```
    complex operator - ( complex c2 );
    complex operator * ( complex c2 );
    complex operator / ( complex c2 );
};
// addition of complex numbers, c3 = c1 + c2
complex complex::operator + ( complex c2 )
{
  complex temp;
  temp.real = real + c2.real;
  temp.imag = imag + c2.imag;
  return( temp );
}
// subtraction of complex numbers, c3 = c1 - c2;
complex complex::operator - ( complex c2 )
{
  complex temp;
  temp.real = real - c2.real;
  temp.imag = imag - c2.imag;
  return( temp );
}
// Multiplication of complex numbers, c3 = c1 * c2
complex complex::operator * ( complex c2 )
{
  complex temp;
  temp.real = real * c2.real - imag * c2.imag;
  temp.imag = real * c2.imag  + imag * c2.real;
  return( temp );
}
// Division of complex numbers, c3 = c1 / c2
complex complex::operator / ( complex c2 )
{
  complex temp;
  float qt;
  qt = c2.real*c2.real+c2.imag*c2.imag;
  temp.real = (real * c2.real + imag * c2.imag)/qt;
  temp.imag = (imag * c2.real- real * c2.imag) /qt;
  return( temp );
}
void main()
{
  complex c1, c2, c3;
  // read complex numbers c1 and c2
  cout << "Enter Complex Number c1 .." << endl;
  c1.getdata();
  cout << "Enter Complex Number c2 .." << endl;
  c2.getdata();
  cout << "Entered Complex Numbers are...";
  c1.outdata( "c1 = " );
  c2.outdata( "c2 = " );
  cout << endl << "Computational results are...";
  c3 = c1 + c2;
```

```
    c3.outdata("c3 = c1 + c2: ");
    c3 = c1 - c2;
    c3.outdata("c3 = c1 - c2: ");
    c3 = c1 * c2;
    c3.outdata("c3 = c1 * c2: ");
    c3 = c1 / c2;
    c3.outdata("c3 = c1 / c2: ");
    c3 = c1 + c2 + c1.+ c2;
    c3.outdata( "c3 = c1 + c2 + c1 + c2: " );
    c3 = c1 * c2 + c1 / c2;
    c3.outdata( "c3 = c1 * c2 + c1 / c2: ");
}
```

Run

```
Enter Complex Number c1 ..
Real Part ? 2.5
Imag Part ? 2.0
Enter Complex Number c2 ..
Real Part ? 3.0
Imag Part ? 1.5
Entered Complex Numbers are...
c1 = (2.5, 2)
c2 = (3, 1.5)
Computational results are...
c3 = c1 + c2: (5.5, 3.5)
c3 = c1 - c2: (-0.5, 0.5)
c3 = c1 * c2: (4.5, 9.75)
c3 = c1 / c2: (0.933333, 0.2)
c3 = c1 + c2 + c1 + c2: (11, 7)
c3 = c1 * c2 + c1 / c2: (5.43333, 9.95)
```

In `main()`, the statement,

```
c3 = c1 + c2 + c1 + c2;
```

is evaluated as

```
((c1.operator+(c2)).operator+(c1)).operator+(c2);
```

from left to right, since all the operators have the same precedence. However, the statement

```
c3 = c1 * c2 + c1 / c3;
```

is evaluated as

```
(c1.operator*(c2)).operator+(c1.operator/(c2))
```

Operators with higher precedence are evaluated first, followed by those with lower precedence.

13.10 Concatenation of Strings

Normally, concatenation of strings is performed by using the library function `strcat()` explicitly. To illustrate this concept, consider the strings `str1` and `str2` which are defined as follows:

```
char str1[50] = "Welcome to ";
char str2[25] = "Operator Overloading";
```

The strings `str1` and `str2` are combined, and the result is stored in `str1` by invoking the function

`strcat()` as follows:

```
strcat( str1, str2 );
```

On execution `str2` remains unchanged. In C++, such operations can also be performed by defining a string class and overloading the + operator. A statement such as,

```
str1 = str1 + str2;
```

for concatenation of string, (where `str1` and `str2` are the objects of a class `string`) would be perfectly valid. The program `string.cpp` defines a `string` class and uses it to concatenate strings.

```
// string.cpp: Concatenation of strings
#include <iostream.h>
#include <string.h>
const int BUFF_SIZE = 50; // length of string
class string               // user defined string class
{
  private:
    char str[BUFF_SIZE];
  public:
    string()               // constructor1 without arguments
    {
      strcpy( str, "" );
    }
    string( char *MyStr )   // constructor2, one argument
    {
      strcpy( str, MyStr );   // MyStr is copied to str
    }
    void echo()    // display string
    {
      cout << str;
    }
    string operator +( string s ) // overloading + operator
    {
      string temp = str;   // creates object and strcpy( temp.str, str );
      strcat( temp.str, s.str ); //  temp.str = temp.str + s.str
      return temp;   // return string object temp
    }
};
void main()
{
  string str1 = "Welcome to ";          // uses constructor2
  string str2 = "Operator Overloading"; // uses constructor2
  string str3;  // uses constructor1, str3.str = NULL
  // display strings of str1, str2, and str3
  cout << "\nBefore str3 = str1 + str2; ..";
  cout << "\nstr1 = ";
  str1.echo();
  cout << "\nstr2 = ";
  str2.echo();
  cout << "\nstr3 = ";
  str3.echo();
  str3 = str1 + str2;  // str1 invokes its operator + function with str2
```

```
    // display strings of str1, str2, and str3
    cout << "\nAfter str3 = str1 + str2; ..";
    cout << "\nstr1 = ";
    str1.echo();
    cout << "\nstr2 = ";
    str2.echo();
    cout << "\nstr3 = ";
    str3.echo();
}
```

Run

```
Before str3 = str1 + str2; ..
str1 = Welcome to
str2 = Operator Overloading
str3 =
After str3 = str1 + str2; ..
str1 = Welcome to
str2 = Operator Overloading
str3 = Welcome to Operator Overloading
```

The prototype of the string concatenation operator function

```
string operator +( string s ) // overloading + operator
```

indicates that the + operator takes one argument of type `string` object and returns an object of the same type. The concatenation is performed by creating a temporary `string` object `temp` and initializing it with the first string. The second string is added to first string in the object `temp` using the `strcat()` and finally the resultant temporary string object `temp` is returned. In this case, the length of `str1` plus `str2` should not exceed `BUFF_SIZE`. If it exceeds, then the behavior of the program may be unpredictable. It can be overcome by testing the length of `str1` plus `str2` before concatenating them in the operator `+()` function of the `string` class and then taking appropriate actions.

13.11 Comparison Operators

Similar to arithmetic operators, the relational operators can be overloaded for comparing the magnitudes of the operands. The relational operators can also operate on the user defined data-types similar to the way they operate on primitive data-types. The program `idxcmp.cpp` demonstrates the overloading of the comparison operator `<` to compare indexes.

```
// idxcmp.cpp: Index comparison with overloading of < operator
#include <iostream.h>
enum boolean { FALSE, TRUE };
class Index
{
    private:
        int value;              // Index Value
    public:
        Index()                 // No argument constructor
        {
            value = 0;
        }
```

```
    Index( int val )          // Constructor with one argument
    {
       value = val;
    }
    int GetIndex()             // Index Access
    {
       return value;
    }
    boolean operator < ( Index idx )   //compare indexes
    {
       return( value < idx.value ? TRUE : FALSE );
    }
};
void main()
{
   Index idx1 = 5;
   Index idx2 = 10;
   cout << "\nIndex1 = " << idx1.GetIndex();
   cout << "\nIndex2 = " << idx2.GetIndex();
   if( idx1 < idx2 )
      cout << "\nIndex1 is less than Index2";
   else
      cout << "\nIndex1 is not less than Index2";
}
```

Run

```
Index1 = 5
Index2 = 10
Index1 is less than Index2
```

The concept of overloading the comparison operator < in the above program is similar to overloading arithmetic operators. The operator function `<()` returns `TRUE` or `FALSE` depending on the magnitudes of the Index operands.

Strings Comparison

The relational operators such as <, >, ==, etc., can be overloaded to operate on strings. These operators return `TRUE` or `FALSE` depending on the contents of the string arguments. The program `strcmp.cpp` illustrates the overloading of relational operators in a `string` class.

```
// strcmp.cpp: Comparison of strings
#include <iostream.h>
#include <string.h>
const int BUFF_SIZE = 50;    // length of string
enum boolean { FALSE, TRUE };
class string                 // user defined string class
{
   private:
      char str[BUFF_SIZE];
   public:
```

```
    string()                // constructor without arguments
    {
       strcpy( str, "" );
    }
    void read()             // read string
    {
       cin >> str;
       //  cout << str;
    }
    void echo()    // display string
    {
       cout << str;
    }
    boolean operator < ( string s ) // overloading < operator
    {
       if( strcmp( str, s.str ) < 0 )
          return TRUE;  // str < s.str in lexicographical order
       else
         return FALSE;
    }
    boolean operator > ( string s ) // overloading > operator
    {
       if( strcmp( str, s.str ) > 0 )
          return TRUE;    // str > s.str in lexicographical order
       else
          return FALSE;
    }
    boolean operator == ( char *MyStr ) // overloading == operator
    {
       if( strcmp( str, MyStr ) == 0 )
          return TRUE;   // str and MyStr are same
       else
          return FALSE;
    }
};
void main()
{
   string str1, str2;  // uses constructor 1
   while( TRUE )
   {
      cout << "\nEnter String1 <'end' to stop>: ";
      str1.read();
      if( str1 == "end" )
          break;
      cout << "Enter String2: ";
      str2.read();
      cout << "Comparison Status: ";
      // display comparison status
      // display format: String1 "comparison status <, >, = " String2
      str1.echo();
```

```
        if( str1 < str2 )
            cout << " < ";
        else
            if( str1 > str2 )
                cout << " > ";
            else
                cout << " = ";
        str2.echo();
    }
    cout << "\nBye.!! That's all folks.!";
}
```

Run

```
Enter String1 <'end' to stop>: C
Enter String2: C++
Comparison Status: C < C++
Enter String1 <'end' to stop>: Rajkumar
Enter String2: Bindu
Comparison Status: Rajkumar > Bindu
Enter String1 <'end' to stop>: Rajkumar
Enter String2: Venugopal
Comparison Status: Rajkumar < Venugopal
Enter String1 <'end' to stop>: HELLO
Enter String2: HELLO
Comparison Status: HELLO = HELLO
Enter String1 <'end' to stop>: end
Bye.!! That's all folks.!
```

The overloaded operator functions of the class `string` uses the library function `strcmp()` to compare the two strings. The `strcmp(..)` operates as follows:

- It returns 0 if both the strings are equal
- It returns a negative value if the first string is less than the second one
- It returns a positive value if the first string is greater than the second one

The terms *less than*, *greater than*, or *equal to* are used in lexicographic sense to indicate whether the first string appears before or after the second in the alphabetical order.

The prototype of string comparison function

```
boolean operator == ( char *MyStr )
```

indicates that the == operator takes one argument of type pointer to character and returns `TRUE` or `FALSE` depending on the operands weightage in lexicographical order. The `strcmp()` in the function body compares the object's attribute `str` with the argument `MyStr`. From this example, it is understood that the arguments to an overloaded operator need not be of the same data-type, but the overloaded operator must be a *member function of the first object*.

13.12 Arithmetic Assignment Operators

Like arithmetic operators, *arithmetic assignment* operators can also be overloaded to perform an arithmetic operation followed by an assignment operation. Such statements are useful in replacing the expressions involving operations on two operands and storing the result in the first operand. For

instance, a statement such as

```
c1 = c1 + c2;
```

can be replaced by

```
c1 += c2;
```

The program `complex4.cpp` illustrates the overloading of arithmetic assignment operators to manipulate complex numbers.

```
// complex4.cpp: Overloading of +=, -=, *=, /= operators for complex class
#include <iostream.h>
class complex
{
   private:
      float real;
      float imag;
   public:
      complex()              // constructor1
      {
         real = imag = 0;
      }
      void getdata()         // read complex number
      {
         cout << "Real Part ? ";
         cin >> real;
         cout << "Imag Part ? ";
         cin >> imag;
      }
      void outdata( char *msg )     // display complex number
      {
         cout << endl << msg;
         cout << "(" << real;
         cout << ", " << imag << ")";
      }
      void operator += ( complex c2 );
      void operator -= ( complex c2 );
      void operator *= ( complex c2 );
      void operator /= ( complex c2 );
};
// addition of complex numbers, c1 += c2 instead of c1 = c1 + c2;
void complex::operator += ( complex c2 )
{
   real = real + c2.real;
   imag = imag + c2.imag;
}
// subtraction of complex numbers, c1 -= c2, i.e., c1 = c1 - c2;
void complex::operator -= ( complex c2 )
{
   real = real - c2.real;
   imag = imag - c2.imag;
}
```

```
// Multiplication of complex numbers, c1 *= c2, instead of c1 = c1*c2
void complex::operator *= ( complex c2 )
{
   complex old = *this; // *this is an object of type complex
   real = old.real * c2.real - old.imag * c2.imag;
   imag = old.real * c2.imag + old.imag * c2.real;
}
// Division of complex numbers, c1 /= c2, i.e., c1 = c1 / c2
void complex::operator /= ( complex c2 )
{
   complex old = *this;
   float qt;
   qt = c2.real*c2.real+c2.imag*c2.imag;
   real = (old.real * c2.real + old.imag * c2.imag)/qt;
   imag = (old.imag * c2.real - old.real * c2.imag) /qt;
}
void main()
{
   complex c1, c2, c3;
   // read complex numbers c1 and c2
   cout << "Enter Complex Number c1 .." << endl;
   c1.getdata();
   cout << "Enter Complex Number c2 .." << endl;
   c2.getdata();
   cout << "Entered Complex Numbers are...";
   c1.outdata( "c1 = " );
   c2.outdata( "c2 = " );
   cout << endl << "Computational results are...";
   // c3 = c1 + c2
   c3 = c1;
   c3 += c2;
   c3.outdata("let c3 = c1, c3 += c2: ");
   // c3 = c1 - c2
   c3 = c1;
   c3 -= c2;
   c3.outdata("let c3 = c1, c3 -= c2: ");
   // c3 = c1 * c2
   c3 = c1;
   c3 *= c2;
   c3.outdata("let c3 = c1, c3 *= c2: ");
   // c3 = c1 / c2
   c3 = c1;
   c3 /= c2;
   c3.outdata("let c3 = c1, c3 /= c2: ");
}
```

Run

```
Enter Complex Number c1 ..
Real Part ? 2.5
Imag Part ? 2.0
Enter Complex Number c2 ..
Real Part ? 3.0
Imag Part ? 1.5
Entered Complex Numbers are...
c1 = (2.5, 2)
c2 = (3, 1.5)
Computational results are...
let c3 = c1, c3 += c2: (5.5, 3.5)
let c3 = c1, c3 -= c2: (-0.5, 0.5)
let c3 = c1, c3 *= c2: (4.5, 9.75)
let c3 = c1, c3 /= c2: (0.933333, 0.2)
```

Observe the difference between the operator function `+()` defined in the program `complex3.cpp` and operator function `+=()` defined in the program `complex4.cpp`. In the former, a new temporary object of `complex` type must be created and returned by the function, so that the resultant object can be assigned to a third `complex` object, as in the statement

```
c3 = c1 + c2;
```

In the latter, the function operator `+=()` is a member function of the object (destination object's class), which receives the result of computation. Hence, the function `operator +=()` has no return value; it returns `void` type. Normally, the result of the assignment operation is not required. In a statement, such as,

```
c3 += c2;
```

the operator alone is used without bothering about the return value.

The use of the arithmetic assignment operator in a complicated statement such as,

```
c3 = c1 += c2;
```

requires a return value. Such requirements can be satisfied by having the function operator `+=()`, which terminates with the statement such as

```
return( *this ); or return complex( real, imag );
```

In the first case, the current object is returned and in the latter case, a nameless object is created with initialization and is returned as illustrated in the program `complex5.cpp`.

```
// complex5.cpp: Overloading of += operator for complex expressions
#include <iostream.h>
class complex
{
   private:
      float real;
      float imag;
   public:
      complex()          // no argument constructor
      {
         real = imag = 0.0;
      }
```

```
    void getdata()        // read complex number
    {
      cout << "Real Part ? ";
      cin >> real;
      cout << "Imag Part ? ";
      cin >> imag;
    }
    complex operator + ( complex c2 );  // complex addition
    void outdata( char *msg )     // display complex number
    {
      cout << endl << msg;
      cout << "(" << real;
      cout << ", " << imag << ")";
    }
    complex operator += ( complex c2 );
};
// addition of complex numbers, c1 += c2 instead of c1 = c1 + c2;
// return complex object *this or build temporary object and return
complex complex::operator += ( complex c2 )
{
  real = real + c2.real;
  imag = imag + c2.imag;
  return( *this );  // *this is current object
}
void main()
{
  complex c1, c2, c3;
  cout << "Enter Complex Number c1 .." << endl;
  c1.getdata();
  cout << "Enter Complex Number c2 .." << endl;
  c2.getdata();
  // Performs 1.  c1 += c2 and 2.  c3 = c1
  c3 = c1 += c2; // c1 += c2 is evaluated first, and assigned to c3
  cout << "\nOn execution of c3 = c1 += c2 ..";
  c1.outdata("Complex c1: ");
  c2.outdata("Complex c2: ");
  c3.outdata("Complex c3: ");
}
```

Run

```
Enter Complex Number c1 ..
Real Part ? 2.5
Imag Part ? 2.0
Enter Complex Number c2 ..
Real Part ? 3.0
Imag Part ? 1.5
On execution of c3 = c1 += c2 ..
Complex c1: (5.5, 3.5)
Complex c2: (3, 1.5)
Complex c3: (5.5, 3.5)
```

13.13 Overloading of new and delete Operators

The memory allocation operators new and delete can be overloaded to handle memory resource in a customized way. It allows the programmer to gain full control over the memory resource and to handle resource crunch errors such as *Out of Memory*, within a class. The main reason for overloading these functions is to increase the efficiency of memory management. An application designed to handle memory allocation by itself through overloading can easily detect memory leaks (improper usage). It can also be used to create the illusion of infinite amount of main memory (virtual memory, which exists in effect but not in reality).

The program resource.cpp illustrates the overloading of new and delete operators. The normal call to the new operator, such as

```
ptr = new vector;
```

dynamically creates a vector object and returns a pointer to that object. The overloaded operator function new in the vector class not only creates an object, but also allocates the resource for its internal data members.

```
// resource.cpp: Overloading of new and delete operators
#include <iostream.h>
const int ARRAY_SIZE = 10;
class vector
{
   private:
      int *array;    // array is dynamically allocatable data member
   public:
      // overloading of new operator
      void * operator new( size_t size )
      {
         vector *my_vector;
         my_vector = ::new vector;  // it refers to global new, otherwise
                                // leads to recursive call of vector::new
         my_vector->array = new int[ARRAY_SIZE]; // calls ::new
         return my_vector;
      }
      // overloading of delete operator
      void operator delete( void* vec )
      {
         vector *my_vect;
         my_vect = (vector *) vec;
         delete (int *) my_vect->array;   // calls ::delete
         ::delete vec;     // it refers to global delete, otherwise
                           // leads to recursive call of vector::delete
      }
      void read();
      int sum();
};
void vector::read()
{
   for( int i = 0; i < ARRAY_SIZE; i++ )
   {
      cout << "vector[" << i << "] = ? ";
```

```
      cin >> array[i];
   }
}
int vector::sum()
{
   int sum = 0;
   for( int i = 0; i < ARRAY_SIZE; i++ )
      sum += array[i];
   return sum;
}
void main()
{
   vector *my_vector = new vector;
   cout << "Enter Vector data ..." << endl;
   my_vector->read();
   cout << "Sum of Vector = " << my_vector->sum();
   delete my_vector;
}
```

Run

```
Enter Vector data ...
vector[0] = ? 1
vector[1] = ? 2
vector[2] = ? 3
vector[3] = ? 4
vector[4] = ? 5
vector[5] = ? 6
vector[6] = ? 7
vector[7] = ? 8
vector[8] = ? 9
vector[9] = ? 10
Sum of Vector = 55
```

In `main()`, the statement

```
vector *my_vector = new vector;
```

invokes the overloaded operator member function

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

defined in the class `vector` as

```
void * operator new( size_t size )
{
   vector *my_vector;
   my_vector = ::new vector;  // it refers to global new, otherwise
   // leads to recursive call of vector::new
   my_vector->array = new int[ARRAY_SIZE]; // calls ::new
   return my_vector;
}
```

In the above function, the statement

```
my_vector = ::new vector;  // it refers to global new, otherwise
```

creates an object of the `vector` class. If scope resolution operator is not used, the overloaded opera-

tor function is called recursively leading to stack overflow. Hence, prefixing of the scope resolution operator to the new operator forces to use the standard new operator supported by the language instead of the one defined in the program. The class vector has a data item of type dynamic array, defined by int *array. Another statement in the above function

```
my_vector->array = new int[ARRAY_SIZE]; // calls ::new
```

creates an array and dynamically allocates memory to it.

Similar to the overloaded new operator function, the overloaded delete operator function handles the process of releasing memory that has been allocated during the dynamic object creation by the new operator; it also releases the memory allocated to the internal data-item array through the function call

```
delete my_vector;
```

It invokes the overloaded operator function

```
void operator delete( void* vec )
```

to release the entire memory resource allocated to the my_vector object and its data members.

13.14 Data Conversion

Representing the same data in multiple forms is a common practice in scientific computations. It involves the conversion of data from one form to another, for instance, conversion from radian to degree, polar to rectangular, and vice versa. Implicit invocation of the conversion procedure in C++ is achieved by overloading the assignment operator, =. The assignment operator assigns the contents of a variable, the result of an expression, or a constant, to another variable. For example,

```
var1 = var2;        // var1 and var2 are defined as integer variables
```

assigns the value of var2 to var1 which are of the same data-type. User defined objects of the same class can also be assigned to one another. In a statement such as

```
c3 = c1 + c2;       // c1, c2, and c3 are objects of complex class
```

the result of addition, which is of type complex is assigned to another object c3 of complex class. The assignment of one variable/object to another variable/object, which are of the same data-type is achieved by copying the contents of all member data-items from source object to the destination object. Such operations do not require any conversion procedure for the data-type conversion. In the above expression, the result of (c1+c2) is of the same data-type as that of the destination object c3. Hence, the compiler does not require any special instruction from the user to perform the assignment of objects.

Thus, assignment of data items are handled by the compiler with no effort on the part of the user, whether they are basic or user defined provided both source and destination data items are of the same data-type. In case the data items are of different types, data conversion interface function must be explicitly specified by the user. These include conversions between basic and user-defined types or between the user-defined data items of different types.

13.15 Conversion between Basic Data Types

Consider the statement

```
weight = age; // weight is of float type and age is of integer type
```

where weight is of type float and age is of type integer. Here, the compiler calls a special routine to convert the value of age, which is represented in an integer format, to a floating-point format, so that

it can be assigned to weight. The compiler has several built-in routines for the conversion of basic data types such as char to int, float to double, etc. This feature of the compiler, which performs conversion of data without the user intervention is known as *implicit type conversion*.

The compiler can be instructed explicitly to perform type conversion using the type conversion operators known as *typecast operators*. For instance, to convert int to float, the statement is

```
weight = (float) age;
```

where the keyword float enclosed between braces is the typecast operator. In C++, the above statement can also be expressed in a more readable form as

```
weight = float( age );
```

The *explicit conversion* of float to int uses the same built-in routine as implicit conversion.

13.16 Conversion between Objects and Basic Types

The compiler supports data conversion of only built-in data types supported by the language. The user cannot rely on the compiler to perform conversion from user-defined data types to primitive data types and vice-versa, because the compiler does not know anything about the logical meaning of user defined data types. Therefore, to perform a meaningful conversion, the user must supply the necessary conversion function. In this case, the conversion process can be from basic data types to user-defined data types or from the user-defined data types to basic data types.

The process of conversion between the user-defined type and basic type is illustrated in the program meter.cpp listed below. In this example, the user-defined type is the class Meter, which represents a unit of length in the MKS measurement system. The basic type is float, which is used to represent a unit of length in CGS measurement system.

The conversion between centimeter and meter can be performed by the following relations:

Length in Cms = Length in Meters * 100

Length in Meters = Length in Cms / 100

Where and How the conversion function should exist ?

To convert data from a basic type to a user-defined type, the conversion function should be defined in user-defined object's class in the form of the constructor. This constructor function takes a single argument of basic data-type as shown in Figure 13.7.

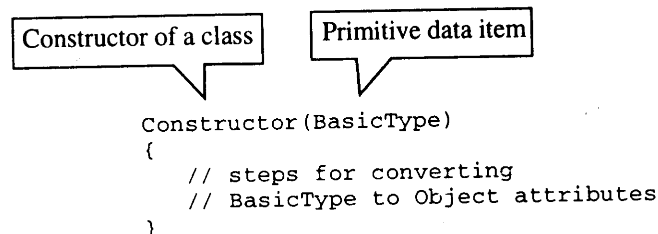

Figure 13.7: Conversion function: basic to user-defined

In the case of conversion from a user-defined type to a basic type, the conversion function should be defined in user-defined object's class in the form of the operator function. The operator function is defined as an overloaded basic data-type which takes no arguments. It converts the data members of an

object to basic data types and returns a basic data-item. The syntax of such a conversion function is shown in Figure 13.8.

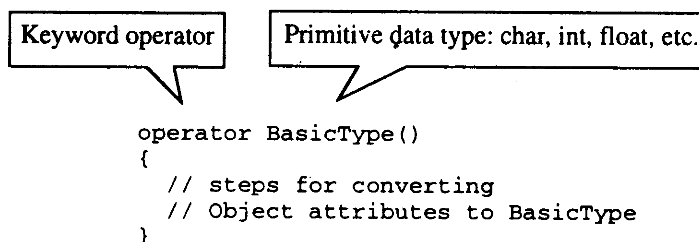

Figure 13.8: Conversion function: user-defined to basic

In the above syntax, it can be observed that the conversion operator function has no return type specification. However, *it should return* `BasicType` *value*. The program `meter.cpp` illustrates the conversion of the `Meter` class's object to `float` representing centimeter and vice-versa.

```
// meter.cpp: Conversion from Meter to Centimeter and vice-versa
#include <iostream.h>
// Meter class for MKS measurement system
class Meter
{
  private:
     float length;              // length in meter
  public:
     Meter()                    // constructor0, no arguments
     {
        length = 0.0;
     }
     // Conversion from Basic data-item to user-defined type
     // InitLength is in centimeter unit
     Meter( float InitLength )     // constructor1, one argument
     {
        length = InitLength / 100.0; // centimeter to meter
     }
     // Conversion from user-defined type to Basic data-item
     // i.e., from meter to centimeter
     operator float()
     {
        float LengthCms;
        LengthCms = length * 100.0; // meter to centimeter
        return( LengthCms );
     }
     void GetLength()
     {
        cout << "\nEnter Length (in meters): ";
        cin >> length;
     }
     void ShowLength()
     {
```

```
        cout << "Length (in meter) = " << length;
    }
};

void main()
{
   // Basic to User-defined conversion demonstration Section
   Meter meter1;      // uses constructor0
   float length1;
   cout << "Enter Length (in cms): ";
   cin >> length1;
   meter1 = length1; // converts basic to user-defined, uses constructor1
   meter1.ShowLength();
   // User-defined to Basic conversion demonstration Section
   Meter meter2;  // uses constructor0
   float length2;
   meter2.GetLength();
   length2 = meter2; //converts user-defined to basic, uses operator float()
   cout << "Length (in cms) = " << length2;
}
```

Run

```
Enter Length (in cms): 150.0
Length (in meter) = 1.5
Enter Length (in meters): 1.669
Length (in cms) = 166.900009
```

Basic to User-Defined Data Type Conversion

In `main()`, the statement

```
meter1 = length1; // converts basic to user-defined, uses constructor1
```

converts basic data item `length1` of `float` type to the object `meter1` by invoking the one-argument constructor:

```
Meter( float InitLength )      // constructor1, one argument
```

This constructor is invoked while creating objects of the class `Meter` using a single argument of type `float`. It converts the input argument represented in centimeters to meters and assigns the resultant value to `length` data member.

The statements such as

```
Meter meter1 = 150.0;
meter1 = length1;
```

invokes the same conversion function. The only difference is, in the case of the first statement, the conversion function is invoked as a part object creation activity, whereas in the case of the second statement, the compiler first searches for the overloaded assignment operator function, and if that is not found, it invokes the one-argument constructor.

The distinction between the function definition and the assignment operator overloading for type conversion is blurred by the compiler; the compiler looks for a constructor if an overloaded = operator function is not available to perform data conversion.

User-Defined to Basic Data Type Conversion

In `main()`, the statement,

```
length2 = meter2; // convert user-defined to basic, uses operator float()
```

converts the object `meter2` to the basic data-item of `float` type by invoking the overloaded operator function:

```
operator float()
{
   float LengthCms;
   LengthCms = length * 100.0; // meter to centimeter
   return( LengthCms );
}
```

The above conversion function can also be invoked explicitly as follows:

```
length2 = ( float ) meter2;
```

or as

```
length2 = float( meter2 );
```

The compiler searches for the appropriate conversion function. First, the compiler looks for an overloaded = operator. If it does not find one, then it looks for a conversion function and invokes the same implicitly for data conversion.

Conversion between Strings and String Objects

The program `strconv.cpp` demonstrates the use of a one argument constructor and a conversion function.

```
// strconv.cpp: conversion between basic string (char *) and class string
#include <iostream.h>
#include <string.h>
const int BUFF_SIZE = 50; // length of string
class string             // user defined string class
{
  private:
    char str[BUFF_SIZE];
  public:
    string()             // constructor1 without arguments
    {
      strcpy( str, "" );
    }
    string( char *MyStr )   // constructor2, one argument
    {
      strcpy( str, MyStr );   // MyStr is copied to str
    }
    void echo()    // display string
    {
      cout << str;
    }
    // conversion function to convert String object item to char * item
    operator char * () // invoked if destination data-item is char* type
    {
```

```
        return str;
    }
};
void main()
{
  // Conversion from string of type char * to string object
  char msg[20] = "OOPs the Great";
  string str1;    // uses constructor 1
  str1 = msg;     // uses the function 'string( char *MyStr )'
  cout << "str1 = ";
  str1.echo();
  // Conversion from object to char * type
  char *receive;
  string str2 = "It is nice to learn";
  receive = str2;   // uses the function 'operator char * ()'
  cout << "\nstr2 = ";
  cout << receive;
}
```

Run

```
str1 = OOPs the Great
str2 = It is nice to learn
```

In the above example, the one argument constructor

```
string( char *MyStr )   // constructor2, one argument
{
   strcpy( str, MyStr );   // MyStr is copied to str
}
```

converts a normal string defined using `char*` to an object of class `string`. The string is passed as an argument to the function; it copies the string `MyStr` to the `str` data member of the object.

The conversion will be applied during creation of the string object with initialization or during the assignment of a normal string to the `string` object. In the statement

```
string str2 = "It is nice to learn";
```

the conversion of normal string to `string` object initialization is performed during creation of the object `str2`. Whereas, in the statement

```
str1 = msg;   // uses the function 'string( char *MyStr )
```

the conversion of normal string defined as `char*` type variable `msg` to `string` object initialization is performed during assignment. The conversion function

```
operator char * ()  // invoked if destination data-item is char * type
{
    return str;
}
```

is used to convert from a string object to a normal string. It is invoked by the the statement,

```
receive = str2; // uses the function 'operator char * ()
```

The object `str2` can also be passed to the indirection operator << to display a string stored in the data member `str` as shown in the statement,

```
cout << str2;
```

The object `str2` is passed as an argument to the overloaded output stream operator <<. But, it does not know anything about the user-defined object `str2`. This is resolved by the compiler by searching for a function which converts the object to a data-type known to the operator `<<()`. In this case, the compiler finds the operator function `char*()`, returning the `char*` type known to the stream operator. If the compiler does not find the conversion function, it reports an error

```
"Operator cannot be applied to these operands in function main()"
```

The program `strconv.cpp` clearly demonstrates the data conversions that take place not only during object creation and in assignment statements, but also in the case of arguments passed to operators (for instance, <<) or functions. Incompatible arguments can also be passed to an operator or a function as long as there exists a conversion function. The incompatibility between the formal arguments of the operator function and actual arguments is resolved by the compiler.

13.17 Conversion between Objects of Different Classes

The C++ compiler does not support data conversion between objects of user-defined classes. The data conversion methods: *one-argument constructor* and *conversion function* can also used for conversions among user defined data types. The choice between these two methods for data conversion depends on whether the conversion function should be defined in the source object or destination object. Consider the following skeleton code:

```
ClassA objecta;
ClassB objectb;
.....
objecta = objectb;
```

where `objecta` and `objectb` are the objects of classes `ClassA` and `ClassB` respectively. The conversion method can be either defined in `ClassA` or `ClassB` depending on whether it should be a one-argument constructor or an operator function.

Conversion Routine in Source Object: `operator` function

The conversion routine in the source object's class is implemented as an operator function. The segment of code shown in Figure 13.9 for class declaration demonstrates the method of implementing a conversion routine in the source object's class.

In an assignment statement such as,

```
objecta = objectb;
```

`objectb` is the source object of the class `ClassB` and `objecta` is the destination object of the class `ClassA`. The conversion function `operator ClassA()` exists in the source object's class.

The program `d2r1.cpp` illustrates the concept of defining a conversion routine in the source object. The conversion of an angle between degrees and radians is achieved by the following relations:

- Angle in Radian = Angle in Degree * PI / 180.0
- Angle in Degree = Angle in Radian * 180.0 / PI, where PI = 22/7

```
// Destination object class
class ClassA
{
   // ClassA stuff here

};
// Source object class
class ClassB
{
   private:
        // attributes of classB
   public:                      → Destination object's class name

        operator ClassA() ←———— Conversion operator function
        {
          // program stuff for converting ClassB object
          // to ClassA object attributes
        }
        . . .
        . . .

};
```

Figure 13.9: Conversion routine in source object

```
// d2r1.cpp: Degree to Radian, Conversion Routine in Source class
#include <iostream.h>
const float  PI = 3.141592654;
class Radian
{
   private:
        float rad;                      // radian
   public:
        Radian()                        // constructor0, no arguments
        {
            rad = 0.0;
        }
        Radian( float InitRad )          // constructor1
        {
            rad = InitRad;
        }
        float GetRadian()               // Access function
        {
            return( rad );
        }
        void Output()                   // Display of radian
        {
            cout << "Radian = " << GetRadian();
        }
};
```

```
class Degree
{
   private:
      float degree;                    // Degree
   public:
      Degree()                         // constructor0, no arguments
      {
         degree = 0.0;
      }
      // radian = degree; conversion routine at the source
      // This function will be called if we try to assign
      // object degree to object of type radian
      operator Radian()
      {
         // convert degree to radian and create an object radian
         // and then return, here radian constructor1 is called
         return( Radian( degree * PI / 180.0 ) );
      }
      void Input()          // Read degree
      {
         cout << "Enter Degree: ";
         cin >> degree;
      }
};
void main( void )
{
   Degree deg1;           // degree using constructor0
   Radian rad1;           // radian using constructor0
   // Read Input values
   deg1.Input();
   rad1 = deg1; // uses 'operator Radian()'
   // display radian and degree
   rad1.Output();
}
```

Run1

```
Enter Degree: 90
Radian = 1.570796
```

Run2

```
Enter Degree: 180
Radian = 3.141593
```

In `main()`, the statement

```
rad1 = deg1; // uses 'operator Radian()'
```

assigns the `deg1` object of class `Degree` to the `rad1` object of the class `Radian`. Since both the objects `deg1` and `rad1` are instances of different classes, the conversion during assignment operation is performed by the member function:

```
operator Radian()
{
   // convert degree to radian and create an object radian
   // and then return, here radian constructor1 is called
   return( Radian( degree * PI / 180.0 ) );
}
```

It is defined in the source object's class `Degree`; it is chosen by the compiler for converting the object `deg1` to `rad1` implicitly.

Conversion Routine in Destination Object: constructor function

The conversion routine can also be defined in the destination object's class as a one-argument constructor. The segment of code shown in Figure 13.10 for class declaration demonstrates the method of implementing a conversion routine in the destination object's class.

```
// Source object class
class ClassB
{
  // ClassB stuff here

};
// Destination object class
class ClassA
{
  private:
     // attributes of classA
  public:

     ClassA(ClassB objectb)
     {

        // program stuff for converting ClassB object
        // to ClassA object attributes
        // Private attributes of ClassB are accessed
        // through its public functions
        ...
        ...
     }
};
```

Destination object's class name
object of a source class
Constructor function

Figure 13.10: Conversion routine in destination object

In an assignment statement such as

```
objecta = objectb;
```

`objectb` is the source object of `ClassB` and `objecta` is the destination object of class `ClassA`. The conversion function (constructor function in this case) `ClassA( ClassB objectb )` is defined in the destination object's class. The program `d2r2.cpp` illustrates the concept of defining conversion function in the destination object.

```
// d2r2.cpp: Degree to Radian. Conversion Routine in the Destination object.
#include <iostream.h>
const float PI = 3.141592654;
```

```
class Degree
{
    private:
        float degree;              // Degree
    public:
        Degree()                   // constructor0, no arguments
        {
            degree = 0.0;
        }
        float GetDegree()          // Access function
        {
            return( degree );
        }
        void Input()               // Read degree
        {
            cout << "Enter Degree: ";
            cin >> degree;
        }
};
class Radian
{
    private:
        float rad;                     // radian
    public:
        Radian()                       // constructor0, no arguments
        {
            rad = 0.0;
        }
        float GetRadian()              // Access function
        {
            return( rad );
        }
        // radian = degree: Conversion routine is in destination object's class
        Radian( Degree deg )
        {
            rad = deg.GetDegree() * PI / 180.0;
        }
        void Output()                  // Display of radian
        {
            cout << "Radian = " << GetRadian();
        }
};
void main( void )
{
    Degree deg1;             // degree using constructor0
    Radian rad1;             // radian using constructor0
    // Read Input values
    deg1.Input();
    rad1 = deg1;         // uses Radian( Degree deg )
    rad1.Output();      // display radian and degree
}
```

Run1

```
Enter Degree: 90
Radian = 1.570796
```

Run2

```
Enter Degree: 180
Radian = 3.141593
```

In `main()`, the statement

```
rad1 = deg1;  // convert degree to radian, uses Radian( Degree deg )
```

assigns the user-defined object `deg1` to another object `rad1`. Since, the objects `deg1` and `rad1` are of different types, the conversion during the assignment operation is performed by a member function

```
Radian( Degree deg )
{
   rad = deg.GetDegree() * PI / 180.0;
}
```

defined in the destination object's class Radian as a one-argument constructor. It is chosen by the compiler for converting the object `deg1's` attributes to `rad1's` attributes implicitly. The constructor must be able to access the private data members defined in the source object's class. The `Degree` class defines the following interface function

```
float GetDegree()               // Access function
{
   return( degree );
}
```

to access the private data members. Note that, the body of the function `main()` in the program `d2r2.cpp` is the same as that in the program `d2r1.cpp`, although the conversion methods have appeared in different forms.

Complete Conversion

The program `degrad.cpp` illustrates the concept of defining conversion functions in the source or destination object's class. In this program, angles in degrees can be converted to radians or angles in radians can be converted to degrees. The class `Degree` has conversion functions: constructor function and operator function. A class can have any number of conversion functions as long their signatures are different.

```
// degrad.cpp: Degree to Radian data conversion and vice-versa
#include <iostream.h>
const float  PI = 3.141592654;
class Radian
{
   private:
      float rad;                // radian
   public:
      Radian()                  // constructor0, no arguments
      {
         rad = 0.0;
      }
```

```
      Radian( float InitRad )     // constructor1, one argument
      {  rad = InitRad;    }
      float GetRadian()           // Access function
      {
         return( rad );
      }
      void Input()                // Read radian
      {
         cout << "Enter Radian: ";
         cin >> rad;
      }
      void Output()               // Display of radian
      {
         cout << "Radian = " << GetRadian() << endl;
      }
};
class Degree
{
   private:
      float degree;          // Degree
   public:
      Degree()               // constructor0, no arguments
      {
         degree = 0.0;
      }
      // degree = radian: Conversion routine at the destination
      Degree( Radian rad )    // constructor1, one-argument constructor
      {
         degree = rad.GetRadian() * 180.0 / PI;
      }
      float GetDegree()           // Access function
      {
         return( degree );
      }
      // radian = degree; conversion routine at the source
      operator Radian()
      {
         // convert degree to radian and create an object radian
         // and then return, here radian constructor 1 is called
         return( Radian( degree * PI / 180.0 ) );
      }
      void Input()                // Read degree
      {
         cout << "Enter Degree: ";
         cin >> degree;
      }
      void Output()               // Display output
      {
         cout << "Degree = " << degree << endl;
      }
};
```

```
void main( void )
{
   Degree deg1, deg2;        // degree using constructor0
   Radian rad1, rad2;        // radian using constructor0
   // degree to radian conversion
   deg1.Input();
   rad1 = deg1;   // convert degree to radian, uses 'operator Radian()'
   rad1.Output();
   // radian to degree conversion
   rad2.Input();
   deg2 = rad2;   // convert radian to degree, uses Degree( Radian rad )
   deg2.Output();
}
```

Run
```
Enter Degree: 180
Radian = 3.141593
Enter Radian: 3.142
Degree = 180.023331
```

One-Argument Constructor or Operator Function ?

From the above discussion, it is evident that either the one-argument constructor or the operator function can be used for converting objects of different classes. A wide variety of classes in the form of class libraries are available commercially. But, they are supplied as object modules (machine code in linkable form) and not as source modules. The user has no control over the modification of such classes. This leads to a problem of conversion between the objects defined using the classes supplied by the software vendors and objects defined using the classes declared by the user. This problem can be circumvented by defining a conversion routine in the user-defined classes. It can be a one-argument constructor or a operator function depending on whether the user-defined object is a source or destination object. The thumb rules for deciding where conversion routine has to be defined are the following:

- If the user-defined object is a source object, the conversion routine must be defined as an operator function in the source object's class.
- If the user-defined object is a destination object, the conversion routine must be defined as a one-argument constructor in the destination object's class.
- If both the source and destination object are the instances of user-defined classes, the conversion routine can be placed either in source object's class as a operator function or in destination object's class as a constructor function.

13.18 Subscript Operator Overloading

The subscript operator [] can be overloaded to access the attributes of an object. It is mainly useful for bounds checking while accessing elements of an array. Consider the following definition

```
int a[10];
```

An expression such as a[20] is syntactically valid though it is accessing an element beyond the range. Such an illegal access can be detected by overloading subscript operators. The user defined class can overload the [] operator and check for validity of accesses to array of objects and permit access to its members only when the index value is valid.

An array of primitive data type can be accessed using integer subscripts only. However, when it is overloaded, it can take parameters other than integer types, i.e., the argument of an operator function `[]` need not be an integer; it can be of any data type. The program `script.cpp` illustrates the concept of overloading the subscript operator `[]`.

```
// script.cpp: Subscripted operator overloading
#include <iostream.h>
#include <string.h>
typedef struct AccountEntry
{
   int number;      // account number
   char name[25];  // name of account holder
} AccountEntry;
class AccountBook
{
   private:
      int aCount;    // account holders count
      AccountEntry account[10];  // accounts table
   public:
      AccountBook( int aCountIn )  // constructor 1
      {
         aCount = aCountIn;
      }
      void AccountEntry();
      int operator [] ( char * nameIn );
      char * operator [] ( int numberIn );
};
// takes name as input, returns account number
int AccountBook::operator [] ( char *nameIn )
{
    for( int i = 0; i < aCount; i++ )
        if( strcmp( nameIn, account[i].name) == 0 )
         return account[i].number; // found name, return its account number
   return 0;
}
// takes number as input, returns name corresponding to account number
char * AccountBook::operator [] ( int numberIn )
{
   for( int i = 0; i < aCount; i++ )
      if( numberIn == account[i].number )
         return account[i].name;
   return 0;
}
void AccountBook::AccountEntry()
{
   for( int i = 0; i < aCount; i++ )
   {
      cout << "Account Number: ";
      cin >> account[i].number;
      cout << "Account Holder Name: ";
      cin >> account[i].name;
```

```
   }
}
void main()
{
   int accno;
   char name[25];
   AccountBook accounts( 5 ); // account having 5 customers
   cout << "Building 5 Customers Database" << endl;
   accounts.AccountEntry();   // read
   cout << "\nAccessing Accounts Information";
   cout << "\nTo access Name Enter Account Number: ";
   cin >> accno;
   cout << "Name: " << accounts[accno]; //operator [] ( int numberIn )
   cout << "\nTo access Account Number, Enter Name: ";
   cin >> name;
   cout << "Account Number: " << accounts[name];
                               // uses, operator [] ( char *nameIn )
}
```

Run

```
Building 5 Customers Database
Account Number: 1
Account Holder Name: Rajkumar
Account Number: 2
Account Holder Name: Kiran
Account Number: 3
Account Holder Name: Ravishanker
Account Number: 4
Account Holder Name: Anand
Account Number: 5
Account Holder Name: Sindhu
Accessing Accounts Information
To access Name Enter Account Number: 1
Name: Rajkumar
To access Account Number, Enter Name: Sindhu
Account Number: 5
```

In `main()`, the statement

```
accounts.AccountEntry();    // read
```

reads a database of five account holders and initializes the object's data members. The statement

```
cout << "Name: " << accounts[accno]; // operator [] ( int numberIn )
```

uses the function

```
char * operator [] ( int numberIn );
```

and returns the name of the account holder for a given account number. The statement

```
cout << "Account Number: " << accounts[name];
```

uses the function

```
int operator [] ( char *nameIn )
```

and returns the account number corresponding to the name of the given account holder's name. The compiler selects the appropriate function which matches with the actual parameter's data type.

13.19 Overloading with Friend Functions

Friend functions play a very important role in operator overloading by providing the flexibility denied by the member functions of a class. They allow overloading of stream operators (<< or >>) for stream computation on user defined data types. The only difference between a friend function and member function is that, the friend function requires the arguments to be explicitly passed to the function and processes them explicitly, whereas the member function considers the first argument implicitly. Friend functions can either be used with unary or binary operators. The syntax of operator overloading with friend functions is shown in Figure 13.11.

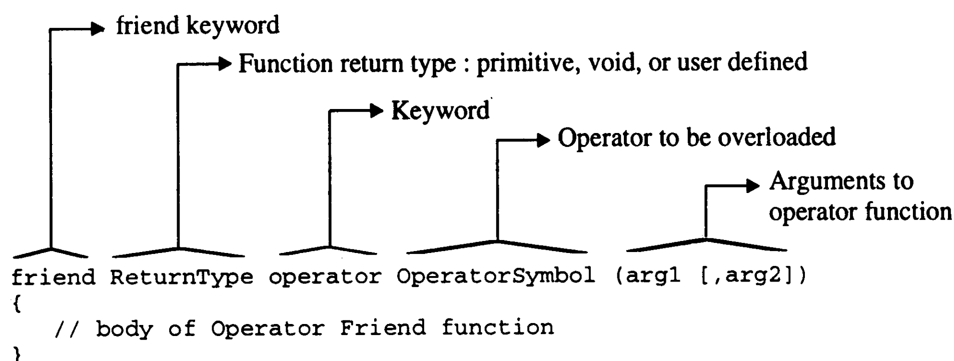

Figure 13.11: Syntax of overloading with friend function

The prototype of the friend function must be prefixed with the keyword `friend` inside the class body. The body of friend function can appear either inside or outside the body of a class. It is advisable to define a friend function outside the body of a class. The definition of the friend function outside the body of a class is defined as normal function and is not prefixed with the `friend` keyword. The arguments of the friend functions are generally objects of friend classes. In a friend function, all the members of a class (to which this function is a friend) can be accessed by using its objects. *Friend function is not allowed to access members of a class (to which it is a friend) directly, but it can access all the members including the private members by using objects of that class.* Hence, a friend function is similar to a normal function except that it can access the private members of a class using its objects.

Unary Operator Overloading using Friend Functions

The program `complex6.cpp` illustrates the concept of negation of complex numbers. The negation function returns negated object without modifying the source object.

```
// complex6.cpp: Negation of complex number with Unary Operator
#include <iostream.h>
class complex
{
   private:
      float real;
      float imag;
   public:
      complex()         // no argument constructor
      {
         real = imag = 0.0;
```

```
    }
    void getdata();   // read complex number
    void outdata( char *msg );    // display complex number
    // overloading of unary minus operator to support c2 = - c1
    friend complex operator - ( complex c1 )
    {
       complex c;
       c.real = -c1.real;
       c.imag = -c1.imag;
       return( c );
    }
    void readdata();
};
void complex::readdata()
{
   cout << "Real Part ? ";
   cin >> real;
   cout << "Imag Part ? ";
   cin >> imag;
}
void complex::outdata( char *msg )
{
   cout << endl << msg;
   cout << "(" << real;
   cout << ", " << imag << ")";
}
void main()
{
   complex c1, c2;
   cout << "Enter Complex c1.." << endl;
   c1.readdata();
   c2 = -c1;          // invokes complex operator - ()
   c1.outdata( "Complex c1 : " );
   c2.outdata( "Complex c2 = -Complex c1: " );
}
```

Run

```
Enter Complex c1..
Real Part ? 1.5
Imag Part ? -2.5
Complex c1 : (1.5, -2.5)
Complex c2 = -Complex c1: (-1.5, 2.5)
```

The complex number negation function without a friend is declared as follows:

```
complex operator - ()
```

In this case, arguments are implicitly assumed. Using the keyword `friend`, it is declared as follows:

```
friend complex operator - ( complex c1 )
```

The above friend operator function cannot access members of the class `complex` directly, unlike its member functions. In `main()`, the statement

```
c2 = -c1;   // invokes unary operator function, complex operator - ()
```

computes the negation of c1 and assigns it to c2. It returns the negated result without negating contents of the c1 object. The object c1 is passed as a value parameter to the negate operator function and any modification to its data members will be reflected in the c1 object.

The negation operation can also be applied to an object to modify its data members. In this case, the same object acts both as a source and a destination object. It is similar to representing a negative number. This can be achieved by passing the object as a reference parameter to the negation operator function so that, the negation of its data members can be also reflected in the calling object. The program complex7.cpp illustrates the concept of negation of complex numbers having the same source and destination operands.

```
// complex7.cpp: Negation of Complex Number with Unary Operator Overloading
#include <iostream.h>
class complex
{
   private:
      float real;
      float imag;
   public:
      complex() { real = imag = 0; }
      void readdata();
      void outdata( char *msg );
      // Note: friend function with explicit reference parameter
      // overloading of unary minus, -c1
      friend void operator - ( complex & c1 ); // definition outside
};
// friend function of the class complex
// Note that, the keyword friend should not prefixed while defining outside
void operator - ( complex & c1 )
{
   c1.real = -c1.real;
   c1.imag = -c1.imag;
}
void complex::readdata()
{
   cout << "Real Part ? ";
   cin >> real;
   cout << "Imag Part ? ";
   cin >> imag;
}
void complex::outdata( char *msg )
{
   cout << endl << msg;
   cout << "(" << real;
   cout << ", " << imag << ")";
}
void main()
{
   complex c1;
```

```
   cout << "Enter Complex c1.." << endl;
   c1.readdata();
   -c1;  // invokes unary operator function, complex operator - ()
   c1.outdata( "Result of -Complex c1: " );
}
```

Run

```
Enter Complex c1..
Real Part ? 1.5
Imag Part ? -2.5
Result of -Complex c1: (-1.5, 2.5)
```

In `main()`, the statement

```
-c1;  // invokes unary operator function, complex operator - ()
```

invokes the function

```
void operator - ( complex & c1 )
```

by passing the object `c1` by reference. Thus, the negation of `c1` in the function is also reflected in the calling object. Note that, the definition of operator friend function is the same as normal functions.

Binary Operator Overloading using Friend Function

The complex number discussed in the program `complex2.cpp` can be modified using a friend operator function as follows:

1. Modify the member function prototype as follows:

```
friend complex operator + ( complex c1, complex c2 )
```

2. Redefine the operator function as follows:

```
friend complex operator + ( complex c1, complex c2 )
{
   complex c;
   c.real = c1.real + c2.real;
   c.imag = c1.imag + c2.imag;
   return( c );
}
```

In the above definition, the input object parameters `c1` and `c2` are handled explicitly without considering the first argument as an implicit argument. The statement

```
c3 = c1 + c2;
```

is equivalent to the statement

```
c3 = operator + ( c1, c2 );
```

The result generated by the friend function is same as that generated by the member function. But, friend functions offer the flexibility of writing an expression as a combination of operands of user defined and primitive data types. For instance, consider the statement

```
c3 = c1 + 2.0;
```

The expression `c1 + 2.0` is made up of the object `c1` and a primitive type. In case of an operator member function, both the operands must be of object's data type. When the friend operator functions are used, both the operands need not be instances of user-defined data type. It requires a parameterized constructor taking a primitive data type parameter. The program `complex8.cpp` illustrates the concept of overloading an operator function as a friend function.

```
// complex8.cpp: Addition of Complex Numbers with friend feature
#include <iostream.h>
class complex
{
  private:
     float real;
     float imag;
  public:
     complex()
     {}
     complex( int realpart )
     {
        real = realpart;
     }
     void readdata()
     {
        cout << "Real Part ? ";
        cin >> real;
        cout << "Imag Part ? ";
        cin >> imag;
     }
     void outdata( char *msg )     // display complex number
     {
        cout << endl << msg;
        cout << "(" << real;
        cout << ", " << imag << ")";
     }
     friend complex operator + ( complex c1, complex c2 );
};
// note that friend keyword and scope resolution operator are not used
complex operator + ( complex c1, complex c2 )
{
  complex c;
  c.real = c1.real + c2.real;
  c.imag = c1.imag + c2.imag;
  return( c );
}
void main()
{
   complex c1, c2, c3 = 3.0;
   cout << "Enter Complex1 c1..:" << endl;
   c1.readdata();
   cout << "Enter Complex2 c2..:" << endl;
   c2.readdata();
   c3 = c1 + c2;
   c3.outdata( "Result of c3 = c1 + c2: " );
   // 2.0 is considered as real part of complex
   c3 = c1 + 2.0;    // c3 = c1 + complex(2.0)
   c3.outdata( "Result of c3 = c1 + 2.0: " );
   // 3.0 is considered as real part of complex
```

```
    c3 = 3.0 + c2;    // c3 = complex( 3.0 ) + c2
    c3.outdata( "Result of c3 = 3.0 + c2: " );
}
```

Run

```
Enter Complex1 c1..:
Real Part ? 1
Imag Part ? 2
Enter Complex2 c2..:
Real Part ? 3
Imag Part ? 4
Result of c3 = c1 + c2: (4, 6)
Result of c3 = c1 + 2.0: (3, 2)
Result of c3 = 3.0 + c2: (6, 4)
```

In `main()`, the statement

```
c3 = c1 + 2.0;    // c3 = c1 + complex(2.0)
```

has an expression, which is a combination of the object `c1` and the primitive floating point constant 2.0. Though, there is no member function matching this expression, the compiler will resolve this by treating the expression as follows:

```
c3 = c1 + complex( 2.0 );
```

The compiler invokes the single argument constructor and converts the primitive value to a new temporary object (here 2.0 is considered as a real part of the complex number) and passes it to the friend operator function:

```
friend complex operator + ( complex c1, complex c2 )
```

The sum of the object `c1` and a new temporary object `complex( 2.0 )` is computed and assigned to object `c3`. The new temporary objects are destroyed immediately after execution of the statement due to which it is created. The above expression can also be written as

```
c3 = 2.0 + c1;
```

Recall that the left-hand operand is responsible for invoking its member function; but this statement has a numeric constant instead of an object. The outcome of either expression is the same, since the compiler treats it as follows:

```
c3 = complex( 2.0 ) + c1;
```

In C++, an object can be used not only to invoke a friend function, but also as an argument to a friend function. Thus, to the friend operator functions, a built-in type operand can be passed either as the first operand or as the second operand.

Overloading Stream Operators using Friend Function

The `iostream` facility of C++ provides an easy means to perform I/O. The class `istream` uses the predefined stream `cin` that can be used to read data from the standard input device. The *extraction* operator >> is used for performing input operations in the `iostream` library. The *insertion* operator << is used for performing output operations in the `iostream` library.

Similar to the built-in variables, the user-defined objects can also be read or displayed using the stream operators. In case of the overloaded operator << function, the `ostream &` is taken as the first argument of a friend function of a class. The return value of this friend function is of type `ostream &` as shown in Figure 13.12.

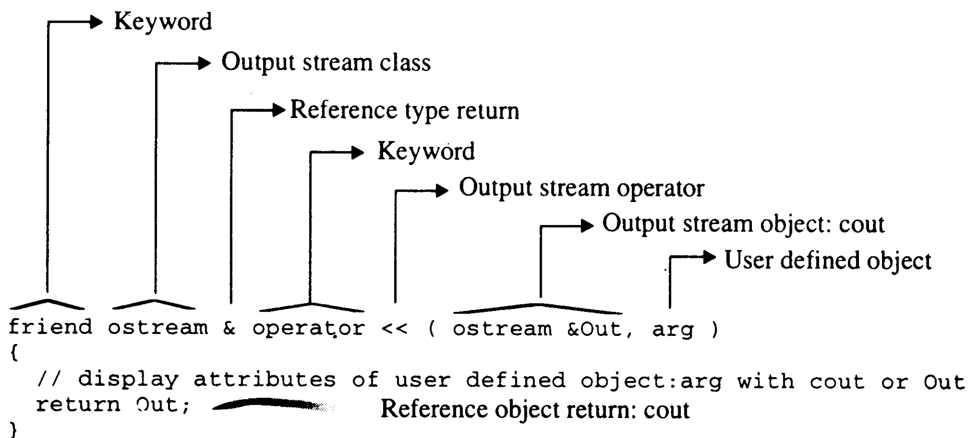

Figure 13.12: Overloading output stream operator as friend function

Similarly, for overloading the >> operator, the `istream &` is taken as the first argument of a friend function of the class. The return value of this friend function is of type `istream &` as shown in Figure 13.13. In both the cases, a reference to an object of the current class is taken as the second argument and the same is returned by reference.

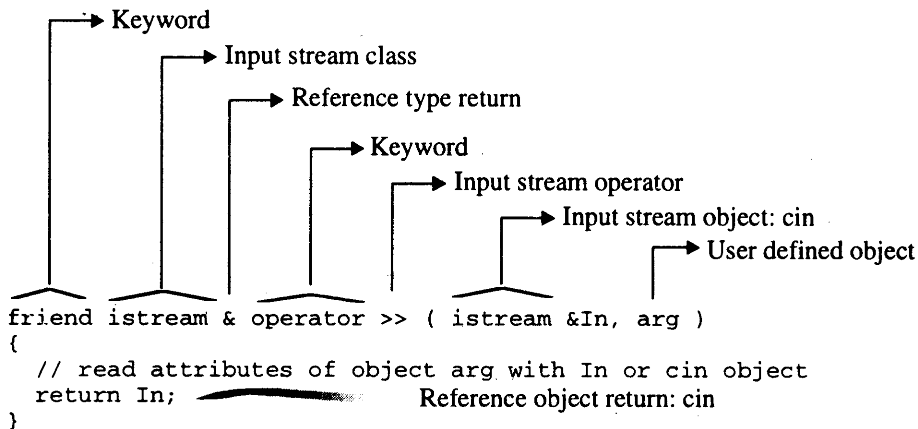

Figure 13.13: Overloading input stream operator as friend function

The program `complex9.cpp` illustrates the flexibility of overloading stream operators and their usage with objects of the user defined data type.

```
// complex9.cpp: Addition of Complex Numbers with stream overloading
#include <iostream.h>
class complex
{
   private:
      float real;
      float imag;
   public:
```

```
      complex() { }
      complex( float InReal )
      {
         real = InReal;
         imag = 0;
      }
      void outdata();
      friend complex operator + ( complex c1, complex c2 )
      {
         complex c;
         c.real = c1.real + c2.real;
         c.imag = c1.imag + c2.imag;
         return( c );
      }
      friend istream & operator >> ( istream &In, complex &c  );
      friend ostream & operator << ( ostream &Out, complex &c  );
};
istream & operator >> ( istream & In, complex & c )
{
   cout << "Real Part ? ";
   In >> c.real;  // cin >> c.real;
   cout << "Imag Part ? ";
   In >> c.imag;  // cin >> c.imag;
   return In;
}
ostream & operator << ( ostream &Out, complex & c )
{
   Out << "(" << c.real;           // or cout << "Real = " << c.real;
   Out << ", " << c.imag << ")"; // cout in place of Out
   return Out;
}
void main()
{
    complex c1, c2, c3 = 3;
    cout << "Enter Complex1 c1..:" << endl;
    cin >> c1;
    cout << "Enter Complex2 c2..:" << endl;
    cin >> c2;
    c3 = c1 + c2;
    cout << "Result of c3 = c1 + c2: ";
    cout << c3;
    // 2.0 is considered as real part of complex
    c3 = c1 + 2.0;   // c3 = c1 + complex(2.0)
    cout<<endl<<"Result of c3 = c1 + 2.0: "; //c3=c1 + complex(2.0);
    cout << c3;
    // 3.0 is considered as real part of complex
    c3 = 3.0 + c2;
    cout<< endl<<"Result of c3 = 3.0 + c2: "; //c3=complex(3.0)+ c2;
    cout << c3;
}
```

Run

```
Enter Complex1 c1..:
Real Part ? 1
Imag Part ? 2
Enter Complex2 c2..:
Real Part ? 3
Imag Part ? 4
Result of c3 = c1 + c2: (4, 6)
Result of c3 = c1 + 2.0: (3, 2)
Result of c3 = 3.0 + c2: (6, 4)
```

In main(), the statements

```
cin >> c1;
cin >> c2;
```

read user-defined class's objects c1 and c2 in the same way as built-in data type variables by using the input stream operator. Also, the sum of the complex numbers c1 and c2 stored in c3 is displayed by the statement,

```
cout << c3;
```

similar to any built-in data item using the output stream operator. The overloaded stream operator functions performing I/O operations with complex numbers are the following:

```
friend istream & operator >> ( istream &In, complex &c  );
friend ostream & operator << ( ostream &Out, complex &c  );
```

The classes istream and ostream are defined in the header file iostream.h, which has been included in the program. C++ does not allow overloading of operators listed in Table 13.2 as friend operator functions. They can, however be overloaded as operator member functions.

Operator Category	Operators
Assignment	=
Function call	( )
Subscribing	[ ]
Class Member Access	->

Table 13.2: Operators that cannot be overloaded as friend operators

13.20 Assignment Operator Overloading

The compiler copies all the members of a user-defined source object to a destination object in an assignment statement, when its members are statically allocated. The data members, which are dynamically allocated must be copied to the destination object explicitly by overloading the assignment operator. Two examples of this process are the assignment operator and the copy constructor. Consider the following statements:

```
vector v1( 5 ), v2( 5 );
v1 = v2;     // operator = invoked
```

```
vector v3 = v2;   // copy constructor is invoked
```

The first statement defines two objects v1 and v2 of the class vector. The second assignment statement

```
v1 = v2;
```

will cause the compiler to copy the data from v2, member-by-member, into v1. The action is similar to the default operation performed by the assignment operator. The next statement

```
vector v3 = v2;
```

initializes one object with another object during definition. This statement causes a similar action after creating the new object v3. The data members from v2 are copied member-by-member into v3. This action is similar to the operation performed by the copy constructor, by default.

The default actions performed by the compiler (to perform assignment operation) are insufficient if the object's state is dynamically varying. Such objects can be processed by overriding these default actions. The program vector.cpp illustrates the concept of overriding default actions by the user-defined overloaded assignment operator and copy constructor.

```
// vector.cpp: overloaded assignment operator for vector elements copying
#include <iostream.h>
class vector
{
      int * v;     // pointer to vector
      int size;    // size of vector v
   public:
      vector( int vector_size )
      {
         size = vector_size;
         v = new int[ vector_size ];
      }
      vector( vector &v2 );
      ~vector()
      {
         delete v;
      }
      void operator = ( vector & v2 );
      int & elem( int i )
      {
         if( i >= size )
            cout << endl << "Error: Out of Range";
         return v[i];
      }
      void show();
};
// copy constructor, vector v1 = v2;
vector::vector( vector &v2 )
{
   cout << "\nCopy constructor invoked";
   size = v2.size;             // size of v1 is equal to size of v2
   v = new int[ v2.size ];     // allocate memory of the vector v1
```

```
    for( int i = 0; i < v2.size; i++ )
        v[i] = v2.v[i];
}
// overloading assignment operator, v1 = v2, v1 is implicit
void vector::operator = ( vector & v2 )
{
    cout << "\nAssignment operation invoked";
    // memory is already allocated to the vector and v1.size = v2.size
    for( int i = 0; i < v2.size; i++ )
        v[i] = v2.v[i];
}
void veccor::show()
{
    for( int i = 0; i < size; i++ )
        cout << elem( i ) << ", ";
}
void mai..()
{
    int i;
    vector v1( 5 ), v2( 5 );
    for( i = 0; i < 5; i++ )
        v2.elem( i ) = i + 1;
    v1 = v2;     // operator = invoked
    vector v3 = v2;   // copy constructor is invoked
    cout << "\nvector v1: ";
    v1.show();
    cout << "\nvector v2: ";
    v2.show();
    cout << "\nvector v2: ";
    v3.show();
}
```

Run

```
Assignment operation invoked
Copy constructor invoked
vector v1: 1, 2, 3, 4, 5,
vector v2: 1, 2, 3, 4, 5,
vector v2: 1, 2, 3, 4, 5,
```

The overloaded = operator function does the job of copying the data members from one object to another. The function also prints a message to assist the user in keeping track of its execution.

The copy constructor

```
vector( vector &v2 );
```

takes one argument, an object of the type `vector`, passed by reference. It is essential to pass a reference argument to the copy constructor. It cannot be passed by value. When an argument is passed by value, its copy is constructed using the copy constructor, i.e., the copy constructor would call itself to make this copy. This process would go on until the system runs out of memory. Hence, *arguments to the copy constructor must be always passed by reference, thus preventing creation of copies.* A copy

constructor also gets invoked when arguments are passed by value to functions and when values are returned from functions. When an object is passed by value, the argument on which the function operates is created using a copy constructor. If an object is passed by its address or reference, the copy constructor of course would not be invoked, and the copies of the objects are not created. When an object is returned from a function, the copy constructor is invoked to create a copy of the value returned by the function.

13.21 Tracing Memory Leaks

Memory fragmentation can affect program performance, but memory *leaks* frequently cause programs to crash. A memory leak occurs when the user program fails to free an allocated memory block. The new operator can be overloaded to write signature bytes for the blocks it allocates. The meaning of *memory leak* is that dynamic memory being allocated (*new*ed) without being releasing (*deleted*). The executable size quickly outgrows the size of memory in the machine, requiring an undesirable amount of swapping activity. The first step in attacking this problem is to find where memory is being requested, used, and not returned.

Approach

In C++, it is easy to overload the built-in `new` and `delete` operators with user-supplied versions and thereby determine when the memory is requested and to which memory location it is bounded. The program `mleak.cpp` overloads `new` and `delete` operators and records the memory location to which the request is bound, in the disk file `space.raw`. It also records all those bindings that are released using explicit memory free request command.

```
// mleak.cpp: Memory leak tracing
#include <iostream.h>
#include <stdio.h>
#include <process.h>
#include <alloc.h>
#include <string.h>
//global information
static space_debug = 1;     // space_debug switch, ON
FILE * fp_space = NULL;     // file pointer to the debug info
void * operator new( size_t size )
{
   void *ptr;
   if( space_debug )
   {
      if( fp_space == NULL )  // first time call to new or delete
      {
         // open leak debug info file which is unopened
         if( (fp_space = fopen( "space.raw", "w" )) == NULL )
         {
            cout << "Error opening space.raw in write mode";
            exit( 1 );
         }
      }
   }
```

```
   if( (ptr = malloc( size )) == NULL )
   {
      cout << "out of memory space";
      exit( 1 );
   }
   if( space_debug )   // debug switch is ON, store memory info
      fprintf( fp_space, "new( %d ) -> %x\n", size, ptr );
   return ptr;
}
void operator delete( void *ptr )
{
   if( space_debug )
   {
      // open leak debug info file which is unopened
      if( fp_space == NULL )  // first time call to new or delete
      {
         if( (fp_space = fopen( "space.raw", "w" )) == NULL )
         {
            cout << "Error opening space.raw in write mode";
            exit( 1 );
         }
      }
   }
   if( ptr )   // if valid pointer
   {
      free( (char *) ptr );
      if( space_debug )  // debug switch is ON, store memory info
         fprintf( fp_space, "free <- %x\n", ptr );
   }
}
void main()
{
   int *vector;
   char *buffer;
   vector = (int *) new int[ 10 ];
   buffer = (char *) new char[ 6 ];
   for( int i = 0; i < 10; i++ )
      vector[i] = i+1;
   strcpy( buffer, "hello" );
   cout << "vector =  ";
   for( i = 0; i < 10; i++ )
      cout << vector[i] << " ";
   cout << endl << "buffer = " << buffer;
   delete vector;    // vector is deallocated
   fclose( fp_space );
}
```

Run

```
vector =  1 2 3 4 5 6 7 8 9 10
buffer = hello
```

The `space_debug` variable allows the programmer to decide whether to trace a particular portion of code or not. When tracing is desired it must be set to a nonzero (debug ON) value. When the following statements:

```
vector = (int *) new int[ 10 ];
buffer = (char *) new char[ 6 ];
```

are invoked in the program, the overloaded `new` operator allocates the requested amount of memory and returns a pointer to the memory location to which it is bound. In addition, it records this memory address to which it is bound, in the disk file `space.raw`. Similarly, the overloaded `delete` operator releases the memory pointed to by the input pointer and also records the memory address in the disk file. In the above *Run*, the information recorded in `space.raw` file is the following:

```
new( 36 ) -> bd2
new( 516 ) -> bfa
new( 36 ) -> e02
new( 516 ) -> e2a
new( 36 ) -> 1032
new( 516 ) -> 105a
new( 10 ) -> 1262
new( 6 ) -> 127a
free <- 1262
free <- bfa
free <- bd2
free <- e2a
free <- e02
free <- 105a
free <- 1032
```

The first six requests are made by the program execution start-up routine. They can be discarded in the memory leak tracing analysis. The seventh and eighth requests are made in the program explicitly. Similarly, the last six memory free requests made by the system, can be discarded during analysis. These requests vary from system to system. The first request to free memory is made by the statement

```
delete vector;     // vector is deallocated
```

The pointer returned for the requests

```
vector = (int *) new int[ 10 ];
buffer = (char *) new char[ 6 ];
```

are the following

```
new( 10 ) -> 1262
new( 6 ) -> 127a
```

By tracing the above allocation address information in the free list, it can be detected that `new(6)` pointer address is not released, leading to memory leak. In the program it can be observed that, the memory allocated for the variable `vector` is released explicitly whereas, the memory allocated for the variable `buffer` is not released. It can also be noticed from the trace of memory debug information.

13.22 Niceties of Operator Overloading and Conversions

Operator overloading and data conversion features of C++ provide an opportunity to the user to redefine the C++ language. Polymorphism feature of C++ is a bonus for the user to customize C++ to their taste. Of course, it can be misused, since C++ does not restrict the user from misusing (exploiting)

the feature of operator overloading. Consider an example of overloading the + operator to perform arithmetic on the user-defined objects x, y, and z. The statement,

```
x = y + z;
```

can represent a different meaning as compared with that conveyed by the operation with basic data types. In the body of overloaded function, even if subtraction operation is performed instead of addition, C++ neither signals an error nor restricts such operation. The above operation can also mean concatenation of strings y and z, and storing the result in x (x, y, and z are object's of String class). Thus, operator overloading provides the ability to redefine the building blocks of the language and allows to manipulate the user-defined data-items in a more intuitive and readable way.

The program misuse.cpp illustrates the misuse of the operator overloading feature in C++. The compiler only validates syntax errors but not the semantics.

```
// misuse.cpp: Misuse of operator overloading, performs subtraction instead
//          of addition operation
#include <iostream.h>
class number
{
   private:
      int num;
   public:
      void read()     // number read function
      {
         cin >> num;
      }
      int get()     // private member num access function
      {
         return num;
      }
      // overloaded operator for number addition
      number operator+( number num2 )
      {
          number sum;
          sum.num = num - num2.num;  // subtraction instead of addition
          return sum;
      }
};
void main()
{
   number num1; num2, sum;
   cout << "Enter Number 1: ";
   num1.read();
   cout << "Enter Number 2: ";
   num2.read();
   sum = num1 + num2;   // addition of number
   cout << "sum = num1 + num2 = " << sum.get();
}
```

Run1

```
Enter Number 1: 20
```

```
Enter Number 2: 10
sum = num1 + num2 = 10
```

Run2

```
Enter Number 1: 5
Enter Number 2: 10
sum = num1 + num2 = -5
```

In `main()`, the statement

```
sum = num1 + num2;     // addition of number
```

is supposed to perform addition of two numbers `num1` and `num2`, but instead it performs subtraction. The statement in the body of the overloaded operator function `number operator+(..)`

```
sum.num = num - num2.num; // instead of addition, subtraction is done
```

performs subtraction instead of addition. Such neglected use of operator overloading is not taken care by the C++ compiler, but it is the responsibility of the programmer.

As operator overloading is only a notational convenience, the language should try to prevent its misuse (but C++ does not prevent). It is indeed said that *the meaning of operators applied to standard data types cannot be redefined. The intent is to make C++ extensible, but not mutable.* Hence, operators cannot be overloaded for enumerations, although it would be sometimes desirable and fully sensible.

Guidelines

It is essential to follow syntax and semantic rules of the language while extending the power of C++ using operator overloading. In fact, operator overloading feature opens up a vast vistas of opportunities for creative programmers (for instance, `new` and `delete` can be overloaded to detect memory leaks as illustrated earlier). The following are some guidelines that needs to be kept in mind while overloading any operators to support user defined data types:

1. Retain Meaning

Overloaded operators must perform operations similar to those defined for primitive/basic data types. The operator + can be overloaded to perform subtraction; operator * can be overloaded to perform division operation. However, such definitions should be avoided to retain the intuitive meaning of the operators. For example, the overloaded operator `+()` function operating on user-defined data-items must retain a meaning similar to addition The operator + could perform the union operation on *set* data type, concatenation on *string* data type, etc.

2. Retain Syntax

The syntactic characteristics and operator hierarchy cannot be changed by overloading. Therefore, overloaded operators must be used in the same way they are used for basic data types. For example, if `c1` and c2 are the objects of `complex` class, the arithmetic assignment operator in the statement

```
c1 += c2;
```

sets `c1` to the sum of `c1` and `c2`. The overloaded version of any operator should do something analogous to the standard definition of the language. The above statement should perform an operation similar to the statement

```
c1 = c1 + c2;
```

3. Use Functions when Appropriate

An operator must not be overloaded if it does not perform the obvious operation. It should not demand the user's effort in order to identify the actual operation performed by the operator. The main aim of overloading is to make the program code more readable. If the meaning of an operation to be performed by the overloaded operator is unpredictable or doubtful to the user, it is advisable to use a more descriptive and meaningful function name.

4. Avoid Ambiguity

The existence of multiple data conversion routines performing the same operations, places the compiler in an ambiguous state. It does not know which one to select for conversion. For instance, existence of a one-argument constructor in the destination object's class and operator function also in the source object's class performing the same conversion function, confuses the compiler; it does not know which one to select and issues an error message. Therefore, avoid defining multiple routines performing the same operation, which become ambiguous during compilation. The program `confuse.cpp` illustrates the ambiguity which arises when multiple conversion routines exists in a program.

```
// confuse.cpp: conversion routines for object A's to object B
class B;     // forward specification
class A      // source class
{
   // data members of the class A
   public:
      A()
      {}
      // conversion routine in source, operator function
      operator B()
      {
         B b_obj;
         // convert A class's object into class B's object, b_obj
         return b_obj;
      }
      // other member functions of the class A
};
class B      // destination class
{
   // data members of the class B
   public:
      B()
      {}
      // conversion routine in destination, one-argument constructor
      B( A a_obj )
      {
        // convert source class A's object to initialize data members of B
      }
      // other member functions of the class B
};
void main( void )
{
   A a_obj;
```

```
    B b_obj;
    b_obj = a_obj;
    // other operations on objects of the classes A and B if necessary
}
```

In `main()`, the statement

```
b_obj = a_obj;
```

leads to the following compilation error:

```
Error confuse.cpp 35: Ambiguity between 'A::operator B()' and 'B::B(A)'
                      in function main()
```

It is because the source object `a_obj`'s class `A` has operator conversion function and the destination object `b_obj`'s class `B` also has conversion function in the form of one-argument constructor function.

5. All Operators Cannot be Overloaded

C++ supports a wide variety of operators, but all of them cannot be overloaded (see Table 13.3) to operate in an analogous way on standard operators. These excluded operators are very few compared to the large number of operators, which qualify for overloading.

Operator Category	Operators
Member access	(dot operator)
Scope resolution	`::` (global access)
Conditional	`?:` (conditional statement)
Pointer to member	`*`
Size of Data Type	`sizeof(..)`

Table 13.3: Non-Overloadable C++ operators

An operator such as `?:` has an inherent meaning and it requires three arguments. C++ does **not support** the overloading of an operator, which operates on three operands. Hence, the conditional operator, which is the only ternary operator in the C++ language, cannot be overloaded.

Review Questions

13.1 What is operator overloading ? Explain the importance of operator overloading.

13.2 List the operators that cannot be overloaded and justify why they cannot be overloaded.

13.3 What is operator function ? Describe operator function with syntax and examples.

13.4 Write a program to overload unary operator, say ++ for incrementing distance in FPS system. Describe the working model of an overloaded operator with the same program.

13.5 What are the limitations of overloading unary increment/decrement operator ? How are they overcome ?

13.6 Explain the syntax of binary operator overloading. How many arguments are required in the definition of an overloaded binary operator ?

13.7 Write a program to overload unary operator for processing counters. It should support both upward and downward counting. It must also support operator for adding two counters and storing the result in another counter.

13.8 Write a program to overload arithmetic operators for manipulating vectors.

13.9 Overload `new` and `delete` operators to manipulate objects of the `Student` class. The `Student` class must contains data members such as `char *name`, `int roll_no`, `int branch`, etc. The overloaded `new` and `delete` operators must allocate memory for the `Student` class object and its data members.

13.10 Design classes called `Polar` and `Rectangle` for representing a point in the polar and rectangle systems. Support data conversion function to support statements such as:

```
Rectangle r1, r2;    Polar p1, p2;
r1 = p1;             p2 = r2;
```

13.11 Write a program to manipulate N student objects. Overload the subscript operator for bounds checking while accessing i^{th} `Student` object.

13.12 Why is the friend function not allowed to access members of a class directly although its body can appear within the class body ?

13.13 Write a program to overload stream operators for reading or displaying contents of `Vector` class's objects as follows:

```
cin >> v1;   cout << v2;
```

13.14 Suggest and implement an approach to trace memory leakage.

13.15 State with reasons whether the following statements are TRUE or FALSE:

(a) Precedence and associativity of overloaded operators can be changed.
(b) Semantics of overloaded operators can be changed.
(c) With overloading binary operator, the left and right operands are explicitly passed.
(d) The overloaded operator functions parameters must be user-defined objects only.
(e) A constructor can be used to convert a user-defined data types only.
(f) An object of a class can be assigned to basic type operand.
(g) Syntax of overloaded operators can be changed.
(h) The parameter type to overloaded subscript `[]` operator can be of any data type.
(i) Friend function can access members of a class directly.
(j) The ternary operator can be overloaded.
(k) The compiler reports an error if overloaded + operator performs - operation.

13.16 Design classes such that they support the following statements:

```
Rupee r1, r2; Dollar d1, d2;
d1 = r2; // converts  rupee (Indian currency) to dollar (US currency)
r2 = d2; // converts dollar (US currency) to rupee (Indian currency)
```

Write a complete program which does such conversions according to the world market value.

13.17 Write a program for manipulating linked list supporting node operations as follows:

```
node = node + 2;   node = node - 3;
```

The first statement creates a new node with node information 2 and the second statement deletes a node with node information 3.

13.18 Write a program for creating a doubly linked list. It must support the following operations:

```
firstnode = node;  firstnode += 10;  Node *n = node1 + node2;
```

The doubly linked list class should have overloaded node creation and deletion operator function should appear in the form of overloaded + and – operator functions respectively.

13.19 Write an interactive operator overloaded program for manipulating matrices. Overload operators such as >>, <<, +, –, *, ==.

13.20 Write an interactive operator overloaded program to manipulate the three-variable polynomial:

$a_n x^n y^n z^n + a_{n-1} x^{n-1} y^{n-1} z^{n-1} + \ldots + a_1 x^1 y^1 z^1 + a_0$

14

Inheritance

14.1 Introduction

Inheritance is a technique of organizing information in a hierarchical form. It is like a child inheriting the features of its parents (such as beauty of the mother and intelligence of the father). In real world, an object is described by using inheritance. It derives general properties of an object by tracing an inheritance tree from one specific instance, upwards towards the primitive concepts at the root.

Inheritance allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. Classes are created by first inheriting all the variables and behavior defined by some primitive class and then adding specialized variables and behaviors. In object oriented programming, classes encapsulate data and functions into one package. New classes can be built from existing ones, just as a builder constructs a skyscraper out of bricks, stone, and other relatively simple material. *The technique of building new classes from the existing classes is called inheritance.*

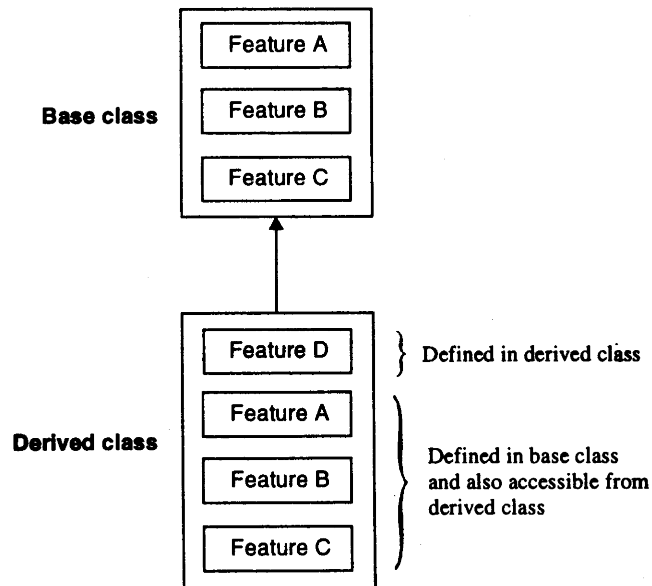

Figure 14.1: Base class and derived class relationship

Inheritance, a prime feature of OOPs can be stated as *the process of creating new classes (called derived classes), from the existing classes (called base classes).* The derived class inherits all the

capabilities of the base class and can add refinements and extensions of its own. The base class remains unchanged. The derivation of a new class from the existing class is represented in Figure 14.1. The derived class inherits the features of the base class (A, B, and C) and adds its own features (D). The arrow in the diagram symbolizes *derived from.* Its direction from the derived class towards the base class, represents that the derived class accesses features of the base class and not vice versa.

A number of terms are used to describe classes that are related through inheritance. A base class is often called the ancestor, parent, or superclass, and a derived class is called the descendent, child, or subclass. A derived class may itself be a base class from which additional classes are derived. There is no specific limit on the number of classes that may be derived from one another, which forms a class hierarchy.

14.2 Class Revisited

C++, not only supports the access specifiers `private` and `public`, but also an important access specifier, `protected`, which is significant in class inheritance. As far as the access limit is concerned, within a class or from the objects of a class, `protected` access-limit is same as that of the `private` specifier. However, the protected specifier has a prominent role to play in inheritance. A class can use all the three visibility modes as illustrated below:

```
class ClassName
{
    private:
        ....        // visible to member functions within
        ....        // its class but not in derived class
    protected:
        ....        // visible to member functions within
        ....        // its class and derived class
    public:
        ....        // visible to member functions within
        ....        // its class, derived classes and through object
};
```

Similar to the private members of a class, the protected members can be accessed only within the class. That is, in the hierarchy of access, privilege code (members and friends) can see the whole structure of an object whereas, the external code can see only the public features. Consider the following definition of a class to illustrate the visibility limit of the various class members:

```
class X
{
  private:
    int a;
    void f1()
    {
     // .. can refer to members a, b, c, and functions f1, f2, and f3
    }
  protected:
    int b;
```