CONSTRUCTORS AND DESTRUCTORS

Constructor is a special member function. Its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created.

The constructor functions have some special characteristics:

1. Constructor name and class name must be equal.
2. Constructor should be declared in the public section.
3. It has no return type. But it may have arguments.
4. They are invoked automatically, when the object is created.

Program:

```
#include<iostream.h>
#include<conio.h>
class sum
{
 public:
  sum()
  {
   int a,b,c;
   cin>>a>>b;
   c=a+b;
   cout<<"Sum="<<c<<endl;
  }
};
void main()
{
 clrscr();
 sum obj;
 getch();
}
output:
10 20
Sum=30
```

Parameterized constructs:

The constructors that can take arguments are called parameterized constructors. We must pass the initial values as arguments to the constructor function when an object is declared.

Program:

```
#include<iostream.h>
#include<conio.h>
class sum
{
 public:
  sum(int a,int b)
  {
   int c;
   c=a+b;
   cout<<c<<endl;
  }
};
void main()
```

```
{
 clrscr();
 sum obj(10,20);
 getch();
}
output:
30
```

Constructor overloading:

The process of sharing the same name by two or more functions is referred to as function overloading. Similarly when more than one constructor function is defined in a class, we say that the constructor is overloaded.

Program:

```
#include<iostream.h>
#include<conio.h>
class sum
{
 public:
  sum()
  {
   int a,b,c;
   cin>>a>>b;
   c=a+b;
   cout<<"Sum="<<c<<endl;
  }
  sum(int a,int b)
  {
   cout<<"Sum="<<a+b<<endl;
  }
  sum(int a,int b,int c)
  {
   cout<<"Sum="<<a+b+c<<endl;
  }
};
void main()
{
 clrscr();
 sum obj1;
 sum obj2(4,5);
 sum obj3(6,45,7);
 getch();
}
output:
10 20
Sum=30
Sum=9
Sum=58
```

Constructor with default argument:

It is possible to define constructors with default arguments. When called with no arguments, it becomes a default constructor.

Program:

```
#include<iostream.h>
#include<conio.h>
class sum
{
 public:
  sum(int a=10,int b=20);
};
sum :: sum(int a,int b)
{
 int c;
 c=a+b;
 cout<<c<<endl;
}
void main()
{
 clrscr();
 sum obj1;
 sum obj2(50);
 sum obj3(50,100);
 getch();
}
output:
30
70
150
```

Copy Constructor

Copy constructor is used to declare and initialize an object from another object.

The general format of the copy consturctor is class-name(class-name &ptr);

Program:

```
#include<iostream.h>
#include<conio.h>
class sum
{
 private:
  int a,b;
 public:
  sum(int p,int q)
  {
   a=p;
   b=q;
   cout<<a+b<<endl;
  }
  sum(sum &ptr)
  {
   cout<<ptr.a*ptr.b;
  }
 };
void main()
{
 clrscr();
 sum e1(20,30);
 e1.display();
 sum e2=e1;  //copy constructor call
 sum e3(e1); //copy constructor call
 getch();
}
```

```
output:
50
600
```

Destructor

A destructor is a function that automatically executes when an object is destroyed.

Syntax for Writing a destructors:

1. The destructor function name is the same as that of the class it belongs except that the first character of the name must be tilde(~)
2. It has no return type and no argument.
3. They are invoked automatically, when the object is deleted.

```
~classname( )
{
   body of the destructor
}
```

Program:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class sum
{
 sum()
 {
  int a,b,c;
  cin>>a>>b;
  c=a+b;
  cout<<"Sum="<<c<<endl;
}
~sum()
{
 cout<<"Object is destroyed"<<endl;
}
};
void main()
{
 clrscr();
 sum b;
 getch();
}
output:
10 20
Sum=30
Object is destroyed
```

10. OPERATOR OVERLOADING

Operators

An operator is a symbol, which indicates the particular operation.

Classification of operators

The operators in C++ are classified as

1. Unary – Operator depends on one operand
2. Binary – Operator depends on two operands
3. Ternary – Operator depends on three operands.

Operator Overloading

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. Operator overloading provides a flexible option for the creation of new definitions for most of the c++ operators.

Rules for overloading Operators

1. Only existing operators can be overloaded new operators cannot be created.

2. The overload operator must have at least one operand that is of user-defined type.

3. We cannot change the basic meaning of an operator.

4. There are some operators that cannot be overloaded.

   A) class member access operator (.)
   B) Scope resolution operator (:)
   C) Size operator (sizeof)
   D) Conditional operator (? :)

5. We cannot use friend functions to overload certain operators.

   A) Assignment operator =
   B) Function call operator ()
   c) Subscripting operator []
   d) Class member access operator ->

6. Operator functions must be either member functions or friend functions. A basic difference between them is that a friend function will have one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators

The process of overloading involves the following steps

1. Create a class that defines that 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.

Member Function

```
Return-type operator op(argument list);
```

Friend Function

```
Friend return-type operator op(argument list);
```

3. Define the operator function to implement the required operations.

Member function

```
Return-type classname :: operator(argument list)
{
      function body
}
```

Friend function:

```
returntype operator op(argument list)
{
      function body
}
```

4. Overloaded operator functions can be invoked by expressions such as

```
Op x for unary operators and x op y for binary operators.
```

Friend function

```
Operator op (x)
Operator op (x, y)
```

Overloading unary operators using member function

Program:

```
#include<iostream.h>
#include<conio.h>
class complex
{
  float x,y;
  public:
  void getdata(float a,float b);
  void display();
  void operator-();
};
void complex :: getdata(float a,float b)
{
```

```
 x=a;
 y=b;
}
void complex :: display()
{
 cout<<"("<<x<<")+i("<<y<<")"<<endl;
}
void complex :: operator-()
{
 x=-x;
 y=-y;
}
void main()
{
 complex z;
 clrscr();
 z.getdata(4.5,6.3);
 z.display();
 -z;
 z.display();
 getch();
}
output:
(4.5)+i(6.3);
(-4.5)+i(-6.3)
```

Overloading unary operators using friend function

Program

```
#include<iostream.h>
#include<conio.h>
class complex
{
  float x,y;
 public:
  void getdata(float a,float b);
  void display();
  friend complex operator-(complex z1);
};
void complex :: getdata(float a,float b)
{
 x=a;
 y=b;
}
void complex :: display()
{
 cout<<"("<<x<<")+i("<<y<<")"<<endl;
}
complex operator-(complex z1)
{
 complex k;
 k.x=-z1.x;
 k.y=-z1.y;
 return(k);
}
void main()
{
 complex z1,z2;
 clrscr();
 z1.getdata(4.5,6.3);
 z1.display();
```

```
 z2=-z1;
 z2.display();
 getch();
}
output:
(4.5)+i(6.3);
(-4.5)+i(-6.3)
```

Overloading binary operators using member function:

Program:

```
#include<iostream.h>
#include<conio.h>
class complex
{
  float x,y;
 public:
  void getdata(float a,float b);
  void display();
  complex operator+(complex c1);
};
void complex :: getdata(float a,float b)
{
 x=a;
 y=b;
}
void complex :: display()
{
 cout<<"("<<x<<")+i("<<y<<")"<<endl;
}
complex complex :: operator+(complex c1)
{
 complex temp;
 temp.x=x+c1.x;
 temp.y=y+c1.y;
 return(temp);
}
void main()
{
 complex z1,z2,z3;
 clrscr();
 z1.getdata(4.5,6.3);
 z2.getdata(3.7,2.6);
 z3=z1+z2;
 z1.display();
 z2.display();
 z3.display();
 getch();
}
output:
(4.5)+i(6.3)
(3.7)+i(2.6)
(8.2)+i(8.9)
```

Overloading binary operators using friend function

Program:

```
#include<iostream.h>
#include<conio.h>
class complex
{
  float x,y;
 public:
  void getdata(float a,float b);
  void display();
  friend complex operator+(complex c1,complex c2);
};
void complex :: getdata(float a,float b)
{
 x=a;
 y=b;
}
void complex :: display()
{
 cout<<"("<<x<<")+i("<<y<<")"<<endl;
}
complex operator+(complex c1,complex c2)
{
 complex temp;
 temp.x=c1.x+c2.x;
 temp.y=c1.y+c2.y;
 return(temp);
}
void main()
{
 complex z1,z2,z3;
 clrscr();
 z1.getdata(4.5,6.3);
 z2.getdata(3.7,2.6);
 z3=z1+z2;
 z1.display();
 z2.display();
 z3.display();
 getch();
}
output:
(4.5)+i(6.3)
(3.7)+i(2.6)
(8.2)+i(8.9)
```

Type conversions

Conversion from basic type to class type

The conversion from basic type to class type is easy to accomplish. This conversion can be done by the constructors. The constructors used for type conversion take a single argument whose type is to be converted.

Conversion from class type to basic type

C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type. The general form of an overloaded casting operator function, usually referred to as a conversion function is:

```
Operator typename()
{
 Function body
}
```

This function converts a class type data to type name.

The casting operator function should satisfy the following conditions:-

1. It must be a class member.
2. It must not specify a return type.
3. It must not have any arguments.

Program:

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class vector
{
 private:
  float a,b,c;

 public:
  void getdata(float x,float y,float z)
  {
   a=x;
   b=y;
   c=z;
  }
  void display(void)
  {
   cout<<"("<<a<<")i+("<<b<<")j+("<<c<<")k"<<endl;
  }
  operator float()
  {
   float k;
   k=sqrt(a*a+b*b+c*c);
   return(k);
  }
};
void main()
{
 vector v1;
 float p;
 clrscr();
 v1.getdata(2,4,3);
 p=float(v1);
 v1.display();
 cout<<p<<endl;
 getch();
}
output:
```

(2)i+(4)j+(3)k
5.385165