UNIT - 4

CONSTRUCTORS & DESTRUCTORS
OPERATOR OVERLOADING

CONSTRUCTORS & DESTRUCTORS

Introduction

Until now in our C++ programs, we are initializing the data members of a class by creating one or more member functions and passing values as parameters to those functions. We know that C++ considers user-defined types on par with pre-defined types. We can initialize a pre-defined type at the time of declaration itself as shown below:

```
int x = 20;
```

Similarly, to initialize the data members when an object is created, C++ provides a special member function called a *constructor* and to de-initialize an object, another special member function called a *destructor* is used.

Constructor

Constructor is a special member function which is invoked automatically when an object is created to initialize the data members of that object. Following are the characteristics of a constructor:

- The name of a constructor is same as the class name.
- A constructor must be declared in public section of the class.
- It should not be called explicitly as it is automatically called when an object is created.
- A constructor doesn't return any value. So, a constructor doesn't have any return type. Not even *void*.
- A constructor cannot be inherited or virtual.
- The address of a constructor cannot be referred in programs. So, pointers and references cannot be used with constructors.
- A constructor cannot be declared as *static*, *volatile*, or *const*.
- Constructors can be overloaded.
- Constructors can have default arguments.

Like a member function, we can define a constructor inside or outside the class. Syntax for defining a constructor inside the class is as follows:

```
class ClassName
{
        private:
        ...
        public:
        ...
        ClassName( )
        {
```

```
		//Body of constructor
	}
	...
};
```

Syntax for defining a constructor outside the class is as follows:

```
class ClassName
{
	private:
	...
	public:
	...
	ClassName( );		//Constructor declaration
	...
};
ClassName::ClassName( )	//Constructor definition
{
	//Body of constructor
}
```

Following is an example which demonstrates constructor defined inside the class:

```
#include <iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
	public:
		Student()	//Constructor
		{
			age = 20;
			branch = "CSE";
		}
		void get_details()
		{
			cout<<"Age of student is: "<<age<<endl;
			cout<<"Branch of student is: "<<branch<<endl;
		}
};
int main()
{
	Student s1;
	s1.get_details();
	return 0;
}
```

Output for the above program is as follows:

```
Age of student is: 20
Branch of student is: CSE
```

Following is an example which demonstrates constructor defined outside the class:

```
#include <iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
	public:
		Student();	//Constructor declaration
		void get_details()
		{
			cout<<"Age of student is: "<<age<<endl;
			cout<<"Branch of student is: "<<branch<<endl;
		}
};
Student::Student()	//Constructor definition
{
	age = 20;
	branch = "CSE";
}
int main()
{
	Student s1;
	s1.get_details();
	return 0;
}
```

Output for the above program is as follows:

```
Age of student is: 20
Branch of student is: CSE
```

Types of Constructors

A constructor can be classified into following types:

1. Dummy constructor
2. Default constructor
3. Parameterized constructor
4. Copy constructor
5. Dynamic constructor

Dummy Constructor

When no constructors are declared explicitly in a C++ program, a dummy constructor is invoked which does nothing. So, data members of a class are not initialized and contains garbage values. Following program demonstrates a dummy constructor:

```
#include <iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
	public:
		void get_details()
		{
			cout<<"Age of student is: "<<age<<endl;
			cout<<"Branch of student is: "<<branch<<endl;
		}
};
int main()
{
	Student s1;
	s1.get_details();
	return 0;
}
```

Output for the above program is as follows:

```
Age of student is: 4883904
Branch of student is:
```

You can see that since age and branch are not initialized, they are displaying garbage values.

Default Constructor

A constructor which does not contain any parameters is called a default constructor or also called as no parameter constructor. It is used to initialize the data members to some default values. Following program demonstrates a default constructor:

```
#include <iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
	public:
		Student()	//Constructor
		{
			age = 20;
			branch = "CSE";
		}
		void get_details()
		{
			cout<<"Age of student is: "<<age<<endl;
			cout<<"Branch of student is: "<<branch<<endl;
		}
};
int main()
{
	Student s1;
	s1.get_details();
	return 0;
}
```

Output for the above program is as follows:

```
Age of student is: 20
Branch of student is: CSE
```

Parameterized Constructor

A constructor which accepts one or more parameters is called a parameterized constructor. Following program demonstrates a parameterized constructor:

```
#include <iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
	public:
		Student(int a, string b)//Constructor
		{
			age = a;
			branch = b;
		}
		void get_details()
		{
			cout<<"Age of student is: "<<age<<endl;
			cout<<"Branch of student is: "<<branch<<endl;
		}
};
int main()
{
        Student s1(21, "CSE");
        s1.get_details();
        return 0;
}
```

Output for the above program is as follows:

```
Age of student is: 21
Branch of student is: CSE
```

Copy Constructor

A constructor which accepts an already existing object through reference to copy the data member values is called a copy constructor. As the name implies a copy constructor is used to create a copy of an already existing object. Following program demonstrates a copy constructor:

```
#include <iostream>
using namespace std;
class Square
{
        private:
                int side;
        public:
                Square(int s)
                {
                        side = s;
                }
                Square(Square &s)
                {
                        side = s.side;
                }
                void display()
                {
                        cout<<"Side of square is: "<<side<<endl;
                }
};
int main()
{
        Square s1(10);
        Square s2(s1);
        Square s3 = s2;
        s1.display();
        s2.display();
        s3.display();
        return 0;
}
```

Output for the above program is as follows:

```
Side of square is: 10
Side of square is: 10
Side of square is: 10
```

In the above program you can see that there are two ways for calling a copy constructor. One way is to call the copy constructor by passing the object as a parameter (in case of s2). Other way is to directly assign an existing object to a new object (in case of s3).

Note: Copying an object through assignment only works during object declaration. After declaring an object, assignment just serves its basic purpose.

Dynamic Constructor

A constructor in which the memory for data members is allocated dynamically is called a dynamic constructor. Following program demonstrates dynamic constructor:

```
#include <iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
		int *marks;
	public:
		Student(int n)  //Constructor
		{
			marks = new int[n];
			cout<<"Enter marks for "<<n<<" subjects: ";
			for(int i=0; i<n; i++)
				cin>>marks[i];
		}
		void display_marks(int n)
		{
			cout<<"Marks are: ";
			for(int i=0; i<n; i++)
				cout<<marks[i]<<" ";
		}
};
int main()
{
	int n;
	cout<<"Enter no. of subjects: ";
	cin>>n;
	Student s1(n);
```

```
        s1.display_marks(n);
        return 0;
}
```

Input and output for the above program are as follows:

```
Enter no. of subjects: 4
Enter marks for 4 subjects: 20 18 19 16
Marks are: 20 18 19 16
```

Constructor with Default Arguments

A constructor like a member function can have default arguments. The default arguments should be declared at the end of the parameters list. Following program demonstrates a constructor with default arguments:

```
#include <iostream>
using namespace std;
class Student
{
        private:
                string name;
                string regdno;
                int age;
                string branch;
                int *marks;
        public:
                Student(string branch, int n = 4)       //Constructor
                {
                        marks = new int[n];
                        cout<<"Enter marks for "<<n<<" subjects: ";
                        for(int i=0; i<n; i++)
                                cin>>marks[i];
                        cout<<"Student marks are: ";
                        for(int i=0; i<n; i++)
                                cout<<marks[i]<<" ";
                        cout<<endl;
                }
};
int main()
{
        int n;
        cout<<"Enter no. of subjects: ";
        cin>>n;
        Student s1("CSE", n);
        Student s2("CSE");
        return 0;
}
```

Output for the above program is as follows:

```
Enter no. of subjects: 3
Enter marks for 3 subjects: 10 20 30
Student marks are: 10 20 30
Enter marks for 4 subjects: 15 18 16 20
Student marks are: 15 18 16 20
```

Constructor Overloading

Declaring multiple constructors with varying number of arguments or types of arguments is known as constructor overloading. All the overloaded constructors will have the same name as that of the class. Following program demonstrates constructor overloading:

```
#include <iostream>
using namespace std;
class Student
{
        private:
                string name;
                string regdno;
                int age;
                string branch;
        public:
                Student()
                {
                        cout<<"Default student constructor is invoked"<<endl;
                }
                Student(int age)
                {
                        cout<<"Student constructor with one parameter is invoked"<<endl;
                }
                Student(string n, string reg, int a, string br)
                {
                        cout<<"Student constructor with four parameters is invoked"<<endl;
                }
};
int main()
{
        Student s1;
        Student s2(20);
        Student s3("Mahesh", "501", 20, "CSE");
        return 0;
}
```

Output for the above program is as follows:

```
Default student constructor is invoked
Student constructor with one parameter is invoked
Student constructor with four parameters is invoked
```

Destructor

A destructor is a special member function that is invoked automatically when the object goes out of scope and is used to perform clean up actions. Following are the characteristics of a destructor:

- Name of the destructor is same as the class name. However, destructor name is preceded by the tilde (~) symbol.
- A destructor is called when a object goes out of scope.
- A destructor is also called when the programmer calls *delete* operator on an object.
- Like a constructor, destructor is also declared in the *public* section of a class.
- The order of invoking a destructor is the reverse of invoking a constructor.
- Destructors do not take any arguments and hence cannot be overloaded.
- A destructor does not return a value.
- The address of a destructor cannot be accessed in a program.
- An object with a constructor or a destructor cannot be used as members of a union.
- Destructor cannot be inherited.
- Unlike constructors, destructors can be virtual.

Syntax for creating a destructor is as shown below:

```
~ClassName( )
{
	//Body of destructor
}
```

Following program demonstrates a destructor:

```
#include <iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
	public:
		Student(string n)
		{
			regdno = n;
			cout<<"Memory allocated for student "<<regdno<<endl;
		}
		~Student()
		{
			cout<<"Memory deallocated for student "<<regdno<<endl;
		}
};
int main()
```

```
{
        Student s1("501");
        Student s2("502");
        Student s3("503");
        return 0;
}
```

Output for the above program is as follows:

```
Memory allocated for student 501
Memory allocated for student 502
Memory allocated for student 503
Memory deallocated for student 503
Memory deallocated for student 502
Memory deallocated for student 501
```

Note: If a destructor is not defined by the programmer, C++ compiler will create a inline public destructor automatically.

Constructors and Destructors

Explicit Invocation

Constructors and destructors can be invoked (called) explicitly from the main() function. A constructor can be invoked as shown below:

ClassName(params-list);

For example, constructor of *Student* class can be called as shown below:

Student( );

In case of a destructor, we need to write the class name along with the destructor name as follows:

object_name.ClassName::~ClassName( );

or

object_name.~ClassName( );

For example, destructor of *Student* class can be called as follows:

s1.~Student( );

A destructor can be called from a constructor as follows:

```
class Student
{
        ...
        Student( )
        {
                this->Student::~Student( );     //Call to destructor
                ...
        }
        ...
};
```

Similarly, a constructor can be called from a destructor as follows:

```
class Student
{
        ...
        ~Student( )
        {
                Student( );     //Call to constructor
                ...
        }
        ...
};
```

Private Constructor and Destructor

Sometimes programmers may want to prevent direct access to a constructor or destructor. In such cases, they can be made *private*. Access to a private constructor or destructor can be granted through *public* member functions or through a *friend* function. Following program demonstrates creating private constructors and a destructor and accessing them using public member functions:

```
#include <iostream>
using namespace std;
class Student
{
        private:
                string name;
                string regdno;
                int age;
                string branch;
                Student()
                {

                }
                Student(string n)
                {
                        regdno = n;
                        cout<<"Memory allocated for student "<<regdno<<endl;
```

```
		}
		~Student()
		{
			cout<<"Memory deallocated for student "<<regdno<<endl;
		}
	public:
		Student* init(string reg)
		{
			return new Student(reg);
		}
		void destroy(Student *s)
		{
			delete(s);
		}
};
int main()
{
	Student *s;
	s = s->init("501");
	s->destroy(s);
	return 0;
}
```

Output for the above program is as follows:

```
Memory allocated for student 501
Memory deallocated for student 501
```

Object Copy

Frequently while programming there will be need to create a copy of an existing object. This can be done in three ways:

Shallow Copy

In this method the copy (object) only copies the addresses (for dynamic memory) of the source object's members. This is similar with the semantics of *static* data members. A same copy is shared among several objects. Similarly, in shallow copy method, both the source and copied objects share the same memory location.

Deep Copy

In this method the copied object maintains a copy of the value of each source object's data members. To achieve this, programmer should explicitly provide a copy constructor to copy all the values of source object's members (including dynamic memory).

Lazy Copy

This method has the advantages of the above two methods. Initially during copying, it uses shallow copying and maintains a counter to keep track of how many objects are sharing the data. When there is an attempt to modify the data, it may initiate deep copy, if necessary.

For more info visit this link: https://www.hackerearth.com/notes/deep-copy-and-shallow-copy/

Constant Objects

When there is a need to create a read-only object, the object can be declared as a constant object using the *const* keyword. Syntax for creating a constant object is as follows:

ClassName const object_name(params-list);

Following are the characteristics of a constant object:

- Constant object can be initialized only through a constructor.
- Data members of a constant object cannot be modified by any member function.
- Constant object are read-only objects.
- Constant objects should be accessed only through constant member functions.

Following program demonstrates a constant object:

```
#include <iostream>
using namespace std;
class Student
{
	private:
		string name;
		string regdno;
		int age;
		string branch;
	public:
		Student(string br)
		{
			branch = br;
		}
		void get_branch() const
		{
			cout<<"Branch is: "<<branch;
		}
};
int main()
{
	Student const s("CSE");
```

```
        s.get_branch();
        return 0;
}
```

Output for the above program is as follows:

```
Branch is: CSE
```

Anonymous Objects

An object which has no name is known as an anonymous object. Syntax for creating an anonymous object is as follows:

ClassName(params-list);

An anonymous object can be created and used in any statement without using any name. It is automatically destroyed after the control moves on to the next statement. Following program demonstrates an anonymous object:

```
#include <iostream>
using namespace std;
class Student
{
        private:
                string name;
                string regdno;
                int age;
                string branch;
        public:
                Student(string reg)
                {
                        regdno = reg;
                        cout<<"Student with regdno "<<regdno<<" is created";
                }
};
int main()
{
        Student("501");         //Anonymous object
        return 0;
}
```

Output for the above program is as follows:

```
Student with regdno 501 is created
```

Following are the advantages of anonymous objects:

- They result in cleaner, shorter, and efficient code.

- There is no need to name objects which are going to be used temporarily.
- No need to declare them.
- Unnamed are objects are destroyed automatically after their use is over.

Anonymous Classes

A class which does not have a name is called an anonymous class. They are generally created when the class body is very short and only one object of that class is required. Following are the restrictions on anonymous classes:

- Anonymous classes cannot have a constructor.
- Anonymous classes can have a destructor.
- Anonymous classes cannot be passed as arguments to functions.
- Anonymous classes cannot be used as return values from functions.

Following program demonstrates an anonymous class:

```
#include <iostream>
using namespace std;
typedef class           //Anonymous class
{
        private:
                string name;
                string regdno;
                int age;
                string branch;
        public:
                void set_branch(string br)
                {
                        branch = br;
                }
                void get_branch()
                {
                        cout<<"Branch is: "<<branch;
                }
}Student;
int main()
{
        Student s;
        s.set_branch("CSE");
        s.get_branch();
        return 0;
}
```

Output for the above program is as follows:

```
Branch is: CSE
```

OPERATOR OVERLOADING

Introduction

The feature of C++ which allows programmers to redefine the meaning of operators to make them work with classes and objects is *operator overloading*.

While evaluating expressions, C++ compiler checks whether the operands are of normal types or user-defined types. If the operands are of built-in or normal type, it performs the regular operation. If the operands are of user-defined type, compiler checks if there is a function available with an operator or not. If no such function is available, it throws an error.

Operator overloading is another type of compile-time polymorphism like function overloading and constructor overloading. Operator overloading is provided by the language, programmer, or both.

Advantages of Operator Overloading

Following are the advantages of operator overloading:

- Operator overloading enables programmers to use notation closer to the target domain. For example we can add two matrices by writing M1 + M2 rather than writing M1.add(M2).
- Operator overloading provides similar syntactic support of built-in types to user-defined types.
- Operator overloading makes the program easier to understand.

Syntax for Operator Overloading

An operator that is to be overloaded is declared in the public section. The syntax for an operator overloading function is as follows:

```
class ClassName
{
        ...
        public:
                ...
                return-type operator op(params-list)
                {
                        //body of the function
                        ...
                }
                ...
};
```

In the above syntax, *op* is the operator to be overloaded and *operator* is a keyword. For example, the function for overloading + operator for adding two complex numbers will be as follows:

```
Complex operator +(Complex &c2)
{
        Complex temp;
        temp.real = real + c2.real;
        temp.imag = imag + c2.imag;
        return temp;
}
```

Now, if c1 and c2 are two objects of *Complex* class, we can add them by writing c3 = c1+ c2.

Complete program for adding two complex numbers is as follows:

```
#include <iostream>
using namespace std;
class Complex
{
        private:
                float real;
                float imag;
        public:
                Complex() {}
                Complex(float a, float b)
                {
                        real = a;
                        imag = b;
                }
                Complex operator +(Complex &c2)
                {
                        Complex temp;
                        temp.real = real + c2.real;
                        temp.imag = imag + c2.imag;
                        return temp;
                }
                void show_details()
                {
                        cout<<real<<"+i"<<imag;
                }
};
int main()
{
        Complex c1(5, 4);
        Complex c2(3, 1);
        Complex c3 = c1 + c2;
        c3.show_details();
        return 0;
}
```

Output of the above program is as follows:

```
8+i5
```

Operators that cannot be Overloaded

Although most of the operators in C++ can be overloaded, there are some operators which cannot be overloaded. They are: scope resolution operator (::), member selection operator (.) and ternary operator (?:). Following are some points to remember about operator overloading:

- Operator overloading extends the semantics without changing its syntax.
- Some operators like assignment operator (=) and address operator (&) are already overloaded by C++.
- Operator overloading does not alter the precedence and associativity of operators.
- New operators cannot be created using operator overloading.
- When operators such as &&, || are overloaded, they lose their special properties of short-circuit evaluation and sequencing.
- Overloaded operators cannot have default arguments.
- All overloaded operators except the assignment operator are inherited by the derived class.
- Number of operands cannot be changed for an operator using operator overloading.

Implementing Operator Overloading

Operator overloading can be implemented in two ways. They are:

1. Using a member function
2. Using a friend function

Differences between using a member function and a friend function to implement operator overloading are as follows:

Member Function	Friend Function
Number of parameters to be passed is reduced by one, as the calling object is implicitly supplied as an operand.	Number of parameters to be passed is more.
Unary operators takes no explicit parameters.	Unary operators takes one explicit parameter.
Binary operators takes only one explicit parameter.	Binary operators takes two explicit parameters.
Left-hand operand has to be the calling object.	Left-hand operand need not be an object of the class.
Writing Obj2 = Obj1 + 10 is allowed but Obj2 = 10 + Obj1 is not allowed	Writing either Obj2 = Obj + 10 or Obj2 = 10 + Obj1 is allowed.

Overloading Unary Operators

Operators which work on a single operand are known as unary operators. Examples are: increment operator(++), decrement operator(--), unary minus operator(-), logical not operator(!) etc.

Using a member function to overload an unary operator

Following program demonstrates overloading unary minus operator using a member function:

```
#include <iostream>
using namespace std;
class Number
{
	private:
		int x;
	public:
		Number(int x)
		{
			this->x = x;
		}
		void operator -()
		{
			x = -x;
		}
		void display()
		{
			cout<<"x = "<<x;
		}
};
int main()
{
	Number n1(10);
	-n1;
	n1.display();
	return 0;
}
```

Output of the above program is as follows:

```
x = -10
```

In the above program the value of *x* is changed and printed using the function *display()*. We can return an object of class *Number* after modifying *x* as shown in the following program:

```
#include <iostream>
using namespace std;
class Number
{
	private:
		int x;
	public:
		Number(int x)
		{
			this->x = x;
		}
		Number operator -()
		{
			x = -x;
			return Number(x);
		}
		void display()
		{
			cout<<"x = "<<x<<endl;
		}
};
int main()
{
	Number n1(10);
	Number n2 = -n1;
	n2.display();
	return 0;
}
```

Output of the above program is as follows:

```
x = -10
```

Using a friend function to overload an unary operator

When a friend function is used to overload an unary operator following points must be taken care of:

- The function will take one operand as a parameter.
- The operand will be an object of a class.
- The function can access the private members only though the object.
- The function may or may not return any value.
- The friend function will not have access to the *this* pointer.

Following program demonstrates overloading an unary operator using a friend function:

```
#include <iostream>
using namespace std;
class Number
{
	private:
		int x;
	public:
		Number(int x)
		{
			this->x = x;
		}
		friend Number operator -(Number &);
		void display()
		{
			cout<<"x = "<<x<<endl;
		}
};
Number operator -(Number &n)
{
	return Number(-n.x);
}
int main()
{
	Number n1(20);
	Number n2 = -n1;
	n2.display();
	return 0;
}
```

Output of the above program is as follows:

```
x = -20;
```

Overloading prefix operators

The syntax for overloading prefix increment and decrement operators is as follows:

```
return-type operator ++( )
{
	//Body of the function
	...
}
```

Following program demonstrates overloading prefix increment and decrement operators:

```
#include <iostream>
using namespace std;
class Number
{
	private:
		int x;
	public:
		Number(int x)
		{
			this->x = x;
		}
		Number operator ++()
		{
			x = x + 1;
			return Number(x);
		}
		Number operator --()
		{
			x = x - 1;
			return Number(x);
		}
		void display()
		{
			cout<<"x = "<<x<<endl;
		}
};
int main()
{
	Number n1(20);
	Number n2 = ++n1;
	n2.display();
	Number n3(20);
	Number n4 = --n3;
	n4.display();
	return 0;
}
```

Output of the above program is as follows:

```
x = 21
x = 19
```

Overloading postfix operators

When overloading postfix increment and decrement operators we have to include an extra integer parameter to avoid confusion with prefix operators. Syntax for overloading postfix increment operator is as follows:

```
return-type operator ++(int)
{
	//Body of function
	...
}
```

The *int* parameter is a dummy parameter and there is no need to give any name for it. Following program demonstrates overloading postfix increment and decrement operators:

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

```
{
	private:
		int x;
	public:
		Number(int x)
		{
			this->x = x;
		}
		Number operator ++(int)
		{
			return Number(x++);
		}
		Number operator --(int)
		{
			return Number(x--);
		}
		void display()
		{
			cout<<"x = "<<x<<endl;
		}
};
int main()
{
	Number n1(20);
	Number n2 = n1++;
	n1.display();
	n2.display();
	Number n3 = n2--;
	n3.display();
	n2.display();
	return 0;
}
```

Output of the above program is as follows:

```
x = 21
x = 20
x = 20
x = 19
```

Overloading Binary Operators

As unary operators can be overloaded, we can also overload binary operators. Syntax for overloading a binary operator using a member function is as follows:

```
return-type operator op(ClassName &)
{
	//Body of function
```

```
        ...
}
```

Syntax for overloading a binary operator using a friend function is as follows:

```
return-type operator op(ClassName &, ClassName &)
{
        //Body of function
        ...
}
```

Following program demonstrates overloading the binary operator + using a member function:

```
#include <iostream>
using namespace std;
class Number
{
        private:
                int x;
        public:
                Number() {}
                Number(int x)
                {
                        this->x = x;
                }
                Number operator +(Number &n)
                {
                        Number temp;
                        temp.x = x + n.x;
                        return temp;
                }
                void display()
                {
                        cout<<"x = "<<x<<endl;
                }
};
int main()
{
        Number n1(20);
        Number n2(10);
        Number n3 = n1 + n2;
        n3.display();
        return 0;
}
```

Output of the above program is as follows:

```
x = 30
```

Following program demonstrates overloading the binary operator + using a friend function:

```
#include <iostream>
using namespace std;
class Number
{
	private:
		int x;
	public:
		Number() {}
		Number(int x)
		{
			this->x = x;
		}
		friend Number operator +(Number &, Number &);
		void display()
		{
			cout<<"x = "<<x<<endl;
		}
};
Number operator +(Number &n1, Number &n2)
{
	Number temp;
	temp.x = n1.x + n2.x;
	return temp;
}
int main()
{
	Number n1(20);
	Number n2(10);
	Number n3 = n1 + n2;
	n3.display();
	return 0;
}
```

Output of the above program is as follows:

```
x = 30
```

Note: Operators such as =, ( ), [ ], and -> cannot be overloaded using friend functions.

Overloading Special Operators

Some of the special operators in C++ are:

- new - used to allocate memory dynamically
- delete - used to free memory dynamically
- ( ) and [ ] - subscript operators

- -> - member access operator

Let's see how to overload them.

Overloading *new* and *delete* operators

C++ allows programmers to overload *new* and *delete* operators due to following reasons:

- To add more functionality when allocating or deallocating memory.
- To allow users to debug the program and keep track of memory allocation and deallocation in their programs.

Syntax for overloading new operator is as follows:

void operator new(size_t size);*

Parameter *size* specifies the size of memory to be allocated whose data type is *size_t*. It returns a pointer to the memory allocated.

Syntax for overloading delete operator is as follows:

void operator delete(void);*

The function receives a parameter of type *void** and returns nothing. Both functions for new and delete are *static* by default and can't access *this* pointer. To delete an array objects, the operator *delete[ ]* must be overloaded.

Following program demonstrates overloading both *new* and *delete* operators:

```
#include <iostream>
using namespace std;
class Number
{
	private:
		int x;
	public:
		Number(int x)
		{
			this->x = x;
		}
		void* operator new(size_t size)
		{
			void *ptr = ::new int[size];	//Using global new operator
			cout<<"Memory allocated of size: "<<size<<endl;
			return ptr;
		}
		void operator delete(void *ptr)
		{
			::delete(ptr);	//Using global delete operator
```

```
                        cout<<"Memory deallocated"<<endl;
                }
                void display()
                {
                        cout<<"x = "<<x<<endl;
                }
};
int main()
{
        Number *n = new Number(10);   //Invokes overloaded new operator
        n->display();
        delete n;         //Invokes overloaded delete operator
        return 0;
}
```

Output of the above program is as follows:

```
Memory allocated of size: 4
x = 10
Memory deallocated
```

In the above program, ::new and ::delete refers to the global new and delete operators. When *new* is called, compiler executes the overloaded function for new and also automatically calls the constructor.

Advantages of overloading *new* and *delete* operators

- The overloaded *new* operator function can receive one or more parameters. This allows flexibility in customizing memory allocation.
- Overloaded *delete* operator provides garbage collection for objects of classes.
- Programmers can add exception handling code while allocating memory.
- Programmers can use memory management functions like *malloc(), realloc(),* and *free()* inside the overloaded functions of *new* and *delete* operators.

Overloading subscript operators [ ] and ( )

The subscript operator [ ] is used to access array elements. The function declared for overloading [ ] or ( ) should be non-static member function of a class. Syntax for overloading [ ] operator is as follows:

```
int& operator [ ](int x)
{
        //Body of function
        ...
}
```

The overloaded function must return an integer value by reference.

Following example demonstrates overloading [ ] operator:

```
#include <iostream>
using namespace std;
class Number
{
	private:
		int x[5];
	public:
		void read(int n)
		{
			cout<<"Enter "<<n<<" numbers: ";
			for(int i=0; i<n; i++)
			{
				cin>>x[i];
			}
		}
		int& operator [](int i)
		{
			return x[i];
		}
};
int main()
{
	Number n1;
	n1.read(5);
	cout<<"Element is: "<<n1[2];
	return 0;
}
```

Input and Output of the above program is as follows:

```
Enter 5 numbers: 1 2 3 4 5
Element is: 3
```

In case of multiple subscripts, we can overload ( ) operator instead of [ ] operator. Syntax for overloading ( ) operator is as follows:

```
int& operator ( ) (int i, int j,...)
{
	//Body of function
	...
}
```

Example for overloading ( ) operator is as follows:

```
#include <iostream>
using namespace std;
class Matrix
{
	private:
		int x[2][2];
	public:
		void read()
		{
			cout<<"Enter 2x2 matrix elements: ";
			for(int i=0; i<2; i++)
			{
				for(int j=0; j<2; j++)
					cin>>x[i][j];
			}
		}
		int& operator ()(int i, int j)
		{
			return x[i][j];
		}
};
int main()
{
	Matrix m;
	m.read();
	cout<<"Element is: "<<m(1,1);
	return 0;
}
```

Input and output for the above program is as follows:

```
Enter 2x2 matrix elements:
1 2
3 4
Element is: 4
```

Overloading class member access operator

Class member access can be controlled by overloading the class member access operator (->). It is an unary operator as it operates on only operand, the object. The overloaded function must be a non-static function and its syntax is as follows:

```
ClassName * operator ->(void)
{
	//Body of function
	...
}
```

Following program demonstrates overloading the member access operator (->):

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

```
{
	public:
		int x;
		Number(int x)
		{
			this->x = x;
		}
		Number * operator ->()
		{
			return this;
		}
};
int main()
{
	Number n1(30);
	cout<<"x = "<<n1->x;
	return 0;
}
```

Output of the above program is as follows:

```
x = 30
```

Type Conversion

In expressions when there are constants or variables of different types (built-in), one or more types are converted to a destination type. Similarly, user-defined types like classes when used in expressions can also be converted to built-in types or vice versa. Following are the different possibilities of type conversion:

- Conversion from basic type to class type
- Conversion from class type to basic type
- Conversion from one class type to another class type

Conversion from basic type to class type

A value or variable of a basic type or built-in type can be converted to a class member type using a constructor. Following program demonstrates conversion of a basic type to a class type:

```
#include <iostream>
using namespace std;
class Number
{
	private:
		int x;
	public:
		Number(int x)
		{
			this->x = x;
		}
		void display()
		{
			cout<<"x = "<<x;
		}
};
int main()
{
	Number n1 = 30;	//Conversion from int to Number
	n1.display();
	return 0;
}
```

Output of the above program is as follows:

```
x = 30
```

Conversion from class type to basic type

A class type can be converted into a basic type by overloading the casting operator. Syntax for overloading the casting operator is as follows:

```
operator typename( )
{
	//Body of function
	...
}
```

Remember following points while overloading the casting operator:

- Function should be defined within the class.
- Function does not have any return type.
- Function does not take any arguments.
- Casting operator is a unary operator which works on only one operand, the object.

Following program demonstrates converting a class type to basic type:

```
#include <iostream>
using namespace std;
class Number
{
	private:
		int x;
	public:
		Number(int x)
		{
```

```
                this->x = x;
        }
        operator int()
        {
                int temp = x;
                return x;
        }
};
int main()
{
        Number n1 = 20;
        cout<<"x = "<<n1;
        return 0;
}
```

Output of the above program is as follows:

```
x = 20
```

Conversion from one class type to another class type

While writing programs it is common to assign an object of one class type to an object of another class type as follows:

destination-object = *source-object;*

In the above notation *destination-object* is an object of destination class type and *source-object* is an object of source class type. The conversion from source class type to destination can be done in two ways based on in which class the conversion is performed.

Conversion in source class

When performing conversion in source class, the type cast operator should be overloaded. Syntax for overloading cast operator is as follows:

```
operator typename( )
{
        //body of function
        ...
}
```

In the above syntax *typename* is generally the destination class name. Following program converts a square to a rectangle using conversion in source class:

```
#include <iostream>
using namespace std;
class Rectangle
{
        private:
```

```
		int length;
		int breadth;
	public:
		Rectangle(int l, int b)
		{
			length = l;
			breadth = b;
		}
		void display()
		{
			cout<<"length = "<<length<<", breadth = "<<breadth;
		}
};
class Square
{
	private:
		int side;
	public:
		Square(int s)
		{
			side = s;
		}
		void display()
		{
			cout<<"side = "<<side;
		}
		operator Rectangle()
		{
			Rectangle r = Rectangle(side, side);
			return r;
		}
};
int main()
{
	Rectangle rect(20, 30);
	Square sq(30);
	rect = sq;	//conversion from square to rectangle
	rect.display();
	return 0;
}
```

Output of the above program is as follows:

```
length = 30, breadth = 30
```

Conversion in destination class

When performing conversion in destination class, a constructor of destination class with one argument of type source class should be created. Syntax of the constructor is as follows:

```
DestinationClass(SourceClass object)
{
	//code of constructor
	...
}
```

Following program converts a square to a rectangle using conversion in destination class:

```
#include <iostream>
using namespace std;
class Square
{
	private:
		int side;
	public:
		Square(int s)
		{
			side = s;
		}
		int get_side()
		{
			return side;
		}
};
class Rectangle
{
	private:
		int length;
		int breadth;
	public:
		Rectangle(int l, int b)
		{
			length = l;
			breadth = b;
		}
		Rectangle(Square s)
		{
			length = breadth = s.get_side();
		}
		void display()
		{
			cout<<"length = "<<length<<", breadth = "<<breadth;
		}
};
int main()
{
	Rectangle rect(20, 30);
	Square sq(30);
	rect = sq;
	rect.display();
	return 0;
}
```

Output of the above program is as follows:

```
length = 30, breadth = 30
```