# Object Oriented Concepts

**The C++ language was designed to facilitate and support the object oriented style of programming. It implements various OOP concepts by applying specific constructs and rules.**

## Class Definitions

**A class in C++ is declared as:** `class class_name { class_body };`

**The class body is private by default, but may consist of any number of sections with specific access rights declared by:**

- **private:** accessed only by members of the class
- **protected:** accessed also by members of derived classes
- **public:** accessed by all

**Data members and methods are declared in the appropriate sections.**

**Methods are usually defined outside the class body, using the notation:**

```
ret_type class_name::method_name(arg_list) {method_body}
```

**Methods of short code may also be defined inline. Inlined methods are copied into the code wherever a call is made, which is more efficient.**

**Inlined Method Definitions**

```
class Point {
    int x, y;
  public:
    Point( int xi, int yi ) { x=xi; y=yi; }  // definitions
    int getX() { return x; }
    int getY() { return y; }
    void move( int dx, int dy );             // declaration
  };
```

# Constructors

Constructors are member functions that return objects of a class. They have the same name as the class. Return type must not be specified.

Constructors are not mandatory. For example, an object of class Point may be defined as follows:

```
Point pt;       // declaration
pt.setX(100);   // initialization
pt.setY(200);
```

Declarations of objects as above are usually allowed only when there is no constructor, or when a null constructor that takes no arguments is defined, e.g. `Point() { x = y = 100; }`

Declaration and definition of objects are usually combined as in

```
Point pt = Point( 10, 20 );
```

given that the class definition contains the constructor

```
Point( int xval, int yval ) { x = xval; y = yval; }
```

Objects can also be defined in an abbreviated form to use the constructor above as in `Point pt( 10, 20 );`

Constructors can be overloaded, when the compiler decides which to use based on number and type of arguments. For example, if both of the constructors above had been defined, then the compiler interprets the following code as given:

```
Point a;            // using Point::Point()
Point b( 2, 4 );    // using Point::Point( int, int )
```

A form of constructor defines an object by copying the data elements of an existing object of same class (called a copy constructor), as in

```
Point( Point p ) { x = p.x; y = p.y; }
```

which can be used as in

```
Point a( 50, 60 );  // using Point::Point( int, int )
Point b( a );       // using Point::Point( Point )
```

# Destructors

Destructors are member functions that provide automatic mechanisms for executing code upon deleting objects when they become invalid, e.g. when leaving scope. The destructor is implicitly called once for each object at its destruction time.

A destructor is of the same name as its class, but prefixed by a tilde character. It has no arguments and no return value.

For example, a destructor of the Point class may be declared as

```
~Point();
```

It can be defined as

```
Point::~Point() {
// code to run upon deleting object
}
```

Destruction of objects takes place when each object leaves its scope of definition, as in

```
int func() {
    Aclass obj;
      . . . . .
}                    // destructor is called implicitly
```

Destruction also takes place for a dynamically allocated object upon release when it is no longer needed, as in

```
Aclass* objp = new Aclass();
. . . . .
delete objp;         // destructor is called implicitly
```

Example: A Shape class has the following methods:

```
void display();       // to display on screen
void remove();        // to remove from screen
~Shape() { remove(); }
```

Note that if there was no destructor code, the displayed objects remain on screen even after they are deleted.

# Operator Overloading

**Operator overloading is a mechanism to allow using standard operators with new operand types. For example, a type for complex numbers (defined as a "Complex" class) needs methods for basic arithmetic.**

```
class Complex {
      double r, i;

   public:
      Complex() { r = 0.0; i = 0.0; }
      Complex( double x, double y ) { r = x; i = y; }
      Complex add( Complex op );
};

Complex Complex::add( Complex op ) {
     double x = r + op.r;
     double y = i + op.i;
     return Complex( x, y );
}
```

**Thus, a complex number calculation ( c = a + b ) can be coded as:**

```
Complex a(1.0, 2.0), b(3.5, 1.2), c;
c = a.add( b );
```

**Although the above is absolutely correct, it would be more convenient to use the usual "+" operator to express the addition of two complex numbers. Fortunately, C++ allows overloading almost all of its operators for newly created types with classes.**

**A "+" operator method can be declared instead of "add" as:**

```
Complex operator+ ( Complex op );
```

**It can be defined in the usual way outside the class body.**

**Thus, a message written as** `c = a.operator+( b );`
**may then be expressed simply as** `c = a + b;`

**Example: Complex class with overloaded operators for basic arithmetic.**

```
class Complex {
      double r, i;
   public:
      Complex() { r = 0.0; i = 0.0; }
      Complex( double x, double y ) { r = x; i = y; }
      Complex operator+ ( Complex op );
      Complex operator- ( Complex op );
      Complex operator* ( Complex op );
      Complex operator/ ( Complex op );
      void print() { cout<<r<<" + "<<i<<" i\n"; }
};
Complex Complex::operator+ ( Complex op ) {
     double x = r + op.r;
     double y = i + op.i;
     return Complex( x, y );
}
Complex Complex::operator- ( Complex op ) {
     double x = r - op.r;
     double y = i - op.i;
     return Complex( x, y );
}
Complex Complex::operator* ( Complex op ) {
     double x = r*op.r - i*op.i;
     double y = r*op.i + i*op.r;
     return Complex( x, y );
}
Complex Complex::operator/ ( Complex op ) {
     double d = op.r*op.r + op.i*op.i;
     double x = (r*op.r + i*op.i)/d;
     double y = (i*op.r - r*op.i)/d;
     return Complex( x, y );
}

void main() {
     Complex a(1.0,2.0), b(3.5,-1.2), c(0.2,1.0), res;
     res = (a + b)/c - a*(a + b);
     res.print();
}
```

# Inheritance

**OOP simplifies designing new classes by using inheritance from existing classes that share common and/or similar elements. A new class may be declared to "inherit from" other classes. The "*subclass*" would thus automatically inherit all elements of its "*superclasses*".**

**For an example, an existing class Point is already defined as follows:**

```
class Point {
    int x, y;
  public:
    Point() { x = y = 0; }
    Point( int xi, int yi ) { x = xi; y = yi; }
    void add( int xi, int yi ) { x += xi; y += yi; }
    int getX() { return x; }
    int getY() { return y; }
    void print();
};
void Point::print() {
     cout << '(' << x << ',' << y << ")\n"; }
```

**Using inheritance, class Point3D may be defined as follows:**

```
class Point3D : public Point {
    int z;
  public:
    Point3D() { z=0; }
    Point3D( int xi, int yi, int zi ) { add(xi,yi); z=zi; }
    void add3( int xi,int yi,int zi ) { add(xi,yi); z+=zi;}
    int getZ() { return z; }
    void print();
};
void Point3D::print() {
     cout << '(' << getX() << ',' << getY()
          << ',' << z << ")\n"; }
```

**Notes:**

- **Superclass null constructor is implicitly called before any of subclass.**
- **If "add3" is named "add" in Point3D, it overrides the inherited one unless referred to as Point::add.**

## Inheritance Access Rights

**Two types of inheritance access rights exist: *public* and *private*. By default, classes are privately derived from each other.**

- **Using private inheritance, every inherited element of the superclass becomes private in the subclass.**
- **Using public inheritance, every inherited element of the superclass remains as already declared.**

| | Type of Inheritance | |
|---|---|---|
| | **private** | **public** |
| private | private | private |
| protected | private | protected |
| public | private | public |

## Construction

**Prior to the execution of a constructor for an object of a subclass, a constructor for each of its superclasses must initialize its part of the created object.**

- **If an object of a subclass is created without explicit calls to its superclass constructors, then implicit calls are made to superclass null constructors that must exist in this case.**
- **Explicit calls to superclass constructors can be specified after a single colon just before the body of the subclass constructor.**

```
// Implicit call to Point()
    Point3D( int xi,int yi,int zi ) { add(xi,yi); z=zi; }

// Explicit call to Point(int,int)
    Point3D( int xi,int yi,int zi ) : Point(xi,yi) {z=zi;}
```

**If there are a number of superclasses, their constructor calls are made as a comma separated list. The explicit method initializes objects directly instead of using default values and additional assignments.**

## Destruction

When an object is destroyed, the destructor of its class is implicitly invoked. If this class is derived from other classes, their destructors are also called in a recursive call chain.

## Multiple Inheritance

In singular inheritance, a class may be derived from more than one superclass in a singular hierarchy or framework, by specifying the immediate superclass, e.g.:

```
Class Animal : public Living { . . . };
class Mammal : public Animal { . . . };
class Cat : public Mammal { . . . };
```

A class may also be derived from more than one superclass in different hierarchies forming a multiple inheritance framework, by specifying the immediate superclasses in a comma separated list, e.g.:

```
class Dog  : public Mammal, public Drawable { . . . };
class Frog : public Animal, public Drawable { . . . };
class Tree : public Living, public Drawable { . . . };
class DatePalm : public Tree { . . . };
```

Tutorial:

Define a Sphere class derived from the Circle class that was previously defined for chapter 3 tutorial. Public methods are:

- Constructors for given int center coordinates and radius, int radius with center at (0,0,0), and double radius with center at (0,0,0).
- Methods for obtaining volume and surface area.
- Methods for getting the values of the data members.
- Methods for resizing and shifting.
- Method for printing values of a Sphere object in a clear form.