Unit-IV

Operator Overloading and Type Conversions

INTRODUCTION

Operator overloading is one of the many exciting features of C++ language. C++ permits us to add two variables of user-defined types with the same syntax that is 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. Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can overload all the C++ operators except

- Class member access operators
- Scope resolution operator
- Size operator
- Conditional operator

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 describes the task. The general form of an operator function is

```
return   type  classname ::   operator   (op-arglist)
{
      Function body
 }
```

Where return type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator. operator op is the function name. Operator functions must be either member functions (non-static) or friend function.

OVERLOADING UNARY AND BINARY OPERATORS

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one opearand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items. The same mechanism can be used to overload a binary operator. The complex statement

```
C = sum(A, B);
```

Was previously used to add two complex numbers using a friend function. The above functional notation can be replaced by a natural expression

C = A + B;

by overloading the + operator using an operator +( ) function.

OVERLOADING BINARY OPERATORS USING FRIENDS

friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitily passed to it, while amember function requires only one.

RULES FOR OVERLOADING OPERATORS

There are certain rstrictions and limitations in overloading the operators. Some of them are listed below.

1. Only exsisting operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user defined type.
3. We cannot change the basic meaning of an operator.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded.
6. We cannot use friend functions to overload certain operators. However member functions can be used to overload them.

TYPE CONVERSIONS

We know that when constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operands as per certain rules. Similarly, an assignment operation also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of vasriable on the left.

The three types of situations in the data conversion are

1. Conversion from basic type to class type.
2. Conversion from class type to basic type.
3. Conversion from one class type to another class type.

DATA CONVERSION EXAMPLE

```
#include <iostream.h>
#include <conio.h>

class invent1
{
 int code;
 int items;
 float price;

  public:

  invent1()
  {}

  invent1(int a,int b,int c)
  {
    code=a;
    items=b;
    price=c;
  }

  void display()
  {
    cout<<"\nCode  : "<<code;
    cout<<"\nItems : "<<items;
    cout<<"\nPrice : "<<price;
  }

  int getcode()
  {return code;}

  int getitem()
  {return items;}

  int getprice()
  {return price;}

};


class invent2
{
  int code;
  floatvalue;

   public:

    invent2()
    {
      code=0;
      value=0;
    }

    invent2(int x,float y)
    {
      code=x;
      value=y;
    }

    void display()
    {
     cout<<"Code  : "<<code<<endl;
     cout<<"Value : "<<value<<endl;
    }

    invent2(invent1 p)
    {
     code=p.getcode();
     value=p.getitem()*p.getprice();
    }
```

```
};

void main()
{
 clrscr();
 invent1 s1(100,5,140);
 invent2 d1;

 d1=s1;  //Invoke Constructor in Invent2 for conversion

 cout<<"\nProduct details - Invent1 type";
 s1.display();

 cout<<"\n\n\nProduct details - Invent2 type\n";
 d1.display();
 getch();
}
```

The compiler does not support automatic type conversions for the user defined data types. We can use casting operator functions to achieve this. 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.