UNIT - 2

Inline function:

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

When a funt^n is small is overhead of execution time ~~cannot~~ can be minimized. One sol^n to this problem is to use macro functions. These are popular in C. The major drawback with macro is they aren't nearly functions and therefore the usual error checking doesn't occur during compilation. C++ has a different sol^n to that problem. To eliminate the cost of calls to small function. C++ proposes a new feature called inline function.

The compiler replaces the funt^n called with the corresponding funt^n code.

```
inline function header {
        function body;
}
```

If is to make a funt^n inline or we need to do is to prefix the keword inline to funt^n definite. We should exercise care before making the function inline. The speed benefits of inline fuction dimnished as the function grows in size. At some point the overhead of function call becomes small compare to the execution of the funt^n and the benefits of inline funt^n may be lost.

Usually the funt^n are made inline when they are small enough to be defined in a one of two lines. Remember that the inline keyword merely sends a request, not a command to the compilere.

Some of the situation where inline funt^n may n't work.

1. For funt^n returning values, loop, switch, goto, exit.
2. If funt^n contains static vabiables.
3. If inline funt^n are recuresine.
4. for funt^n not returning value, if a return statement exsist.

```
# include <iostream>
using namespace std;
inline float mul(float x, float y) { return x*y); }
Inline float div( floatl, float q) { retur(l/q); }
```

```
int main() {
    float a = 12.34
    float b = 5.02;
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 manger and scientist have defined. We would like to use a fun" income tax to operate on the objects of both these classes. In such situation C++ fun" allows a common fun" to be made friendly with both the classes, thereby allowing the fun" to have access to the private data of these classes.

```
class ABC
    {
    public:
    friend void xyz (void);
    };
```

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

A friend funt^n 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 funt^n without the help of any object.
4. Unlike member funt^n, it can't access 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(...
}
int main{
sample xy;
xy.setvalue();
   mean(xy);
return 0;
}
```

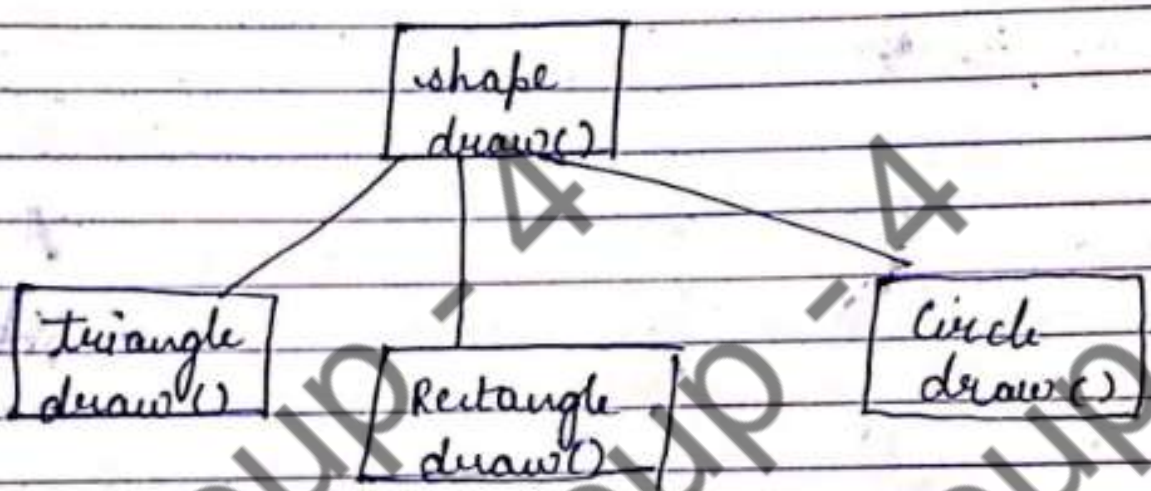
* Member funt" of one class can be friend of another class. In such cases they are defined using the scope resolution operator.

```
class X {
    int funt"();
};
class y {
    friend int x::funt();
};
```

The funt" 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 funt" of another class in such cases the class is called a friend class.

Polymorphism

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 or operator overloading.)
2. Run time polymorphism

Function over-loading

When there are multiple funt^n with same name but different parameters but these functions are said to be over loaded.

```
class Hello {
    public:
        // funt$^n$ with 1 parameter
```

```
void function (int x)
{
  cout << " value of x is " << x;
}
// fun" with same name but double parameter.
  void function (double x)
{ cout << " value of x is " << x;
}
// fun" with same name and 2 int parameter
  void function (int x, int y)
{
  cout << x << y;
}
int main (){
        Hello obj 1;
        obj 1. func (10);
        obj 1. func (11.12);
        obj 1. func (20.30);
        return 0;
}
```

Operator overloading

C++ also provide option to overload ~~operator~~ option To overload operator.

foreg: We can make the operator for string class to concadenate 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 fun^{n} overriding. fun^{n} overriding occurs when a derived class has a definition for one of the member fun^{n} of the base class. That base fun^{n} is said to be over written.

Constructor and destructor

We have used member fun^{n} such as putdata(), setvalue() to provide initial values to the private member variables.

eg: x.setvalue(...)

It passes the initial value has arguments to the fun^{n}. All these fun^{n} 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 initiallise a class type variable when it is declared when a variable of built in type goes out of scope, the compiler automatically destroy the variable but it hasn't happened with the objects we have study.

C++ provides a special member funt" called constructor which enables an object to initialise itself when it is created. This is known automatically initiallisation of objects. It also provide another member funt" called distructor that destroys the object when they are no longer required.

· Constructor

The constructor is invoke whenever object of it associated class is created. It is called constructor beior 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 funt" have some special characteristics :-

→ They should declare in the public section.
→ They are invoked automatically when the object are created.
→ 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++ funtions they can defaul arguments.
→ constructor can't be virutual.
→ 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++ permits us to achieve this objective by passing ~~this~~ argument to the constructor funt" when the object are created. The constructors that can take arguments are called parametrised constructor.

integer int 1;

The object declaration statement such that may not work, we may pass the initial values as arguments to the constructor funt. This can be done in 2 ways.

1. By calling the constructor implicit :- integer int q (100,200); shortcut method

2. By calling the constructor explicit :- integer int q = 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
};
```

copy constructor

```
# include <iostream>
using namespace std;
class point {
    int x, y;
public :
    point () {
        x = 0;
        y = 0;
    }
    point (int a, int b ) {   // explicit default or no arg.
        x = a;
        y = b;
    }
```

```
    point (point &i){ // copy cons.
        x = i.x;
        y = i.y;
    }
    void display(){
        cout << "(" << x << "," << y << ")\n";
    }
};
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. 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 2.0 to real and 3.0 to image. The actual parameter when specified overwrites the default value.

~~Dynamic initialization of objects~~

```
#include <iostream>
using namespace std;
class complex{
    float x, y;
public:
    complex(){};
    complex(float a){x=y=a;}
    complex(float real, float imag.)
        {x=real; y=imag;}
friend complex sum(complex, complex);
friend void show(complex);
};
```

Global declaration of functions

```
complex sum(complex c1, complex c2)
{
    complex c3;
    c3.x = c1.x + c2.x;
    c3.y = c1.y + c2.y;
    return (c3); }
void show (complex c)
{
    cout << c.x << "+j" << c.y << "\n";
}
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);
}
```

main

— Destructor

The destructor is a special member funct^n that work just opposite to ~~dest~~ 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.

⇒ <u>When does the destructor does call</u> :--

- A destructor is automatically called when :-

1. the program finish execution
2. A scope containing local variable ends.
3. You call the delete operator.

→ When do we need to write a user define destructor

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

The default distructor 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 relise memory before the class instant is destroy. This is must be done to avoid memory leak.

```
#include<iostream>
using namespace std;
class hellow world {
        Public:
    hello world () {
        cout << "constructor is called";
            }
~ Hello world {
        cout << "destructor is called";
    }
Void display () {
        cout << "hello world!" << endl;
    }
};
int main() {
        hello would 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 declaring 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.

Syntax

{ pointer variable = new data type;

ex:- like memory allocⁿ

① int * p = new int; like dynamic memory allocⁿ

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 * aage = new int (5);
char * gender = new char;
   * age = 0;           // not neccessary but good prog. habit
   * gender = 'm';
cout << "enter age";
cin >> * age;
cout << "age is" << * age << "gender is:" << *gender;
return 0;
}
```

Delete operator

Delete operater is used to deallocates the dynamic allocated memory. Since the neccessity of dyanic memory is usually limited to Speific moment within a program, once its is no longer needed. It should be freed so that the memory becomes available for other requests of dyanamic 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 element.

this pointer

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

1. Each object gets its own topic copy of the data members.
2. All excess the same funt^n defination as present in the code segment.
3. Then now question is if only one copy of each member funt^n exist abd is used by multiple

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

- This pointer is fast as a hidden argument to call non-static member funt" calls & is available as a local variable within the body of all non-static funt".
- "This pointer" is not available in static member funt" as static member funt" 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 funt".

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

```
class class name {
        int datamember;
        public:
        method ( int a) {
            If "this" pointer stores the address
        of object and access data member this -> data pointer
                                                = a;
}
```

```
int main{
    classname obj;
    obj.method(5-);
    return 0;
    }
```

#1. Application of 'this' pointer :-

1. Return object :-
One of the important application of using this pointer is to return the object it points, return *this; Inside a member fun" will return the object that calls the fun".

2. Method changing :- After returning the object from a fun" a 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 fun" 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^n input & output. Input funt^n recieve two integer parameter a & b, which are of same name as data member of class sample to distinguish the local variable to input funt^n of class, 'this' pointer is used when input funt^n is called, the data of object inside it is represented as this → a; this → b; while the local variable of funt^n 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(p1);
  cout << " elder is:\n";
  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 data type. The mechanism of giving such special meanings to an operator is known as operator overloading. We can overload all the c++ operator except the following:-

class member also operator()
scope resolution
size
conditional operator (?)

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

Although the syntax 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 fun^{n}, called operator fun^{n} which describes the task. The general form of an operator fun^{n} is

returntype classname

where return type is the type of value it

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

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

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 funt" (operator OP ()) operate in the public part of the class.
3. Define the operator funt" 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 <<" ";
  cout <<"y=" <<y <<" ";
  cout <<"z=" <<z <<"\n";
}
void space :: operator-()
{
  this -> x = -x;
```

```
    this -> y = -y;
    this -> z = -z;
}
int main()
{
  space s;
  s.getdata(10,-20,30);
  cout<<"S:";
  s.display();

  -s;
  cout<<"-S:";
  s.display();
  return 0;
}
```

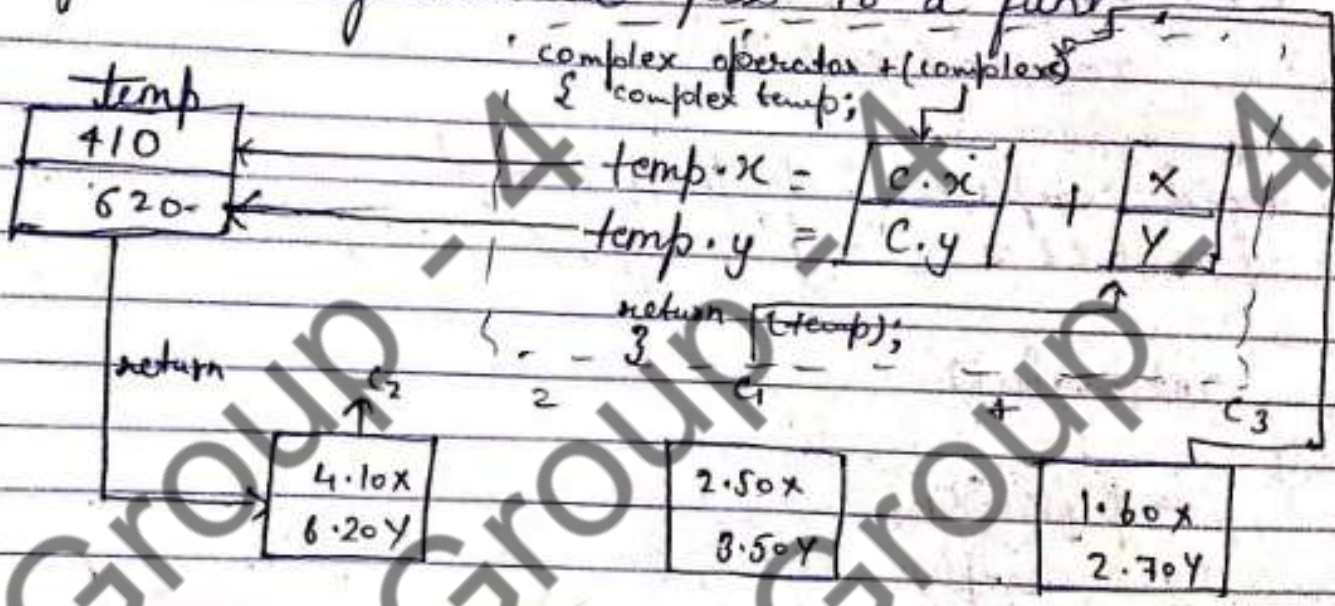
Overloading Binary Operator

```
# include <iostream>
using namespace std;
class complex
{
  float x;
  float y;
  public:
  complex(){}
  complex(float real, float imag)
    { x = real, y = imag;}
```

→ We should note the following features of Binary operator :-

1. It recieves only one complex type argument
2. It returns a complex type value.
3. It is a member fun^n of complex

⟹ $c_3 = c_1 + c_2$, object c_1 takes the responsibility of invoking the fun^n & c_2 place the role of an argument i.e pass to a fun^n.



→
```
#include <iostream>
using namespace std;
class distance {
    Public:
    int feet, inch;
    // no parameter constructor
    distance ()
    {
        this → feet = 0;
        this → inch = 0;
    }
```

// constructor to initialize objects
distance (int f, int i)
{
this → feet = f;
this → inch = i;
}
// declaring friend function
friend distance operator + (distance &, distance &);
};
Implementary friend function with two parameters.
distance operator + (distance & d1, distance & d2)
{
distance d3;
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;
}