the `inline` keyword merely sends a request, not a command to the compiler.

Some of the situations where inline function may not work.

1. For function returning values, loop, switch, goto, exit.
2. If function contains static variables.
3. If inline functions are recursive.
4. for function not returning value, if a return statement exists.

```
#include <iostream>
```

```
using namespace std;
```

```
inline float mul(float x, float y) { return x * y; }
```

```
inline float div(float p, float q) { return (p/q); }
```

```
int main() {
```

```
    float a = 12.34
```

```
    float b = 5.0;
```

```
    cout << mul(a, b) << "\n";
```

```
    cout << div(a, b) << "\n";
```

```
    return 0;
```

```
}
```

## Friend function:-

We have saying that private members can't be accessed from outside the class. A known member can't have an access to the private data of a class.

For eg:- Consider a case where two classes manager and scientist have defined. We would like to use a function income tax to operate on the objects of both these classes. Is such situation C++ function allows a common function to be made friendly with both the classes thereby allowing the function to have access to the private data of these classes.

```
class ABC
```

```
{
```

```
public :
```

```
friend void eye(void);
```

```
{};
```

The `friend` declaration should be preceded by the keyword `friend`. The `func` is defined in the prog. like a normal prog. in C++. The `func` tho can be declared as a friend in any no. of classes. A `friend func`, although not a member `func`, has full access by to the private members of the class. A `friend func` characteristics:-

A `friend func` characteristics

1. It is not in the scope of the class to which it has been declared as a friend.
2. Since it is not in the scope of the class it can't called using the scope of that class.
3. It can be invoked like a normal `func` without the help of any object.
4. Unlike member `func`, it can't excess the member names directly and has to use an object name and `(.)` membership operator with each member name.

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

Class sample

```
{
```

```
    int a;
```

```
    int b;
```

```
public:
```

```
    void setvalue () {
```

```
        friend float mean (sample s);
```

```
}
```

find mean (sample s)

{ return float (

f int main {

sample ny;

ny. getvalue ( );

mean ( ny );

return 0;

}

\* Member fun<sup>n</sup> of one class can be friend of another class. In such cases they are defined using the scope resolution operator.

class X {

int fun<sup>n</sup> ();

};

class y {

friend int n :: fun<sup>n</sup> ();

};

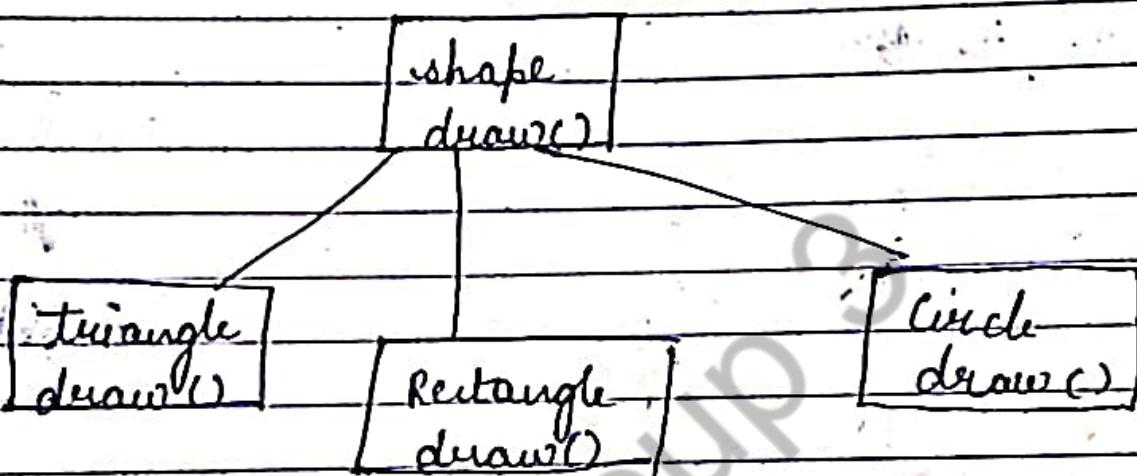
The fun<sup>n</sup> first one is a member of class x and friend of class y. We can also declare all the members of one class a friend fun<sup>n</sup> of another class. In such cases the class is called a friend class.

Using friend function to add data object  
of two different classes

## Polymerphism

The word polymorphism means having many forms.

In simple words we can define polymorphism as the ability of a msg to be displayed in more than one form. A real life eg. of polymorphism is a person at the same time can have different characteristics.



In c++ polymorphism is mainly divided into 2 types.

1. Compile time polymorphism → this type of polymorphism is achieved by function overloading.

2. Run time polymorphism

Function over-loading

when there are multiple "fun<sup>t</sup>" with same name but different parameters but these functions are said to be over loaded.

```
class Hello {
```

```
public:
```

1) fun<sup>t</sup> with 1 parameter

void function (int x)

```
{ cout << "value of x is " << x ; }
```

// first with same name but double parameter

```
void function (double x);
```

```
{ cout << "value of x is " << x ; }
```

// first with same name and a int parameter

```
void function (int x, int y)
```

```
{ cout << x << y ; }
```

```
int main () {
```

```
Hello obj' 1 ;
```

```
obj' 1 . func (10) ;
```

```
obj' 1 . func (11, 132) ;
```

```
obj' 1 . func (20, 30) ;
```

```
} return 0 ;
```

Operator overloading

C++ also provide option to overload operator  
option to overload operator

for eg:- We can make the operator for string  
class to concatenate to strings. We know  
that this the addition operator whose task is  
to add two operands. So a single operator  
(+) when b/w integer operands, adds them and  
when placed b/w string operand concade.

## Run-time polymorphism

This type of polymorphism is achieved by function overriding. Function overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overwritten.

## Constructor and destructor

We have used member function such as `putdata()`, `setvalue()` to provide initial values to the private member variables.

e.g.: `x.setvalue( - )`

If passes the initial value has arguments to the function. All these function called statements are used with the appropriate object that have already been created. These functions can't be used to initialise the member variables at the time of creation of their objects. We should be able to initialise a class type variable when it is declared. When a variable of built-in types goes out of scope, the compiler automatically destroy the variable but it hasn't happened with the objects we have studied.

C++ provides a special member func<sup>n</sup>t called constructor which enables an object to initialise itself when it is created. This is known as automatic initialisation of objects. It also provides another member func<sup>n</sup>t called destructor that destroys the object when they are no longer required.

### Constructor

The constructor is invoke whenever object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

```
eg:- class integer{  
    int m, n;  
public:  
    integer(void); // constructor  
    void run(5); // normal  
};  
integer::integer()  
{  
    m=5;  
    n=5;  
}
```

→ The constructor func<sup>n</sup>t have some special characteristics :-

- They should declare in the public section.
- They are invoked automatically when the object are used.
- They do not have return types, not even void and therefore they can't return values.
- They can't be inherited, a derived class can call the base class constructor like other C++ functions they can default arguments.
- constructor can't be virtual.
- We can't refer to their address.
- They make implicit call to the operators new and delete when memory allocation is required.

### Parameterised constructor

The constructor integer) initialize the data members of all the objects to zero. However in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permit us to achieve this objective by passing this argument to the constructor function when the object are created. The constructors that can take arguments are called parameterised constructor.

integer int 1;

The object declaration statement such that may we can pass the initial values as arguments to the constructor function. This can be done in 2 ways.

1. By calling the constructor implicit :-  $\text{integer int } q(100, 200)$   
shortened method
2. By calling the constructor explicit :-  $\text{integer int } q = \text{integer}(100, 200)$

### Parameterised constructor

The parameters of a constructor can be of any type except that of class to which it belongs.

eg:- class A  
{  
public :  
    A(A); // illegal  
};

class A  
{  
public :  
    A(A&); // illegal  
};

# include <iostream>

using namespace std;

class point {  
    int x, y;  
public :

    point () {

        x = 0;

        y = 0;

}

point C int a, int b) { // explicit default or no arg.

    n = a;

    y = b;

{

```
point (point & i) { // copy cons.
```

```
x = i.x;
```

```
y = i.y;
```

```
}
```

```
void display() {
```

```
cout << "(" << x << ", " << y << ")" << endl;
```

```
}
```

```
}
```

```
int main()
```

```
{
```

```
point p1(1,1); // invoke parameterised constructor
```

```
point p2(5,10);
```

```
point p3(p2);
```

```
point p4;
```

```
cout << "point p1 = ";
```

```
p1.display();
```

```
cout << "point p2 = ";
```

```
p2.display();
```

```
cout << "point p3 = ";
```

```
p3.display();
```

```
cout << "point p4 = ";
```

```
p4.display();
```

```
return 0;
```

```
}
```

## Constructors with default Arguments

It is possible to define constructors with default arguments. A constructor complex can be declared as follows:

The default value of argument image is 0 then the statement complex c(5.0), so it assign the value 5.0 to the real variable & 0.0 to image by default. However the statement complex c assign then it assign 0.0 to real and 3.0 to image. The actual parameter when specified overrides the default value.

## Dynamic initialization of objects

```
#include <stdio.h>
using namespace std;

class complex {
    float x, y;
public:
    constructor complex() {} ;
    constructor complex(float a) { x = y = a; }
    constructor complex(float real, float img.) {
        { x = real, y = img; }
    }
    friend complex sum(complex, complex);
    friend void show(complex);
}
```

~~global declaration  
of functions~~

C complex sum(complex c<sub>1</sub>, complex c<sub>2</sub>)

complex c<sub>3</sub>;

c<sub>3</sub>.x = c<sub>1</sub>.x + c<sub>2</sub>.x;

c<sub>3</sub>.y = c<sub>1</sub>.y + c<sub>2</sub>.y;

return(c<sub>3</sub>); }

void show(complex c)

cout << c.x << " + " << c.y << " i ";

int main()

complex A(2.7, 3.5);

complex B(1.6);

complex C;

C = sum(A, B);

cout << "A = " ; show(A);

cout << "B = " ; show(B);

cout << "C = " ; show(C);

## - Destructor

The destructor is a special member function that work just opposite to constructor. Unlike constructors that are used for initialising an object, destructor destroy the object. Similar to constructor the destructor

name should exactly match with the class name.  
A destructor declaration should always begin with the (~) symbol.

class :: class-name

{

public :

class-name () : // constructor

~ class-name () : // destructor

}

A destructor neither takes any arguments nor it does return a value. Destructor can't be overloaded.

### Destructor Uses :-

1. Releasing memory of the objects.
2. Releasing memory of the pointer variables.
3. Closing files & resources.

→ When does the destructor does call :-

- A destructor is automatically called when :-
  1. the program finish execution.
  2. A scope containing local variable ends.
  3. If we call the delete operator.

→ When do we need to write a user defined destructor.

If we don't write our own destructor in class, compiler creates a default destructor.

The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory we should write a destructor to release memory before the class instant is destroyed. This is must be done to avoid memory leak.

```
#include <iostream>
using namespace std;
class hello_world {
public:
```

```
    hello_world () {
        cout << "constructor is called";
    }
```

```
~Hello_world {
    cout << "destructor is called";
```

```
    void display () {
        cout << "hello world! " << endl;
```

```
    }
}
```

```
int main () {
    hello_world obj;
    obj.display ();
    return 0;
}
```

```
#include <iostream>
using namespace std;
class Rectangle
{
    int length;
    int breadth;
public:
    void setDimension (int l, int b)
    {
        length = l;
        breadth = b;
    }
    int getArea ()
    {
        return length * breadth;
    }
    Rectangle () // constructor
    {
        cout << "constructor";
    }
    ~Rectangle () // destructor
    {
        cout << "destructor";
    }
    int main ()
    {
        Rectangle rt;
        rt.setDimension (7, 4);
        cout << rt.getArea ();
        return 0;
    }
}
```

## New and delete operator

New is used to allocate memory for a variable, object, array, etc. at run time.

To declare memory at run time is known as dynamic memory allocation. We use memory by this method when it is not known in advance how much memory space is needed. It is also known as free space operator.

### System

{ pointer variable = new data type; }

ex:-

- like  
memory  
allocn
- (1) int \* p = new int; like dynamic memory allocation
  - (2) int \* p = new int (value);
  - (3) float \* q = new float (5);
  - (4) int x;

cin >> x

int \* p = new int [x];

- (5) complex \* ptr = new complex;

user defined data type or class

ex:- #include <iostream>

int main()

int \* age = new int; // int \* age = new int (5)

char \* gender = new char;

\* age = 0;

// not necessary

\* gender = 'm';

but good prog. habit

cout << "enter age";

cin >> \* age;

cout << "age is " << \* age << "gender is " << gender;

return 0;

}

## Delete operator

Delete operator is used to deallocate the dynamic allocated memory. Since the necessity of dynamic memory is usually limited to specific moment within a program, once it is no longer needed. It should be freed so that the memory becomes available for other regards of dynamic memory. This is the purpose of the operator delete.

(I) Delete pointer variable:

(II) delete [ ] pointer variable

The first expression should be used to delete memory allocated for a single element and the second one for memory allocated for array of elements.

this pointer automatically points to current object.

To understand 'this pointer' it is important to know how object look at just and data members of a class.

1. Each object gets its own ~~topic~~ copy of the data members.

2. All excess the same function definition as present in the code segment.

3. This raises a question is if only one copy of each member function exist that is used by multiple objects.

objects, how are the proper data members are accessed and updated. The compiler supplies an implicit pointer along with the names of the function as 'this' passed

\* This pointer is just as a hidden argument to call non-static member function calls & is available as a local variable within the body of "all non-static func".

"This pointer" is not available in "static member func" as "static member func" can be called without any object (with class name).

\* In C++ 'this' pointer is used to represent the address of an object inside a member func.

eg:- consider an object obj calling one of its member func say method() as obj.method then this pointer will hold the address of object inside the func" member method.

```
class class_name {
```

```
    int datamember;
```

```
public:
```

```
method (int a) {
```

stores

If "this" pointer stores the address of object and access data member this->data

pointer

```
}
```

```
int main{  
    classname obj;  
    obj.method();  
    return 0;  
}
```

## II. Application of 'this' pointer :-

### 1. Return object :-

One of the important application of using this pointer is to return the object it points, returning "this". Inside a member func "will return the object that calls the func".

### 2. Method changing :- After returning the object from a func "is very useful application would be to change the methods for a clearer code.

```
position obj → setX(14);
```

```
position obj → setY(15);
```

```
position obj → setZ(16);
```

### (3) Distribution data member :-

Another application of 'this' pointer is distinguishes data members from local variable of member func "if they have same name".

```
#include <iostream>
```

```
using namespace std;
```

```
class sample
```

```
{  
    int a, b;
```

```

public:
    void input (int a, int b)
    {
        this -> a = a + b;
        this -> b = a - b;
    }

    void output ()
    {
        cout << "a=" << a;
        cout << " b=" << b;
    }
};

int main()
{
    sample X;
    X . input (5, 8);
    X . output ();
    return 0;
}.

```

A class sample is created in the prog. with data members a & b and member funt' input & output. Input funt' receive two integer parameter a & b , which are of samename as datamember of class sample to distinguish the local variable to input funt' of class , 'this' pointer is used when Input funt' is called , the data of object inside it is represented as this -> a ; this -> b ; while the local variable of funt' is represent by a & b.

'This' pointer

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

class person {
    char name[20];
    float age;
public:
    person(char *s, float a)
    {
        strcpy(name, s);
        age = a;
    }
    person & greater(person & x)
    {
        if (x.age >= age)
            return x;
        else
            return *this;
    }
    void display()
    {
        cout << "age " << age;
        cout << "name " << name;
    }
};
```

```

int main()
{
    person p1 ("prakash", 58.50);
    person p2 ("deepak", 57.50);
    person p3 ("ravi", 56.50);
    person p = p2.greater(p3);
    cout << "elder is: " << p;
    p.display();
    return 0;
}

```

### Operator overloading

Operator overloading is one of the many exciting features for C++ language. C++ tries to make the user define data type behave in much the same way as the built-in types. C++ permits us to add 2 variables of user defined type with the same syntax i.e. applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a date type. The mechanism of giving such special meanings to an operator is known as operator overloading. We can overload all the C++ operators except the following:-

class member also operator(),  
scope resolution  
size

conditional operator (?)

The major reason why we can't overload the operator that these operators take names (eg: class name) as their operand instead of values.

Although the symmetry of an operator can be extended, we can't change its syntax.

Remember, when an operator is overloaded, its original meaning isn't lost. for eg: the operator (+), which has been overloaded to add two vectors, can still be used to add 2 integers.

Defining operator overloading  
To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function called "operator function" which describes the task. The general form of an operator function is:

`return_type class_name::operator<operator>(const reference_type& right)`

where return type is the type of value is

returned by the specified operation and OP is the operator being overloaded. Operator OP is the "funct" name where operator is the keyword. Operator funct must be either member funct (non-static) or friend funct.

Basic difference b/w them a friend funct will have only one argument for unary operators and two for binary operators, while a member funct has no arguments for unary operators & only 1 for binary operators. This is because the object used to invoke the member funct is passed implicitly and therefore available for the member funct. This is not the case with friend funct.

The process of overloading involves following steps.

1. Create a class that defines the data type i.e. to be used in the overloading operations.
2. declare the operator funct (operator OP ()) operator in the public part of the class.
3. Define the operator funct to implement the required operations.

```
#include <iostream>
using namespace std;
class space
{
    int x;
    int y;
    int z;
public:
    void getdata (int a, int b, int c);
    void display (void);
    void operator-();
};

void space :: getdata (int a, int b, int c)
{
    x=a;
    y=b;
    z=c;
}

void space :: display (void)
{
    cout << "x=" << x << endl;
    cout << "y=" << y << endl;
    cout << "z=" << z << endl;
}

void space :: operator-()
{
    this->x=-x;
}
```

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

```
this->z = -z;
```

```
}
```

```
int main()
```

```
{
```

```
shape s;
```

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

```
cout << "S:";
```

```
s.display();
```

```
-s;
```

```
cout << "-S:";
```

```
s.display();
```

```
return 0;
```

```
}
```

## Overloading Binary Operator



```
#include <iostream>
```

```
using namespace std;
```

```
class complex
```

```
{
```

```
float x;
```

```
float y;
```

```
public:
```

```
complex () {}
```

```
complex (float real, float imag.)
```

```
{ x = real, y = imag; }
```

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

```
};  
complex complex::operator+(complex c)  
{
```

```
    complex temp;
```

```
    temp.x = x + c.x;
```

```
    temp.y = y + c.y;
```

```
    return(temp);
```

```
};  
void complex::display(void)
```

```
{ cout << x << " + " << y << "\n"; }
```

```
int main()
```

```
{  
    complex c1, c2, c3;
```

```
    c1 = complex(2.5, 3.5);
```

```
    c2 = complex(1.6, 2.7);
```

```
    c3 = c1 + c2;
```

```
    cout << "c1 = " ; c1.display();
```

```
    cout << "c2 = " ; c2.display();
```

```
    cout << "c3 = " ; c3.display();
```

```
    return 0;
```

```
}
```

→ We should <sup>note</sup> trace the following features of Binary operator :-

1. It receives only one complex type argument
2. It returns a complex type value.
3. It is a member func<sup>n</sup> of complex

→  $c_3 = c_1 + c_2$ , object  $c_1$  takes the responsibility of invoking the func<sup>n</sup> &  $c_2$  place the role of an argument i.e. pass to a func<sup>n</sup>.

temp

410

620

return

$c_2$

complex operator +(complex)  
{ complex temp;

temp.x = | c.x | , x  
temp.y = | c.y | , y

return {temp};

4.10x  
6.20y

2.50x  
3.50y

1.60x  
2.70y

→ #include <iostream>

using namespace std;

class distance {

public :

int feet, inch ;

// no parameter constructor

distance ()

{

this → feet = 0 ;

this → inch = 0 ;

}

1) constructor to initialize objects

distance (int f, int i)

this → feet = f;

this → inch = i;

2) declaring friend function

friend distance operator + (distance & d1, distance & d2);

Implementation friend function with two parameters.

distance operator + (distance & d1, distance & d2)

distance d2;

d3.feet = d1.feet + d2.feet;

d3.inch = d1.inch + d2.inch;

return d3;

int main()

distance d1(8,9);

distance d2(10,2);

distance d3;

d3 = d1 + d2;

cout << "In feet & inches " << d3.feet << d3.inch;

return 0;