Unit 2: Functions in OOP

By : Prof. Abhijit Pande

Syllabus: Functions in OOP, function overloading, friendly functions, Passing & returning Objects, pointers to members, constructors and destructors, copy constructor, operator overloading. Access specifiers and packages..

COURSE CODE: CT2202

COURSE NAME: OBJECT ORIENTED PROGRAMMING

COURSE OBJECTIVE

1. To introduce object oriented programming features and its diagrammatic representation of its model components.
2. To understand concept of class, handling its features and the reusability concept in object oriented language.
3. To understand the mechanism to make use of files and standard libraries.
4. To introduce the exception handling mechanism and the MVC architecture along with web components to design the software solution.
5. To introduce how to perform the event driven programming.

COURSE OUTCOME

1. Able to analyze the problem and can proposed the solution in OO approach.
2. Able to implement the solution using suitable reusability technique provided in OOP language.
3. Able to implement the solution using files and standard template library.
4. Able to design the error free software solution using the standard architecture patterns.
5. Able to design and implement the event driven solution for the problem.

Functions in OOP

- A function is a set of statements that take inputs, do some specific computation and produces output.
- The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can call the function.
- A function **declaration** tells the compiler about a function's name, return type, and parameters.
- A function **definition** provides the actual body of the function.
- The general form of a C++ function definition is as follows

```
return_type function_name( parameter list ) {
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

Function Arguments

- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

- The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.
- While calling a function, there are two ways that arguments can be passed to a function –

1. **Call By Value**

   - This method copies the actual value of an argument into the formal parameter of the function.
   - In this case, changes made to the parameter inside the function have no effect on the argument.

2. **Call by Reference**

   - This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call.
   - This means that changes made to the parameter affect the argument.

Function Overloading

- Overloading refers to the use of the same thing for different purposes.
- In function overloading we can use the same function name to create functions that perform a variety of different task.
- The function would perform different operations depending on the argument list in the function call.
- The correct function to be invoked is determined by checking the number and type of arguments but not on function type.
- Examples:

```
int Add( int a, int b)
float Add(float a,float b)
double Add(double a, int b)
```

- Overloaded functions are extensively used for handling class objects.

The function selection involves the following steps:

1. The compiler first tries to ***find an exact match*** in which the types of the actual arguments are same and use that function.
2. If an exact match is not found, then compiler uses the ***integral promotions*** to the actual arguments such as *char to int, float to double* to find a match.
3. When either of them fails, the compiler tries to ***use built in conversion*** to the actual arguments and then uses the function whose match is unique.
4. If all of the steps fails, then the compiler will try ***user defined conversions in combination with integral promotions and built in conversion*** to find unique match.

```
#include<iostream.h>
#include<conio.h>
class fn
{
        public:
        void area(int); //circle
        void area(int,int); //rectangle
        void area(float ,int,int); //triangle
};
void fn::area(int a)
{
        cout<<”Area of Circle:”<<3.14*a*a;
}
void fn::area(int a, int b)
{
        cout<<”Area of Circle:”<<a*b;
}
void fn::area(float t, int a, int b)
{
        cout<<”Area of triangle:”<<t*a*b;
}
```

```
void main( )
{
        fn obj;
        obj.area(5);
        obj.area(4,6);
        obj.area(4.5,8,9);
        getch();
}
```

Friend Functions

- A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class.
- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
- A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.
- To declare a function as a friend of a class, precede the function prototype in the class definition with keyword **friend** as follows:

  ```
  class Box
  {
          double width;
          public:
          double length;
          friend void printWidth( Box box );
          void setWidth( double wid );
  };
  ```

- To declare all member functions of class ClassTwo as friends of class ClassOne, place a following declaration in the definition of class ClassOne:

  ```
          friend class ClassTwo;
  ```

Characteristics of friend Function

1. It is not in the scope of the class to which it has been declared as friend.
2. It cannot be called using the object of that class.
3. It can be invoked like a normal function without the help of any object.
4. Unlike member functions, it cannot access member names directly and has to use an object name and dot membership operator with each member name.
5. It can be declared wither in the public or the private part of a class without affecting its meaning.
6. Usually, it has objects as arguments.

Example

```
#include <iostream.h>
class Box
{
        double width;

        public:
        friend void printWidth( Box box );
        void setWidth( double wid );
};
// Member function definition
void Box::setWidth( double wid )
{
        width = wid;
}
// Note: printWidth() is not a member function of any class.
void printWidth( Box box )
{
        /* Because printWidth() is a friend of Box, it can directly access any member of this
        class */
        cout << "Width of box : " << box.width <<endl;
}
// Main function for the program
int main( )
{
        Box box;
        box.setWidth(10.0);
        printWidth( box );
        return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Width of box : 10

Passing and Returning Objects in C++

- In C++ we can pass class's objects as arguments and also return them from a function the same way we pass and return other variables.

Passing an Object as argument

- To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.
- Syntax:

function_name(object_name);

```
#include <iostream.h>
class A
{
          public:
            int n=100;
            char ch='A';
          void disp(A a)
        {
            cout<<a.n<<endl;
            cout<<a.ch<<endl;
        }
};
int main()
{
  A obj;
  obj.disp(obj);
  return 0;
}
```

- Here in class A we have a function disp() in which we are passing the object of class A. Similarly we can pass the object of another class to a function of different class.

Return object from a function

- Syntax:

object = return object_name;

```
#include <iostream>
using namespace std;
class Student
{
        public:
          int stuId;
          int stuAge;
          string stuName;
Student input(int n, int a, string s)
{
    Student obj;
    obj.stuId = n;
    obj.stuAge = a;
    obj.stuName = s;
    return obj;
  }

void disp(Student obj){
    cout<<"Name: "<<obj.stuName<<endl;
    cout<<"Id: "<<obj.stuId<<endl;
    cout<<"Age: "<<obj.stuAge<<endl;
  }
};
int main()
{
  Student s;
  s = s.input(1001, 29, "Negan");
  s.disp(s);
  return 0;
}
```

- In this example we have two functions, the function input() returns the Student object and disp() takes Student object as an argument

Constructor

- A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.
- It is special because its name is same as that of class name.
- It is called constructor because it constructs the values of data members of the class. Constructors can be very useful for setting initial values for certain member variables.

Characteristics of Constructor

1. They should be declared in the public section.
2. They are invoked automatically when the objects are created.
3. They do not have return types even void.
4. They cannot be inherited, though a derived class can call the base class constructor.
5. Like other c++ functions, they can have defaults arguments.
6. Constructors cannot be virtual.
7. We cannot refer to their addresses.
8. An object with a constructor cannot be used as a member of a union.
9. They make implicit calls to the operators new and delete when memory allocation is required.

```
#include <iostream.h>
class Line
{
        public:
                void setLength( double len );
                double getLength( void );
                Line(); // This is the constructor
        private:
                double length;
};

Line::Line(void)
{
        cout << "Object is being created" << endl;
}

void Line::setLength( double len )
{
        length = len;
}
double Line::getLength( void )
{
        return length;
}
int main( )
{
Line line;
// set line length
line.setLength(6.0);
cout << "Length of line : " << line.getLength() <<endl;
return 0;
}
```

Output:

```
Object is being created
        Length of line : 6
```

1. Default Constructor

- A Default constructor is that will either have no parameters, or all the parameters have default values.
- If no constructors are available for a class, the compiler implicitly creates a default parameterless constructor without a constructor initializer and a null body.

Example:

```
#include <iostream.h>
class Defal
{
        public:
        int x;
        int y;
        Defal(){x=y=0;}
};
int main()
{
        Defal A;
        cout << "Default constructs x,y value::"<<
        A.x <<" , "<< A.y << "\n";
        return 0;
}
```

2. Parameterized Constructor

- A default constructor does not have any parameter, but if you need, a constructor can have parameters.
- This helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
#include <iostream.h>
class Line
{
public:
        Line(double len); // This is the constructor
        private:
        double length;
};
// Member functions definitions including constructor
Line::Line( double len)
{
        cout << "Object is being created, length = " << len << endl;
        length = len;
}
// Main function for the program
int main( )
{
        Line line(10.0);
        return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created, length = 10
```

3. Copy Constructor

- The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
- The copy constructor is used to:
  1. Initialize one object from another of the same type.
  2. Copy an object to pass it as an argument to a function.
  3. Copy an object to return it from a function.
- If a copy constructor is not defined in a class, the compiler itself defines one.
- If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.
- The most common form of copy constructor is shown here:

```
classname (const classname &obj)
{
// body of constructor
}
```

Destructor

- A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
- A destructor will have exact same name as the class prefixed with a tilde (~).
- It can neither return a value nor can it take any parameters.
- Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Characteristics:

1. A destructor is invoked automatically by the compiler upon exit from the program.
2. A destructor does not return any value.
3. A destructor cannot be declared as static.
4. A destructor must be declared in the public section of the class.
5. A destructor does not accept arguments and does it cannot be overloaded.

Following example explains the concept of destructor:

```
#include <iostream.h>
class Line
{
        public:
        Line(); // This is the constructor declaration
        ~Line(); // This is the destructor: declaration
        private:
        double length;

};
Line::Line(void)
{
        cout << "Object is being created" << endl;
}
Line::~Line(void)
{
        cout << "Object is being deleted" << endl;
}
int main( )
{
        Line line;
        return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Object is being deleted
```

Operator Overloading

- Operator overloading is one of the exciting features of C++ language.
- C++ tries to make the user defined data types behave in much the same way as the built in types.
- C++ has the ability to provide the operators with a special meanings to an operator is known as *operator overloading*.
- Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators.

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 describe a task.
- The general form of an operator function is:

```
returntype classname :: operator op(arglist)
{
function body //task defined
}
```

- where *returntype* is the type of value returned by the specific operations and *op* is the operator being overloaded.
- The *op* is preceded by the keyword **operator**. **operator** *op* is the function name.
- Operator function must be either member function or friend function.
- Arguments may be passed either by value or by reference.

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op( ) in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required operations.

- Overloaded operator functions can be invoked by expression such as
  1. *op x or x op* for unary operators
  2. *x op y* for binary operators
  3. *operator op (x)* or *operator op (x,y)* for friend function

Rules for Operator overloading

1. Only existing operators can be overloaded. New operator cannot be created.
2. The overloaded operator must have atleast one operand that is of user defined type.
3. We cannot change the basic meaning of operator. That is we cannot redefine + operator to subtract one value from other.
4. Overloaded operators follow syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded
   i. Class member access operator (., .*)
   ii. Scope resolution operator (::)
   iii. Size operator (sizeof)
   iv. Conditional operator (?:)

6. We cannot use friend functions to overload certain operators.
    i. Assignment operator =
    ii. Function call operator ( )
    iii. Subscripting operator [ ]
    iv. Class member access operator ->
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit value. Those overloaded by means of friend function, take one reference argument.
8. Binary operator overloaded by means of a member function, take one explicit argument and those overloaded by means of friend function, takes two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +, -, *, / must explicitly return a value.

<u>Overloading Unary Operators</u>

```
#include<iostream.h>
#include<conio.h>
class Negation
{
        private:
        int a,b;
        public:
        void getdata( );
        void putdata( );
        void operator –( );
};
```

```
void Negation::getdata( )
{
        cout<<”Enter the value of A and B”;
        cin>>a>>b;
}
void Negation::putdata( )
{
        cout<<”A=”<<a<<endl;
        cout<<”B=”<<b<<endl;
}

void Negation::operator –( )
{
        a=-a;
        b=-b;
}
void main( )
{
        Negation N;
        N.getdata( );
        N.putdata( );
        cout<<”After Overloading”<<endl;
        -N;
        N.putdata( );
        getch( );
}
```

Overloading Binary Operators

```
#include<iostream.h>
#include<conio.h>
class complex
{
        int a,b;
        public:
        void getvalue()
        {
                cout<<"Enter the value of Complex Numbers a,b:";
                cin>>a>>b;
        }
        complex operator+(complex ob)
        {
                complex t;
                t.a=a+ob.a;
                t.b=b+ob.b;
                return(t);
        }

        complex operator-(complex ob)
        {
                complex t;
                t.a=a-ob.a;
                t.b=b-ob.b;
                return(t);
        }
        void display()
        {
                cout<< a<<"+"<<b<<"i"<<"\n";
        }
};
```

```
void main()
{
        clrscr( );
        complex obj1,obj2,result,result1;
        obj1.getvalue();
        obj2.getvalue();
        result = obj1+obj2;
        result1=obj1-obj2;
        cout<<"Input Values:\n";
        obj1.display();
        obj2.display();
        cout<<"Result:";
        result.display();
        result1.display();
        getch();
}
```

Overloading Binary Operators Using Friends

- Friend function may be used in the place of member functions for overloading a binary operator.
- The only difference is that a friend function requires two arguments to be explicitly passed to it, while a member function requires one.

```
#include <iostream.h>
class myclass
{
int a;
int b;
public:
```

```
myclass(){}
myclass(int x,int y)
{
        a=x;b=y;
}
void show()
{
        cout<<a<<endl<<b<<endl;
}
friend myclass operator+(myclass,myclass);
friend myclass operator-(myclass,myclass);
};
myclass operator + (myclass ob1,myclass ob2)
{
         myclass temp;
        temp.a = ob1.a + ob2.a;
        temp.b = ob1.b + ob2.b;
        return temp;
}
myclass operator - (myclass ob1,myclass ob2)
{
        myclass temp;
        temp.a = ob1.a - ob2.a;
        temp.b = ob1.b - ob2.b;
        return temp;
}
void main()
{
        myclass a(10,20);
        myclass b(100,200);
        a=a+b;
        a.show();
}
```

Pointers to Class Members

- Just like pointers to normal variables and functions, we can have pointers to class member functions and member variables.

```
class Simple
{
   public:
   int a;
};

int main()
{
   Simple obj;
   Simple* ptr;   // Pointer of class type
   ptr = &obj;

   cout << obj.a;
   cout << ptr->a;  // Accessing member with pointer
}
```

- Here you can see that we have declared a pointer of class type which points to class's object.
- We can access data members and member functions using pointer name with **arrow ->** symbol.

Using Pointers with Objects

- For accessing normal data members we use the dot . operator with object and -> qith pointer to object.
- But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

  *Object.*pointerToMember*

and with pointer to object, it can be accessed by writing,

*ObjectPointer->*pointerToMember*

```
class Data
{
   public:
   int a;
   void print()
   {
      cout << "a is "<< a;
   }
};

int main()
{
   Data d, *dp;
   dp = &d;     // pointer to object
   int Data::*ptr=&Data::a;   // pointer to data member 'a'
   d.*ptr=10;
   d.print();
   dp->*ptr=20;
   dp->print();
}
```

```
a is 10 a is 20
```

Packages

- A package is a named collection of declarations that may span several files.
- A package defines the scope of the declarations it contains and may be separated into an interface package and an implementation package.

Packages are used for:

1. Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
2. Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
3. Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses.
4. A default member (without any access specifier) is accessible by classes in the same package only.
5. Packages can be considered as data encapsulation (or data-hiding).

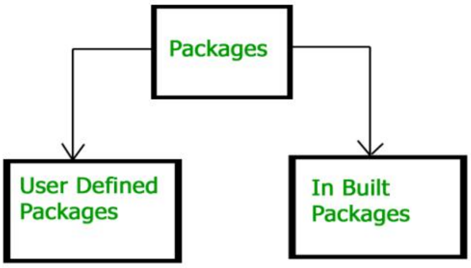
How packages work?

- Package names and directory structure are closely related.
- For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present *college*.

Package naming conventions

- Packages are named in reverse order of domain names, i.e., org.com.practice.
- For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

Types of packages:



- C++ provides several types of scopes: global, file, class, and block.
- We can also create a package scope using a **namespace declaration**:

namespace NAME { DECLARATION ... }

Subpackages

- Packages that are inside another package are the **subpackages**.
- These are not imported by default, they have to imported explicitly.
- Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.