11.4 IMPLEMENTING OPERATOR OVERLOADING

Operator overloading is usually implemented in two ways as follows:

- Through member function
- Through friend function

Programming Tip: Operator overloading cannot be used to define new operators.

Although operators can be easily overloaded using any of these techniques, the choice of technique is just a matter of programmer's convenience. However, you must consider the major difference when overloading using a member and/or a friend function as shown in Table 11.2.

Table 11.2 Differences between operator overloading using member function and friend function

Member function	Friend function
• Number of explicit parameters is reduced by one, as the calling object is implicitly supplied as an operand. • Unary operators take no explicit parameters. • Binary operators take one explicit parameter. • Left-hand operand has to be the calling object. • `obj2 = obj1 + 10;` is permissible but `obj2 = 10 + obj1;` is not permissible.	• Number of explicit parameters is more. • Unary operators take only one parameter. • Binary operators take two parameters. • Left-hand operand need not be an object of the class. • `obj2 = obj1 + 10;` as well as `obj2 = 10 + obj1;` is permissible.

11.5 OVERLOADING UNARY OPERATORS

Unary operators work only on a single operand. Some examples of unary operators are as follows:

- Increment operator (`++`)
- Decrement operator (`--`)
- Unary minus operator (`-`)
- Logical not operator (`!`)

As stated earlier, unary operators can be overloaded using a friend function or a member function. In both the cases, the unary operator operates on the object for which it was called. Usually, the operator is specified on the left side of the object, as in `++obj`, `-obj`, and `!obj`. However, the operator may appear on the right side of the object when it is a post-fix increment or a post-fix decrement such as `obj++` or `obj--`.

11.5.1 Using a Member Function to Overload a Unary Operator

The syntax for operator overloading using a member function can be given as

```
return_type operator op()
```

where `return_type` is the return type and *op* is the operator to be overloaded. Let us consider a small program that overloads the unary operator using member function. We know that if we have a number say, `7`, then the `unary` operator when applied to it will make it negative, which is `-7`. However, if we apply the same operator to a number say `-10`, then the result would be `10` (positive).

Example 11.1 Overloading `unary` operator with member function

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

OUTPUT

```
x = 10
```

11.5.3 Returning a Nameless Object

The function code operator overloading could be made simpler and shorter by returning a nameless object. This could be done by writing

Programming Tip: Binary operator will be applied on two operands.

```
Number operator -()
{       x = -x;
        return Number(x);
}
```

Note There is no restriction on the return types of the unary operators.

11.5.4 Using a Friend Function to Overload a Unary Operator

When a unary operator is overloaded using a friend function, then you must ensure the following:

- The function will take one operand as an argument.
- This operand will be an object of the class.
- The function will use the private members of the class only with the object name.
- The function may take the object by using value or by reference. However, if the object is passed using value, then any changes made to the data members of the object in the function will not be reflected back in the calling function. Therefore, if you want the changes to persist, pass the object by reference.
- The function may or may not return any value. It depends on the usage. Since we had to assign the value to another object we have returned.
- The friend function does not have access to the `this` pointer.

The syntax for operator overloading using a member function can be given as

```
friend return_type operator op (class_name object)
```

where `return_type` is the return type, and `op` the operator to be overloaded and object is an instance of the class on which the operator has to be applied. The program given here performs the same operation but using a friend function.

Example 11.3 Friend function for operator overloading

```
using namespace std;
#include<iostream>
class Number
{  private:
      int x;
   public:
      Number()
      {     x = 0;       };
      Number(int n)
      {     x = n;       }
      void show_data()
      {     cout<<"\n x = "<<x;       }
      friend Number & operator-(Number &);      //friend function declared
```

11.6 OVERLOADING BINARY OPERATORS

Binary means two and binary operators mean operators that work with two opearnds. Like unary operators, binary operators such as +, -, *, /, =, <. >, !, %, ^, &&, ||, <<, and >> can also be overloaded. Binary operators are also overloaded either using member functions or friend functions. While the member function will be invoked using an object of the class and will accept one argument, the friend function, on the other hand, will accept two arguments.

The syntax of overloading a binary operator using a member function can be given as

```
return_type operator op(arg)
```

Here, `operator` is the keyword, `op` is the operator to be overloaded, and `arg` is the argument of any type that will be used in the function. While the first operand or the object which invokes the function, is taken implicitly, the other operand on the other hand, is passed explicitly. Therefore, the first operand's data members can be accessed directly without using the dot operator but the second object's data members must be accessed using the dot operator. For example, if `c1` and `c2` are objects of complex type, then overloaded + operator can be called `c3 = c1 + c2;` this is equivalent to `c3 = c1.operator+(c2);`

Note Binary operators must explicitly return a value. They might not attempt to change the original values of the arguments.

Similarly, the syntax to overload a binary operator using a friend function is

```
friendreturn_type operator op(arg1, arg2)
```

Here, `arg1` and `arg2` are arguments of any type. However, one of them must compulsorily be of the class type. When the binary + operator is overloaded in class complex using friend function, then it will be invoked as `c3 = c1 + c2;` which is equivalent to writing `c3 = operator+(c1, c2);`

Note There is no restriction on the return type of a binary operator overloaded function.

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

Program 11.8 Write a program to add two arrays using classes and operator overloading.

Programming Tip: Operator overloaded function can be invoked only by an object of the class.

```
          using namespace std;
          #include<iostream>
          class Array
          {     private:
                      int arr[10];
                      int size;
                public:
                      Array();
                      Array(int);
           void show_data();
           Array operator+(Array &);
};
Array :: Array()
{     for(int i=0;i<10;i++)
           arr[i] = 0;
      size = 0;
}
Array :: Array(int n)
{     size = n;
```

```
        {     cout<<str;     }
        String & operator+=(String &s);
        int operator==(String &s);
        int operator>(String &s);
};
String & String :: operator+=(String &S)
{   len += strlen(S.str) + 1;
    strcat(str, S.str);
    return *this;
}
int String :: operator==(String &S)
{   if(strcmp(str, S.str)==0)
        return 1;
    else return 0;
}
int String :: operator>(String &S)
{   if(strcmp(str, S.str)>0)
        return 1;
    else return 0;
}
main()
{   char s[10];
    String S1("Hello");
    cout<<"\n Enter a string : ";
    cin>>s;
    String S2(s);
    if(S1==S2)
        cout<<"\n EQUAL STRINGS";
    if(S1>S2)
        cout<<"\n FIRST STRING IS GREATER THAN SECOND";
    else
        cout<<"\n SECOND STRING IS GREATER THAN FIRST";
    S1 += S2;
    cout<<"\n AFTER CONCATENATION : ";
    S1.show_data();
}
```

OUTPUT

```
Enter a string : World
SECOND STRING IS GREATER THAN FIRST
AFTER CONCATENATION: HelloWorld
```

11.7 OVERLOADING SPECIAL OPERATORS

In this section, we will discuss about overloading some special operators such as `<<`, `>>`, `[]`, `()`, `->`, `new`, and `delete`. Before we discuss overloading of these operators, let us quickly revise their functions.

`new`—to dynamically allocate memory
`delete`—to free the memory allocated dynamically
`<<`—to display a message
`>>`—to accept input from users
`[]` and `()`—are subscript operators
`->`—to access a class member

11.7.1 Overloading New and Delete Operators

C++ allows programmers to overload the new and delete operators because of the following reasons:

- To allow users to allocate memory in a customized way.
- To allow users to debug the program and keep track of memory allocation and deallocation in their programs.
- To allow users to perform additional operations while allocating or deallocating memory.

The syntax for overloading the new operator can be given as follows:

```
void* operator new(size_t size);
```

The overloaded new operator receives a parameter size of type size_t, which specifies the number of bytes of memory to be allocated.

The return type of the overloaded new must be void*. The overloaded function returns a pointer to the beginning of the block of memory allocated.

Similarly, the syntax for the overloaded delete operator can be given as

```
void operator delete(void*);
```

The function receives a parameter ptr of type void* which has to be deleted. Note that the function should not return anything.

Note that both of these overloaded new and delete operator functions are static members by default. Therefore, they do not have access to the this pointer.

Note To delete an array of objects, the operator delete[] must be overloaded.

The program code given here demonstrates the use of overloaded new and delete operators.

Example 11.6 Overloading new and delete operators

```
using namespace std;
#include<iostream>
class Array
{  private:
       int *arr;
   public:
       void * operator new(size_t size)
       {   void *parr = ::new int[size];  // dynamicallyallocatingspaceusingoverloadednewoperator
           return parr;
           }
           void operator delete(void *parr)
           {    ::delete parr;   }  // dynamicallyde-allocatingspaceusingoverloadeddeleteoperator
             void get_data();
             void show_data();
};
void Array :: get_data()
{   cout<<"\n Enter the elements : ";
   for(int i=0;i<5;i++)
        cin>>arr[i];
}
void Array :: show_data()
{   cout<<"\n The array is : ";
for(int i=0;i<5;i++)
        cout<<" "<<arr[i];

}
main()
{   Array *A = new Array;   // calls the overloaded new operator
    A->get_data();
    A->show_data();
    delete A;              // calls the overloaded delete operator

}
OUTPUT
Enter the elements : 1 2 3 4 5
The array is : 1 2 3 4 5
```

In the program,`::new` and `::delete` refer to the global `new and delete operators provided` by the C++ library.

Advantages of Overloading

- The overloaded new operator function can accept arguments; therefore, a class can have multiple overloaded new operator functions. This gives the programmer more flexibility in customizing memory allocation for objects. For example,

```
void * operator new(size_t size, char c)
{     void *ptr;
      ptr = malloc(size);
      if(ptr != NULL)
            *ptr = c;
      return ptr;
}
main()
{     char *ch = new('#') char;
}
```

Programming Tip: The return type of the overloaded new must be void*.

This code will not only allocate memory for a single character but will also initialize the allocated memory with the #character.

- The overloaded new operator also enables programmers to squeeze some extra performance out of their programs. For example, in a class, to speed up the allocation of new nodes, a list of deleted nodes is maintained so that their memory can be reused when new nodes are allocated. In this case, the overloaded `delete` operator will add nodes to the list of deleted nodes and the overloaded new operator will allocate memory from this list rather than from the heap to speedup memory allocation. The global `new` operator can be used only when the list of deleted nodes is empty.
- Overloaded `new` or `delete` operators also provide garbage collection for class's objects.
- Programmers can add exception handling routine in the overloaded `new` operator function.
- Programmers can use C++ memory allocation functions such as `malloc()` and `realloc()` to allocate and re-allocate memory dynamically. For example, the function codes given here allocate, re-allocate, and free memory dynamically.

```
void * operator new(size_t size)
{     void *ptr = malloc(size);
      if(ptr == NULL)
```

```
        {     cout<<"\n Memory could not be allocated";
                exit(1);
        }
        else return *ptr;
}
void * realloc(void * ptr, size_t size);
void operator delete(void * ptr)
{       free(ptr);
}
```

The `realloc()` is used to change the size of the allocated block at address `ptr` with the size. The address pointed to by `ptr` may change if the block is shifted to another location in memory. This can happen when size bytes are not available in the previous allocated space. For example, if `10` bytes were allocated and now the user has called `realloc()` to allocate `20` bytes, then the address pointed by `ptr` may change to point at the address where `20` bytes are available contiguously. The `realloc()` returns the new value of `ptr`. However, if `realloc()` fails, it returns `NULL` and does not free the original memory. Therefore, when using `realloc()`, you must save the previous pointer value.

Note While a user can have any number of overloaded new operators, there can be only one overloaded `delete` operator. `delete` is different from `delete[]`.

11.7.2 Overloading Subscript Operators [ ] and ( )

The subscript operator—`[]`—is used to access array elements and can be overloaded to enhance its functionality with classes. The syntax of overloading the subscript operator can be given as follows:

```
Identifier[expression]
```

where `identifier` is the object of the class. The syntax is interpreted as

```
identifier.operator[](expression)
```

From the syntax, it is clear that the subscript operator is a binary operator in which the first operator is an object of the class and the second operand is an integer index.

Note The overloaded operator [ ] ( ) must be defined as a non-static member function of a class.

Example 11.7 Overloading `[]` operator

```
using namespace std;
#include<iostream>
class Array
{     private:
            int arr[10];
      public:
            Array();
            void get_data();
            void show_data();
            int & operator[](int i);
};
Array :: Array()
{     for(int i=0;i<10;i++)
```

```
        arr[i] = 0;
}
void Array :: get_data()
{
    cout<<"\n Enter the array elements : ";
    for(int i=0;i<10;i++)
        cin>>arr[i];
}
void Array :: show_data()
{
    cout<<"\n The Array is : ";
    for(int i=0;i<10;i++)
        cout<<" "<<arr[i];
}
int& Array :: operator[](int i)
{
    return arr[i];      //returns value at specified index
}
main()
{
    Array A;   //create object
    A.get_data();    //invokes member function
            A.show_data();
            cout<<"\n Modified Array Elements Are : ";
            for(int i=0;i<10;i++)
                cout<<" "<<A[i] * 2; //invokes overloaded [] operator
}
```

Programming Tip: Overloaded new operators speed up the memory allocation process.

OUTPUT

```
Enter the array elements : 1 2 3 4 5 6 7 8 9 10
The Array is : 1 2 3 4 5 6 7 8 9 10
Modified Array Elements Are : 2 4 6 8 10 12 14 16 18 20
```

The overloaded subscript operator given here is a very simple one that returns the element at the specified index. You can enhance the function code to check for array bounds by writing,

```
int & Array :: operator[](int i)
{    if(i<0 || i>9)   //checking for valid indexes
    {    cout<<"\n Array index out of bounds";
         exit(1);
    }
    else
         return arr[i];
}
```

You can have another version of the overloaded subscript operator function as,

```
const int & Array :: operator[](int i) const
{    return arr[i];
}
```

The const version of the subscript operator is called when its object itself is const.

Note The overloaded subscript operator function must return a value by reference.

Overloading Subscript Operator () rather than []

When there are multiple subscripts, it is better to overload the `operator()` rather than `operator[]`. This is because the `operator[]` always takes exactly one parameter, but `operator()` can take any number of parameters. Let us consider a small piece of code to learn how `operator()` is overloaded for a two-dimensional matrix.

Example 11.8 Overloading`()` for a two-dimensional matrix

```
using namespace std;
#include<iostream>
#include<stdlib.h>
class Matrix
{     private:
          int arr[2][2];
      public:
          Matrix();
          void get_data();
          void show_data();
          int& operator()(int i, int j); //()operator overloaded function declaration
};
Matrix :: Matrix()
{     for(int i=0;i<2;i++)
          for(int j=0;j<2;j++)
             arr[i][j] = 0;
}
void Matrix :: get_data()
{     cout<<"\n Enter the Matrix elements : ";
      for(int i=0;i<2;i++)
          for(int j=0;j<2;j++)
               cin>>arr[i][j];
}
void Matrix :: show_data()
{     cout<<"\n The Matrix is : ";
      for(int i=0;i<2;i++)
      {    cout<<"\n";
            for(int j=0;j<2;j++)
             cout<<" "<<arr[i][j];
      }
}
int & Matrix :: operator()(int i, int j) // overloaded () operator
{    if(i<0 || i>9 || j<0 || j>9)
     {           cout<<"\n Matrix index out of bounds";
                 exit(1);
     }
          else
          return arr[i][j];
}
main()
{     Matrix M;
      M.get_data();
```

```
    M.show_data();
    cout<<"\n Modified Matrix Elements Are : ";
    for(int i=0;i<2;i++)
    {      for(int j=0; j<2;j++)
             cout<<" "<<M(i,j)*2;    //invoking overloaded operator function
    }
}
OUTPUT
Enter the Matrix elements : 1 2 3 4
The Matrix is :
1 2
3 4
Modified Matrix Elements Are : 2 4 6 8
```

11.7.3 Overloading Class Member Access Operator (->)

C++ allows programmers to control class member access by overloading the member access operator (->). The -> operator is a unary operator as it takes only one operand, that is the object of the class. The syntax for overloading the -> operator can be given as,

```
class_ptr *operator->()
```

where `class_ptr` is a pointer of class type. Before overloading the class member access operator, the overloaded -> operator function must be a non-static member function of the class. The program given here demonstrates the functionality of overloaded member access operator.

Example 11.9 Overloading -> operator

```
using namespace std;
#include<iostream>
class Sample
{    public:
        int num;
        Sample(int n)
        {   num = n;  }
        Sample * operator->(void)    //overloaded operator ->function
        {   return this;    }
};
main()
{    Sample *ptr = new Sample(10);
     cout<<"\n Number = "<<ptr->num;    // using normal object pointer
     Sample S(20);
     cout<<"\n Number = "<<S->num;    //using overloaded -> operator
}

OUTPUT
Number = 10
Number = 20
```

Explanation: In the program, `ptr` is an object pointer and S is an object of class sample. The class member access operator has been overloaded to return a pointer to the invoking object. Therefore, the statement, `s->num` is equivalent to writing `this->num`.

11.7.4 Overloading Input and Output Operators

C++ allows input and output of built-in data types using the stream extraction operator >> and the stream insertion operator <<. The capabilities of these operators can be extended to perform input and output for user-defined types. However, before overloading the extraction and insertion operators, you must know the following aspects:

- `cin` and `cout` are defined in class `iostream`.
- While `cin` is an object of `istream` class, `cout` is an object of the `ostream` class. We will read more on the relationship between `istream, ostream, iostream, cin,` and `cout` later in chapter File Handling.
- The insertion and extraction of operators will be overloaded by using a friend function because we need to call these functions without any object.
- The insertion and extraction operators must return the value of the left operand—the `ostream` or `istream` object—so that multiple << or >> operators may be used in the same statement. The syntax for overloading the >> function can be given as

Programming Tip: () operator is overloaded when there are multiple subscript values.

```
friend ostream & operator << (ostream & output , My_class&obj)
{     // code
}
```

Example 11.10 Overloading << and >> operators

```
using namespace std;
#include<iostream>
class Date
{     private:
            int dd, mm, yy;
      public:
           friend istream & operator >> (istream & input , Date &D)
           {     input>>D.dd>>D.mm>>D.yy;
                 return input;
           }
           friend ostream & operator << (ostream & output , Date &D)
           {     cout<<D.dd<<" - "<<D.mm<<" - "<<D.yy;
                 return output;
           }
};
main()
{     Date D;
      cout<<"\n Enter the Date : ";
      cin>>D;   //overloaded >> is invoked
      cout<<"\n DATE : ";
      cout<<D;   //overloaded << is invoked
}
```

OUTPUT

```
Enter the Date : 17 2 2007
DATE : 17 - 2 - 2007
```

Explanation: The input and output of the program shows that a class object can be input and output in the same way as is done for a variable of basic data type.

11.8 TYPE CONVERSIONS

In Chapter 2, we have seen that when constants and variables of different basic data types are mixed together in a single expression, automatic type conversions takes place. Since C++ treats user-defined data types same as built-in types, it should also allow expressions consisting of mixed user-defined and basic data types. However, the main problem in user-defined types is that the compiler has no idea about the user-defined data types and about their conversion to other data types.

Therefore, to overcome this problem, programmers must specifically design routines that convert basic data types to user-defined data types or vice versa. With regard to data conversion, there can be three possibilities as given in Fig. 11.1. These are as follows:

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

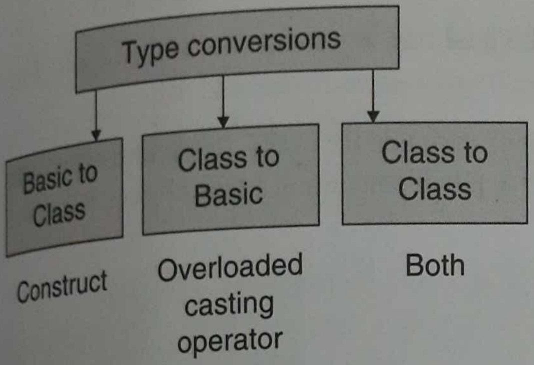

Figure 11.1 Type conversions

11.8.1 Conversion from Basic to Class Type

A variable of basic data type is converted into a data member of a user-defined class with the help of a constructor function defined in the class as had been the practice so far. This is given in Fig. 11.2. For example, when we used to write

```
String(char *s)
{     strcpy(str, s);
}
```

Programming Tip: Overloaded -> operator function must be a non-static member function of the class.

We are building the string object from the basic data type of char.

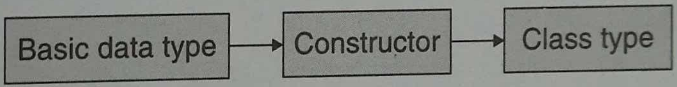

Figure 11.2 Conversion from basic to class type

From the `main()`, we invoke the constructor by writing,

```
String S1 = name; or
String S1(name);
```

In both the cases, the object of the class must appear on the left.

For example, String S1 = “Hello”;

Or, String S1(“Hello”);

11.8.2 Conversion from Class to Basic Data Type

A user-defined class type can be converted into a basic type using the overloaded casting operator as shown in Fig. 11.3.

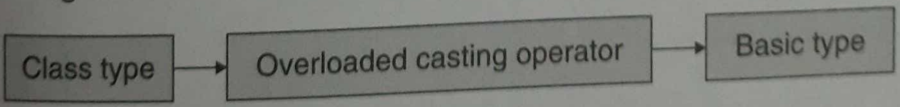

Figure 11.3 Conversion from Class type to basic data type

The syntax of an overloaded casting operator can be given as follows:

```
operator type_name()
{     -----
```

```
        FUNCTION BODY
        ------
}
```

While defining the overloaded casting operator, the following aspects must be adhered to:

- The function can be declared and/or defined within the class.
- The function converts the class type into the `type_name`. For example, operator `int()` will convert a class type to `int`.
- The function does not have any return type.
- The function does not take any argument.
- The casting operator is a unary operator that takes just one argument that is the object of the class that invokes this function.

Let us consider a small program that converts minutes `(int)` into time (of `class Type`) consisting of data members in `hrs` and `mins`. The program also does the reverse job of converting a `Time` object into `minutes`.

Example 11.11 Converting basic data type to class type

```
using namespace std;
#include<iostream>
class Time
{     private:
             int h, m;
    public:
          Time()
          {   h = m = 0;   }
          Time(int t)   // converting int to class
          {   h = t / 60;
              m = t % 60;
          }
          void get_data()
          {   cin>>h>>m;  }
          void show_data()
          {   cout<<h<<" hrs"<<m<<" mins";    }
          operator int()     //converting class to int
          {   int t = h*60 + m;
              return t;
          }
};
main()
{    int min;
     cout<<"\n Enter the minutes : ";
     cin>>min;
     Time T1;
     T1 = min;           //constructor called
     T1.show_data();
     cout<<"\n Enter the number of hrs and mins : ";
     T1.get_data();
     min = T1;     //casting operator called
```